Proceedings of the Linux Symposium

July 23th–26th, 2003
Ottawa, Ontario
Canada

Contents

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Alan Cox, *Red Hat, Inc.*
Andi Kleen, *SuSE, GmbH*
Matthew Wilcox, *Hewlett-Packard*
Gerrit Huizenga, *IBM*
Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Martin K. Petersen, *Wild Open Source, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

UML Simulator

Werner Almesberger[*]
werner@almesberger.net

Abstract

umlsim extends user-mode Linux (UML) with an event-driven simulation engine and other instrumentation needed for deterministically controlling the flow of time as seen by the UML kernel and applications running under it.

umlsim will be useful for a wide range of applications in research and kernel development, including simulations involving the networking code, regression tests, proof of race conditions, validation of configuration scripts, and also performance analysis.

This paper describes the design and implementation of umlsim, gives a brief overview of the scripting language, and shows a real-life usage example.

1 Introduction

Simulation is an effective means for examining properties of systems that are too complex, too volatile, too expensive, or simply too large to build and test in real life.

In the development of the Linux kernel, simulations only play a niche role, and are rarely used for more than helping in the design of individual components. Also for performance evaluation, there is broad reliance on benchmark suites, but little is done with simulations that would allow to pinpoint bottlenecks and regressions with much more accuracy.

umlsim provides an environment that allows the use of regular Linux kernel or application code in event-driven simulations. It consists of an extension of user-mode Linux (UML, [1]) to control the flow of time as seen by the UML kernel and applications running under it, and a simulation control system that acts like a debugger, and that is programmed in a C and Perl-like scripting language.

The key feature of umlsim is that—unlike most other simulators, which implement an abstract model of the system being simulated—it uses the original Linux kernel code, with only minor changes. This reduces the risk of creating simulations that differ in some important details from the original, avoids code forking, and generally shortens the process of designing and building a simulation.

The simulation environment is deterministic, i.e. running a simulation multiple times will produce exactly the same results, although one can of course also introduce real or pseudo randomness. This makes umlsim suitable for regression tests, and for exercising specific execution patterns that exhibit problems.

The project's home page is at `http://umlsim.sourceforge.net/`

One of the first uses of umlsim is to examine the behaviour of Linux TCP in gigabit networks [2], but it will also be useful for many other applications in research and kernel de-

[*]The work presented in this paper is conducted as part of the FAST project at the California Institute of Technology, `http://netlab.caltech.edu/FAST/`

velopment, including regression tests, examination of race conditions and other kernel bugs, validation of configuration scripts, and performance analysis.

This paper is intended for two different audiences: first, it aims to introduce the capabilities and concepts of umlsim to prospective users. Second, it gives other kernel developers an overview of the kernel changes, and describes mechanisms that could also be useful in other projects.

This introduction continues with the historical background and related work. Section 2 discusses overall design and implementation aspects, and section 3 describes the most important elements of the scripting language. A real-life simulation example is given in section 4. We conclude with a discussion of future uses and improvements.

1.1 History

The basic concept behind umlsim, namely to use original kernel and user space code in simulations, was already explored in the earlier tcng ("Traffic Control Next Generation" [3]) project.

The main component of tcng is a compiler that translates traffic control configurations from a high-level language to the low-level commands understood by the tc command-line utility. In that project, a simulator called tcsim is used to validate that these commands are formally correct, and that they also yield the desired behaviour. In particular, since the configuration process involves many inter-related parameters with poorly documented semantics, it happened quite frequently that the use of some parameters or constructs was mis-interpreted.

Simulators usually implement an abstracted model of the system they simulate. In the case of tcng, this approach could lead to a simulator that contains the same mis-interpretations as the program being tested, so both would happily agree on incorrect results.

To avoid this problem, tcsim reduces the amount of abstraction needed by building the simulation environment from portions of the original traffic control code of the kernel, and the tc configuration utility. The structure of tcsim is depicted on the left-hand side of Figure 1.

This approach also allows the use of powerful user-space debugging tools like ElectricFence [4] and valgrind [5] to find bugs in the original code.

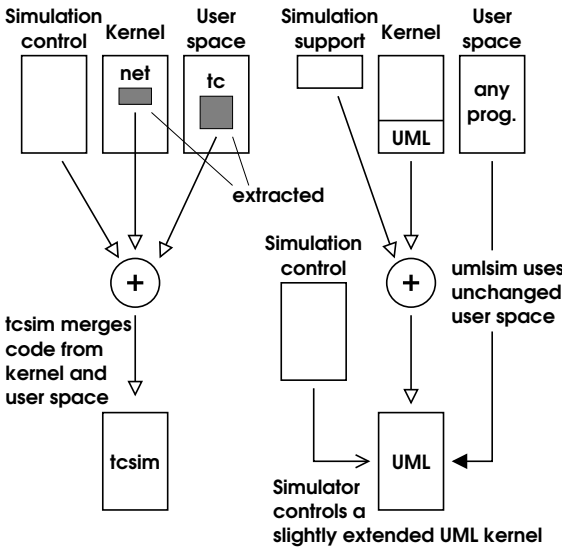


Figure 1: tcsim uses a monolithic approach, with many dependencies on kernel and user space internals. umlsim is modular, and requires only very minor changes.

The difficult part in writing tcsim was to extract precisely the right amount of kernel code, and to make it fit in the simulation environment. In many cases, some small code modifications are needed to eliminate unwanted references to structure elements, variables, and functions not available in the simulator. All this makes the extraction procedure very sensitive to even the

smallest changes.

tcsim was written for 2.4 kernels. Early in 2.5 development, it became clear that the networking code had changed sufficiently to require a major rewrite of the extraction process.

Another limitation of tcsim is that it only covers a very small part of the networking stack. For instance, it would be interesting to use TCP as a reactive traffic source.

The bottom line of the experience with tcsim is that, while using the original source also for the simulator works well, the process of extracting it causes problems and confines the simulator to only a small part of the system. So, why not avoid the extraction step at all, and use the entire kernel?

This is the approach chosen for umlsim, as shown on the right-hand side of Figure 1: instead of extracting the interesting bits from the kernel, it builds on UML, where all the work of making the Linux kernel run in user space has already been done, and adds a few functions for simulation control to it. User space is left completely unchanged.

1.2 Other simulators

Particularly in the area of networking, simulators are rather common tools. In many cases, they focus only on a very limited set of functions, such as a specific protocol. Among the more general simulators, the network simulator ns-2 [6] is certainly the one most widely known.

ns-2 consists of a modular simulation core written in C++, which is configured through scripts written in an extended version of the Tcl scripting language. The core provides network elements, protocol engines, and traffic generators.

umlsim also has a "core," but this core provides only very low-level primitives, and higher level functions are implemented by scripts. On the other hand, large subsystems, such as TCP, are simply reused without needing any special treatment in the simulator, and they behave in every detail like in a real system.

ns-2 is much faster than umlsim, and will probably always be, while umlsim is more general and can also be used for simulations involving other subsystems, instead of or in addition to networking.

2 Simulator design

umlsim consists of a simulation control process (we shall call it simply "the simulator"), and the UML systems that are being studied in the simulation. Besides UML systems, a simulation can also include other processes, e.g. to implement communication services. The general structure of a simulation system is shown in Figure 2.

The simulator executes a script in a C/Perl-like language. Scripts serve two purposes: (1) they define the simulation and control its execution, and (2) they provide the "glue" between the actual simulation and the processes used in it, and also between elements in these processes.

A simulation can choose how closely the simulator and the UML systems interact, i.e. the simulator may just watch a few variables and perform basic synchronization, but exercise no further control over execution, but it may as well intercept even the slightest activity, manipulate variables in the UML kernel, and even alter the flow of execution. Typically, umlsim controls UML at a very low level, but hides most of these interactions inside the simulator and behind library functions that provide a higher level of abstraction.

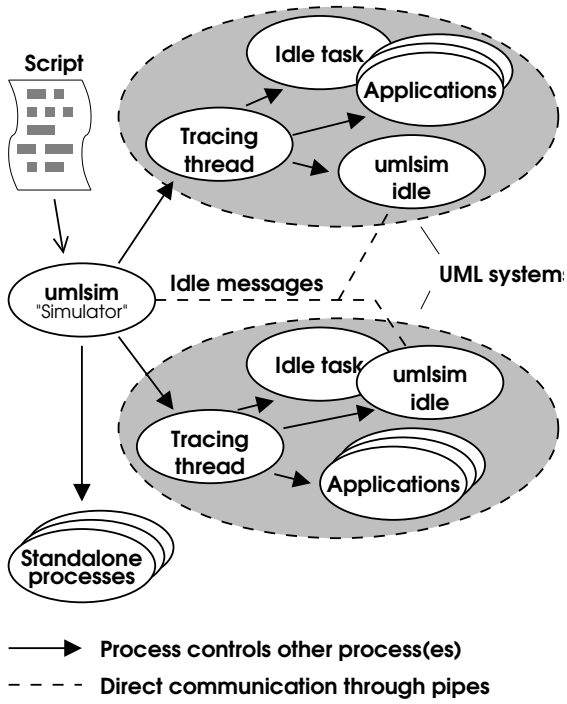


Figure 2: The simulator controls UML systems and other processes. Each UML system in turn consists of several processes.

The simulator basically acts like a debugger, and places breakpoints into the UML kernel. When the kernel is stopped, the simulator can read and change variables. The simulator can also call functions, make them return, etc.

In addition to this, the simulator exchanges time updates with the umlsim idle thread described in the next section directly through a pair of pipes.

Figure 3 shows the current structure of libraries. Work in this area of umlsim is still very much in progress.

2.1 Virtual time

It is frequently desirable to run simulations in a deterministic virtual time instead of real time. umlsim can accomplish this by adding code to the kernel that intercepts all functions reporting

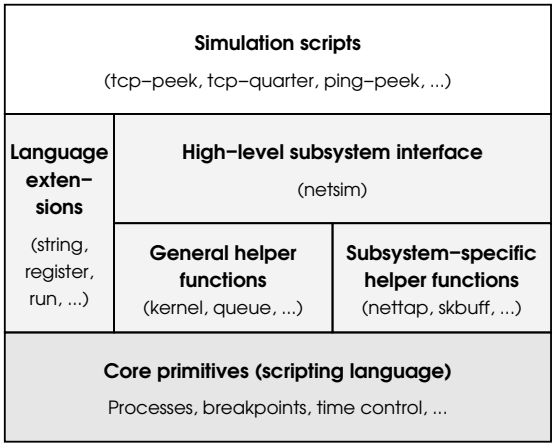


Figure 3: Organization of libraries in umlsim.

or advancing time, and puts them under its own control. This code also introduces a umlsim-specific idle thread that yields to all other tasks, except the kernel's regular idle task.

Whenever the kernel is idle (i.e. no process is scheduled to run), the umlsim code in the kernel does one of the following:

- if a soft-interrupt is pending: it generates a timer interrupt, but does not advance time
- if a timer will expire within the next jiffy:[1] it generates a timer interrupt, and allows `do_timer` to advance time by one jiffy
- if the next timer will expire in the future: the UML kernel reports this back to the simulator, and waits for further instructions

Since the kernel always runs some timers (such as `neigh_periodic_timer` and `rt_check_expire`), umlsim does not need to handle the case of a timeout without further activity.

[1] The "jiffy" is the basic time unit in the kernel. One jiffy typically equals 1–10 ms.

The kernel can also become active when a device interrupt arrives. umlsim currently only handles network events. Instead of using signals (which correspond to interrupts in UML), it calls the functions invoked by interrupt handlers directly.

When all kernels in a simulation report that they are waiting for a timer, the simulator picks the earliest expiration time among these timers (or a timeout specified in the `wait` command, if it is earlier), sets the global simulation time to that value, and updates the local time in all kernels.

2.2 Running UML under a debugger

When running UML under a debugger or a similar program (such as strace), the tracing thread watches the debugger with `ptrace`, intercepts calls to functions like `ptrace` and `waitpid`, and emulates them or redirects them to the process currently executing the kernel. This part of UML is called the "`ptrace` proxy."

Unfortunately, this design allows only a single UML system per debugger, because a process can be watched with `ptrace` by at most one process at any given time.

In order to control multiple UML systems, umlsim forks a forwarder process for each such system. This process communicates with the main simulator through pipes, and executes the `ptrace` calls on its behalf. This is shown in Figure 4.

As an example, Figure 5 shows a simplified flow of control when the simulator performs a `ptrace` call on a UML system.

2.3 Debugging the kernel

There are several idiosyncrasies of kernel code and of the way gcc compiles it that need special attention in umlsim. This section describes some of them.

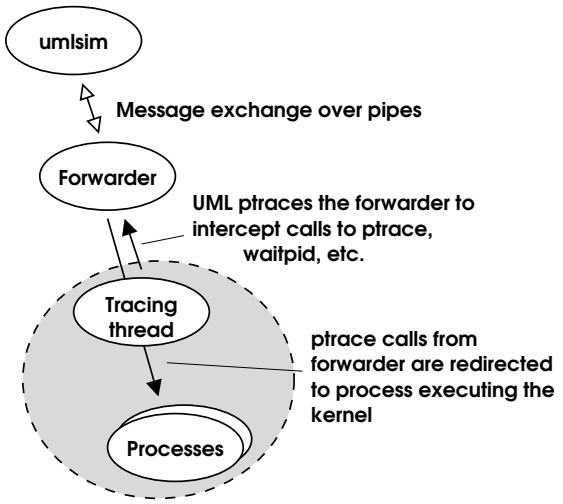


Figure 4: umlsim uses an intermediate forwarder process to "debug" the UML system.

Because the `jiffies` variable is defined in the linker, the debugging information generated by the compiler only contains its declaration, but not its location. umlsim therefore retrieves this information from the symbol table of the kernel executable, and augments the declaration with it.

Some functions use registers instead of the stack to pass arguments (e.g. those declared with `FASTCALL`). umlsim currently does not support or even recognize this.

The kernel makes heavy use of inline functions. One peculiarity of inline functions is that breakpoints in an inline function need to be repeated for each instance of this function. Furthermore, gcc rearranges and sometimes even removes labels (the ones used as targets for `goto` statements) when optimizing. umlsim introduces a mechanism called "reliable markers" that includes an explicit label in the function, which can then be used for breakpoints. Reliable markers also work in functions that

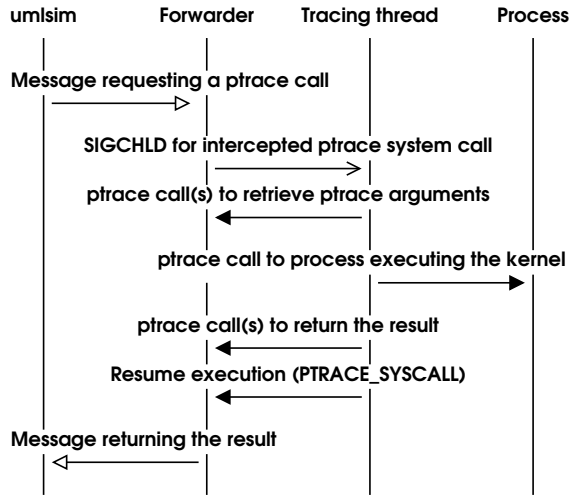


Figure 5: Simplified control flow when the simulator performs a `ptrace` call on a UML system.

are not inlined, and are used as follows:

```
void some_function(int a)
{
    int b = 10;

    MARKER(label_name,a,b);
...
```

Variables that may be accessed by the simulator when stopped at the location of the marker are listed after the label name. This makes sure that the variables in question have a memory location, that they are not cached in registers when passing the marker, and that no code accessing these variables gets moved across the marker. (E.g. in the example above, the compiler might otherwise try to move the initialization of `b` after the marker.)

Also, since many low-level service functions are declared `static inline`, they cannot be called directly. umlsim generates callable instances of the most common inline functions by including their definitions in a file compiled with `-fkeep-inline-functions`.

2.4 Kernel changes

The kernel changes required for umlsim are comparably minor, and most of them are in the area specific to the UML architecture.

umlsim requires the following changes in generic kernel code:

- `calibrate_delay` explicitly waits for a timer interrupt, which would never happen under umlsim, because at that time, the umlsim idle thread does not yet exist. Therefore, umlsim simply skips `calibrate_delay` when using virtual time, and sets `loops_per_jiffy` to one.
- functions are added to `timer.c` to retrieve the expiration time of the next timer.

umlsim replaces the following functions using the linker's `--wrap` mechanism:

- `do_gettimeofday` and `gettimeofday` return the simulation time instead of the system's real time.
- `setitimer` becomes a no-op, because umlsim generates all timer interrupts under its own control.
- a switch is added to control whether a timer interrupt invokes `do_timer`. This way, timer interrupts can be used to run soft-interrupts without advancing the jiffies count.
- `idle_sleep` leads to the timeout handling code of umlsim.

The timeout handling code decides which actions to take (e.g. to raise a timer interrupt

if there are pending soft-interrupts), communicates with the simulator, and maintains the various "current time" variables.

The umlsim patches also add the reliable markers, and callable definitions of common inline functions, which are both described in the previous section.

2.5 Network simulation

When simulating network elements, umlsim builds on the infrastructure used by uml_switch, but the script intercepts the transmit function and replaces most of the UML-specific part of the stack. Figure 6 shows the key functions invoked when sending and receiving packets.

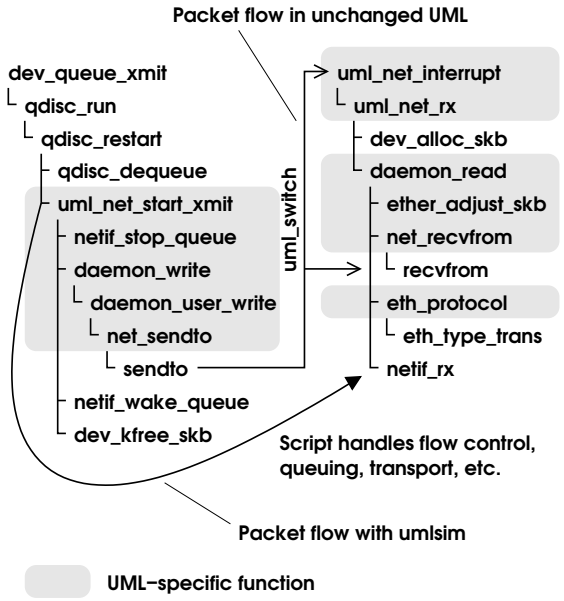


Figure 6: Call sequence and packet flow with and without umlsim (simplified).

With umlsim, the UML-specific networking part is only used for device setup, but all the transport and low-level packet manipulation functionality is provided directly by the simulation script.

umlsim currently only provides a single-link model, which will be extended and generalized in the near future. Figure 7 shows how the device interface and link are implemented. This code can be found in the files `include/netsim.umlsim` and `include/nettap.umlsim` of the umlsim distribution.

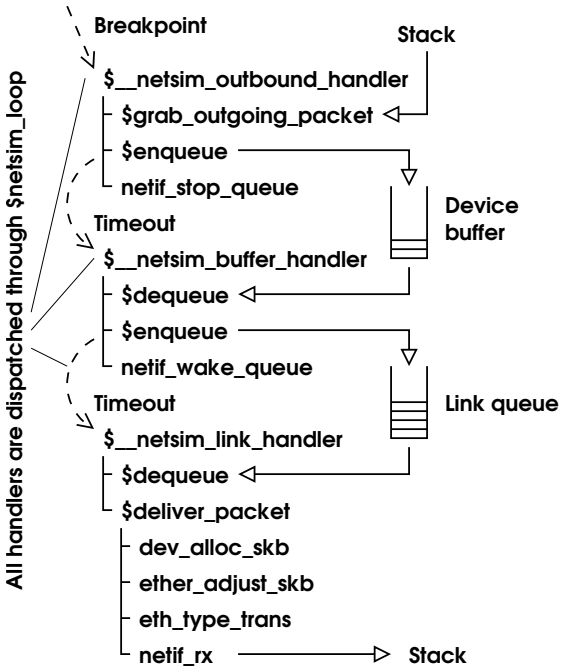


Figure 7: Control flow in script implementing network device and link.

When reaching the breakpoint `uml_net_start_xmit`, umlsim retrieves the packet, calculates the queuing delay, and stores it in an internal queue. If the device queue is full, the script calls the flow-control function `netif_stop_queue`.

When the packet is due for sending, it is dequeued and put into the link queue, from which it emerges after the transfer delay. umlsim then invokes basically the same functions as the original code, and finally pushes the packet to the stack by calling `netif_rx`.

3 The scripting language

The scripting language is mainly based on C and Perl, but it also borrows concepts from the Bourne shell, Pascal, and LISP. This section gives a brief overview of the most important concepts, and differences to similar languages.

umlsim passes scripts through the C preprocessor, so the usual comment handling, macro capabilities, and include files are available.

3.1 Variables and functions in the simulator

The names of variables in the simulator always begin with a dollar sign, like in Perl. Also like in Perl, variables can be used without prior declaration, their value can be of any type, and the type can be change.

An uninitialized variable has the so-called *undefined value*. The undefined value can also be used explicitly, with the construct `undef`, and one can test whether an expression yields the undefined value with `defined` *expression*.

The scripting language, like C, uses lexical scoping, i.e. the visibility of a variable is determined by its location in the program, but not by the sequence of function calls that leads to a specific access.

By default, variables are visible within the enclosing function, but not in other functions. This can be changed by either declaring them `local`, which creates a new, uninitialized instance that is visible in the current block and any blocks inside it, or by declaring them `global`, which creates a new instance in the current function, which is visible also in functions defined inside this function. Example:

```
$a = 5;
{
    local $a = 0;
    $a++;
    {
        $a++;
        printf("inner %d\n",$a);
    }
    printf("middle %d\n",$a);
}
printf("outer %d\n",$a);
```

yields

```
inner 2
middle 2
outer 5
```

The goal of these slightly unusual scoping rules is to avoid explicit declarations as much as possible, but also to avoid the problem of functions accidentally altering global variables, which is common in other scripting languages.[2]

Functions are anonymous, similar to lambda expressions in LISP. In order to reference a function by name, the function has to be stored in a variable. Example:

```
global $gcd = function ($a,$b)
{
    if ($a == $b) return $a;
    return $a > $b ?
      $gcd($a-$b,$b) : $gcd($a,$b-$a);
};

print $gcd(300,90);
```

The scripting language also supports associative arrays. Indices can be integers, pointers, strings, processes, or breakpoints. Elements can be of any type, including arrays. Examples:

[2] Only time—and users—will tell whether this is indeed an improvement over more traditional scoping rules. Users preferring to declare all their variables can set the `-Wundeclared` option to enable warnings when trying to access undeclared variables.

```
$a[0] = 3;
$a["string"] = $a;
print $a["string"][0];
```

3.2 Printing and files

The scripting language has two output statements: the C-like `printf`, and the "smart" and somewhat Perl-like `print`.

`print` accepts a list of items to print, appends a newline after the last item, and it pretty-prints structured types. Example:

```
$proc = uml("linux");
print "xtime = ",xtime;
```

yields

```
xtime = {
    tv_sec = 0 (0x0)
    tv_nsec = 0 (0x0)
}
```

`print` outputs integers as decimal and as hexadecimal numbers, enumeration type members by name, strings and signed character arrays as text strings, and arrays of unsigned characters as a hexdump. Like in Perl, a separator between printed arguments can be introduced by setting the special variable `$,`.

To send output to a file, the file first has to be opened with the `open` function, which has a file name argument like Perl's `open`, but returns a file handle. Then, the file handle can be used as the first argument of `print` or `printf`. Example:

```
$file = open(">tmp");
print $file,"example data";
printf($file,"answer = %d\n",42);
close($file);
```

Data can be read from files with the `read` function, but this is rarely used.

3.3 Control statements

`if-else`, `while` (with `break` and `continue`), and `for` work exactly like in C. There is no `do-while` loop, because `while` can be used in its stead.[3]

`switch-case` is similar to C, with the difference that variable expressions can be used for case labels.

There is no `goto`.

3.4 Processes

Simple programs are started with the function `run`, and UML systems are started with the function `uml`. Both functions return a handle that identifies the process. They also set the "magic" variable `$$` to this value. `$$` always identifies the *current* process, i.e. the process that has most recently been created or stopped, and that is currently being manipulated. `$$` can be changed by the simulator (when a different process becomes current) and by the script (if one wants another process to be current).

`run` and `uml` also support some basic IO-redirection, e.g. `run("/bin/date", ">/tmp/xyz")`

After `run` or `uml`, the process is in the *starting* state, but not yet running. A starting or *stopped* process is run with the `continue` statement.[4]

The `wait` statement is used to make the simulator wait for the next event (process termination, breakpoint, timeout, etc.). If the event is related to a process, `wait` sets `$$` to this process. Example:

[3] . . . and because the way programs are represented internally makes `do-while` somewhat difficult to express. It may be added at a later time.

[4] This `continue` has the process handle as argument, in parentheses, and therefore differs syntactically from the `continue` control statement. If continuing the current process, the parentheses can be left empty.

```
$proc =
  run("/bin/echo","hello world");
continue();
wait();
print $$ == $proc;
```

yields[5]

```
hello world
1 (0x1)
```

3.5 Breakpoints, functions, and timeouts

Breakpoints can be placed at function entry, at the point to which the current function returns, at labels, and at the reliable markers described in section 2.3. Breakpoints are set with the `break` function, which returns a handle that identifies the breakpoint.

In the example below, we set breakpoint `$b1` at the entry of the `main` function in the current process, breakpoint `$b2` at the label or reliable marker `label` inside the main function, and breakpoint `$b3` at the location to which the current function returns.

```
$b1 = break(main);
$b2 = break(main.label);
$b3 = break(return);
```

When reaching a breakpoint, umlsim sets `$$` to the process in which the breakpoint is located, and `$!` to the breakpoint handle.

When calling a function in the process, also a breakpoint is generated. This breakpoint is triggered when the function returns. The return value of the function is stored in the special variable `$?`. Example:

```
$b = call fn(1,2,3);
continue();
...
wait();
if ($! == $b)
    printf("result = %d\n",$?);
```

This example shows an *asynchronous* call, because other breakpoints, timeouts, or events in other processes can be handled before the function returns. If this flexibility is not needed, one can use the simpler *synchronous* form, which does not change `$!` or `$?`. Example:

```
printf("result = %d\n",fn(1,2,3));
```

Breakpoints can be removed implicitly, by destroying all references to them, or explicitly with `delete(`*breakpoint*`);`

A script can not only call functions in a process, but it can also make a function return. For example, `$__netsim_outbound_handler` in Figure 7 forces `uml_net_start_xmit` to return, without executing any code of that function, with `$$.return 0;`.

Besides terminating or reaching a breakpoint, a process may also stop with a timeout. Timeouts are specified with a time argument to `wait`. When the specified absolute time is reached, `wait` sets `$$` to the undefined value, and the "current time" variable `$@` to the timeout, rounded up to the next nanosecond. Example:

```
wait(10.2);
  /* wait until t = 10.2 seconds */
if (!defined $$) print $@;
```

If more than one timeout can occur at a given time (e.g. packets arriving within the same

[5] After warning that `/bin/echo` has neither symbols nor debugging information, so there is very little umlsim can do with this process.

nanosecond at different points in the simulation), `wait($@)` must be called after handling each event, so that breakpoints reached when handling a timeout can be processed before handling further timeouts. An example for using `wait($@)` can be found in the event loop at the end of `include/netsim.umlsim` in the umlsim distribution.

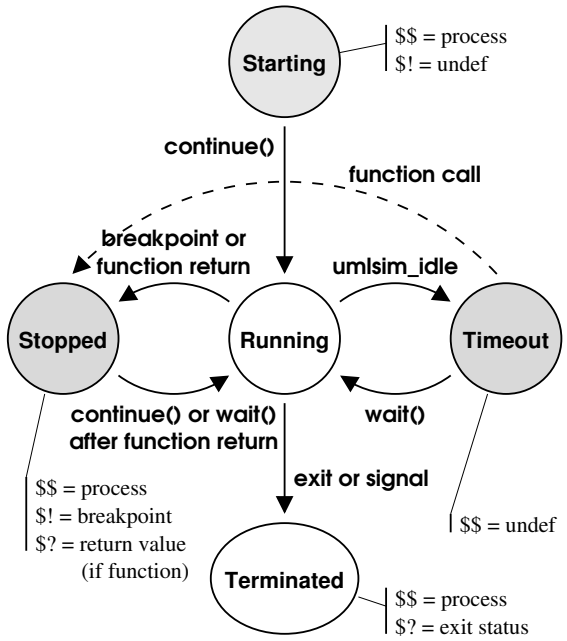


Figure 8: User-visible process states. "Function call' and "return" refer to asynchronous function calls.

Figure 8 summarizes the process states described in this section. States shown in grey allow manipulations of the process, such as the creation of new breakpoints, access to variables, or function calls. A function call from *timeout* puts the process in a state equivalent to *stopped*, but it does not affect any of the special variables.

3.6 Data in a process

umlsim scripts can directly read and write variables in a process, follow pointers, select struct or union members, and so on.

The basic operation is to access a variable. In many cases, simply specifying the variable's name is enough, e.g. given the example program below, `foo` retrieves the value 42.

```
static int foo = 42;

int main(void)
{
    static int bar = 5;

    MARKER(stop_here,bar);
    return bar;
}
```

Accessing `bar` is more complicated. If the program has not been started yet, umlsim looks for variables only in the global scope. To access a variable local to a function, it has to be qualified with the function name, i.e. `main.bar`.

If the program is stopped at the label `main.stop_here`, umlsim searches the local scope first, so just `bar` is sufficient.

Variables, functions, and labels can also be qualified with the process and the compilation unit. Compilation units are in double quotes. Examples:

```
$b1 = break($proc.main);
$b2 = break("fs/ext2/super.c".
  parse_options);
"drivers/net/tun.c".debug = 1;
"tun.c".debug = 0;
```

Since distinct processes may use the same name for different types, also struct or union tags can be qualified, e.g. `struct $proc_a.sk_buff`.

Type definitions with `typedef` differ from C in that umlsim cannot usefully distinguish at parse time between typedef names and other identifiers. Therefore, typedef names are always prefixed with the keyword `typedef`,

e.g. `typedef pte_addr_t`. Like all other names, they can be qualified. For convenience, the C99 standard integer types `uint32_t`, `int8_t`, etc. are predefined.

Conflicts between C identifiers and keywords of the scripting language (e.g. `printf`) can be resolved by escaping the word with a backslash when the C identifier is meant, e.g. `\printf`.

A peculiarity in the way umlsim handles data are array copies: when accessing an object of array type, the entire array is copied. To obtain a pointer to the array, the & operator must be used. Example:

C program fragment:

```
int a[10];
int b[10];
```

umlsim script:

```
$array = a;
b = a; /* memcpy equivalent */
$ptr = &a;
```

This can also be used in type casts. E.g. the following construct copies the content of a network packet:

```
$pkt = (unsigned char [skb->len])
  skb->data;
```

4 Simulation example

In this section, we use umlsim to demonstrate a bug in Linux TCP, and to show the effect of a possible fix. The problem in question, which was first observed on a simulator by Cheng Jin, is that Linux TCP[6] decreases the congestion window (*cwnd*, TCP's estimate of how many packets can be "in flight" for a given connection) too much if there are multiple packet losses in a single round-trip time.

When a packet is lost, TCP assumes that this was due to congestion, and reduces *cwnd* by half. However, if multiple losses occur within a single round-trip time, they should be treated only like a single loss. Linux TCP does not do this, and may reduce *cwnd* to as low as a quarter of the original value. This causes TCP to send data a little slower than it would be allowed to.

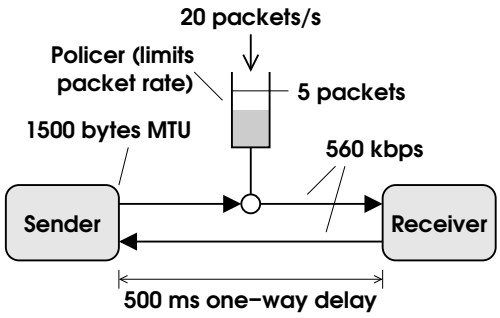


Figure 9: Network setup used in the simulation.

Figure 9 shows the network configuration used in the simulation: the TCP sender and receiver are connected by a single link with a round-trip time of one second. The maximum throughput is rate-limited to twenty packets per second. We simulate the transfer of a 1 MB file.

The left-hand side of Figure 10 shows the transfer with an unchanged 2.5.66 kernel. `snd_cwnd` is the congestion window, in segments. `snd_ssthresh` marks the point where TCP switches between "slow start" and "congestion avoidance" mode. `snd_cwnd` should not fall below `snd_ssthresh`. `snd_una` is the number of bytes that have been acknowledged by the receiver. `packets lost` is the cumulative number of packets dropped by the rate limiter.

For the second simulation, we use the same

[6] Most if not all 2.4 and 2.5 kernels are affected. At the time of writing, this bug still exists in the mainstream kernel. The entire discussion can be found at [7].

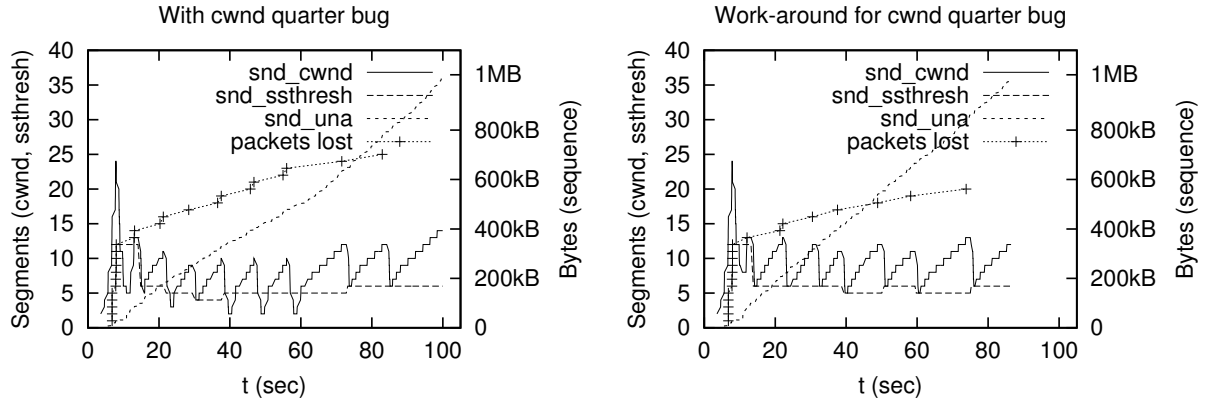


Figure 10: Simulated transfer with and without the "cwnd quarter" bug.

kernel, but set a breakpoint at the beginning of `tcp_cwnd_down`, and execute a replacement in the script instead of the original function. This replacement implements a fix that keeps *cwnd* from being lowered too far.

With this work-around in place, `snd_cwnd` never falls below `snd_ssthresh`, and the transfer finishes considerably earlier than in the buggy version.

5 Future work

As a complex but relatively young project, umlsim still has shortcomings in many areas. This section discusses some of the problems, and outlines approaches for solving them.

Future work on umlsim will primarily focus on the needs of network simulations, and in particular the analysis of TCP performance.

5.1 Functionality

Networking simulations are essentially limited to a single-link scenario at the time of writing. Work is under way for providing building blocks that allow the construction of arbitrary network topologies.

Another issue all but ignored so far is portability to architectures with other byte order or word size than ia32. Also support for multiprocessing is absent so far.

The simulator has currently no direct control over processes running in the user space under a UML kernel. It would be useful if simulations could treat such processes like ordinary processes, i.e. by launching them with a simple command, by placing breakpoints, etc.

It would be interesting to explore the possibility of using umlsim to reconstruct the internal state of the kernel, based on traces obtained from "live" systems. For example, this could be used to explain anomalies in network activity captured with tcpdump. The open issue here is how quickly unavoidable time differences and events not recorded in the trace (such as soft-interrupt execution after a hardware interrupt) will cause the simulation to diverge from the original system, and how such errors can be compensated.

5.2 Usability

umlsim today is clearly a hacker's toy. Most users will want high-level components when implementing their simulations, and the script-

ing language could also use some minor cleanup.

Dark corners of the language include the cast operator, pointers to data in the simulator, inconsistencies in the syntax (e.g. normally, `$$.`*thing* is equivalent to just *thing*, but `return` is very different from `$$.return`), and subtle differences in the semantics of array indices and `case` expressions.

To be useful outside the kernel hacker community, umlsim needs libraries with application-oriented building blocks that provide a convenient level of abstraction. At the time of writing, such a library is slowly emerging for networking, with the main focus on TCP.

Beyond libraries, also preprocessors that translate simpler application-oriented languages, like the one used by tcsim, to umlsim may be useful.

One of the most important aspects of simulations is the visualization of results. While it is desirable to retain a maximum of flexibility, examples for data formats, and visualization packages for common tasks will help users to obtain results more rapidly.

Also, as befits a hacker's toy, documentation is incoherent and spotty.

5.3 Performance

At the time of writing, umlsim is rather slow. While some optimization work has been done to reduce startup time and to accelerate some lookup operations, and more recently also to accelerate the communication between the simulator and the UML processes, several areas remain where major speed improvement are possible.

`ptrace` is a rather inefficient means for accessing process memory. It would be better if the simulator entirely bypassed the tracing thread when reading or changing variables, and accessed the address space of the UML processes directly.

Also the performance of UML itself is the object of on-going work [8]. In particular, the so-called "skas mode" ("skas" stands for "Separate Kernel Address Space") has been added recently, to accelerate context switches of processes under UML [9]. By following these changes, umlsim will permit UML to run faster, which in turn will benefit overall system performance, and may perhaps also itself be able to access UML systems more efficiently.

Last but not least, several algorithms and data structures inside the simulator are rather inefficient, and will have to be improved for larger simulations. For example, associative arrays just store their elements in a linear list. Also, results of identifier lookups could be cached.

6 Conclusion

umlsim provides the infrastructure for turning the (UML) Linux kernel into a versatile event-driven simulator, that can be customized using a scripting language most programmers will find easy to learn.

The next challenges in the project will be to bring performance closer to that of comparable simulators, to improve overall usability, to apply umlsim to concrete problems, and to use experience gained from such real-life applications to further improve the simulator.

References

[1] Dike, Jeff *et al. The User-mode Linux Kernel Home Page*, `http://user-mode-linux.sourceforge.net/`

[2] C. Jin, D. Wei, S.H. Low, G. Buhrmaster, J. Bunn, D.H. Choe, R.L.A. Cottrell, J.C. Doyle, W. Feng, O. Martin, H. Newman, F. Paganini, S. Ravot, S. Singh. *FAST TCP: From Theory to Experiments*, Submitted for publication in IEEE Communications Magazine, Internet Technology Series, April 1, 2003. `http://netlab.caltech.edu/pub/papers/fast-030401.pdf`

[3] Almesberger, Werner. *Linux Traffic Control—Next Generation*, Proceedings of the 9th International Linux System Technology Conference (Linux-Kongress 2002), pp. 95–103, September 2002. `http://www.linux-kongress.org/2002/papers/lk2002-almesberger.html`

[4] Perens, Bruce. *Electric Fence*, `ftp://ftp.perens.com/pub/ElectricFence/`

[5] Seward, Julian; Nethercote, Nick. *Valgrind, an open-source memory debugger for x86-GNU/Linux*, `http://developer.kde.org/~sewardj/`

[6] *The Network Simulator – ns-2*, `http://www.isi.edu/nsnam/ns/`

[7] Almesberger, Werner; Jin, Cheng; Kuznetsov, Alexey. *snd_cwnd drawn and quartered*, Discussion thread on the netdev mailing list, December 25, 2002. `http://marc.theaimsgroup.com/?t=104078129500001`

[8] Dike, Jeff. *Making Linux Safe for Virtual Machines*, Proceedings of Ottawa Linux Symposium 2002, pp. 107–116, June 2002. `http://www.linux.org.uk/~ajh/ols2002_proceedings.pdf.gz`

[9] Dike, Jeff. *skas mode*, `http://user-mode-linux.sourceforge.net/skas.html`

SCSI Mid-Level Multipath

Michael Anderson, Patrick Mansfield
IBM Linux Technology Center
andmike@us.ibm.com, patmans@us.ibm.com

Abstract

Multipath IO is the ability to address the same storage device over multiple connections, providing improved reliability and availability. This concept is not new to Linux®. Multipath capabilities exist in the volume management layer, SCSI upper level, and in the SCSI lower level device driver. This paper examines an approach to providing multipath support in the Linux 2.5+ SCSI mid-level. An implementation at this level gives the reduced resource usage and better performance of lower level implementations, along with the device independent capabilities of upper level implementations.

The target audience is developers knowledgeable about SCSI or Linux SCSI internals that are also interested in multipath storage support.

1 Introduction

This section provides an overview of the characteristics of the multiple paths and multiple ports presented to the Linux kernel by the storage IO transport and by the storage device itself.

A path is the connection between the host system and the storage device. Multiple paths to a device result from the storage device having more than one port (multi-port storage device) or the host system having multiple connections into a given bus or fabric that connects to a storage device port (multi-initiated interconnect).

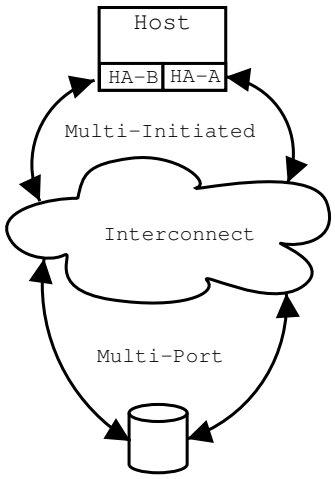


Figure 1: Multi-Initiated and Multi-ported multipath configuration

Utilization of a multipath device can increase the availability of the storage device presented to the operating system by reducing the loss of access due to a single transport problem. Multipath device support may also provide an increase in performance due to load balancing if the performance attributes of the storage device are greater than a single transport can deliver. When a system architecture like NUMA exhibits increased latency between local memory and non-local adapters multipath with NUMA aware routing can be used to route IO to adapters with the lowest latency.

Although the necessity for multipath in enterprise systems is clear, the selection of where to

implement it has lead to several different approaches in the Linux kernel. This SCSI mid-level multipath solution was created to address the following:

- Implementation at a level higher in the stack than vendor unique lower level driver solutions, while not exposing device specific knowledge to layers above SCSI.
- Reduction of kernel resources while still utilizing the existing IO scheduler and interfaces of the block layer.
- Binding paths to devices without obtaining information from the devices media, allowing support for both block and character devices.
- The ability to distinguish between path and device errors.
- Selection of the optimal path for IO based on Lower Level Device Driver (LLDD) attributes, NUMA topology, and device attributes.

1.1 Multi-Initiated Interconnect

A multi-initiated interconnect results from attaching multiple host adapters to a single interconnect. For example, a multi-initiated SCSI bus or multi-initiated Fibre Channel.

Multi-initiated paths to a single storage device normally present equal characteristics. Some hardware platforms can create performance inequalities down separate paths to a storage device port when different latencies exist between a host adapter and the memory it is referencing. A NUMA architecture based platform can present such latencies; depending on the magnitude of the latency, platform specific routing policies can increase performance. Path selection will be discussed in detail later in the document.

1.2 Multi-Port Storage Device

A storage device can present multiple protocol communication ports to an IO interconnect. These ports can be accessed from the host through a single host bus adapter (single initiator) or multiple host bus adapters (multi-initiated).

While the performance characteristics of IO down multiple paths to a single port of a device are nominally equal (excluding NUMA), the performance to different ports of the device can vary greatly due to the architecture of the storage device.

These different storage device architectures can be grouped into three behavior models based on the device's differing response to IO sent to more than one port. A device may exhibit differing port behavior only on ports that cross logical unit ownership boundaries. Some storage devices can be configured to operate in more than one behavior mode.

- **Failover** – When a device is operating with Failover behavior, IO to a secondary port must be preceded by control commands indicating a redirection of all IO to an alternate port. Once the storage device has transitioned, "Failed over" IO may be directed to an alternate port. This behavior model is exhibited in devices with some performance penalty in the transition of logical unit ownership. Clustered shared storage systems may use this model to keep port thrashing from degrading performance.
- **Transparent Failover** – IO to a single volume should only be sent to a single port until an availability condition arises to cause IO to be redirected to a secondary port. The storage device will transition transparently to using the secondary port

on the receipt of the first IO to this port. The storage device transition delta for this first IO is significant when compared to subsequent IOs, such that performance would degrade noticeably if port transitions occurred frequently (i.e., if round robin routing policy were used).

- **Active Load Balancing** – IO to a single storage device volume can be sent down any path without degrading performance (note: some cache warmth benefits may be achieved by using more sophisticated path selection algorithms, but this is vendor unique). These storage devices usually have a cache that can be symmetrically accessed from any input port or a single cache that all input ports feed into.

1.3 Linux Multipath Implementations

Multipath support to a storage device can be implemented at different levels in the Linux operating system's IO stack. This support can be provided by storage vendors, adapter vendors, and the base kernel.

- **Volume Management** – Multipath at this level is usually implemented as a modified case of existing RAID support. Multiple block devices exposed by the operating system point to the same storage device and are configured to be failover paths for IO. The "md" driver [3] and LVM multipath patch [2] are examples of support at this level.

- **Upper Level** – Support at this level involves chaining or linking the multiple block devices exposed by the operating system as failover paths. An example of this type of implementation is the T3 Multipath failover driver written by Linuxcare Inc. [5].

- **Mid-Level** – This is the level of implementation described in this document.

- **Lower Level** – An implementation at this level involves a binding of common vendor adapters exposing only one device to the operating system. On failure, the adapter driver re-drives the IO through another adapter previously paired as a failover adapter. An example of this type of implementation is the Qlogic Fibre Channel failover driver.

2 Data Model

2.1 Current Linux SCSI Device Data Model

This section provides a high level overview of the Linux 2.5 SCSI subsystem data structures with a focus on providing a background for later discussion on multipath support. General Linux SCSI information can be obtained by viewing the "The Linux 2.4 SCSI subsystem HOWTO" [1] and [4] listed in the Reference section.

Each LLDD that wishes to register with the Linux SCSI sub-system provides a SCSI host template (`Scsi_Host_Template`) data structure that describes the capabilities of the driver and interface functions.

The LLDD can register with the SCSI subsystem in two ways. One method is a Legacy interface, which is driven from the SCSI mid-level code and calls into the LLDD detect routine, which causes `scsi_register()` to be called for each adapter card detected by the driver. The other method allows the LLDD to call `scsi_register()` directly and then call `scsi_add_host()` when it is initialized and ready to be scanned. These two methods result in a `Scsi_Host` data structure being allocated for each instance of an adapter card.

If an adapter contains multiple busses or channels (not a PCI bridge of two cards), there will be only one SCSI host structure (`Scsi_Host`) created.

After a kernel boot, a insmod of a LLDD, or hotplug event, a list (`scsi_hostlist`) will contain a SCSI host structure representing each adapter detected.

During device scanning a SCSI device data structure (`scsi_device`) will be allocated for each logical unit discovered. Each SCSI device structure will be added to a linked list member of its SCSI host parent.

See figure 2 for a diagram of the data structures and their relationships.

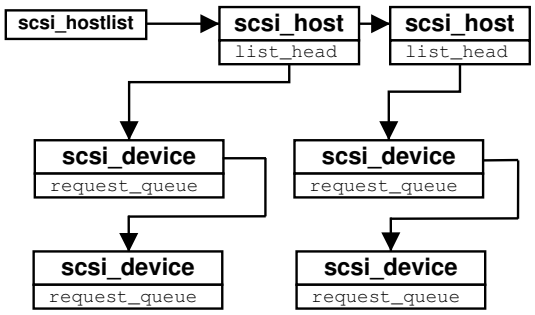


Figure 2: Linux SCSI Data Structures

Once the scanning phase is complete a `struct scsi_device` will be associated to only one `struct Scsi_Host`.

2.2 Mid-Level Multipath Data Model

When no multipath capabilities are enabled in the Linux SCSI subsystem, multiple paths result in a SCSI device structure being created and eventually exposed through the block or character layer for each path. This redundancy wastes system resources and creates a non-optimal presentation of structures to the block layer and user level.

The mid-level multipath implementation coalesces these extraneous SCSI device structures while still maintaining the information relating to the paths.

The child relationship of the SCSI device structure to the SCSI host structure is removed and a new relationship is created using the mid-level multipath structures. The SCSI multipath structure (`struct scsi_mpath`) is a container for all the paths to the storage device. The SCSI mpath structure also contains information on the path routing policy, a count of active paths, and a reference to where the last IO was routed. The `scsi_mpath` structure is associated with the SCSI device structure through a new member, `sdev_paths`.

Each path is represented by a SCSI path structure (`struct scsi_path`). This path structure contains a fast reference to the next sibling path, the state of the path, and a SCSI nexus structure (`struct scsi_nexus`).

The nexus object contains the transport specific knowledge to communicate with the storage device. In an ideal world, this nexus would be an opaque object (i.e., a handle) that was handed to the SCSI mid-level during the device scanning process.

See figure 3 for a diagram of the multipath data structures and their relationships.

2.2.1 Multipath Data Model General Applicability

The data model presented for multipath has advantages even in non-multipath cases.

Because the mid-level model has separated the request queue presented to the block layer from the nexus object that is associated with the SCSI hosts, paths containing nexus objects can be added to or removed from a SCSI device

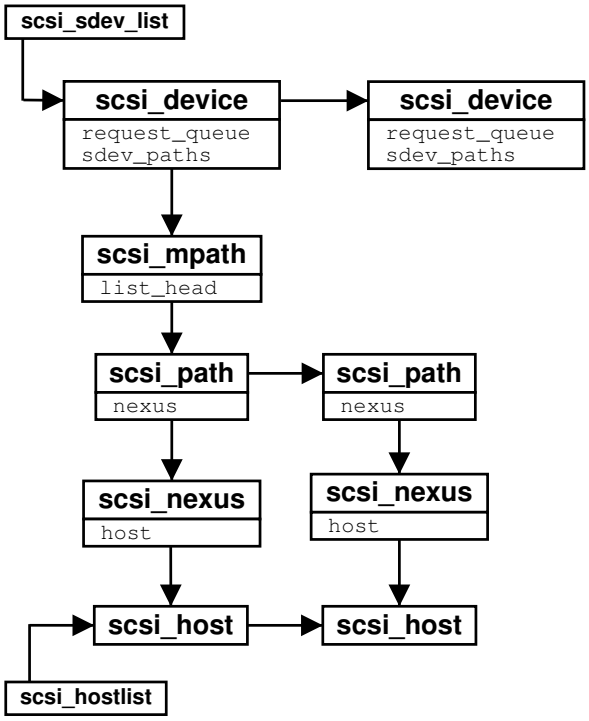


Figure 3: Mid-Level Multipath Data Structures

structure.

When all paths have failed or there are no paths to a device, a policy could be created to allow the SCSI device structure to remain in place, but suspend SCSI IO request processing. This would allow block and file system level attachments to remain established while transport connectivity is in flux.

If UUID authentication can be ensured (this would be the case for multipath devices supported by this implementation) new paths could bind to the SCSI device and IO request processing would resume. If authentication cannot be ensured, lower level resources can be released in less time than current structures allow.

Depending on the future direction of the SCSI mid layer, this separation could be used to add or remove SCSI subsystem components while IO is active.

3 Mid-Level Multipath

3.1 Linux SCSI Scanning Overview

Following is an overview of the SCSI scan algorithm for a given logical unit within `scsi_scan.c` as pertains to modifications for use with multipath (per code in linux version 2.5.68).

A call to `scsi_alloc_sdev` allocates and initializes a `scsi_device` (sdev). Note that a `sdev`, logical unit, and I_T_L nexus are all equivalent in the current linux SCSI code. The LLDD `slave_alloc()` function is called for the `sdev` [4].

The `sdev` is sent an INQUIRY. Device attributes settings are obtained by calling `scsi_get_device_flags()`.

If the logical unit responds, and it has a logical unit configured, the `sdev` is left in place, otherwise it is removed.

`scsi_load_identifier()` function is called to get a UUID (universal unique identifier) via SCSI INQUIRY VPD pages [6], and the result is stored in `sdev->name`.

`scsi_device_register()` is called, generating a hotplug event.

Last of all, the LLDD `slave_configure()` function is called for the `sdev`.

Upper level attaches are done after all scanning (on insmod or initialization of a LLDD) or the upper level attach is done after a single logical unit is scanned via /proc/scsi/scsi (in `scsi_add_device()`). These in turn can generate their own set of hotplug events as the upper level drivers (sd, st, sr, and sg) attach to

each `sdev`.

This means that a series of hotplug events occurs for many scsi_devices, followed by a series of hotplug events for each upper level device (such as a block device).

3.2 Scan Modifications for Multipath

The scanning code is changed as follows for mid-level multipath support.

The allocation of an `sdev` is modified to not only allocate the actual `scsi_device`, but to also allocate and add a single path (`struct scsi_path`, including a `struct scsi_nexus`) to the `sdev`. `slave_alloc()` is modified to take both an `sdev` and `scsi_nexus` as an argument, such that it can access both logical unit data (in the `sdev`) and nexus specific data.

Future plans are to supply a set of parallel slave_nnn interfaces for use with multipath, so that existing drivers not supporting the new interfaces will behave as if the multipath patch were not applied (each path to a storage device will generate a new `scsi_device`).

After a UUID is retrieved, all existing `sdev`'s are searched for a match.

If no match is found, this is the first path to the device, and it is handled the same way as the current non-multipath code.

If a match is found, the new path is added to the matching `sdev` (the paths are coalesced), and the current `sdev` is freed.

`slave_configure()` is also modified to take both an `sdev` and a `scsi_nexus` as arguments.

3.3 UUID

The immutability of the UUID is key to determining if more than one nexus can access the same storage device. This is not a simple problem to deal with, as some devices return no UUID, some return a UUID that is not unique, and others require device specific methods to retrieve a truly unique UUID. Future changes (such as a UUID white list) are required to properly handle the UUID in all cases; user level scanning would simplify the problem.

Discussions were actively in progress at the time this paper was written on whether or not to keep the existing UUID retrieval code in the kernel. Depending on the outcome, and amount of time available, the multipath patch might have to carry the UUID retrieval.

The primary problem with moving UUID retrieval to user level (for use with multipath, assuming full user level scanning is beyond the scope of the current implementation) is that the current scan and upper level attach are initiated without the ability for user level intervention - all upper level devices are attached to all `scsi_devices` with no synchronization from user space.

Without the coalescing of paths as described above devices can show up multiple times, leading to potential resource shortages (memory as well as major/minor numbers), and potential problems for applications dependent on the hiding of duplicate paths.

Further investigation is needed to determine if it is practical to modify the scan and upper level attach to be user initiated (versus the more difficult problem of complete user level scanning). Such that all devices are scanned in kernel code, and then from user level: UUID's retrieved, coalesced, and then upper level attaches triggered.

3.4 Current SCSI I/O Request Flow Overview

Following is an overview of the current IO request flow as it pertains to functions modified for use with SCSI mid-level multipath IO.

A SCSI device structure request queue member (`request_queue`) is registered with the block layer at SCSI scan time for en-queuing requests. Both SCSI character and block devices utilize this queue, as do the commands issued during SCSI scan and by upper level attachment.

For block IO devices, an IO request is sent to the block layer via the `__make_request()` function. In turn it eventually calls the SCSI block request function, `scsi_request_fn()`.

For SCSI character devices, scanning, and commands sent during upper level attaches, `scsi_do_req()` or `scsi_wait_req()` are used to send SCSI commands to the `scsi_device`. These functions setup the `sr_done` function pointer, insert a request, and trigger a call to the `scsi_request_fn()` via a call to the block queue `blk_insert_request()`.

The `scsi_request_fn()` function retrieves a request and calls the `scsi_prep_fn()` via a call to the `elv_next_request()`.

`scsi_prep_fn()` allocates and initializes the `scsi_cmnd`. The `scsi_cmnd done` function is set in upper level drivers via calls to their `init_command` functions. For users of `scsi_wait_req()` or `scsi_wait_req()`, the `done` function is set to the `sr_done`.

The `scsi_cmnd` is the key data structure used to issue a request to a LLDD.

Control continues in `scsi_request_fn()`, where resource limitations and hardware limits (such as queue depth) are checked via calls to `scsi_dev_queue_ready()` and `scsi_host_queue_ready()`.

If resources are available, `scsi_dispatch_cmd()` is called, it adds a timeout, and transfers control to the LLDD by calling the `scsi_host queuecommand` function, passing the `scsi_cmnd`, and `scsi_done()`.

The LLDD is responsible for sending the command to the actual logical unit. After the request is submitted, `queuecommand` returns.

Upon completion of the IO request, the LLDD calls the `scsi_cmnd scsi_done()` function.

`scsi_done()` puts the completed command onto a per-CPU queue, and raises the SCSI_SOFTIRQ.

`scsi_softirq()` determines the completion status of each `scsi_cmnd` via calls to `scsi_decide_disposition()`.

`scsi_decide_disposition()` classifies the completion status of the IO (the scsi_cmnd) and returns the following values to `scsi_done()`, that lead to further actions:

`SUCCESS`: the IO has completed without error, the command is completed by calling `scsi_finish_command()`. This includes an IO completion with failures (for example, an IO went to a disk, but had media errors).

`ADD_TO_MLQUEUE`: the IO completed with a SCSI QUEUE FULL status. The command is re-queued for a retry via `scsi_queue_insert()`, effectively resending the command.

`NEEDS_RETRY`: the IO had a temporary or other condition such that it can be immediately

resent, resend the IO (without re-queuing it) by calling `scsi_retry_command()`.

`FAILURE` or any other value: a device or transport error occurred. Call `scsi_eh_scmd_add()` to queue the failed command for error handling; when the host adapter has no more IO outstanding (when the `active_count` equals the `host_failed` count) the error handler wakes up and handles all failed commands.

`scsi_finish_command()` is the main path for IO completion, it calls the `scsi_cmnd done` function, either the upper level completion function (for sd, `sd_rw_intr`) or the function specified in `scsi_wait_req()` or `scsi_wait_done()`.

3.5 Modifications for IO Path Selection and Path Failures

The SCSI code is modified as follows for mid-level multipath.

A path (including nexus) is selected via a call to `scsi_get_best_path()` from `scsi_request_fn()`.

Path selection is affected by the number of paths, path state, path policy, and NUMA topology.

A list of all available paths and all active paths to a device are kept. In addition, for NUMA systems, there is a list of paths local to a given node.

If no active paths are available, the `scsi_request_fn()` function fails the IO request.

Currently, path selection policy is controlled via the global variable `scsi_path_dflt_path_policy`. This is set via the kernel config and can be modified at boot time, with future plans to allow setting this both per device and via sysfs.

Setting `scsi_path_dflt_path_policy` to SCSI_PATH_POLICY_LPU (1) sets the path selection policy to last path used. This means that another path will only be used on failure.

Setting `scsi_path_dflt_path_policy` to SCSI_PATH_POLICY_ROUND_ROBIN (2) sets the path selection policy to round robin. This means that paths will be rotated across all available paths on every request sent to the device.

A last-path used policy is safest for general purpose use (for example with a device using a transparent failover model). Future plans are to add device specific attribute hooks and code to fully support a transparent failover model, so that round-robin path selection can be done across a subset (lowest path weight) of all paths, not just a single path.

NUMA path selection where possible picks a path local to the node containing the memory to be used for the IO operation. If no local path is available, all paths are valid for selection. So, for round-robin path selection, path selection is either round-robin with respect to all paths local to a given node, or for all paths to a device.

Current NUMA multipath support is limited to a one-to-one mapping of path to node. Future plans are to support multiple nodes connected to the same path (the same bus). Changes are required in the current kernel NUMA topology in order to support topologies that have varied or unequal distances (that is, where the inter-node distances can vary), or for cases where a node contains no CPU's.

For multipath, the `scsi_cmnd device` is changed from a `struct scsi_device`

pointer to a `struct scsi_nexus` pointer; the `scsi_nexus` retains the same fields as those used by the LLDD in order to determine the nexus (that is, the `scsi_nexus` contains a `scsi_host` pointer, `channel`, `id` and `lun`).

Then as part of the path selection, the `scsi_cmnd` device pointer is set to point to the selected path's nexus. (Renaming the `scsi_cmnd` device to `nexus` would be appropriate.)

The LLDD queuecommand function is called, passing the `scsi_cmnd` that includes a pointer to a `scsi_nexus`. Existing LLDD code can then be used, with no changes required to the core of the LLDD.

Upon IO completion, `scsi_path_decide_disposition()` categorizes failures as transport (path failure) or device specific (device failure). Path specific failures cause a path to fail, and the IO can be retried on any remaining paths. Device specific failures generally offline the device, and do not allow an IO to be retried.

`scsi_check_paths()` is then called with an indication as to whether a path failure has occurred or not, it updates the path state, and returns a result specifying the action to take: the standard `SUCCESS` (meaning the IO has completed, not that it has completed successfully), `FAILURE`, or a new `REQUEUE` value signaling that the IO should be re-queued. `NEEDS_RETRY` is no longer returned.

In `scsi_softirq()`, when a `REQUEUE` result is returned from `scsi_decide_disposition()`, the IO is re-queued via `scsi_queue_insert()`.

So, on a path failure, the IO is re-queued, and in `scsi_request_fn()` the IO can be retried on any of the remaining paths.

3.6 User Space Interface

3.6.1 Procfs

The mid-level multipath code provides a procfs interface for viewing and setting attributes related to paths. The path to the procfs file is `/proc/scsi/scsi_path/paths`. The file supports both read and write operations, and displays attributes about the paths. The table below provides a description of each of the columns in the output.

Column	Description
1	UUID
2	Host Number
3	Host Channel
4	Target ID
5	Lun
6	State
7	Failures
8	Weight

Table 1: Procfs Columns

An edited output to show only one device of a read is shown as follows:

```
#cat /proc/scsi/scsi_path/paths
...
2000002037171f24 3 0 1 0 1 0 0
2000002037171f24 4 0 1 0 1 0 0
...
```

Writing to the file allows path attributes to be modified. Currently the only meaningful write operation is to modify path state. A path state may be modified from dead to good or good to dead. A good path state has a value of "1" and a dead path has a value of "3".

An example of failing a path is shown as follows:

```
echo '2000002037171f24 3 0 1 0 3 0 0'
```

```
> /proc/scsi/scsi_path/paths
```

In the near term the procfs interface will be replaced with a sysfs interface. At the time of this writing, the interface was not complete and is discussed in the Future Work section.

4 Future Work

4.1 Multipath Device Personality

The use of device-neutral information through standard SCSI interfaces limits the set of multipath capabilities that can be supported on a given device to a minimal set known to be safe for all devices. To utilize the extended capabilities of some storage device's, the device attributes or a device's "personality" needs to be exposed.

The interfaces provided for obtaining this personality knowledge will not be restricted to kernel space. Some data can be set from user space, but other operations will need to be kernel resident to avoid deadlock. Further direction toward user level scanning will affect these interfaces.

The single kernel config time path policy can be enhanced with device attribute information allowing support for device specific path policies.

Path weighting values related to a device's attributes would allow proper primary and secondary paths to be determined. The ability to determine preferred paths to assist in the balancing of load across a storage device's port can also be determined.

4.2 SCSI Reservations

The support of SCSI Reserve/Release affects the type of path selection policy that can be selected for a storage device restricting it to the last path used. Support of SCSI persistent reserve requires an interface to accept a reservation key and special IO operations before paths can used for normal IO. Future work is trying to meet the requirements of SCSI reservation with a general purpose path preparation capability.

4.3 Sysfs Interface

As mentioned in a previous section, the procfs interface to mid-level multipath is being migrated to a sysfs-based interface. The migration to a sysfs interface allows for the linkage to the device tree for increased path topology information and the utilization of common kernel code infrastructure, reducing code duplication.

5 Availability

The SCSI Mid-Level Multipath project page is located at:

http://www-124.ibm.com/storageio/multipath/scsi-multipath/

Legal Statement

References

[1] Douglas Gilbert, "The Linux 2.4 SCSI subsystem HOWTO" `http://www.linuxdoc.org/HOWTO/SCSI-2.4-HOWTO/index.html`

[2] “LVM multipath support”, `http://oss.software.ibm.com/linux390/useful_add-ons_lvm.shtml`

[3] “MD Multiple Devices driver”, `drivers/md/*`

[4] “SCSI mid_level - lower_level driver interface”, `Documentation/scsi/scsi_mid_low_api.txt`

[5] “Linux T3 Driver”, `http://open-projects.linuxcare.com/t3/`

[6] “SCSI Primary Commands - 3 (SPC-3)”, `ftp://ftp.t10.org/t10/drafts/sam3/sam3r06.pdf`

[7] “SCSI Architecture Model - 3 (SAM-3)”, `ftp://ftp.t10.org/t10/drafts/sam3/sam3r06.pdf`

IPv4/IPv6 Translation

Allowing IPv4 hosts to communicate with IPv6 hosts without modifying the software on the IPv4 or IPv6 hosts

J. William Atwood[*]
Kedar C. Das
Xing (Scott) Jiang
Concordia University
Department of Computer Science
Montréal, Québec H3G 1M8
bill@cs.concordia.ca, http://www.cs.concordia.ca/~faculty/bill

Abstract

As the Internet makes the transition from IP version 4 to IP version 6, it will be necessary to allow IPv4-based clients to access IPv6-based servers, and IPv6-based clients to access legacy services. Network Address Translation–Protocol Translation (NAT-PT) can provide network protocol translation, and Application Layer Gateways (ALGs) can handle the cases where peer addresses are embedded in application-layer messages. We describe an implementation on a Linux router/translation server, the necessary configuration of the IPv4 and IPv6 environments, and the operation of ALGs for the File Transfer Protocol and for the Session Initiation Protocol. We will present a demonstration of basic communication (web client to web server), and of multimedia communication (based on the open-source VOCAL project).

[*]On leave at Ericsson Research Canada, Open Systems Research Laboratory, Montréal, Québec, Canada

1 Introduction

IPv6 is the next generation protocol designed by the IETF to replace the current version of the Internet Protocol, IPv4. During the last decade, IP has conquered the world's networks. Most of today's Internet uses IPv4, which has been remarkably resilient in spite of its age, but it is beginning to have problems.

One motivation for developing IPv6 was the anticipated exhaustion of addresses for individual hosts. While the rate of depletion has been slowed through the use of Network Address Translation (NAT) [1], it does continue, and the other virtues of IPv6 (routing and network autoconfiguration, and enhanced support for IP Security (IPsec) and IP Mobility (Mobile IPv6)), will encourage its deployment much more widely in coming years.

Although a significant percentage of the clients and servers will be *dual stack* (i.e., capable of using either IPv4 or IPv6), there will be a large number of existing clients and servers (legacy systems) that will only be capable of using IPv4, and there will be a growing number of

clients and servers that will only be capable of using IPv6. For example, the Third Generation Partnership Project (3GPP) has mandated that third generation cellular networks will be "All-IP," and that the "IP" will be IP version 6 *only*.

In the same way as NAT has been used to connect hosts on private networks [2] with hosts on the public network, Network Address Translation–Protocol Translation (NAT-PT) [3] has been standardized as a way of connecting hosts in the IPv4 address space and hosts in the IPv6 address space. NAT and NAT-PT work by altering the IP headers. For NAT, only the address fields are replaced; for NAT-PT, the entire header is changed. This use of NAT-PT solves the network-layer problem for the IPv4/IPv6 transition, but it requires some auxilliary services to operate properly, and it does not solve a number of application-layer problems associated with crossing the IPv4/IPv6 boundary.

In this paper, we discuss the auxilliary services needed, report an experimental validation of the requirements, outline the solution to some of the application-layer problems, and speculate on the solution to the rest.

2 System Architecture

Figure 1 identifies the typical components that will be used to support communication between IPv4 hosts and IPv6 hosts. The IPv4 region represents the entire IPv4-based Internet of today. The IPv6 stub region contains the hosts that are to be granted the privilege of accessing legacy (IPv4-based) services. The IPv6 region represents the rest of the IPv6 address space. The v4/v6 Border Router provides the connection between the IPv6 stub hosts and the IPv4 hosts. The v6/v6 Border Router provides the connection between the IPv6 stub region hosts and the rest of the IPv6 address space. In some systems, the v4/v6 Border Router and the v6/v6 Border Router will be co-located. However, we leave them separate in the following, to make the explanations clearer.

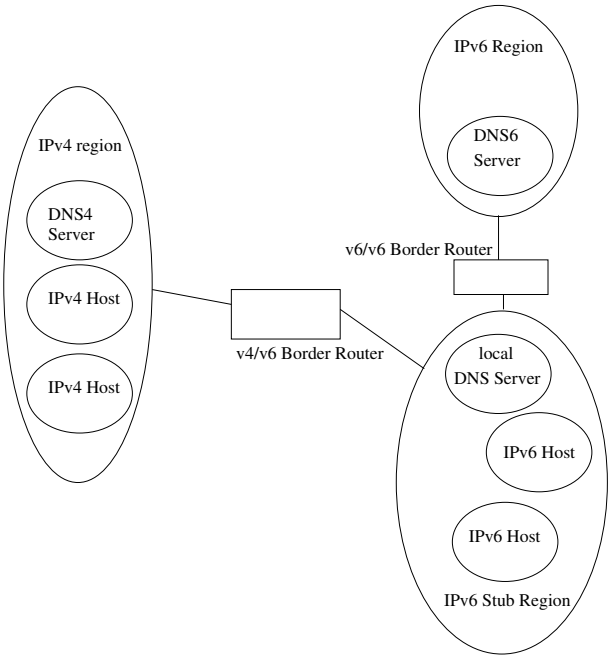


Figure 1: System Architecture

Each host has a *host name* and a *host address*. The host name is a (globally unique) character string that is intended to be human-readable. The host address is a (globally unique) 32-bit (IPv4) or 128-bit (IPv6) number. A particular host may have more than one name, and more than one address, especially if it has multiple interfaces.

Two address pools are associated with the v4/v6 Border Router. The *IPv4 address pool* is a sequence of addresses that are associated (temporarily) with the IPv6 hosts that are communicating with IPv4 hosts. Given the scarcity of IPv4 addresses, this pool will be sized to correspond to the number of IPv6 hosts (in the IPv6 stub region) that are *actively* communicating with IPv4 hosts at a particular time. The IPv6 address pool is a sequence of addresses that represent hosts in the IPv4 region. Given the large size of the IPv6 address space, this

pool is structured as a 96-bit prefix, catenated with a 32-bit IPv4 address. A motivation for this will be presented later, in Section 3.6.

2.1 Translation Requirements

As packets move between the IPv4 region and the IPv6 region, it is necessary to rebuild their headers, since the IPv4 and IPv6 packet headers have different formats. This process is stateless—the necessary mapping information is determined by tables in the v4/v6 Border Router (for packets travelling from the IPv4 region to the IPv6 stub region) or by information carried in the packet address (for packets travelling from the IPv6 stub region to the IPv4 region). For NAT-PT, the mapping between an IPv4 pool address and the corresponding IPv6 host address is one-to-one. When there are insufficient IPv4 pool addresses available, then NAPT-PT can be used, with a mapping from (IPv4 address, IPv4 port) to (IPv6 address). This allows about 65,000 IPv6 hosts to be serviced using a single IPv4 address, as long as the IPv4 application does not care about the port that is being used to access it.

Certain packets require special treatment. In general, these packets contain application data that have embedded IPv4 or IPv6 addresses. They are identified when their headers are processed (usually by noting which port they are using), and they are handled by application-specific code called an *Application Layer Gateway* (ALG). The ALGs are application-specific, because they need to be able to parse the packets being exchanged by the application end-points. The specific ALG then modifies the contents of the packet, to reflect the address translation that has just taken place in the headers.

2.2 Centralized Architecture

One approach to handling the entire requirement is to co-locate the NAT-PT software and the set of ALGs needed to support the desired applications. In this case the interaction between the ALG and the translation tables in the NAT-PT software is simplified. However, the v4/v6 Border Router must provide processing power for all functions, which could overload it.

2.3 Distributed Architecture

An alternate approach is to separate the ALGs from the NAT-PT software. This lowers the processing requirements for the v4/v6 Border Router, but it introduces a requirement to define a protocol for interaction between the ALG and the NAT-PT. This approach is favoured when the application makes use of some form of "proxy" server, because the proxy functions and the ALG functions can often be advantageously combined in a single host, and separated from the v4/v6 Border Router. Commercial systems will need to adopt this approach, to handle the large number of IPv6 hosts that will be in a typical IPv6 stub region. However, our project was concerned with exploring issues relating to establishing the right environment, and we did not require high performance at this time.

3 NAT-PT Tool

The experimental system was based on a user-space NAT-PT implementation developed at ETRI [4]. The original implementation was based on a Linux 2.4.0 kernel, and required modification to make it work on the more recent (2.4.20) Linux kernel.

3.1 General Flow

To establish communication between IPv4 and IPv6 using NAT-PT we need interactions of at least 3 modules. These are-NAT, PT, ALG.

3.2 Network Address Translation (NAT)

The NAT module implements the transport/network layer translation mechanism. It uses a pool of IPv4 addresses for assigning to IPv6 nodes dynamically, and this assignment is done when sessions are initiated across the v4/v6 boundary.

3.3 Protocol Translation (PT)

As all the fields of IPv6 headers are not the same as that of the IPv4 header, the PT module translates IP/ICMP headers to make end-to-end communication possible. Due to the address translation function and because of possible port multiplexing, PT also makes appropriate adjustments to the upper layer protocol (TCP/UDP) headers, e.g., the checksum.

The IPv4-to-IPv6 translator replaces the IPv4 header of IPv4 packet with an IPv6 header to send it to the IPv6 host. Except for ICMP packets, the transport layer header and data portion of the packet are left unchanged. In IPv6, path MTU discovery is mandatory but it is optional in IPv4. This implies that IPv6 routers will never fragment a packet—only the sender can do fragmentation. Path MTU discovery is implemented by sending an ICMP error message to the packet-sender stating that the packet is too big. When an IPv6 router sends an ICMP error message, it will pass through a translator, which will translate the ICMP error to a form that the IPv4 sender can understand. In this case an IPv6 fragment header is only included if the IPv4 packet is already fragmented. The presence of df flag in the IPv4 header is the indication of Path MTU discovery.

However, if the IPv4 sender does not perform path MTU discovery, the translator has to ensure that the packet does not exceed the path MTU on the IPv6 side. The translator fragments the IPv4 packet so that it fits in a 1280 byte IPv6 packet, since IPv6 guarantees that 1280 byte packets never need to be fragmented. Also, when the IPv4 sender does not perform path MTU discovery the translator must always include an IPv6 fragment header to indicate that the sender allows fragmentation.

3.4 Application Layer Gateway (ALG)

Several applications send IP addresses and host names within the payload of the IP packet. Because NAT-PT does not snoop the payload, it requires some Application Level Gateways (ALG) to extract that address to replace it either by IPv4 or IPv6. That is, ALG could work in conjunction with NAT-PT to provide support for many such applications. Two examples are the FTP-ALG and the SIP-ALG. These are discussed in Section 3.8 and Section 4.

While the FTP-ALG and the SIP-ALG are optional (provision of a specific ALG depends on the desire to support that particular application), the DNS-ALG is essential, because it is the trapping of the DNS queries that allows NAT-PT to discover the need for mapping between IPv4 addresses and IPv6 addresses.

In the DNS, “A” records represent IPv4 addresses and “AAAA” or “A6” records represent IPv6 addresses.

3.5 Communication from V4 to V6

Any packet originating on the IPv4 side destined to the IPv6 stub network should cross the v4/v6 Border Router, which is the NAT-PT device. When any packet is received on the IPv4 side, NAT-PT will check the destination address. If it is an IPv4 pool address, and if a

mapping exists to an IPv6 host, then the IPv4 packet header is converted to an IPv6 packet header, otherwise the packet is dropped. The IPv6 source address will be the PREFIX catenated with the IPv4 source address; the IPv6 destination address will be the mapped IPv6 host address.

If the packet is a DNS query, then the DNS-ALG will change the query type from "A" to "AAAA", and alter the format of strings ending in "IN-ARPA.ARPA" to the IPv6 format ending in "IP6.INT". When the response is returned, then the DNS-ALG will change the "AAAA" record to an "A" record (if the resolution was successful), and change the resolved IPv6 address to the corresponding IPv4 (pool) address.

When any other response packet is returned to the NAT-PT, the IPv4 destination address will be found by removing the PREFIX from the IPv6 destination address. The source address to be used in the generated IPv4 packet is the IPv4 pool address corresponding to the IPv6 host.

3.6 Communication from V6 to V4

When a packet is received from the IPv6 side, its destination address will consist of the PREFIX catenated with the actual IPv4 destination, so the IPv4 packet can be created using this address as the destination, and the IPv4 pool address corresponding to the IPv6 host as the source address.

The key to the creation of the mapping is the DNS queries. If a DNS query is received from an IPv6 host, it is not known *a priori* whether the target host is a v4 host or a v6 host. The DNS-ALG will therefore split the query into an "A" query sent to the v4 DNS, and an "AAAA" query sent to the v6 DNS. If a v6 response is received, then any v4 response is discarded (the v6 path is preferred). Otherwise, a received v4 response triggers creation of a mapping entry, and then an "AAAA" response is generated using PREFIX catenated with the v4 address.

Returning traffic on the IPv4 side will arrive at a pool address. This is used to determine the correct IPv6 destination address, so that the packet can be forwarded. The IPv6 source address will be the PREFIX catenated with the original IPv4 source address.

3.7 TCP/UDP/ICMP Checksum Update

NAT-PT retains the mapping between a specific IPv6 host address and an IPv4 address from the pool of IPv4 addresses available. The mapping between IPv6 address and an IPv4 from the pool of IPv4 addresses is used in the translation of packets passing through NAT-PT. With the translation of IP header, TCP/UDP/ICMP checksum is also updated in NAT-PT according to specific algorithm.

3.8 FTP Application Layer Gateway (FTP-ALG)

Existing FTP works with IPv4 addresses. Two important FTP commands PORT and PASV will no longer exist in IPv6. The PORT command is used to specify a port different from the default one, and it contains the IPv6 address information. So it can't be used without translation. The PASV command is used to put the server into passive mode, which means the server listens on a specific data port rather than initiating the transfer. This command includes the host name and address of the FTP server and therefore does not work over IPv6 without modification. The PORT command is replaced by the EPRT command, which allows the specification of an extended address for the connection. The extended address specifies the network protocol (IPv6 or IPv4), as well as the IP address and the port to be used. The EPSV

command replaces the PASV command. The EPSV command has an optional argument that allows it to specify the network protocol, if necessary. The server reply contains only the port number on which it listens, but the format of the answer is similar to the one used for the EPRT command and has a placeholder for the network protocol and address information might be used in the future to provide flexibility in using FTP through firewalls and NATs. An FTP control session carries the IP address and TCP port information for the data session in its payload; an FTP-ALG provides application level transparency.

If a V4 host originates the FTP session and uses PORT or PASV command, the FTP-ALG will translate these commands into EPRT and EPSV commands respectively prior to forwarding to the V6 node. Likewise, EPSV response from V6 nodes will be translated into PASV response prior to forwarding to V4 nodes.

If a V4 host originated the FTP session and was using EPRT and EPSV commands, the FTP-ALG will simply translate the parameters to these commands, without altering the commands themselves.

3.9 Payload Modifications for V6 originated FTP sessions

If a V6 host originates the FTP session the FTP-ALG has two approaches:

In the first approach, the FTP-ALG will leave the command strings “EPRT” and “EPSV” unaltered and simply translate the <net-prt>, <net-addr> and <tcp-port> arguments from V6 to its NAT-PT (or NAPT-PT) assigned V4 information. <tcp-port> is translated only in the case of NAPT-PT. The same goes for the EPSV response from V4 node. With this approach, the V4 hosts must have their FTP application upgraded to support EPRT and EPSV extensions to allow access from V6 hosts.

In the second approach, the FTP-ALG will translate the command strings “EPRT” and “EPSV” and their parameters from the V6 node into their equivalent NAT-PT assigned V4 node info and attach to “PORT” and “PASV” commands prior to forwarding to the V4 node. However, the FTP-ALG would be unable to translate the command “EPSVALL” issued by V6 nodes. In such a case, the V4 host, which receives the command, may return an error code indicating unsupported function, and this error response may cause FTP applications to simply fail. The benefit of this approach is that is does not impose any FTP upgrade requirements on V4 hosts.

3.10 Header updates for FTP control packets

All the payload translations considered in the previous sections are based on ASCII encoded data. As a result, these translations may result in a change in the size of packet. If the new size is the same as the previous, only the TCP checksum needs adjustment as a result of the payload translation. If the new size is different from the previous, TCP sequence numbers should also be changed to reflect the change in the length of the FTP control session payload. The IP packet length field in the V4 header or the IP payload length field in the V6 header should also be changed to reflect the new payload size. A table is used by the FTP-ALG to correct the TCP sequence and acknowledgement numbers in the TCP header for control packets in both directions.

The table entries should have the source address, source data port, destination address and destination data port for V4 and V6 portions of the session, sequence number delta for outbound control packets and sequence number delta for inbound control packets.

4 SIP-ALG Operation

Many communication applications on the Internet require a session protocol to negotiate and maintain the data exchange between endpoints in a session.

Moreover, as the evolution of the mobile computing and wireless networks, a session is required to handle user mobility, different media type, and media addition and removal in an existing session. Upon these requirements, the Internet Engineering Task Force (IETF) issued the Session Initiation Protocol (SIP) to enable the Internet endpoints (called user agents in SIP) to discover one another and to agree on the parameters of a session.

4.1 SIP overview

"SIP is an application-layer control protocol that can establish, modify, and terminate multimedia sessions (conferences) such as Internet telephony calls" [5]. SIP is specified as an agile and general-purpose tool that works independently of underlying transport protocols and without dependency on the various media types.

SIP establishes sessions by its invitations in which session descriptions are used to negotiate a set of compatible media types to be shared among participants. In addition, SIP can invite participants to join in an already existing session. SIP transparently supports name mapping and redirect services. SIP proxy severs could be used to facilitate routing SIP requests to the user's current location. SIP also provides a registration function that stores users' current locations that could be used by proxy servers to redirect requests.

The characteristics of SIP are simplicity and flexibility. SIP is not a complete communication system. SIP is rather a component that can cooperate with other IETF protocols to provide complete services to the users. Typically, the Real-time Transport Protocol (RTP) [6] could be combined with SIP to support real-time data transfer and provide QoS feedback; the Session Description Protocol (SDP) [7] could be used for describing multimedia sessions; the Real-Time Streaming Protocol (RSTP) [8] could be used to control delivery of streaming media; and the Media Gateway Control Protocol (MEGACO) [9] could be used for controlling gateways to the Public Switched Telephone Network (PSTN).

It should be noted that SIP does not depend on any of the protocols above that provide services. Rather, SIP provides basic functionalities and operations that can be used to implement different services. Furthermore, "SIP provides a suite of security services, which include denial-of-service prevention, authentication (both user to user and proxy to user), integrity protection, and encryption and privacy services" [5].

4.2 SIP Messages

SIP defines two distinct types of messages: requests and responses. "A SIP message is either a request from a client to a server, or a response from a server to a client. ... Both types of messages consist of a start-line, one or more header fields, an empty line indicating the end of the header fields, and an optional message-body" [5].

4.3 SIP Behavior

SIP requests can be sent directly from a user agent client to a user agent server, or they can traverse one or more proxy servers along the way. User agents send requests either directly to the address indicated in the SIP URI or to a designated proxy (outbound proxy), independent of the destination address. The current

destination address is carried in the Request-URI. Each proxy can forward the request based on local policy and information contained in the SIP request. The proxy may rewrite the request URI.

4.4 SIP-ALG Behavior

There exists an increasing need of IP address translation because the networks based on IPv6 addresses have being extended and because the supply of IPv4 addresses is inadequate.

SIP is an application layer control protocol for establishing media sessions. It encounters problems with NAT-like devices, because the payloads of SIP packets carry the addresses for the sessions to be established. However, the NAT function in NAT-PT is application unaware and does not snoop the payloads. Along with NAT-PT and DNS-ALG, a SIP-ALG is needed on the boundary between IPv4 and IPv6.

This section describes a simple implementation of a SIP ALG to enable simple SIP sessions to pass through a NAT-PT box based on the Vocal system (an open source SIP implementation created by Vovida.org [10]). Rather than attempt to make a full specification for SIP-ALG, we have implemented a subset of the functionalities that is sufficient for typical use.

An IP packet carrying a SIP message is identified by NAT-PT box by the characteristic of the SIP protocol, using the port 5060 as the destination.

Whenever a Vocal user agent (UA) initiates a SIP session by the host name of a callee, it looks up the IP address from DNS services. DNS servers transparently provide the address as normal except that the DNS-ALG will setup an address mapping once the DNS query is traversing a boundary of IPv4 and IPv6, which has been described in the previous sections. If a mapping occurs, the SIP message is sent to the NAT box. The SIP-ALG in the NAT box will build a table storing two pairs of the source address and the corresponding destination address for both IPv4 and IPv6 domains. According to the table, the various fields in the SIP message will be modified. If the Content-Type is SDP, the SDP message and the Content-Length will also be adjusted. After that, the modified message is forwarded to another IP version of the network. Similarly, the response messages will be modified back to the original messages correspondingly when they return to the NAT box. Thus, it seems as if the IPv4 UA and the IPv6 UA are communicating with the NAT-PT box respectively.

5 DNS Considerations

The IPv4 region has a DNS server, which returns Address (A) records when queried about a host name that exists in the IPv4 region. The IPv6 stub region has a local DNS server, which returns IPv6 address (AAAA) records when queried about a hostname that exists in the IPv6 stub region. In addition, there is a "global" IPv6 DNS server.

If any IPv6 hosts in the IPv6 stub region are to provide services to IPv4 clients (initiated by the IPv4 clients), the NAT-PT must permanently associate an IPv4 (pool) address to the IPv6 address of the serving host. In addition, one of the following must be true:

1. The IPv4 DNS server must map a host name (probably different from its IPv6 host name) to the associated IPv4 (pool) address, *or*
2. The IPv4 DNS server must refer the DNS request for the associated host name to a DNS server located at a specific IPv4 pool address. Queries sent to this address are

processed by the DNS-ALG at the Border Router, and sent to the local IPv6 DNS (in the stub region). The returned reply is processed by the DNS-ALG, and sent to the original requesting host as a DNS "A" record. This record will contain the IPv4 pool address that was assigned to the IPv6 server.

We had to install and configure a *local* IPv4 DNS server to filter the requests for resolution of host names associated with the IPv6 stub domain. This was done on the host that would serve as the client, to minimize the impact on the rest of the system. Queries about names ending in ".ip6.lmc.ericsson.se" were sent to the statically assigned IPv4 pool address that was associated with the IPv6 DNS server. All other queries were forwarded to the regular IPv4 DNS server. (Note: this "extra" DNS server would be unnecessary in a production environment. The problem would be addressed by putting the necessary directives in the IPv4 DNS server itself.)

6 Conclusion and Future Work

We have demonstrated that it is possible to protect the investment in past hardware and systems, by installing a NAT-PT on the Border Router that provides access to IPv6-based equipment. To do so requires that special care be taken in configuring the Domain Name System servers (for both IPv4 and IPv6), the NAT-PT itself, and the IPv6 hosts (so that their DNS requests are sent to the NAT-PT).

The NAT-PT solution requires one IPv4 pool address for each IPv6 host that is concurrently accessing the IPv4 address space. This could clearly result in exhaustion of the IPv4 address pool. A potential solution is to use NAPT-PT [3], where host/port number pairs on the IPv6 side are mapped to a single IPv4 pool address and multiple port numbers on the IPv4 side. NAPT-PT bears the same relationship to NAPT that NAT-PT bears to NAT; the implementation details are more complex, but the use of NAPT-PT will be necessary when large IPv6 stub regions are able to use only a small pool of IPv4 addresses to provide the desired services.

7 Acknowledgments

The authors acknowledge the support of Ericsson Research Canada through the use of its laboratory. The first author acknowledges the support of the Natural Sciences and Engineering Research council of Canada, through its Discovery Grants program.

8 Availability

A link to the developed implementation is on the web site

`http://www.linux.ericsson.ca/ipv6`

References

[1] K. Egevang and P. Francis. The IP network address translator (NAT). Request for Comments 1631, Internet Engineering Task Force, May 1994.

[2] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. De, and E. Lear. Address allocation for private internets. Request for Comments 1918, Internet Engineering Task Force, February 1996.

[3] G. Tsirtsis and P. Srisuresh. Network address translation - protocol translation (NAT-PT). RFC 2766, Internet Engineering Task Force, February 2000.

[4] ETRI-PEC. User space NAT-PT implementation. http://www.ipv6.or.kr/english/natpt-overview.htm.

[5] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. R. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261, Internet Engineering Task Force, June 2002.

[6] Henning Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: a transport protocol for real-time applications. Request for Comments 1889, Internet Engineering Task Force, January 1996.

[7] M. Handley and V. Jacobson. SDP: Session Description Protocol. Request for Comments 2327, Internet Engineering Task Force, April 1998.

[8] H. Schulzrinne, A. Rao, and R. Lanphier. Real Time Streaming Protocol (RTSP). Request for Comments 2326, Internet Engineering Task Force, April 1998.

[9] F. Cuervo, N. Greene, A. Rayhan, C. Huitema, B. Rosen, and J. Segers. Megaco protocol version 1.0. Request for Comments 3015, Internet Engineering Task Force, November 2000.

[10] VOVIDA. VOCAL home page. http://www.vovida.org.

Building Enterprise Grade VPNs

A Practical Approach

Ken S. Bantoft
freeswan.ca / MDS Proteomics
ken@freeswan.ca, http://www.freeswan.ca

Abstract

As Linux scales its way into the enterprise, things like high availabilty and redundancy become more and more important.

As is the case with many Open Source applications, you can integrate several applications together to build in as much redundancy and failover as you need for your network.

By combining several Open Source applications, including Linux [1], FreeS/WAN [2], GNU/Zebra [3] and Heartbeat [4] we are able to build a very reliable, robust VPN solution.

1 Building an Enterprise VPN

FreeS/WAN has been used for several years by many Linux system administrators to build Virtual Private Networks (VPNs) between sites, end-users, and business partners. It is very configurable, inter-operates well with other IPSec implementations and is generally quite stable. It has few limits, however two of the more common limits sysadmins run into are:

1. IPSec doesn't tunnel all IP traffic—it does not handle Multicast or Broadcast traffic. This means if we wanted to do Multicasting over our VPN, or run OSPF from Zebra on ipsec0 for dynamic routing, we can't. While it's possible to run OSPF in NMBA mode, we ran into problems with this as well.

2. You need one tunnel per network combination pair—thus if you have 4 IP subnets behind Secure Gateway #1, and 2 behind Secure Gateway #2, you will need to configure 8 separate tunnels (unless you can aggregate your IP network space to /23s, /16s or other CIDR compatible blocks).

Point 1 isn't too much of a limit, since many networks don't need multicast and broadcast traffic to be routed between sites. In large networks, point 2 quickly becomes an adminitrative nightmare to deal with.

This can quickly get out of hand for large networks, especially when more than two sites are involved. The ideal solution is to setup a single tunnel between each site, and then route all traffic from site 1 destined for site 2 over the VPN, and hope the other side accept it. The problem here is that IPSec policies (which FreeS/WAN enforces) prevents this—if there is no explcit tunnel defined for the Source + Destination pair, the packet is dropped.

The solution here is to use GRE—Generic Routing Encapsultion. This has been part of the Linux kernel for quite some time, and allows us to solve both problems identified above. By setting up a GRE tunnel over one of our IPSec tunnels, we can then route any-

thing we want over the GRE tunnel, including Multicast traffic.

Once we are using GRE, we need some way to dynamically inform the other side of the tunnel which networks we know about, and how to get to them. GNU/Zebra can provide this functionality, using either OSPF or BGPv4.

2 Challenges & Solutions

Integration of all of the applications used presented several challenges, since none of them were designed to work together. Some had to be extended, and others had to be scripted around in order for them to notify each other of various different sorts of failures.

Luckily, with source in hand, this was much easier than expected.

2.1 Zebra

Zebra was the easiest to integrate, as no direct code changes were required. There were initially some problems with using OSPF on aliased interfaces (eg: eth0:0) but those were solved by an upgrade to the latest version (0.91a or 0.93 are known to work).

2.2 Heartbeat

Heartbeat needed some modifications so it was aware of the status of the physical interfaces, and then just needed a basic configuration and a lot of scripting.

One of the most significant differences between commercial grade routers and Linux is that if an interface is physically unplugged, or the switch/hub on the other side goes down, a commercial router drops the interface, and all routes that travel over it are removed. Linux desperatly needs this capability, but until recently many network cards were unable to report the link status.

Donald Becker[5] wrote a handy toolset for this—mii-tool/mii-diag. During my tenure at IBM Canada, one of the developers I worked with took this and turned it into a patch against Heartbeat. If Heartbeat detects a physical problem with the network card/cable/switch, Heartbeat initates a failover event. This code supported the Intel 10/100 (eepro.o|e100.o) as well as the Intel Gigabit Ethernet adapters (e1000.o).

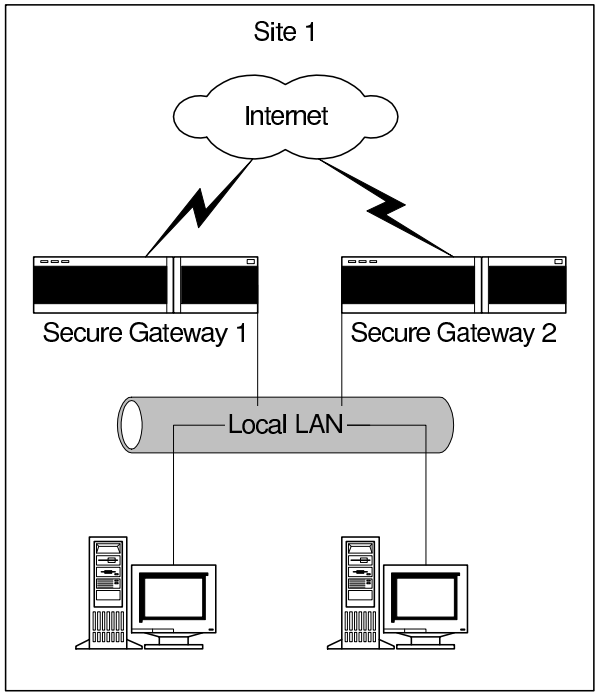
The bulk of the time went into the rewriting of many scripts to properly bring interfaces up and down, to restart Zebra's bgpd correctly, and to cleanly restart FreeS/WAN.

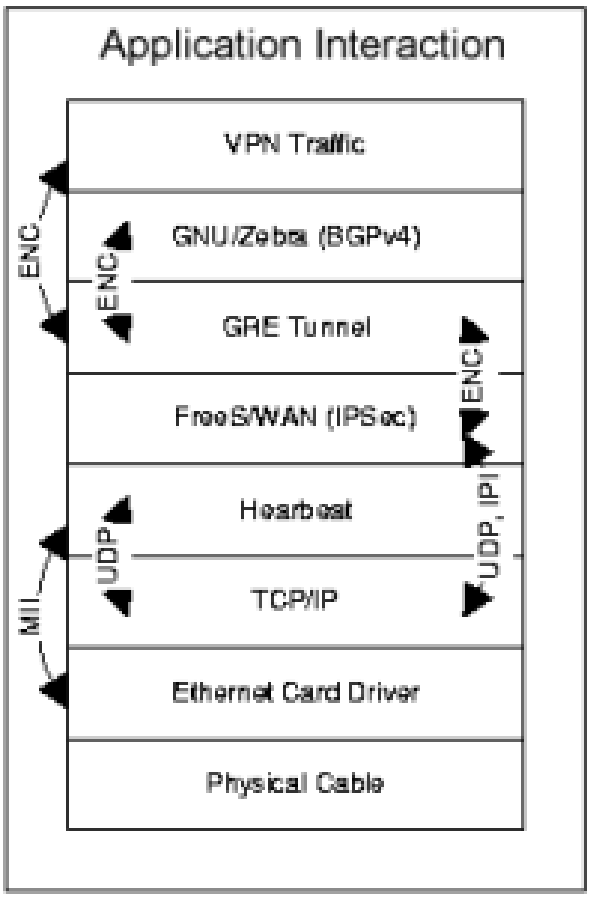
2.3 FreeS/WAN

FreeS/WAN required no code changes, only configuration and some scripting to keep the configuration syncronised between each set of Secure Gateways. We used SSH for this.

3 Gluing it all Together

The complicated part is making all of these applications and protocols work together seamlessly. Our basic network layout is below:





We have two Secure Gateways, each with a connection to the internet. Ideally, each would have its own link to the Internet, preferably redundant, but you could share the link if needed. Each gateway also has a connection to the local lan, which Heartbeat sends keep-alives over. If possible, use a Null-modem cable between each pair of gateways for out of band keep alives. If this isn't possible, Heartbeat also supports UDP keep-alives over any network interface.

Ensure each gateway has all of the applications installed, and the GRE and FreeS/WAN configurations should be syncronised. Heartbeat and GNU/Zebra configurations differ slightly, so they should not be shared.

A diagram showing the key interactions between the applications:

Heartbeat is the control center in this setup, as it monitors each node in the group for failure, and checks its own Ethernet devices via MII calls. Heartbeat also starts and stops various scripts (from /etc/rc.d/init.d) which bring up FreeS/WAN, the GRE tunnels, and GNU/Zebra.

3.1 Dealing with Startup Scripts

FreeS/WAN and GNU/Zebra both provide scripts suitable for use in /etc/rc.d/init.d, so we used those. We also needed to add a GRE tunnel, so we wrote our own startup scripts for that. A quick example:

```
#!/bin/sh
# chkconfig: 2345 50 64
# description: Set up a GRE tunnel from
```

```
# here to somewhere else

case "$1" in
  start)
    ip tunnel add MYTunnel mode gre \
    remote 216.1.1.1 local 116.1.1.1 ttl 255
    ip link set MYTunnel up
    ip addr add 172.16.0.1 dev MYTunnel
    ip route add 172.16.0.2/32 dev MYTunnel
    ;;
  stop)
    ip route del 172.16.0.2/32 dev MYTunnel
    ip addr del 172.16.0.1 dev MYTunnel
    ip link set MYTunnel down
    ;;
  restart)
    $0 stop
    $0 start
    ;;
  *)
    echo "Usage: $0 {start|stop|restart}" >&2
    exit 2
esac

exit 0

}
```

Our script supports the traditional arguments start, stop, and restart, and will bring the GRE tunnel up and down when called from Heartbeat.

3.2 Heartbeat Configuration

Full details on configuring Heartbeat are available from the package itself [7], so I will cover only the /etc/ha.d/haresources config file here. From Heartbeat, we need to control the IP address takeover, and the services (/etc/rc.d/init.d scripts) we start and stop when a node fails. This can be done with a simple, single line entry in /etc/ha.d/haresources:

```
cluster1 116.1.1.1/28 192.168.0.1/24 \
ipsec gre zebra bgpd
}
```

The above line tells Heartbeat to do IP address takeover on 159.18.124.254 (our External IP address) 192.168.0.1 (our Internal IP address). It also lists the scripts (in order) to run when a takeover happens. It passes each of these scripts a parameter—either "start" or "stop" determined by what is occurring—taking over the IP, or releasing it. (I.e., `/etc/rc.d/init.d/ipsec start`.)

3.3 FreeS/WAN Configuration

FreeS/WAN configuration was straightforward. PSK (pre shared secrets) use is not recommended, as recent bugtraq postings have shown some potential security flaws. We recommend RSASig's, however X.509 Digital certificates can also be used. The following example uses PSK authentication for one remote site:

```
config setup
    interfaces="ipsec0=eth0:0"
    klipsdebug=none
    plutodebug=none
    plutoload=%search
    plutostart=%search
    uniqueids=yes

conn %default
    keyingtries=0

conn site1tosite2
authby=rsasig
    left=116.1.1.1
    leftnexthop=116.1.1.30
    leftrsasigkey=0xA0S8PIPI...
    leftid=@site1.company.com
    right=216.1.1.1
    rightnexthop=216.1.1.30
    rightrsasigkey=0xA0QKJ986...
    rightid=@site2.company.com
    auto=start
}
```

The critical line of the config is `interfaces="ipsec0=eth0:0"`, as by default FreeS/WAN won't bind to an aliased interface. Since Heartbeat brings up the service IP addresses on aliases, we need to bind our ipsec interface to the alias.

3.4 GNU/Zebra Configuration

From GNU/Zebra, we use the BGPv4 daemon to handle our dynamic routing. This gives us much more control over which routes we share than OSPF would, as well as makes configuration simple.

Sample bgpd.conf file:

```
!
hostname torcofw1
password a_secure_password
enable password a_more_secure_password
log file bgpd.log
log stdout

router bgp 65432
 bgp router-id 172.16.0.1
 network 172.16.0.0/30
 redistribute kernel
 redistribute connected
 redistribute static
 neighbor 172.16.0.2 remote-as 65432
 neighbor 172.16.0.2 next-hop-self
!
}
```

We use a reserved AS number (65432) in case we ever need to do BGP with peers from another company, or the Internet. All of the network and neighbour statements refer to our GRE tunnel IP addressing, as we wish to communicate with our BGP peer over the GRE tunnel—not the IPSec tunnel. neighbour 172.16.0.2 next-hop-self is critical—we need our BGP peer to send any traffic destined for our local networks through us directly, since we have an established tunnel.

4 Conclusions

It took a few weeks to get this setup stable, during which we changed from OSPF to BGPv4, which cleared up several problems we encountered with neighbours failing to exchange routes consistantly. The current design has been running in production at 4 sites for over 2 years now, and we have had several successful failovers (several faulty network cards, a bad switch port, and the more common system administrator error).

Connections that do not pass though netfilter connection tracking (i.e., NAT/MASQ) are usually unaffected—with a keepalive time of 2 seconds, and a deadtime of 10 seconds, dead peer detection is fairly quick. This can be optimized down to about 5 seconds if needed. Changing over the IP addresses, starting FreeS/WAN, GRE tunnels and GNU/Zebra takes less than 10 seconds on modern hardware, so our total time between failover is less than 20 seconds.

5 Future Improvements

There are more improvements to be made that could bring detection and failover down into the 1–3 second range.

Heartbeat seems currently limited to 1-second keepalives—this could be brought down to 1/4 second over the serial interface, meaning a deadtime of 1 second would be reasonable (3 missed polls).

FreeS/WAN has a routing limitation whereby you can't have two tunnels for the same source + destination pair going to two different remote gateways. Hopefully, this limitation will not be present in either kernel 2.6's IPSec implementation, or future versions of FreeS/WAN that implement the MAST [8] device.

Connections that do utilize the netfilter connection tracking are currently cut off, since the secondary firewall is not aware of the current state of the conntrack table on the primary firewall. There was some discussion at the OLS 2002 Netfilter BOF, and on the Netfilter Failover list [9] on how to handle syncronization of the conntrack table, however no code

has emerged.

6 Acknowledgments

I would like to acknowledge the FreeS/WAN team, and extra thanks to Michael Richardson and JuanJo Ciarlante who helped me setup a UML environment setup so I could demonstrate much of this. I'd also to acknowledge IBM Canada, who employed me during the initial development of this solution, and my current employer, MDS Proteomics who allows me to continue to refine it. Also, thanks to Astaro Corporation for funding some of my development of Super FreeS/WAN, and covering my hosting costs for freeswan.ca.

7 Availability

Software:

```
http://www.freeswan.ca
http://www.zebra.org
http://www.linux-ha.org
```

Documentation:

```
http://www.freeswan.ca/docs/HA
```

References

[1] Linux, `http://www.linux.org`

[2] The FreeS/WAN Project,
`http://www.freeswan.org`
`http://www.freeswan.ca`

[3] The GNU Zebra Project,
`http://www.zebra.org`

[4] The Linux-HA Project,
`http://www.linux-ha.org`

[5] Donald Becker, Scyld Computing Corporation, `http://www.scyld.com/diag#mii-diag`

[6] Ken S. Bantoft, *Building HA VPNS with FreeS/WAN*, `http://www.freeswan.ca/docs/HA/`

[7] Getting Started with Heartbeat, `http://www.linux-ha.org/download/GettingStarted.html`

[8] John S. Denker, *Next-Generation IPsec Packet Handling*, `http://www.quintillion.com/moat/ipsec+routing/mast.html`

[9] Netfilter Failover Archives, `http://lists.netfilter.org/pipermail/netfilter-failover/`

Linux memory management on larger machines

Martin J. Bligh
mbligh@aracnet.com

David Hansen
haveblue@us.ibm.com

Abstract

A large amount of work has gone into the memory management subsystem during the 2.5 series of Linux® kernels, and it is more stable under a wide variety of workloads than the 2.4 VM (virtual memory subsystem). Many scalability problems have been solved, making memory managment perform much better on larger machines (meaning either with more than 1GB of RAM, or more than one processor, or both). Some of these changes also benefit smaller machines.

During the 2.4 series of kernels, the main Linux distributions diverged massively from the mainline kernel, particularly in the area of VM. This causes ongoing maintainance problems, and wasted duplicated effort in problem solving and feature implementation. Many of the enhancements made by the distributions have been brought back into the mainline kernel during the 2.5 series, under the leadership of Andrew Morton, providing a solid base for future development, and a greater potential for co-operative work.

This paper discusses the changes made to the Linux VM system during 2.5 that will significantly impact larger machines. It also covers changes that are proposed for the future, most of which are currently available as separate patches. Larger machines also have to cope with a larger number of simultaneous tasks—I have focused on up to 5000.

For the sake of simplicity, clarity, and brevity, we assume an IA32 machine with PAE mode (3 level pagetables) and normal memory layout settings throughout the paper. Unless otherwise specified, measurements were taken on a 16-CPU NUMA-Q® system (PIII/700MHz/2MB L2 cache) with 16GB of RAM.

1 Introduction

Market economics dictate the prevalance of large 32 bit systems, despite the software complexity involved. Though cheap 64 bit chips are beginning to appear, they are still not available as large systems. However, the techniques and discoveries described in this paper are by no means only applicable to such machines.

2 The global kernel virtual area

The fundamental problem with 32 bit machines is the lack of virtual address space for both user processes and the kernel—32 bits limits us to 4GB total. Each user processes' address space is local to that process, but the kernel address space is global. In order to ensure efficient operation, the user address space is shared with the global kernel address space (see Figure 1).

The default address space split for Linux 2.4 and 2.5 is 3GB user: 1GB kernel. It is possible to change this split, but it is often not desirable—some applications (such as

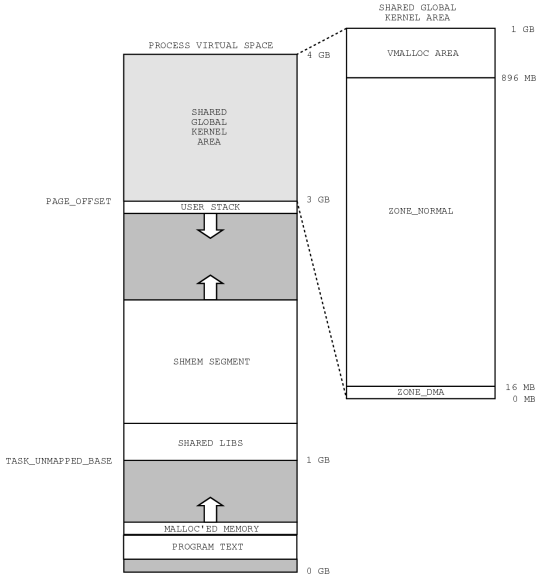


Figure 1: The process address space

databases) want as much address space for the process as possible for the application, whilst the kernel also wants as much space as possible for its data structures.

The first 896MB of physical memory is mapped 1:1 into the shared global kernel address space. This memory range is known as low memory (ZONE_NORMAL), and memory above the 896MB boundary is known as high memory (ZONE_HIGHMEM). The more physical memory we add to the machine, the bigger the kernel control structures need to be, but the control area is fixed size by the virtual space limitation.

Thus the more RAM we add to the machine, the more pressure there is on the global kernel area. The standard Linux 2.4 kernel copes very badly with large amounts of memory, perhaps limited to 4GB at best. The Linux 2.4 enterprise distributions will work with 16–32GB of memory, depending on the distribution. Linux 2.5 will cope with approximately 32GB of memory.

Unfortunately, most of the data that is put into the kernel address space is not swapable, and the Linux kernel often does not shrink the data gracefully under memory pressure. Thus, the failure condition is often difficult to diagnose; kswapd goes into a flat spin, all kernel memory allocations stop, and the system appears to have hung. Monitoring the "lowfree" field of /proc/meminfo in the runup to the system hang will often help to detect this condition.

The main space consumers for the kernel space are:

- mem_map (physical page control structures)
- Slab caches, particularly:
  - buffer_head
  - dentry_cache
  - inode_cache
- Pagetables

mem_map is an array of page control structures, one for each physical page of RAM on the system. On a 16GB machine, that takes 19% of the kernel's address space. For 64GB, it takes 78% of all the space we have, leaving insufficient space for the normal kernel text and data. Whilst the machine may boot, it will not be usable.

William Irwin and Hugh Dickins are implementing a technology called "page clustering," that makes one page control structure govern a group of pages, thus dramatically reducing the space taken (e.g. 8 page groups reduces us from 78% of space to 9%).

3 kmap

The kernel has permanent direct access to low memory, but needs to perform special opera-

tions to map high memory (however, note that user space can directly map high memory). High memory is usually mapped one 4K page at a time, via two main mechanisms: persistent kmap, and atomic kmap.

Persistent kmap uses a pool of 512 entries. All entries start out as clean; each entry is used in turn, and has a usage count associated with it. As entries are freed (usage count falls to 0) they are marked as dirty. When we reach the end of the pool, all dirty entries with 0 usage are marked as clean, a system-wide tlbflush is invoked, and now the buffers may be reused. All of these operations are global, and done under a global lock (kmap_lock).

Atomic kmap has a small number of entries per CPU—one for each of a few specific operations (that may need to be done in conjunction, so one entry is not sufficient). To reuse an atomic kmap slot, a single TLB entry needs to be flushed, and only on 1 CPU. This allows lockless operation, and CPU local data management (i.e., no cacheline bouncing). However, due to the CPU-local nature of the mapping, it is not possible to sleep, or reschedule onto another CPU whilst holding the mapping.

The problem comes in that persistent kmap turns out to be heavily used and rather slow. Not only is the data & locking global, but the global TLB flushes are very expensive (particular on machines without tlb_flush_range, such as IA32). Persistent kmap scales as $O(N^2)$ where N is the number of CPUs in the system (N times the frequency that the pool is exhausted * N times the impact from the tlb flushes). As CPU:memory speed ratios continue to grow, the caches become ever more important, and such algorithms are not suitable for heavy use.

The heaviest users were copy_to/from_user and related functions (copying data between kernel and userspace). It is possible that these operations would take a pagefault on the userspace page, and thus sleep (and thus cannot directly use atomic kmap). Other heavy users included one implementation of putting user pagetables into highmem—workloads with heavy pagetable manipulation (e.g. kernel compiles) were observed to spend more than half of their time just mapping and unmapping pte pages.

After much discussion on the subject during 2002, the following solution was agreed upon, and Andrew Morton implemented it. In essence, we now use atomic kmap for operations such as copy_to/from_user, but touch the page first to ensure it is faulted in, making it extremely unlikely that we will take a pagefault. In the unlikely event that a pagefault does occur, we handle the fault, then retry the copy operation using persistent kmap (in practice, this was never found to occur). Truly persistent global operations (typically for the lifetime of the OS instance) where performance it is not a concern can still use persistent kmap.

4 Pagetables

The pagetables map the process' virtual addresses to the physical addresses of the machine. For an IA32 machine with PAE, each PTE entry controlling a 4K page consumes 8 bytes of space, resulting in a fully populated 3GB process address space consuming 6MB of PTE entries. In other words, the overhead of PTEs is 0.2% of physical RAM if we have no sharing going on.

In most workloads, however, there is significant amounts of space shared between processes, either in shared libraries, or as shared memory segments. In particular, database workloads often use large shared segments (e.g. 2GB) shared between large numbers of processes. Whilst the memory itself is shared

between processes, the pagetables are duplicated; one copy for each process. Thus for 5000 processes sharing a 2GB shmem segment, the PTE overhead for that segment is now 20GB of RAM (i.e. the overhead is 1000% of the consumed space).

These levels of heavy sharing are for real workloads analysed, not a theoretical projection, and for a machine that might otherwise run happily with 8GB of RAM.There are two obvious ways to reduce the overhead: either we share the pagetables, or we reduce the size of each copy substantially.

Sharing the PTE level of the pagetables has been implemented by Dave McCracken. This enables us to share identical mappings over large areas between different processes, thereby reducing the mapping overhead for the case under consideration from 20GB to 4MB. It is totally transparent to applications, but is not currently in the 2.5 kernel as of 2.5.68.

The locking required for shared pagetables makes the patch slightly complex, and sharing can only occur on a pte-page sized basis (2MB of memory). Due to the mechanisms of shared libraries writing to certain areas of the shlib (and thus causing a copy-on-write split for the 2MB area) and the alignment requirements, it is not generally useful as a mechanism for reducing the overhead of shared libraries. It is, however, extremely effective on large shared memory segments, and reduces the overhead of fork+exec for large processes.

Support for large hardware pagesizes (aka hugetlbfs) can dramatically reduce the overhead of each process' pagetables. On IA32 PAE, there is only 1 large page size available (2MB), and this reduces the overhead by a factor of approximately 512. Memory consumption for the case under consideration is reduced for this case from 20GB to about 60MB. This is in the Linux 2.5 kernel, but requires small modifications to applications in order to use this facility.

A static pool of memory reserved for large pages is established at boot time, and handed out to applications that request it via a flag to shared memory create calls. Future work is planned to make a more flexible mechanism, whereby it is not necessary to reserve a static number of pages, and the kernel automatically uses large pages where appropriate.

In order to accomodate the large numbers of pagetables that are potentially needed on larger systems, it is possible to put the third level of the pagetables (PTEs) into the high memory area, rather than the main global kernel space. While this can greatly alleviate the space consumption problem, it comes at a price in terms of time.

Though modern implementations of highmem pagetables use atomic kmap, the cost of setting up the mappings for access, and the subsequent TLB flush is still expensive for such heavy usage, especially for workloads that create and destroy processes frequently. For kernel compiliation, the overhead of highpte was an increase of approximately 8% of system time.

5 UKVA

The shortage of virtual space on IA32 keeps the kernel from directly mapping everything that it might like to, especially things which are in high memory. We have mechanisms to do this temporarily with kmap() and kmap_atomic(), but both of these mechanisms impose significant overhead in data management and tlb flushing.

For workloads with large numbers of processes, one of the largest consumers of virtual space is page tables, specifically the bottom-level PTE pages. There is an option (high-

pte) in the kernel to put these pages in high memory and map them via kmap_atomic() as needed, but this incurs an 8% increase in system overhead. It would be much more efficient to permanently map the PTE pages, but into a per-process area instead of a global one, thus giving efficient operation without wasting large amounts of virtual space.

UKVA (User-kernel virtual addressing) provides a per-process kernel memory area. The same virtual space in each process is mapped to different physical pages, just like the userspace addresses (thus the "U"), but with the protections of kernel space (hence the "K"). A previous implementation actually located this area in the current user area. However, that implementation would have made it difficult to locate things other than user PTEs in the area. The implementation described here will locate the area inside the current kernel virtual area, and concentrate on locating PTEs in the area.

The first question is, "How big does it need to be?" An IA32 machine with PAE enabled has 4k pages, each controlled by a 8-byte pte entry. Each process will only need enough UKVA space to map in its own page tables.

4GB / (4K / page) = 1M pages

1M pages * 8 bytes/pte = 8MB of virtual space for ptes

For the purpose of this example, this space will be from 4G–8MB through 4GB; however, it does not really matter *where* this area is. It does not *need* to be aligned in a certain way, but this does make it easier to work with. First, the area should be aligned on a PTE page/PMD entry boundary (2MB with PAE). This will make it certain that the whole area can itself be mapped with only 4 PTE pages (more on this below). Secondly, the area should not straddle a PMD boundary, to avoid the initial setup being required to map more than 1 page, which makes it more expensive.

To map the required 8MB of virtual space, we require 2048 4K page table entries. We can fit 512 pte entries per 4K page, thus we require 4 UKVA PTE pages. Each time a pte page needs to be mapped in to the UKVA area, one of these 2048 pte entries contained in the 4 UKVA PTE pages will need to be set.

5.1 Initialization

Since the UKVA area will be the primary means for access to all PTE pages, it must be available for the entire life of the process's pagetables. For this reason, the initialization will occur in pgd_alloc(), at the same time as the top level pagetable entry is created.

On IA32, a pmd entry and a pte entry are the same size and PTRS_PER_PMD (the count of pmd entries per pmd page) is the same as PTRS_PER_PTE (the count of pte entries per pte page). Also, every time a pte page is allocated, a pmd entry is pointed to it. Each time you want to map an allocated PTE page, a pte entry is made, somewhere (highpte uses kmap() to do this). Instead of setting kmap() ptes, we will use UKVA ptes. This means there will be a 1:1 relationship between PMD pages/entries and UKVA PTE pages/entries.

During pgd_alloc(), the 4 UKVA PTE pages are allocated as soon as the PMD page which will point to them is allocated. The 4 pmd entries are made for the 4 pte pages, as are the corresponding 4 pte entries. However, making the PTE entries is slightly complex. One of the goals of UKVA is to replace HIGHPTE, which means that all of the PTE pages will be allocated in highmem, including the special 4 UKVA PTE pages. This means that the pte page that contains the 4 pte entries will need to be mapped via atomic kmap to make the entries. However, after this is done, they may

be accessed directly, without ever using kmap again.

This is the time where keeping the 8MB area from crossing a PMD boundary is important. Because of the 1:1 relationship, if the 4 pages are covered by more than 1 PMD page, they will also be covered by more than 1 PTE page, possibly doubling the amount of number of kmap calls which must occur.

5.2 Runtime

The bulk of the UKVA work is done in pte_alloc_map(). In keeping with the 1:1 relationship, each time a pmd_populate() is done, a UKVA PTE is also set. However, there are 2 possible ways to set a PTE with UKVA.

The first method is to simply index into the UKVA area, and set the PTE directly. Since the UKVA area is virtually contiguous, it can be accessed just like an array of every PTE in the system. The PTE contolling the first page of memory is at the start of the UKVA PTE space, just as the last PTE in the space controls the last page of RAM. It will always be known where the PTE for any given virtual address will be mapped. This also means the the 4 UKVA PTE pages themselves will be mapped to constant, known places.

The second method is used when pte_alloc_map() is asked to allocate a PTE for another process. Since the UKVA only contains the current process's pagetables, the pagetables of the other process must be walked, and appropriate entries made. During the walking process, the UKVE PTE pages must be mapped via kmap_atomic() so that they can be altered.

6 Hot & Cold pages

As the ratio of CPU speed to memory speed grows over time and CPU architectures change in ways such as pipelining, the efficient utilisation of processor caches becomes increasingly important. The hot and cold pages mechanism in Linux 2.5 (and its predecessors such as percpu pages) provide an important way to help increase the efficiency of the data cache. This is important for UP systems, but provides even greater benefit on SMP.

For each CPU in the system, for each zone of memory, we provide two queues for data pages: a hot queue, and a cold queue. The general precept is that pages in the hot queue are cache hot on that CPU, and pages on the cold queue are cache cold. Only 0-order pages (single page groups) are kept in these queues, the higher order allocations (multipage groups) are managed directly by the buddy allocator.

Both the hot and cold page lists allocate pages and free pages en masse from the buddy allocator for greater efficiency. This allows us to take multiple pages under one holding of the lock, whilst those codepaths and data management elements are cache hot. The lists have low watermarks, below which they will be refilled, and high watermarks, above which they will be emptied. Default batch size for allocations is 16 pages at a time; watermarks are 32–96 pages for the hotlists, and 0–32 pages for the cold lists.

The hot queue is managed as a LIFO stack—pages freed via the normal free_pages() route are pushed onto the hot stack (i.e. assumed to be cache warm). Further tuning in this area may be needed—the caller has better information about the cache warmth of the pages they are freeing than the generic routines. By default, page allocations come out of the hot list, unless __GFP_COLD is specified.

The cold list basically just functions as a batching mechanism for page allocations. It is used for pages that will not be first touched by the CPU in question (e.g. pagecache pages that

will be filled by DMA before they are read). This preserves the valuable cache hot pages for other uses, and saves cacheline invalidates for the CPU's cache. shrink_list(), shrink_cache(), and refill_inactive_zone() all free pages back into the cold list via the pagevec mechanism.

Below is a comparison of the kernel profiles of an equivalent workload (kernel compile) with and without the hot & cold pages mechanism, measuring how many ticks are spent in each routine, and which routines see the greatest change. Those labelled '+' get more expensive with hot & cold pages, those labelled '-' get cheaper. The 17.5% overall reduction in the total number of ticks spent is evident.

```
  Ticks  Percent Routine
   +243     0.0% buffered_rmqueue
   +197     6.6% page_remove_rmap
   +131     0.0% handle_mm_fault
   +116     0.0% fget
...
    -77   -15.7% release_pages
    -78   -34.4% atomic_dec_and_lock
    -89   -18.5% d_lookup
    -97   -16.2% __get_page_state
   -155   -35.6% link_path_walk
   -155   -23.8% copy_page_range
   -178   -27.8% shmem_getpage
   -193   -24.6% do_no_page
   -209  -100.0% pte_alloc_one
   -210   -26.0% zap_pte_range
   -303  -100.0% pgd_alloc
   -365  -100.0% __free_pages_ok
   -532   -28.0% do_anonymous_page
   -650   -37.0% do_wp_page
   -700  -100.0% rmqueue
  -4595   -17.5% total
```

The cost of rmqueue, pgd_alloc, pte_alloc_one, and __free_pages_ok has shifted into buffered_rmqueue, but it takes much less time to execute. The main cost for do_anonymous_page was in zeroing newly allocated pages, and the cost of do_wp_page is in copying pages from one to the other. Both obviously benefit greatly from the better cache warmth of the system. The profiles only show kernel time—userspace is actually the biggest beneficiary from this mechanism.

7 Page reclaim

In 2.5, the LRU lists were converted from global to per-zone. This makes it easier to free up one particular type of memory (e.g. ZONE_NORMAL) without affecting other types. It also breaks up the global locks and reduces cross-node cacheline traffic for NUMA machines. Following is the most significant elements from kernel profile data from a 2.4.18 kernel + NUMA patches doing a kernel compile on a 16-way NUMA-Q:

```
  2763 _text_lock_dcache
  2499 _text_lock_swap
  1199 do_anonymous_page
   763 d_lookup
   651 lru_cache_add
   646 __free_pages_ok
   612 do_generic_file_read
   573 lru_cache_del
...
```

The _text_lock_swap entry is the pagemap_lru_lock

The page-reclaim daemon (kswapd) must touch large amounts of data, both the user pages being manipulated, and their control structures (e.g. the LRU lists). However, on NUMA systems this is extremely problematic, as it causes a lot of cross-node memory traffic. Hence the global daemon was replaced with a per-node daemon, each of which only scans its own nodes pages, which is much more efficient.

One of the last major global locks in the VM was pagemap_lru_lock. Andrew Morton's pagevec implementation reduced contention on it by 98% by batching page operations together

into ‘pagevecs’—vectors of pages that could be manipulated together as groups more efficiently.

8 rmap

Whilst the pagetables of each process provide a mapping from that virtual address space to the physical addresses backing it, in Linux 2.4, there is no easy way to map back from a physical address to a virtual address. This is what rmap provides—a “reverse” mapping from the physical address back to the set of virtual addresses mapping it.

To reclaim memory, 2.4 used a “virtual scan”—walk each process, and see if we can unmap the physical pages it is using. 2.5 uses “physical scan”—walk each page of RAM, and see if it can be freed (this requires the rmap mechanism). This new mechanism has several advantages, perhaps the most important of which is stability. It has proven significantly more robust under pressure than the virtual scan in 2.4 code.

Whilst the usage of the rmap mechanism has remained fairly stable, the method of keeping the data for the reverse mapping has been a source of more contention and trouble. The current 2.5 code as of 2.5.68 uses a mechanism called “pte-chains” which keeps (for each physical page) a simple linked list of pointers back to the pte entries of the processes mapping each page.

These pte-chains have several problems:

- Locking
- Space consumption
- Time consumtion

The locking was at first implemented as a global lock, which was actually shared with the worst existing global VM lock (pagemap_lru_lock). This caused massive lock contention (data from a 12-way NUMA-Q), as seen in Table 1.

Therefore, the locking was changed to a per-chain lock, which was subsequently compacted into a 1 bit lock embedded in the flags field of the struct page to avoid more space consumption problems. This reduced kernel compile times by more than half on 16-way NUMA-Q (from 85s to 40s).

The next problem with pte-chains is the space consumption. A simple singly linked list will consume 4 bytes per entry for the pointer to the PTE and 4 bytes per entry for the pointer to the next entry. Two methods were used to alleviate this:

1. Pages with only a single mapping can use the “page-direct” optimisation—instead of storing the pointer to the linked list in the struct page, we use the same space to point directly to the only PTE by using the pte union introduced into struct page:

   ```
   union {
       struct pte_chain *chain;
       pte_addr_t direct;
   } pte;
   ```

   And this switch is governed by the PG_direct flag from the flags field in the struct page.

2. The lists are grouped by cacheline, allowing multiple PTE pointers per ‘list next’ pointer. Not only does this reduce the size of the linked list by almost half (assuming sufficient grouping), but it also greatly increase the data locality and cache efficiency for walking the chain.

Moreover, this space consumption all comes from low memory, an extremely precious resource on large 32-bit machines. To take

SPINLOCKS		HOLD		WAIT						
UTIL	CON	MEAN	MAX	MEAN	MAX	% CPU	TOTAL	NOWAIT	SPIN	NAME
45.5%	72.0%	8.5us	341us	138us	11ms	44.3%	3067414	28.0%	72.0%	pagemap_lru_lock
0.03%	31.7%	5.2us	30us	139us	3473us	0.02%	3204	68.3%	31.7%	deactivate_page+0xc
6.0%	78.3%	6.8us	87us	162us	9847us	9.4%	510349	21.7%	78.3%	lru_cache_add+0x2c
6.7%	73.2%	7.6us	180us	120us	8138us	6.4%	506534	26.8%	73.2%	lru_cache_del+0xc
12.7%	64.2%	7.1us	151us	140us	10ms	13.4%	1023578	35.8%	64.2%	page_add_rmap+0x2c
20.1%	76.0%	11us	341us	133us	11ms	15.0%	1023749	24.0%	76.0%	page_remove_rmap+0x3c

Table 1: Massive lock contention on 12-way NUMA-Q

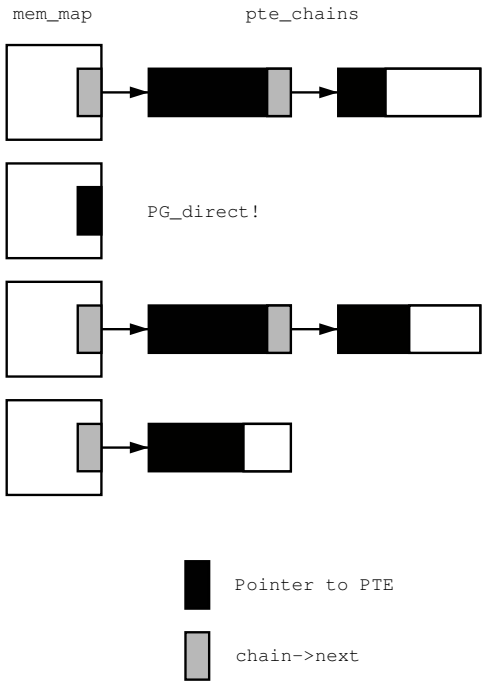


Figure 2: pte-chain based rmap

our example of 5000 processes sharing a 2GB memory segment again, not only do we now have 20GB of pagetables, but 10GB of pte_chains. Whilst the 20GB of pagetables can at least be moved off into high memory, this is not easy to do for pte chains. In order to move the chains into high memory, the "next element" pointers would need to become physical addresses, instead of virtual ones. Not only are these larger (36 bits instead of 32), they also need to be mapped into virtual addresses before use, an incredibly expensive procedure for walking the linked lists.

Last, but not least, of the problems is the time consumption. For every page used, and for every process that uses it, we must take a lock (using an expensive atomic operation) and create a page entry. Worse still, when we tear down the mapping, we must take that same lock, and then walk the pte-chain looking for the element to free. This takes approximately a linear amount of time, depending on the number of elements sharing that page. Even for just a load of 128 on SDET, the kernel profile shows the rmap functions massively dominating:

```
86159 page_remove_rmap
38690 page_add_rmap
17976 zap_pte_range
14431 copy_page_range
10953 __d_lookup
 9978 release_pages
 9369 find_get_page
 7483 atomic_dec_and_lock
 6924 __copy_to_user_ll
 6830 kmem_cache_free
```

The problem is especially acute under a workload such as SDET that does significant amounts of fork/exec/exit traffic, where mappings must be continually built up and then torn down again. I see this problem as fundamental with a page based approach; though it may be alleviated somewhat by tuning, it is still a per-page operation, and thus too expensive. Even for a simple kernel compile, the page_remove_rmap is still the most expensive function in the whole kernel:

```
23222 page_remove_rmap
14034 do_anonymous_page
 7638 __d_lookup
 6406 page_add_rmap
 5188 __copy_to_user_ll
 3656 find_get_page
 3429 __copy_from_user_ll
 3126 zap_pte_range
 2108 do_page_fault
 1925 atomic_dec_and_lock
 1852 path_lookup
```

8.1 rmap shadow pages

Ingo Molnar has proposed a new rmap method which I shall call "rmap shadow pages," which alleviates some of the problems with pte-chains, but is still page based. At the time of writing, there is no implementation available, but some of its properties may be determined by analysis.

Instead of the chains being allocated as needed on a cacheline sized block, it is proposed to allocate two "shadow pages" for each pte-page (page filled with PTE entries). These would form a doubly-linked list with other shadow pages. To retrieve the list of PTE entries for a particular page, one would consult the pte-chain pointer in the relevant struct page, and walk the list (similarly to PTE-chains).

The PTE entry itself need not be stored inside the rmap shadow page, but the rmap pages are implicitly "linked" to the pte page in some fashion (e.g. being placed together in some contiguous group of two pages). However, the cacheline locality characteristics seem to be against this method for scanning, as it involves touching a separate cacheline for every element in the list.

To add a page to the linked list, we would take the pte_chain lock, and add ourselves to the head of the list (one modification to the shadow page, plus one to the struct page). To remove a page from the list would not require walking the list (as for pte_chains), but we would traverse from the PTE page to the corresponding shadow page, map both its prev and next element pages, and perform a regular unlink for a doubly-linked list.

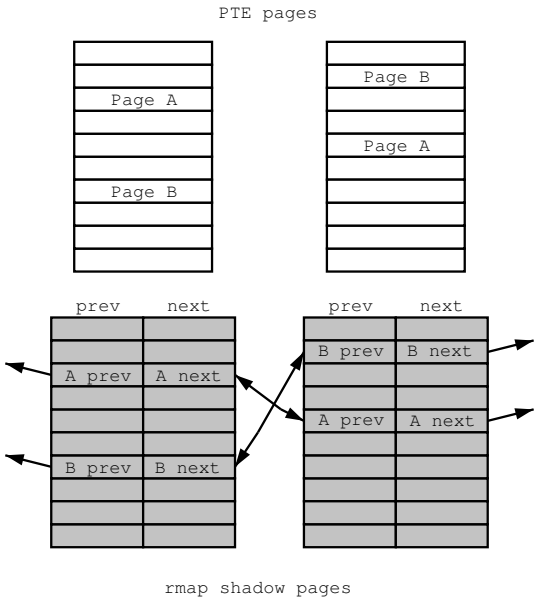


Figure 3: rmap shadow pages

One of the major advantages of the rmap shadow pages method is that the rmap data can be more easily moved into the highmem area. However, this is not without cost—each page accessed must be mapped via kmap, which has proven expensive for PTE pages in the highpte implementation.

Whilst rmap shadow pages may fix some of the problems of pte-chains, it is still page-based, and thus requires a large amount of data manipulation. It is therefore unlikely to solve the fundmantal time and space problem, though moving the chains into high memory may be worthwhile.

8.2 Object based rmap

Other operating systems have taken a different approach to the physical to virtual address mapping problem. K42 has taken an approach based on file objects (akin to the Linux address_space structure), which seemed to be promising after initial discussions with Orran Krieger and other K42 engineers.

Instead of keeping a reverse mapping for each page in the system, we can discover the list of virtual addresses by going from the struct page to the address_space object. From the address_space object, we can walk a list of vmas—areas of process virtual memory which map that object. By adding the offset within the file (stored in struct page as "index") to the base virtual address of each vma, we can derive the virtual address within each process. From there, we can walk the pagetables to find the appropriate PTE.

The key advantage of this method is that there is no overhead at all for the setup and teardown of each page. This comes at the cost of higher overhead to find the PTEs at scanning time (the list of VMAs and the pagetables for each process must be walked). However, many workloads will run without memory pressure, or only have pressure on the caches, which are easily freed, so the approach seems promising.

However, there are a few fundamental problems with an object-based approach within Linux. For one, there is not a backing file object for every page in the system, some pages (e.g. private process data allocated via malloc/sbrk) are anonymous, i.e. not associated with any file. Whilst it would be possible to create a file object for anonymous pages, this would not be a simple change.

Another problem is that the calculation of adding the offset to the base virtual address within the process assumes that the vma is linear. Whilst this used to be true, the 2.5 kernel contains a new mechanism called "sys_remap_file_pages" that allows for non-linear VMAs.

Bearing in mind that pte_chains are most expensive under heavy sharing (the linked list must be walked for page_remove_rmap), some analysis was taken of the length of the pte-chains for both file-backed and anonymous objects. This showed that the anonymous objects were only mapped once for nearly all pages, and the shared mappings were nearly all file-backed.

Based on these observations, and the simplicity of implementation, a "partially object-based" scheme was proposed. This used the object-based mappings for file-backed pages, and the pte-chains method for anonymous memory (and for nonlinear mappings). Dave McCracken implemented this scheme, and it was very effective in reducing both the space and time taken by pte-chains.

Kernbench-16: (make -j 256 vmlinux)				
	Elapsed	User	System	CPU
pte-chains	47.21	569.17	139.55	1500.67
partial objrmap	46.09	568.19	121.83	1496.67

Note the 12% drop in total system time.

SDET 64 (see disclaimer)		
	Throughput	Std. Dev
2.5.68	100.0%	0.2%
2.5.68-objrmap	121.2%	0.3%

Again, kernel profiles clearly show the reduction:

```
-30518   -78.9%   page_add_rmap
-72197   -83.8%   page_remove_rmap
```

Partial objrmap also drastically reduced the number of pte_chain objects in the slab cache, graphically demonstrating that file_backed chains are predomininant. For a `make -j256 vmlinux`:

pte-chains	24116	pte_chain objects in slab cache
objrmap	716	pte_chain objects in slab cache

(a 97% reduction).

However, at the time of writing, there are still a couple of remaining objections to partial objrmap:

1. The interaction with sys_remap_file_page's nonlinear vmas is complex and convoluted due to the conversion to and from pte_chains which may be necessary. However, it is generally recongnised that sys_remap_file_pages is a special case. If those vmas are pre-declared as nonlinear, most of the problems disappear. Furthermore, for the intended use of sys_remap_file_pages (windowing onto large database shared segments), it is acceptable to lock those pages into memory (this is normally done in any case), in which case none of this information will ever be needed, so it is not necessary to keep it.

2. If 100 processes each map 100 vmas onto the same address space, then objrmap would have to scan 10,000 regions, not simply the 100 mappings that the page-based methods might have had for their chains. Whether this corner case is important or not is a matter of debate, as it was exactly the case that sys_remap_file_pages was designed to fix, and callers should be using that method for such an unusual situation. However, a simple optimisation is proposed that should alleviate the problem in any case:

For each distinct range of addresses mapped by a vma inside the address_space, we define an address_range. This takes advantage of the fact that we are likely to remap the same range repeatedly (e.g. for shared libraries). From each shared range, we attatch a list of vmas that map that range. Furthermore, we sort the list of address ranges by start address.

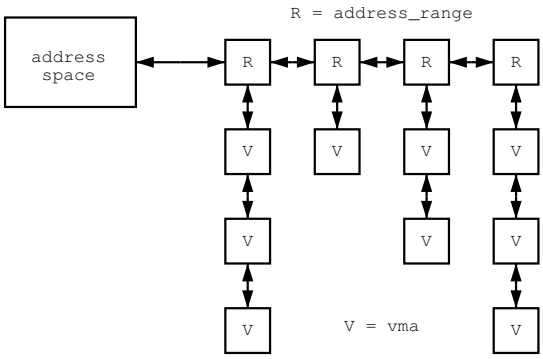


Figure 4: list of lists

```
struct address_range {
    unsigned long           start;
    unsigned long           end;
    struct list_head        ranges;
    struct list_head        vmas;
};
```

9 NUMA support

On NUMA systems, it is more efficient to access node-local memory than remote memory. Thus Linux tries to allocate memory to a process from the node it is currently running on, providing for more efficient performance. This also allows for locality of memory and control structures, reducing cross-node cacheline traffic.

By default, we allocate memory from the local node (if some is free), then round robin amongst the remaining nodes by node number, starting at the local node, and progressing upwards. Kernel code can also request memory on specific nodes via alloc_pages_node().

Matt Dobson has created a simple NUMA binding interface for userspace, allowing processes to request memory from a specific node,

or group of nodes. This is very useful for large database applications, which wish to bind "partitions" of the database to certain nodes.

Several critical control structures (e.g. the mem_map array—the control structures for the physical RAM pages, and the pgdat—the node control structure) are now allocated on the nodes own memory, providing for better performance on NUMA systems. More items (e.g. the scheduler data, and per-cpu data) should be migrated into node-local memory in the future.

Of the three main memory allocators (alloc_pages, vmalloc, slab cache), only the slab cache is not NUMA aware. Manfred Spraul has created patches to do this, but we have not seen observeable performance benefits from this method yet. Part of the problem is the inherently global nature of many of the caches (eg the directory cache), which will need to be attacked first.

Some of the kernel architectures (ones with hardware assistance from the CPU) have kernel text replication functioning. This makes a copy of the kernel data to each node, and processes will use their own node's local copy of the data, reducing backplane traffic, and interconnect cache pollution. Replicating the read-only portion of shared libraries also seems promising, though this has not yet been implemented in Linux. Replicating any data that is not read-only is likely to be too complex to be beneficial.

10 Legal

Integrating DMA Into the Generic Device Model

James E.J. Bottomley
SteelEye Technology, Inc.
http://www.steeleye.com
James.Bottomley@steeleye.com

Abstract

This paper will introduce the new DMA API for the generic device model, illustrating how it works and explaining the enhancements over the previous DMA Mapping API. In a later section we will explain (using illustrations from the PA-RISC platform) how conversion to the new API may be achieved hand in hand with a complete implementation of the generic device API for that platform.

1 Introduction

Back in 2001, a group of people working on non-x86 architectures first began discussing radical changes to the way device drivers make use of DMA. The essence of the proposal was to mandate a new DMA Mapping API[1] which would be portable to all architectures then supported by Linux. One of the forces driving the adoption of this new API was the fact that the PCI bus had expanded beyond the x86 architecture and being embraced by non-x86 hardware manufacturers. Thus, one of the goals was that any driver using the DMA Mapping API should work on *any* PCI bus independent of the underlying microprocessor architecture. Therefore, the API was phrased entirely in terms of the PCI bus, since PCI driver compatibility across architectures was viewed as a desirable end result.

1.1 Legacy Issues

One of the issues left unaddressed by the DMA Mapping API was that of legacy buses: Most non-x86 architectures had developed other bus types prior to the adoption of PCI (e.g. sbus for the sparc; lasi and gsc bus for PA-RISC) which were usually still present in PCI based machines. Further, there were other buses that migrated across architectures prior to PCI, the most prominent being EISA. Finally, some manufacturers of I/O chips designed them not to be bus based (the LSI 53c7xx series of SCSI chips being a good example). These chips made an appearance in an astonishing variety of cards with an equal variety of bus interconnects.

The major headache for people who write drivers for non-PCI or multiple bus devices is that there was no standard for non-PCI based DMA, even though many of the problems encountered were addressed by the DMA Mapping API. This gave rise to a whole hotchpotch of solutions that differed from architecture to architecture: On Sparc, the DMA Mapping API has a completely equivalent SBUS API; on PA-RISC, one may obtain a “fake” PCI object for a device residing on a non-PCI bus which may be passed straight into the PCI based DMA API.

1.2 The Solution

The solution was to re-implement the DMA Mapping API to be non bus specific. This goal was vastly facilitated by the new generic device architecture[5] which was also being implemented in the 2.5 Linux kernel and which finally permitted the complete description of device and bus interconnections using a generic template.

1.3 Why A New API

After all, apart from legacy buses, PCI is the one bus to replace all others, right? so an API based on it must be universally applicable?

This is incorrect on two counts. Firstly, support for legacy devices and buses is important to Linux, since being able to boot on older hardware that may have no further use encourages others who would not otherwise try Linux to play with it, and secondly there are other new non-PCI buses support for which is currently being implemented (like USB and firewire).

1.4 Layout

This paper will describe the problems caused by CPU caches in section 2, move on to introducing new struct device based DMA API[2] in section 3 and describe how it solves the problems, and finally in section 4 describe how the new API may be implemented by platform maintainers giving specific examples from the PA-RISC conversion to the new API.

2 Problems Caused By DMA

The definition of DMA: Direct Memory Access means exactly that: direct access to memory (without the aid of the CPU) by a device transferring data. Although the concept sounds simple, it is fraught with problems induced by the way a CPU interacts with memory.

2.1 Virtual Address Translation

Almost every complex CPU designed for modern Operating Systems does some form of Virtual Address Translation. This translation, which is usually done inside the CPU, means that every task running on that CPU may utilise memory as though it were the only such task. The CPU transparently assigns each task overlapping memory in virtual space, but quietly maps it to unique locations in the physical address space (the memory address appearing on the bus) using an integral component called a MMU (Memory Management Unit).

Unfortunately for driver writers, since DMA transfers occur without CPU intervention, when a device transfers data directly to or from memory, it must use the physical memory address (because the CPU isn't available to translate any virtual addresses). This problem isn't new, and was solved on the x86 by using functions which performed the same lookups as the CPU's MMU and could translate virtual to physical addresses and vice versa so that the driver could give the correct addresses physical addresses to the hardware and interpret any addresses returned by the hardware device back into the CPU's virtual space.

However, the problems don't end there. With the advent of 64 bit chips it became apparent that they would still have to provide support for the older 32 bit (and even 24 bit) I/O buses for a while. Rather than cause inconvenience to driver writers by arbitrarily limiting the physical memory addresses to which DMA transfers could be done, some of the platform manufacturers came up with another solution: Add an additional MMU between the I/O buses and the processor buses (This MMU is usually called the IOMMU). Now the physical address lim-

itation of the older 32 bit buses can be hidden because the IOMMU can be programmed to map the physical address space of the bus to anywhere in the `physical` (not virtual) memory of the platform. The disadvantage to this approach is that now this bus physical to memory physical address mapping must be programmed into the IOMMU and must also be managed by the device driver.

There may even be multiple IOMMUs in the system, so a physical address (mapped by a particular IOMMU) given to a device might not even be unique (another device may use the same bus address via a different IOMMU).

2.2 Caching

On most modern architectures, the speed of the processor vastly exceeds the speed of the available memory (or, rather, it would be phenomenally expensive to use memory matched to the speed of the CPU). Thus, almost all CPUs come equipped with a cache (called the Level 1 [L1] cache). In addition, they usually expose logic to drive a larger external cache (called the Level 2 [L2] cache).

In order to simplify cache management, most caches operate at a minimum size called the cache width.[1] All reads and writes from main memory to the cache must occur in integer multiples of the cache width. A common value for the cache width is sixteen bytes; however, higher (and sometimes for embedded processors, lower) values are also known.

The effect of the processor cache on DMA can be extremely subtle. For example, consider a hypothetical processor with a cache width of sixteen bytes. Referring to Figure 1, supposing I read a byte of data at address `0x18`. Because of the cache burst requirement, this will

[1] also called the "cache line size" in the PCI specification

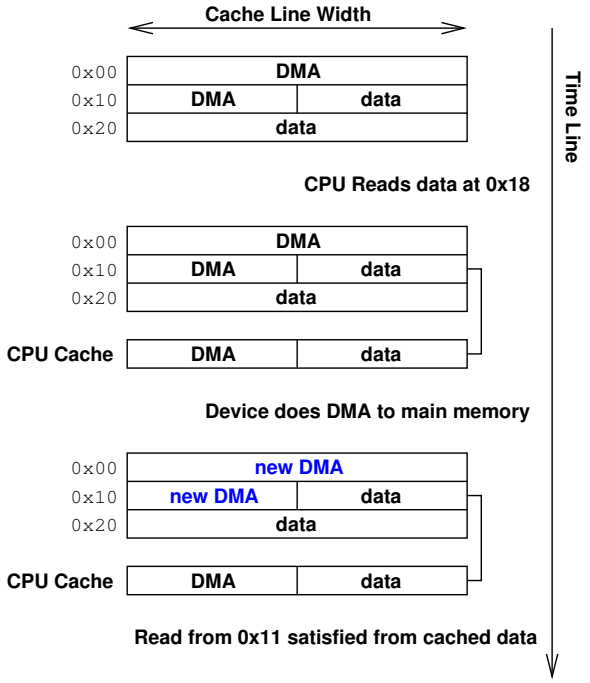


Figure 1: Incorrect data read because of cache effects

bring the address range `0x10` to `0x1f` into the cache. Thus, any subsequent read of say `0x11` will be satisfied from the cache without any reference to main memory. Unfortunately, If I am reading from `0x11` because I programmed a device to deposit data there via DMA, the value that I read will not be the value that the device placed in main memory because the CPU believes the data in the cache to be still current. Thus I read incorrect data.

Worse, referring to Figure 2, supposing I have designated the region `0x00` to `0x17` for DMA from a device, but then I write a byte of data to `0x19`. The CPU will probably modify the data in cache and mark the cache line `0x10` to `0x1f` dirty, but not write its contents to main memory. Now, supposing the device writes data into `0x00` to `0x17` by DMA (the cache still is not aware of this). However, subsequently the CPU decides to flush the dirty cache line from `0x10` to `0x1f`. This flush will overwrite (and thus destroy) part of the the data

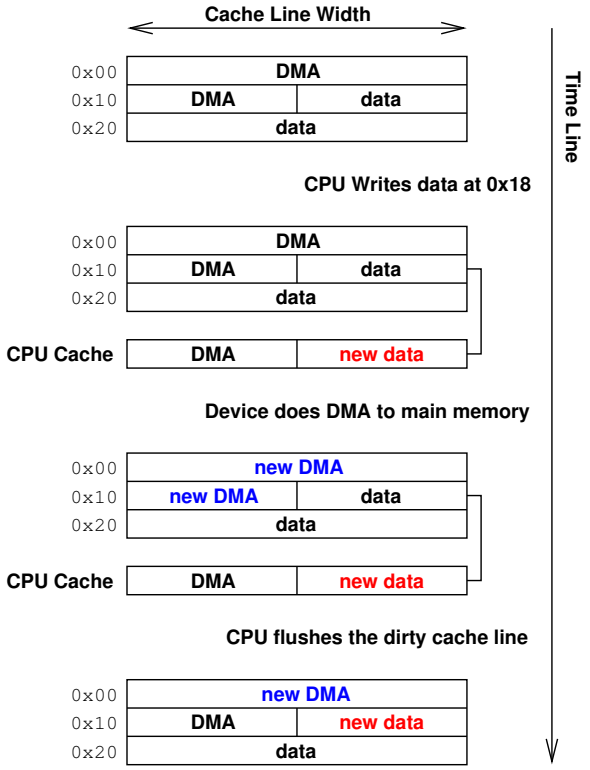


Figure 2: Data destruction by interfering device and cache line writes

that was placed at `0x10` to `0x17` by the DMA from the device.

The above is only illustrative of some of the problems. There are obviously many other scenarios where cache interference effects may corrupt DMA data transfers.

2.3 Cache Coherency

In order to avoid the catastrophic consequences of caching on DMA data, certain processors exhibit a property called "coherency." This means that they take action to ensure that data in the CPU cache and data in main memory is identical (usually this is done by snooping DMA transactions on the memory bus and taking corrective cache action before a problem is caused).

Even if a CPU isn't fully coherent, it can usually designate ranges of memory to be coherent (in the simplest case by marking the memory as uncacheable by the CPU). Such architectures are called "partially coherent."

Finally there is a tiny subset of CPUs that cannot be made coherent by any means, and thus the driver itself must manage the cache so as to avoid the unfortunate problems described in section 2.2.

2.4 Cache Management Instructions

Every CPU that is not fully coherent includes cache management instructions in its repertoire. Although the actual instruction format varies, they usually operate at the level of the cache line and they usually perform one of three operations:

1. Writeback (or flush): causes the cache line to be synced back to main memory (however, the data in the cache remains valid).

2. Invalidate: causes the cache line to be eliminated from the cache (often for a dirty cache line this will cause the contents to be erased). When the cpu next references data in that cache line it will be brought in fresh from main memory.

3. Writeback and Invalidate: causes an atomic sequence of Writeback followed by Invalidate (on some architectures, this is the only form of cache manipulation instruction implemented).

When DMA is done to or from memory that is not coherent, the above instructions must be used to avoid problems with the CPU cache. The DMA API contains an abstraction which facilitates this.

2.5 Other Caches

Although the DMA APIs (both the PCI and generic device ones) are concerned *exclusively* with the CPU cache, there may be other caches in the I/O system which a driver may need to manage. The most obvious one is on the I/O bus: Buses, like PCI, may possess a cache which they use to consolidate incoming writes. Such behaviour is termed "posting." Since there are no cache management instructions that can be used to control the PCI cache (the cache management instructions only work with the CPU cache), the PCI cache employs special posting rules to allow driver writers to control its behaviour. The rules are essentially:

1. Caching will not occur on I/O mapped spaces.
2. The sequence of reads and writes to the memory mapped space will be preserved (although the cache may consolidate write operations).

One important point to note is that there is no defined time limit to hold writes in the bus cache, so if you want a write to be seen by the device it must be followed by a read.

Suffice it to say that the DMA API does not address PCI posting in any form. The basic reason is that in order to flush a write from the cache, a read of somewhere in the device's memory space would have to be done, but only the device (and its driver) know what areas are safe to read.

3 The New DMA API for Driver Writers

By "driver writer", we mean any person who wishes to use the API to manage DMA coherency without wanting to worry about the underlying bus (and architecture) implementation.

This part of the API and its relation to the PCI DMA Mapping API is fairly well described in [2] and also in other published articles [3].

3.1 DMA Masks, Bouncing and IOMMUs

Every device has a particular attachment to a bus. For most, this manifests itself in the number of address lines on the bus that the device is connected to. For example, old ISA type buses were only connected (directly) to the first twenty four address lines. Thus they could only directly access the first sixteen megabytes of memory. If you needed to do I/O to an address greater than this, the transfer had to go via a bounce buffer: e.g. data was received into a buffer guaranteed to be lower than sixteen megabytes and then copied into the correct place. This distinction is the reason why Linux reserves a special memory zone for legacy (24 bit) DMA which may be accessed using flag (`GFP_DMA`) or'd into the allocation flags.

Similarly, a 32 bit PCI bus can only access up to the first four gigabytes of main memory (and even on a 64 bit PCI bus, the device may be only physically connected to the first 32 or 24 address lines).

Thus, the concept of a *dma mask* is used to convey this information. The API for setting the dma mask is:

```
int dma_set_mask(struct
device *dev, u64 mask)
```

- `dev`—a pointer to the generic device.
- `mask`—a representation of the bus connection. It is a bitmap where a 1 means the line is connected and a zero means it isn't. Thus, if the device is only con-

nected to the first 24 lines, the `mask` will be `0xffffff`.

- returns true if the bus accepted the mask.

Note also that if you are driving a device capable of addressing up to 64 bits, you must also be aware that the bus it is attached to may not support this (i.e. you may be a 64 bit PCI card in a 32 bit slot). So when setting the DMA mask, you must start with the value you want but be prepared that you may get a failure because it isn't supported by the bus. For 64 bit devices, the convention is to try 64 bits first but if that fails set the mask to 32 bits.

Once you have set the DMA mask, the troubles aren't ended. If you need transfers to or from memory outside of your DMA mask to be bounced, you must tell the block[2] layer using the

```
void blk_queue_bounce_limit
(request_queue_t *q, u64
mask)
```

- `q`—the request queue your driver is connected to.
- `mask`—the mask (again 1s for connected addresses) that an I/O transfer will be bounced if it falls outside of (i.e. `address & mask != address`)

function that it should take care of the bouncing. Note, however, that bouncing *only* needs to occur for buses without an IOMMU. For buses with an IOMMU, the mask serves only as an indication to the IOMMU of what the range of physical addresses available to the device is. The IOMMU is assumed to be able to address the full width of the memory bus and therefore transfers to the device need not be bounced by the block layer.

Thus, the driver writer must know whether the bus is mapped through an IOMMU or not. The means for doing this is to test the global `PCI_DMA_BUS_IS_PHYS` macro. If it is true, the system generally has no IOMMU[3] and you should feed the mask into the block bounce limit. If it is false, then you should supply `BLK_BOUNCE_ANY` (informing the block layer that no bouncing is required).

3.2 Managing block layer DMA transfers

It is the responsibility of the device driver writer to manage the coherency problems in the CPU cache when transferring data to or from a device and also mapping between the CPU virtual address and the device physical address (including programming the IOMMU if such is required).

By and large, most block devices are simply transports: they move data from user applications to and from storage without much concern for the actual contents of the data. Thus they can generally rely on the CPU cache management implicit in the APIs for DMA setup and tear-down. They only need to use explicit cache management operations if they actually wish to access the data they are transferring. The use of device private areas for status and messaging is covered in sections 3.5 and 3.6.

For setup of a single *physically contiguous*[4]

[2] Character and Network devices have their own ways of doing bouncing, but we will consider only the block layer in the following

[3] This is an inherent weakness of the macro. Obviously, it is possible to build a system where some buses go via an IOMMU and some do not. In a future revision of the DMA API, this may be made a device specific macro

[4] physically contiguous regions of memory for DMA can be obtained from `kmalloc()` and `__get_free_pages()`. They may specifically *not* be allocated on the stack (because data destruction may be

DMA region, the function is

```
dma_addr_t dma_map_single
(struct device *dev,
void *ptr, size_t size,
enum dma_data_direction
direction)
```

- `ptr`—pointer to the physically contiguous data (virtual address)
- `size`–the size of the physically contiguous region
- `direction`—the direction, either to the device, from the device or bidirectional (see section 3.4 for a complete description)
- returns a bus physical address which may be passed to the device as the location for the transfer

and the corresponding tear-down after the transfer is complete is achieved via

```
void dma_unmap_single
(struct device *dev,
dma_addr_t dma_addr,
size_t size, enum
dma_data_direction
direction)
```

- `dma_addr`—the physical address returned by the mapping setup function
- `size`, `direction`—the exact values passed into the corresponding mapping setup function

The setup and tear-down functions also take care of all the necessary cache flushes associated with the DMA transaction.[5]

caused by overlapping cache lines, see section 2.2) or `vmalloc()`

[5]they use the `direction` parameter to get this right, so be careful when assigning directions to ensure that they are correct.

3.3 Scatter-Gather Transfers

By and large, almost any transfer that crosses a page boundary will not be contiguous in physical memory space (because each contiguous page in virtual memory may be mapped to a non-contiguous page in physical memory) and thus may not be mapped using the API of section 3.2. However, the block layer can construct a list of each separate page and length in the transfer. Such a list is called a Scatter-Gather (SG) list. The device driver writer must map each element of the block layer's SG list into a device physical address.

The API to set up a SG transfer for a given `struct request *req` is

```
int blk_rq_map_sg
(request_queue_t *q,
struct request *req, struct
scatterlist *sg)
```

- `q`—the queue the request belongs to
- `sg`—a pointer to a pre allocated physical scatterlist which must be at least `req->nr_phys_segments` in size.
- returns the number of entries in `sg` which were actually used

Once this is done, the SG list may be mapped for use by the device:

```
int dma_map_sg(struct
device *dev, struct
scatterlist *sg, int nents,
enum dma_data_direction
direction)
```

- `sg`—a pointer to a physical scatterlist which was filled in by

- `nents`—the allocated size of the SG list.
- `direction`—as per the API in section 3.2
- returns the number of entries of the `sg` list actually used. This value must be less than or equal to `nents` but is otherwise not constrained (if the system has an IOMMU, it may chose to do all SG inside the IOMMU mappings and thus always return just a single entry).
- returns zero if the mapping failed.

Once you have mapped the SG list, you may loop over the number of entries using the following macros to extract the busy physical addresses and lengths

```
dma_addr_t sg_dma_address
(struct scatterlist *sge)
```

- `sge`–pointer to the desired entry in the SG list
- returns the busy physical DMA address for the given SG entry

```
unsigned int sg_dma_len
(struct scatterlist *sge)
```

- returns the length of the given SG entry

and program them into the device's SG hardware controller. Once the SG transfer has completed, it may be torn down with

```
void dma_unmap_sg (struct
device *dev, struct
scatterlist *sg, int nents,
enum dma_data_direction
direction)
```

- `nents` should be the number of entries passed in to `dma_map_sg()` *not* the number of entries returned.

3.4 Accessing the Data Between Mapping and Unmapping

Since the cache coherency is normally managed by the mapping and unmapping API, you may not access the data between the map and unmap without first synchronizing the CPU caches. This is done using the DMA synchronization API. The first

```
void dma_sync_single(struct
device *dev, dma_addr_t
dma_handle, size_t size,
enum dma_data_direction
direction)
```

- `dma_handle`—the physical address of the region obtained by the mapping function.
- `size`—The size of the region passed into the mapping function.

synchronizes only areas mapped by the `dma_map_single` API, and thus only works for physically contiguous areas of memory. The other

```
void dma_sync_sg
(struct device *dev,
struct scatterlist
*sg, int nelems, enum
dma_data_direction
direction)
```

- The parameters should be identical to those passed in to `dma_map_sg`

synchronizes completely a given SG list (and is, therefore, rather an expensive operation). The correct point in the driver code to invoke these APIs depends on the `direction` parameter:

- DMA_TO_DEVICE—Usually flushes the CPU cache. Must be called *after* you last modify the data and *before* the device begins using it.

- DMA_FROM_DEVICE—Usually invalidates the CPU cache. Must be called *after* the device has finished transferring the data and *before* you first try to read it.

- DMA_BIDIRECTIONAL—Usually does a writeback/invalidate of the CPU cache. Must be called *both* after you finish writing it but before you hand the data to the device *and* after the device finishes with it but before you read it.

3.5 API for coherent and partially coherent architectures

Most devices require a control structure (or a set of control structures), to facilitate communication between the device and its driver. For the driver and its device to operate correctly on an arbitrary platform, the driver would have to insert the correct cache flushing and invalidate instructions when exchanging data using this.

However, almost every modern platform has the ability to designate an area of memory as coherent between the processor and the I/O device. Using such a coherent area, the driver writer doesn't have to worry about synchronising the memory. The API for obtaining such an area is

```
void * dma_alloc_coherent
(struct device *dev,
size_t size, dma_addr_t
*dma_handle, int flag)
```

- dev—a pointer to the generic device

- size—requested size of the area

- dma_handle—a pointer to the area the physically usable address will be placed (i.e. this should be the address given to the device for the area)

- flag—a memory allocation flag. Either GFP_KERNEL if the allocation may sleep while finding memory or GFP_ATOMIC if the allocation may not sleep

- returns the virtual address of the area or NULL if no coherent memory could be allocated

Note that coherent memory may be a constrained system resource and thus NULL may be returned even for GFP_KERNEL allocations.

It should also be noted that the tricks platforms use to obtain coherent memory may be quite expensive, so it is better to minimize the allocation and freeing of these areas where possible.

Usually, device drivers allocate coherent memory at start of day, both for the above reason and so an in-flight transaction will not run into difficulties because the system is out of coherent memory.

The corresponding API for releasing the coherent memory is

```
void dma_free_coherent
(struct device *dev,
size_t size, void *vaddr,
dma_addr_t dma_handle)
```

- vaddr—the virtual address returned by dma_alloc_coherent

- dma_handle—the device physical address filled in at allocation time

3.6 API for fully incoherent architectures

This part of the API has no correspondance with any piece of the old DMA Mapping API. Some platforms (fortunately usually only older ones) are incapable of producing any coherent memory at all. Even worse, drivers which may be required to operate on these platforms usually tend to have to also operate on platforms which can produce coherent memory (and which may operate more efficiently if it were used). In the old API, this meant it was necessary to try to allocate coherent memory, and if that failed allocate and map ordinary memory. If ordinary memory is used, the driver must remember that it also needs to enforce the sync points. This leads to driver code which looks like

```
memory = pci_alloc_coherent(...);
if (!memory) {
  dev->memory_is_not_coherent = 1;
  memory = kmalloc(...);
  if (!memory)
    goto fail;
  pci_map_single(...);
}

....

if (dev->memory_is_not_coherent)
  pci_dma_sync_single(...);
```

Which cannot be optimized away. The non-coherent allocation additions are designed to make this code more efficient, and to be optimized away at compile time on platforms that can allocate coherent memory.

```
void *dma_alloc_noncoherent
(struct device *dev,
size_t size, dma_addr_t
*dma_handle, int flag)
```

- the parameters are identical to those of `dma_alloc_coherent` in section 3.5.

The difference here is that the driver *must* use a special synchronization API,[6] to synchronize this area between data transfers

```
dma_cache_sync (void
*vaddr, size_t size,
enum dma_data_direction
direction)
```

- `vaddr`—the virtual address of the memory to sync (this need not be at the beginning of the allocated region)
- `size`—the size of the region to sync (again, this may be less than the allocated size)
- `direction`—see section 3.4 for a discussion of how to use this.

Note that the placement of these synchronization points should be exactly as described in section 3.4. The platform implementation will choose whether coherent memory is actually returned. However, if coherent memory is returned, the implementation will take care of making sure the synchronizations become nops (on a fully coherent platform, the synchronizations will compile away to nothing).

Using this API, the above driver example becomes

```
memory = dma_alloc_noncoherent(...);
if (!memory)
  goto fail;

...

dma_cache_sync(...);
```

The (possibly) non-coherent memory area is freed using

[6] The driver could also use the API of section 3.4 but the point of having a separate one is that it may be optimized away on platforms that are partially non-coherent

```
void dma_free_noncoherent
(struct device *dev,
size_t size, void *vaddr,
dma_addr_t dma_handle)
```

- The parameters are identical to those of `dma_free_coherent` in section 3.4

Since there are very few drivers that need to function on fully non-coherent platforms, this API is of little use in modern systems.

3.7 Other Extensions in the New API

There are two other extensions over the old DMA Mapping API. They are

```
int dma_get_cache_alignment
(void)
```

- returns the cache alignment width of the platform (see section 2.2). Note, the value returned guarantees only to be a power of two and greater than or equal to the current processor cache width. Thus its value may be relied on to separate data variables where I/O caching effects would destroy data.

```
int dma_is_consistent
(dma_addr_t dma_handle)
```

- returns true if the physical memory area at `dma_handle` is coherent

And

```
void dma_sync_single_range
(struct device *dev,
dma_addr_t dma_handle,
unsigned long offset,
size_t size, enum
dma_data_direction
direction)
```

- `offset`—the offset from the `dma_handle`
- `size`—the size of the region to be synchronized

which allows a partial synchronization of a mapped region. This is useful because on most CPUs, the cost of doing a synchronization is directly proportional to the size of the region. Using this API allows the synchronization to be restricted only to the necessary parts of the data.

4 Implementing the DMA API for a Platform

In this section we will explore how the DMA API should be implemented from a platform maintainer's point of view. Since implementation is highly platform specific, we will concentrate on how the implementation was done for the HP PA-RISC[4] platform.

4.1 Brief Overview of PA-RISC

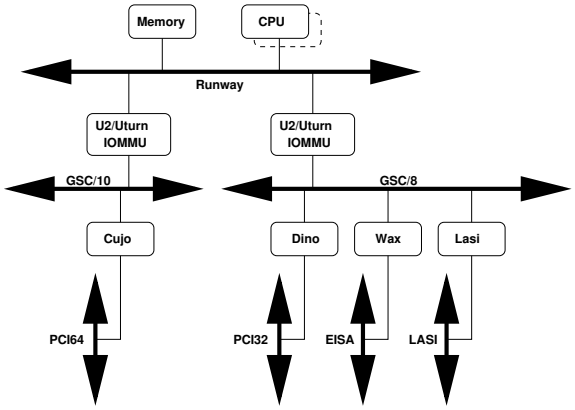


Figure 3: An abridged example of PA-RISC architecture

An abridged version of a specific PA-RISC architecture[7] is given in Figure 3. It is particu-

[7]The illustration is actually from a C360 machine, and does not show every bus in the machine

larly instructive to note that the actual chips involved (Cujo, U2/Uturn, Wax etc.) vary from machine to machine, as do the physical bus connections, so the first step that was required for PA-RISC was to build a complete model of the device layout from the runway bus on down in the generic device model[5].

4.2 Converting to the Device Model

Since PA-RISC already had its own device type (`struct parisc_device`), it was fairly simple to embed a generic device in this, assign it to a new `parisc_bus_type` and build up the correct device structure. For brevity, instead of giving all the PA-RISC specific buses (like Lasi, GSC, etc) their own bus type, they were all simply assigned to the `parisc_bus_type`.

The next problem was that of attaching the PCI bus. Previously, PCI buses could only have other PCI buses as parents, so a new API[8] was introduced to allow parenting a PCI bus to an arbitrary generic device. At the same time, others were working on bringing the EISA bus under the generic device umbrella [6].

With all these core and architecture specific changes, PA-RISC now has a complete generic device model layout of all of its I/O components and is ready for full conversion to the new DMA API.

4.3 Converting to the DMA API

Previously for PA-RISC, the U2/Uturn (IOMMU) information was cached in the PCI device `sysdata` field and was placed there at bus scan time. Since the bus scanning was done from the PA-RISC specific code, it knew which IOMMU the bus was connected to. Unfortunately, this scheme doesn't work for any other bus type, so an API[9] was introduced to obtain a fake PCI device for a given `parisc_device` and place the correct IOMMU in the `sysdata` field. PA-RISC actually has an architecture switch (see `struct hppa_dma_ops` in `asm-parisc/dma-mapping.h`) for the DMA functions. All the DMA functions really need to know is which IOMMU the device is connected to and the device's `dma_mask`, making conversion quite easy since the only PCI specific piece was extracting the IOMMU data.

Obviously, in the generic device model, the field `platform_data` is the one that we can use for caching the IOMMU information. Unfortunately there is no code in any of the scanned buses to allow this to be populated at scan time. The alternative scheme we implemented was to take the generic device passed into the DMA operations switch and, if `platform_data` was `NULL`, walk up the `parent` fields until the IOMMU was found, at which point it was cached in the `platform_data` field. Since this will now work for every device that has a generic device (which is now every device in the PA-RISC system), the fake PCI device scheme can be eliminated, and we have a fully implemented DMA API.

4.4 The Last Wrinkle—Non-Coherency

There are particular PA-RISC chips (the PCX-S and PCX-T) which are incapable of allocating any coherent memory at all. Fortunately, none of these chips was placed into a modern system,or indeed into a system with an IOMMU, so all the buses are directly connected.

Thus, an extra pair of functions was added to the DMA API switch for this platform

[8] `pci_scan_bus_parented()`

[9] `ccio_get_fake()`

which implemented noncoherent allocations as a `kmalloc()` followed by a map, and also for the `dma_cache_sync()` API. On the platforms that are able to allocate coherent memory, the noncoherent allocator is simply the coherent one, and the cache sync API is a nop.

5 Future Directions

The new DMA API has been successful on platforms that need to unify the DMA view of disparate buses. However, the API as designed is really driver writer direct to platform. There are buses (USB being a prime example) which would like to place hooks to intercept the DMA transactions for programming DMA bridging devices that are bus specific rather than platform specific. Work still needs doing to integrate this need into the current framework.

Acknowledgements

I would like to thank Grant Grundler and Matthew Wilcox from the PA-RISC linux porting team for their help and support transitioning PA-Linux to the generic device model and subsequently the new DMA API. I would also like to thank HP for their donation of a PA-RISC C360 machine and the many people who contributed to the email thread[7] that began all of this.

References

[1] David S. Miller, Richard Henderson, and Jakub Jelinek *Dynamic DMA Mapping* Linux Kernel 2.5 `Documentation/DMA-mapping.txt`

[2] James E.J. Bottomley *Dynamic DMA mapping using the generic device* Linux Kernel 2.5. `Documentation/DMA-API.txt`

[3] Jonathan Corbet *Driver Porting: DMA Changes* Linux Weekly News, `http://lwn.net/Articles/28092`

[4] The PA-RISC linux team `http://www.parisc-linux.org`

[5] Patrick Mochel *The (New) Linux Kernel Driver Model* Linux Kernel 2.5 `Documentation/driver-model/*.txt`

[6] Marc Zyngier *sysfs stuff for EISA bus* `http://marc.theaimsgroup.com/?t=103696564400002`

[7] James Bottomley *[RFC] generic device DMA implementation* `http://marc.theaimsgroup.com/?t=103902433500007`

Linux® Scalability for Large NUMA Systems

Ray Bryant and John Hawkes
Silicon Graphics, Inc.
raybry@sgi.com hawkes@sgi.com

Abstract

The SGI® Altix™ 3000 family of servers and superclusters are nonuniform memory access systems that support up to 64 Intel® Itanium® 2 processors and 512GB of main memory in a single Linux image. Altix is targeted to the high-performance computing (HPC) application domain. While this simplifies certain aspects of Linux scalability to such large processor counts, some unique problems have been overcome to reach the target of near-linear scalability of this system for HPC applications. In this paper we discuss the changes that were made to Linux® 2.4.19 during the porting process and the scalability issues that were encountered. In addition, we discuss our approach to scaling Linux to more than 64 processors and we describe the challenges that remain in that arena.

1 Introduction

Over the past three years, SGI has been working on a series of new high-performance computing systems based on its NUMAflex™ interconnection architecture. However, unlike the SGI® Origin® family of machines, which used a MIPS® processor and ran the SGI® IRIX® operating system, the new series of machines is based on the Intel® Itanium® Processor Family and runs an Itanium version of Linux.

In January 2003, SGI announced this series of machines, now known as the SGI Altix 3000 family of servers and superclusters. As announced, Altix supports up to 64 Intel Itanium 2 processors and 512GB of main memory in a single Linux image. The NUMAflex architecture actually supports up to 512 Itanium 2 processors in a single coherency domain; systems larger than 64 processors comprise multiple single-system image Linux systems (each with 64 processors or less) coupled via NUMAflex into a "supercluster." The resulting system can be programmed using a message-passing model; however, interactions between nodes of the supercluster occur at shared memory access times and latencies.

In this paper, we provide a brief overview of the Altix hardware and discuss the Linux changes that were necessary to support this system and to achieve good (near-linear) scalability for high-performance computing (HPC) applications on this system. We also discuss changes that improved scalability for more general workloads, including changes for high-performance I/O. Plans for submitting these changes to the Linux community for incorporation in standard Linux kernels will be discussed. While single-system-image Linux systems larger than 64 processors are not a configuration shipped by SGI, we have experimented with such systems inside SGI, and we will discuss the kernel changes necessary to port Linux to such large systems. Finally, we present benchmark results demonstrating the results of these changes.

2 The SGI Altix Hardware

An Altix system consists of a configurable number of rack-mounted units, each of which SGI refers to as a brick. Depending on configuration, a system may contain one or more of the following brick types: a compute brick (C-brick), a memory brick (M-brick), a router brick (R-brick), or an I/O brick (P-brick or PX-brick).

The basic building block of the Altix system is the C-brick (see Figure 1). A fully configured C-brick consists of two separate dual-processor systems, each of which is a bus-connected multiprocessor or node. The bus (referred to here as a Front Side Bus or FSB) connects the processors and the SHUB chip. Since HPC applications are often memory-bandwidth bound, SGI chose to package only two processors per FSB in order to keep FSB bandwitdh from being a bottleneck in the system.

The SHUB is a proprietary ASIC that implements the following functions:

- It acts a memory controller for the local memory on the node
- It provides an interface to the interconnection network
- It provides an interface to the I/O subsystem
- It manages the global cache coherency protocol
- It supports global TLB shoot-down and inter-processor interrupts (IPIs)
- It provides a globally synchronized high-resolution clock

The memory modules of the C-brick consist of standard PC2100 or PC2700 DIMMS. With

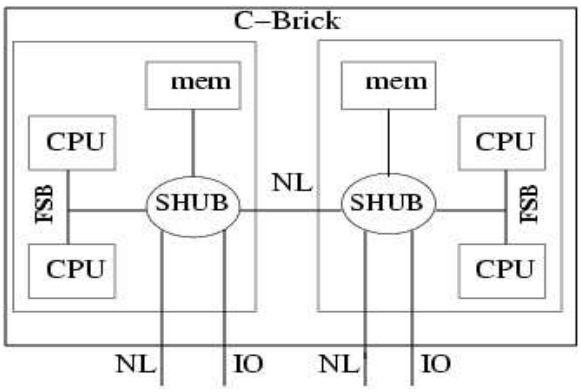


Figure 1: Altix C-Brick

1GB DIMMS, up to 16GB of memory can be installed on a node. For those applications requiring even more memory, an M-brick (a C-brick without processors) can be used.

Memory accesses in an Altix system are either local (i.e., the reference is to memory in the same node as the processor) or remote. Local memory references have lower latency; the Altix system is thus a NUMA (nonuniform memory access) system. The ratio of remote to local memory access times on an Altix system varies from 1.9 to 3.5 depending on the size of the system and the relative locations of the processor and memory module involved in the transfer.

As shown in Figure 1, each SHUB chip provides three external interfaces: two NUMAlink™ interfaces (labeled NL in the figure) to other nodes or the network and an I/O interface to the I/O bricks in the system. The SHUB chip uses the NUMAlink interfaces to send remote memory references to the appropriate node in the system. Depending on configuration the NUMAlink interfaces provide up to 3.2GB/sec of bandwidth in each direction.

I/O bricks implement the I/O interface in the Altix system. (These can be either an IX-brick or a PX-brick. Here we will use the generic term I/O brick.) The bandwidth of the I/O

channel is 1.2GB/sec in each direction. Each I/O brick can contain a number of standard PCI cards; depending on configuration a small number of devices may be installed directly in the I/O brick as well.

Shared memory references to the I/O brick can be generated on any processor in the system. These memory references are forwarded across the network to the appropriate node's SHUB chip and then they are routed to the appropriate I/O brick. This is done in such a way that standard Linux device drivers will work against the PCI cards in an I/O brick, and these device drivers can run on any node in the system.

The cache-coherency policy in the Altix system can be divided into two levels: local and global. The local cache-coherency protocol is defined by the processors on the FSB and is used to maintain cache-coherency between the Itanium processors on the FSB. The global cache-coherency protocol is implemented by the SHUB chip. The global protocol is directory-based and is a refinement of the protocol originally developed for DASH [11] (Some of these refinements are discussed in [10]).

The Altix system interconnection network uses routing bricks (R-bricks) to provide connectivity in system sizes larger than 16 processors. (Smaller systems can be built without routers by directly connecting the NUMAlink channels in a ring configuration.) For example, a 64-processor system is connected as shown in Figure 2.

One measure of the bandwidth capacity of the interconnection network is the bisection bandwidth. This bandwidth is defined as follows: draw an imaginary line through the center of the system. Suppose that each processor on one side of this line is referencing memory on a corresponding node on the other side of this line. The bisection bandwidth is the total amount of data that can be transferred across this line with all processors driven as hard as possible. In the Altix system, the bisection bandwidth of the system is at least 400MB/sec/processor for all system sizes.

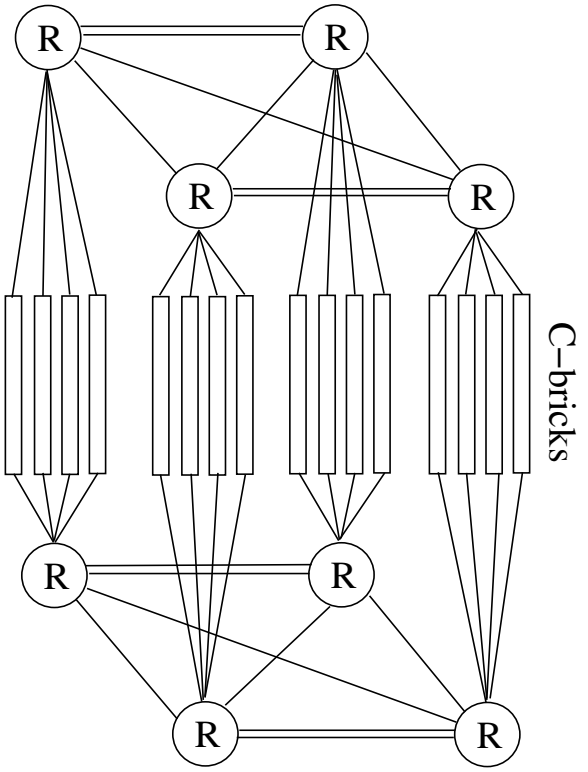


Figure 2: 64-CPU Altix System (R's represent router bricks)

3 Benchmarks

Three kinds of benchmarks have been used in studying Altix performance and scalability:

- HPC benchmarks and applications
- AIM7
- Other open-source benchmarks

HPC benchmarks, such as STREAM [12], SPEC® CPU2000, or SPECrate® [22] are simple, usermode, memory-intensive benchmarks

to verify that the hardware architectural design goals were achieved in terms of memory and I/O bandwidths and latencies. Such benchmarks typically execute one thread per CPU with each thread being autonomous and independent of the other threads so as to avoid interprocess communication bottlenecks. These benchmarks typically do not spend a significant amount of time using kernel services.

Nonetheless, such benchmarks do test the virtual memory subsystem of the Linux kernel. For example, virtual storage for dynamically allocated arrays in Fortran is allocated via *mmap*. When the arrays are touched during initialization, the page fault handler is invoked and zero-filled pages are allocated for the application. Poor scalability in the page-fault handling code will result in a startup bottleneck for these applications.

Once these benchmark tests had been passed, we then ran similar tests for specific HPC applications. These applications were selected based on a mixture of input from SGI marketing and an assessment of how difficult it would be to get the application working in a benchmark environment. Some of these benchmark results are presented in section 7 on page 85.

The AIM Benchmark Suite VII (AIM7) is a benchmark that simulates a more general workload than that of a single HPC application. AIM7 is a C-language program that forks multiple processes (called tasks), each of which concurrently executes similar, randomly ordered set of 53 different kinds of subtests (called jobs). Each subtest exercises a particular facet of system functionality, such as disk-file operations, process creation, user virtual memory operations, pipe I/O, or compute-bound arithmetic loops.

As the number of AIM7 tasks increases, aggregated throughput increases to a peak value. Thereafter, additional tasks produce increasing contention for kernel services, they encounter increasing bottlenecks, and the throughput declines. AIM7 has been an important benchmark to improve Altix scalability under general workloads that consist of a mixture of throughput-oriented runs or of programs that are I/O bound.

Other open-source benchmarks have also been employed. *pgmeter* [6] is a general file system I/O benchmark that we have used to evaluate file system performance[5]. *Kernbench* is a parallel make of the Linux kernel, (e.g., */usr/bin/time make -j 64 vmlinux*). *Hackbench* is a set of communicating threads that stresses the CPU scheduler. Erich Focht's *randupdt* stresses the scheduler's load-balancing algorithms and ability to keep threads executing near their local memory and has been used to help tune the NUMA scheduler for Altix.

4 Measurement and Analysis Tools

A number of measurement tools and benchmarks have been used in our optimization of Linux kernel performance for Altix 3000. Among these are:

- Lockmeter [4, 17], a tool for measuring spinlock contention
- Kernprof [18], a kernel profiling tool that supports hierarchical profiling
- VTune™ [20], a profiling tool available from Intel.
- pfmon [13, 15], a tool written by Stephane Eranian that uses the *perfmon* system calls of the Linux kernel for Itanium to interface with the Itanium processor performance measurement unit

5 Scaling and the Linux Kernel on Altix

"Perfect scaling"—a linear one-to-one relationship between CPU count and throughput for all CPU counts—is rarely achieved because one or more bottlenecks (software or hardware) introduce serial constraints into the otherwise independently parallel CPU execution streams. The common Linux bottlenecks - lock contention and cache-line contention - are true for uniform-memory-access multiprocessor systems as well as for NUMA multiprocessor systems and Altix 3000. However, the performance impact of these bottlenecks on Altix can be exaggerated (potentially in a nonlinear way) by the high processor counts and the directory-based global cache-coherency policy.

Analysis of lock contention has therefore been a key part of improving scalability of the Linux kernel for Altix. We have found and removed a number of different lock-contention bottlenecks whose impacts are significantly worse on Altix than they are on smaller, non-NUMA platforms. Specific examples of these changes are discussed below, in sections 5.2 through 5.8.

Cache-line contention is usually more subtle than lock contention, but cache-line contention can still be a significant impediment to scaling large configurations. These effects can be broadly classified as either "false cache-line sharing" or cache-line "ping-ponging." "False cache-line sharing" is the unintended co-residency of unrelated variables in the same cache-line. Cache-line "ping-ponging" is the change in exclusive ownership of a cache-line as different CPUs write to it.

An example of "false cache-line sharing" is when a single L3 cache-line contains a location that is frequently written and another location that is only being read. In this case, a read will commonly trigger an expensive cache-coherency operation to demote the cache-line from exclusive to shared state in the directory, when all that would otherwise be necessary would be adding the processor to the list of sharing nodes in the cache-line directory entry. Once identified, "false cache-line sharing" can often be remedied by isolating a frequently dirtied variable into its own cache-line.

The performance measurement unit of the Itanium 2 processor includes events that allow one to sample cache misses and record precisely the data and instruction addresses associated with these sampled misses. We have used tools based on these events to find and remove false sharing in user applications, and we plan to repeat such experiments to find and remove false sharing in the Linux kernel.

Some cache-line contention and ping-ponging can be difficult to avoid. For example, a multiple-reader single-writer *rwlock_t* contains an contending variable: the read-lock count. Each *read_lock()* and *read_unlock()* request changes the count and dirties the cache-line containing the lock. Thus, an actively used *rwlock_t* is continually being ping-ponged from CPU to CPU in the system.

5.1 Linux changes for Altix

To get from a `www.kernel.org` Linux kernel to a Linux kernel for Altix, we apply the following open-source community patches:

- The IA-64 patch maintained by David Mossberger [14]
- The "discontiguous memory" patch originally developed as part of the Atlas project [1]
- The O(1) scheduler patch from Erich Focht [7]

- The LSE rollup patch for the Big Kernel Lock [19]

This open-source base is then modified to support the Altix platform-specific addressing model and I/O implementation. This set of patches and the changes specific to Altix comprise the largest set of differences between a standard Linux kernel and a Linux kernel for Altix.

Additional discretionary enhancements improve the user's access to the full power of the hardware architecture. One such enhancement is the CpuMemSets package of library and kernel changes that permit applications to control process placement and memory allocation. Because the Altix system is a NUMA machine, applications can obtain improved performance through use of local memory. Typically, this is achieved by pinning the process to a CPU. Storage allocated by that process (for example, to satisfy page faults) will then be allocated in local memory, using a first-touch storage allocation policy. By making optimal use of local memory, applications with large CPU counts can be made to scale without swamping the communications bandwidth of the Altix interconnection network.

Another enhancement is XSCSI, a new SCSI midlayer that provides higher throughput for the large Altix I/O configurations. Figure 3 shows a comparison of I/O bandwidths achieved at the device-driver level for SCSI and XSCSI on a prototype Altix system. XSCSI was a tactical addition to the Linux kernel for Altix in order to dramatically improve I/O bandwith performance while still meeting product-development deadlines.

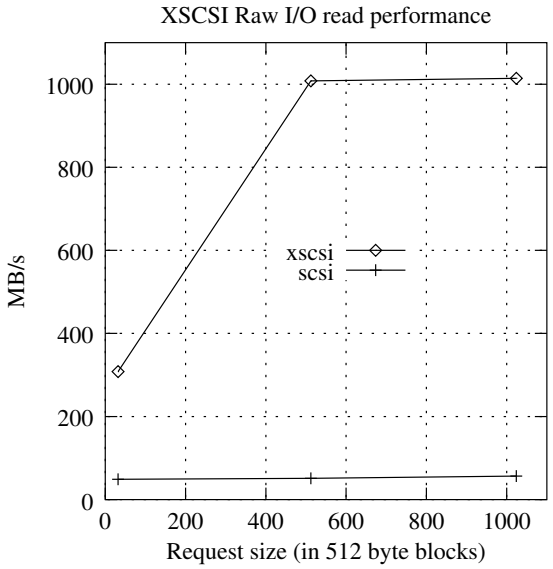


Figure 3: Comparison of SCSI and XSCSI on Prototype Altix Hardware

5.2 Big Kernel Lock

The classic Linux multiprocessor bottleneck has been the *kernel_flag*, commonly known as the Big Kernel Lock (BKL). Early Linux MP implementations used the BKL as the primary synchronization and serialization lock. While this may not have been a significant problem for workloads on a 2-CPU system, a single gross-granularity spinlock is a principal scaling bottleneck for larger CPU counts.

For example, on a prototype Altix platform with 28 CPUs running a relatively unimproved 2.4.17 kernel and with Ext2 file systems, we found that with an AIM7 workload, 30% of the CPU cycles were consumed by waiting on the BKL, and another 25% waiting on the *runqueue_lock*. When we introduced a more efficient multiqueue scheduler that eliminated contention on the *runqueue_lock*, we discovered that the *runqueue_lock* contention simply became increased contention pressure on the BKL. This resulted in 70% of the system CPU cycles being spent waiting on the BKL, up from 30%, and a 30% drop in AIM7 peak throughput performance. The lesson learned here is that we should attack the biggest bottlenecks first, not the lesser bottlenecks.

We have attempted to solve the BKL problem using several techniques. One approach has been in our preferential use of the XFS® file system, vs. Ext2 or Ext3, as the Altix file system. XFS uses scalable, fine-grained locking and largely avoids the use of the BKL altogether. Figure 4 shows a comparison of the relative scalability of several different Linux file systems under the AIM7 workload [5].

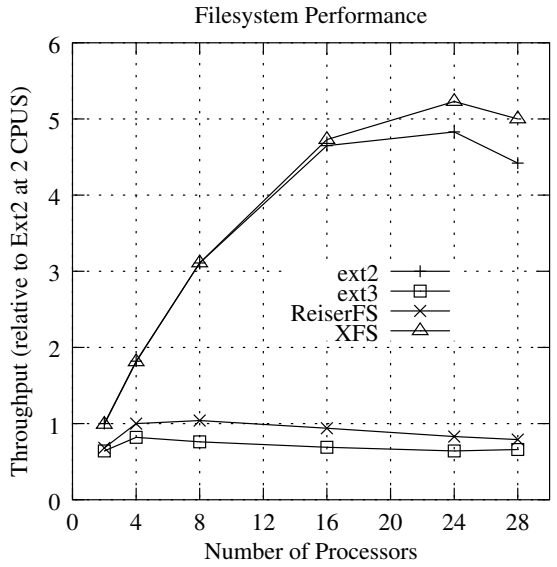


Figure 4: File System Scalability of Linux® 2.4.17 on Prototype Altix Hardware

Other changes were targeted to specific functionality, such as back porting the 2.5 algorithm for process accounting that uses a new spinlock instead of using the BKL.

The largest set of changes was derived from the LSE rollup patch [19]. The aggregate effect of these changes has reduced the fraction of cycles spent spinning (out of all CPU cycles) for the BKL lock from 50% to 5% when running the AIM7 benchmark, with 64 CPUs and XFS filesystems on 64 disks.

5.3 TLB Flush

For the Altix platforms, a global TLB flush first performs a local TLB flush using the Intel *ptc.ga* instruction. Then it writes TLB flush address and size information to special SHUB registers that trigger the remote hardware to perform a local TLB flush of the CPUs in that node. Several changes have been incorporated into the Linux kernel for Altix to reduce the impact of TLB flushing. First, care is taken to minimize the frequency of flushes by eliminating NULL entries in the *mmu_gathers* array and by taking advantage of the larger Itanium 2 page size that allows flushing of larger address ranges. Another reduction was accomplished by having the CPU scheduler remember the node residency history of each thread; this allows the TLB flush routine to flush only those remote nodes that have executed the thread.

5.4 Multiqueue CPU Scheduler

The CPU scheduler in the 2.4 (and earlier) kernels is simple and efficient for uniprocessor and small MP platforms, but it is inefficient for large CPU counts and large thread counts [3]. One scaling bottleneck is due to the use of a single global *runqueue_lock* spinlock to protect the global runqueue, thereby making it heavily contended.

A second inefficiency in the 2.4 scheduler is contention on cache-lines that are involved with managing the global runqueue list. The global list was frequently searched, and process priorities were continually being recomputed. The longer the runqueue list, the more CPU cycles were wasted.

Both of these problems are addressed by new Linux schedulers that partition the single global runqueue of the standard scheduler into multiple runqueues. Beginning with the Linux® 2.4.16 for Altix version, we began to

use the Multiqueue Scheduler contributed to the Linux Scalability Effort (LSE) by Hubertus Franke, Mike Kravetz, and others at IBM [8]. This scheduler was replaced by the O(1) scheduler contributed to the 2.5 kernel by Ingo Molnar [16] and subsequently modified by Erich Focht of NEC and others. The Linux kernel for Altix now uses a version of the 2.5 scheduler with some "NUMA-aware" enhancements and other changes that provide additional performance improvements for typical Altix workloads.

What is relatively peculiar to Altix workloads is the commonplace utilization of the task *cpus_allowed* bitmap to constrain CPU residency. This is typically done to pin processes to specific CPUs or nodes in order to efficiently use local storage on a particular node. The result is that the Altix scheduler commonly encounters processes that cannot be moved to other CPUs for load-balancing purposes. The 2.5 version of the O(1) scheduler's *load_balance()* routine searches only the "busiest" CPU's runqueue to find a candidate process to migrate. If, however, the "busiest" CPU's runqueue is populated with processes that cannot be migrated, the 2.5 version of the O(1) scheduler does not then search other CPUs' runqueues to find additional migration candidates. What is done in the Linux kernel for Altix is that *load_balance()* builds a list of "busier" CPUs and searches all of them (in decreasing order of imbalance) to find candidate processes for migration.

A more subtle scaling problem occurs when multiple CPUs contend inside *load_balance()* on a busy CPU's runqueue lock. This is most often the case when idle CPUs are load-balancing. While it is true that lock contention among idle CPUs is often benign, the problem is that that "busiest" CPU's runqueue lock has now become highly contended, and that stalls any attempt to *try_to_wake_up()* a process on that queue or on the runqueue of any of the idle CPUs that are spinning on another runqueue lock. Therefore, it is beneficial to stagger the idle CPUs' calls to *load_balance()* to minimize this lock contention. We continue to experiment with additional techniques to reduce this contention.

At the time this paper was being written, we have begun to use an adaptation of the O(1) scheduler from Linux® 2.5.68 in the Linux kernel for Altix. To this version of the scheduler, we have added a set of "NUMA-aware" enhancements that were contributed by various developers and that have been aggregated into a roll-up patch by Martin Bligh [2]. Early performance tests show promising results. AIM7 aggregate CPU time is roughly 6% lower at 2500 tasks than without these "NUMA-aware" improvements. We will continue to track further changes in this area and to contribute SGI changes back to the community.

5.5 dcache_lock

In the 2.4.17 and earlier kernels, the *dcache_lock* was a modestly busy spinlock for AIM7-like workloads, typically consuming 3% to 5% of CPU cycles for a 32-CPU configuration. The 2.4.18 kernel, however, began to use this lock in *dnotify_parent()*, and the compounding effect of that additional usage made this lock a major CPU cycle consumer. We have solved this problem in the Linux kernel for Altix by back porting the 2.5 kernel's finer-grained *dparent_lock* strategy. This has returned the contention on the *dcache_lock* to acceptable levels.

5.6 lru_list_lock

One bottleneck in the VM system we have encountered is contention for the *lru_list_lock*. This bottleneck cannot be completely eliminated without a major rewrite of the Linux

2.4.19 VM system. The Linux kernel for Altix contains a minor, albeit measurably effective, optimization for this bottleneck in *fsync_buffers_list()*. Instead of releasing the *lru_list_list* and immediately calling *osync_buffers_list()*, which reacquires it, *fsync_buffers_list()* keeps holding the lock and instead calls a new *__osync_buffers_list()*, which expects the lock to be held on entry. For a highly contended spinlock, it is often better to double the lock's hold-time than to release the lock and have to contend for ownership a second time. This particular change produced 2% to 3% improvement in AIM7 peak throughput.

5.7 xtime_lock

The *xtime_lock* read-write spinlock is a severe scaling bottleneck in the 2.4 kernel. A mere handful of concurrent user programs calling *gettimeofday()* can keep the spinlock's read-count perpetually nonzero, thereby starving the timer interrupt routine's attempts to acquire the lock in write mode and update the timer value. We eliminated this bottleneck by incorporating an open-source patch that converts the *xtime_lock* to a lockless-read using *frlock_t* functionality (which is equivalent to *seqlock_t* in the 2.5 kernel).

5.8 Node-Local Data Structures

We have reduced memory latencies to various per-CPU and per-node data structures by allocating them in node-local storage and by employing strided allocation to improve cache efficiency. Structures where this technique is used include *struct cpuinfo_ia64*, *mmu_gathers*, and *struct runqueue*.

6 Beyond 64 Processors

The Altix platform hardware architecture supports a cache-coherent domain of 512 CPUs. Although SGI currently supports a maximum of 64 processors in a single Linux image, we believe there is potential interest in SSI Linux systems that are larger than 64 CPUs (judging from our experience with IRIX systems, where some customers run 512-CPU SSI systems). Additionally, testing on systems that are larger than 64 CPUs can help us find scalability problems that are present, but not yet obvious in smaller systems.

The Linux kernel currently defines a CPU bit mask as an *unsigned long*, which for an Itanium architecture provides enough bits to specify 64 CPUs. To support a CPU count that exceeds the bit count of an *unsigned long* requires that we define a *cpumask_t* type, declared as *unsigned long[N]*, where *N* is large enough to provide sufficient bits to denote each CPU. While this is a simple kernel coding change, the change affects numerous files. Moreover, it also affects some calling sequences which today expect to pass a *cpumask_t* as a call-by-value argument or as a simple function return value. Most problematic is when *cpumask_t* is involved in a user-kernel interface, such as we have with the SGI CpuMemSets functions like *runon* and *dplace*. Our plan is to follow the approach of the 2.5 kernel in this area (c.f., *sched_set/get_affinity*).

Our initial 128-CPU investigations have so far not yielded any great surprises. The Altix hardware architecture scales memory access bandwidths to these larger configurations, and the longer memory access latencies are small (65 nanoseconds per router "hop" in the current systems) and well understood.

The large NR_CPU configurations benefit from anti-aliasing the per-node data and from dynamically allocating the *struct cpuinfo_ia64* and *mmu_gathers*. Dynamic allocation of the *runqueue_t* elements reduces the static size of the kernel, which otherwise produces a "gp

overflow" at link time. Inter-processor interrupts and remote SHUB references are made in physical addressing mode, thereby avoiding the use of TLB entries and reducing TLB pressure. Finally, several calls to *__cli()* were eliminated or converted into locks.

7 HPC Application Benchmark Results

In this section, we present some benchmark results for example applications in the HPC market segment.

7.1 STREAM

The STREAM Benchmark [12] is a "simple synthetic benchmark program that measures sustainable memory bandwidth (in MB/sec) and the corresponding computation rate for simple vector kernels" [12]. Since many HPC applications are memory bound, higher numbers for this benchmark indicate potentially higher performance on HPC applications in general. The STREAM Benchmark has the following characteristics:

- It consists of simple loops
- It is embarrassingly parallel
- It is easy for compiler to generate scalable code for this benchmark
- In general, only simple optimization levels are allowed

Here we report on what is called the "triad" portion of the benchmark. The parallel Fortran code for this kernel is shown below:

```
!$OMP PARALLEL DO
    DO j = 1, n
       a(j) = b(j) + s*c(j)
    CONTINUE
```

To execute this code in parallel on an Altix system, the data is evenly divided among the nodes, and each processor is assigned to do the calculations on the portion of the data on its node. Threads are pinned to processors so that no thread migration occurs during the benchmark. The scalability results for this benchmark on Altix 3000 are as shown in figure 5. As can be seen from this graph,

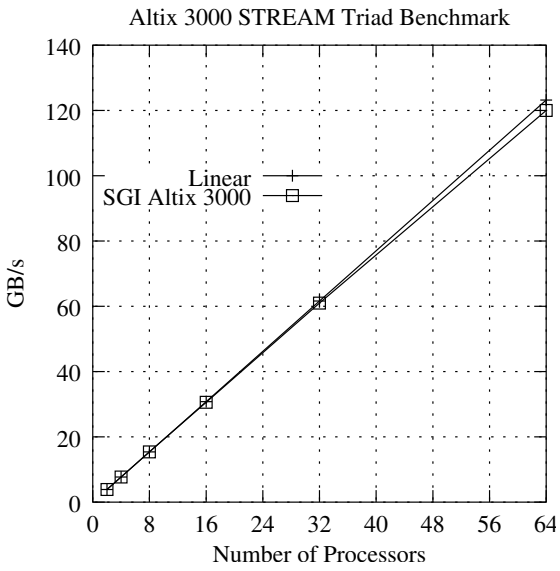


Figure 5: Scalability of STREAM TRIAD benchmark on Altix 3000

the results scale almost linearly with processor count. In some sense, this is not surprising, given the NUMA architecture of the Altix system, since each processor is accessing data in local memory. One can argue that any non-shared memory cluster with a comparable processor could achieve a similar result. However, what is important to realize here is that not all multiprocessor architectures can achieve such a result. A typical uniform-memory-access multiprocessor system, for example, could not achieve such linear scalability because the interconnection network would become a bottleneck. Similarly, while it is true that a non-

shared memory cluster can achieve good scalability, it does so only if one excludes the data distribution and collection phases of the program from the test. These times are trivial on an Altix system due to its shared memory architecture.

We have also run the STREAM benchmark without thread pinning, both with the standard 2.4.18 scheduler and the O(1) scheduler. The O(1) scheduler produces a throughput result that is nearly six times better than the standard Linux scheduler. This demonstrates that the standard Linux scheduler causes dramatically more thread migration and cache damage than the O(1) scheduler.

7.2 Star-CD™

Star-CD [21] is a fluid-flow analysis system marketed by Computational Dynamics, Ltd. It is an example of a computational fluid dynamics code. This particular code uses a message-passing interface on top of the shared-memory Altix hardware. This example uses an "A" Class Model, with 6 million cells. The results of the benchmark are shown in Figure 6.

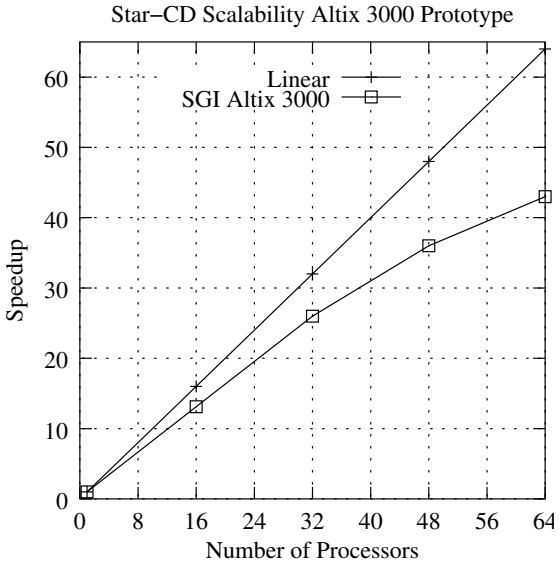


Figure 6: Scalability of Star-CD Benchmark

7.3 Gaussian® 98

Gaussian [9] is a product of Gaussian, Inc. Gaussian is a computational chemistry system that calculates molecular properties based on fundamental quantum mechanics principles. The problem being solved in this case is a sample problem from the Gaussian quality assurance test suite. This code uses a shared memory programming model. The results of the benchmark are shown in Figure 7.

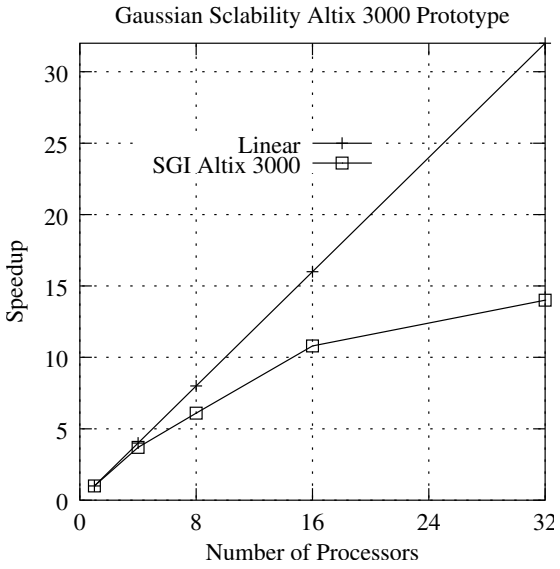


Figure 7: Scalability of Gaussian Benchmark

8 Concluding Remarks

With the open-source changes discussed in this paper, SGI has found Linux to be a good fit

for HPC applications. In particular, changes in the Linux scheduler, use of the XFS file system, use of the XSCSI implementation, and numerous small scalability fixes have significantly improved scaling of Linux for the Altix platform. The combination of a relatively low remote-to-local memory access time ratio and the high bandwidth provided by the Altix hardware are also key reasons that we have been able to achieve good scalability using Linux on the Altix system.

Relatively speaking, the total set of changes in Linux for the Altix system is small, and most of the changes have been obtained from the open-source Linux community. SGI is committed to working with the Linux community to ensure that Linux performance and scalability continues to improve as a viable and competitive operating system for real-world environments. Many of the changes discussed in this paper have already been submitted to the open-source community for inclusion in community maintained software. Versions of this software are also available at `oss.sgi.com`.

We anticipate future scalability efforts in multiple directions. One is an ongoing reduction of lock contention, as best we can accomplish with the 2.4 source base. We have work in progress on CPU scheduler improvements, reduction of the *pagecache_lock* contention, and reductions in other I/O-related locks.

Another class of work will be analysis of cache and TLB activity inside the kernel, which will presumably generate patches to reduce overhead associated with poor cache and TLB usage.

Large-scale I/O is another area of ongoing focus, including improving aggregate bandwidth, increasing transaction rates, and spreading interrupt handlers across multiple CPUs.

References

[1] `sourceforge.net/projects/discontig`

[2] `www.kernel.org/pub/linux/kernel/people/mbligh/`

[3] Ray Bryant and Bill Hartner, Java technology, threads, and scheduling in Linux, *IBM Developerworks* `www-106.ibm.com/developerworks/library/j-java2/index.html`

[4] Ray Bryant and John Hawkes, Lockmeter: Highly-Informative Instrumentation for Spin Locks in the Linux Kernel, *Proceedings of the Fourth Annual Linux Showcase & Conference*, Atlanta, Ga. (2000), `oss.sgi.com/projects/lockmeter/als2000/als2000lock.html`

[5] Ray Bryant, Ruth Forester, and John Hawkes, Filesystem Performance and Scalability in Linux 2.4.17, *Proceedings of the Freenix Track of the 2002 Usenix Annual Technical Conference*, Montery, Ca., (June 2002).

[6] Ray Bryant, David Raddatz, and Roger Sunshine, PenguinoMeter: A new File-I/O Benchmark for Linux, *Proceedings of the 5th Annual Linux Showcase and Conference*, Oakland, Ca. (October 2001).

[7] `home.arcor.de/efocht/sched`

[8] M. Kravetz, H. Franke, S. Nagar, and R. Ravindran, Enhancing Linux Scheduler Scalability, *Proceedings of the Ottawa Linux Symposium, Ottawa, CA*, July 2001.

[9] Gaussian, Inc., Carnegie Office Park, Building 6, Suite 230, Carnegie, PA 15106 USA. `www.guassian.com`

[10] James Laudon and Daniel Lenoski, The SGI Origin: a ccNUMA Highly Scalable Server, *ACM SIGARCH Computer Architecture News*, Volume 25, Issue 2, (May 1997), pp. 241-251.

[11] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennesy, The DASH prototype: Logic overhead and performance, *IEEE Transacctions on Parallel and Distributed Systems*, 4(1):41-61, January 1993.

[12] John D. McCalpin, STREAM: Sustainable Memory Bandwidth in High Performance Computers, `www.cs.virginia.edu/stream`

[13] David Mosberger and Stephane Eranian, *IA-64 Linux Kernel, Design and Implementation*, Prentice-Hall (2002), ISBN 0-13-061014-3, pp. 405-406.

[14] `www.kernel.org/pub/linux/kernel/ports/ia64`

[15] `www.hpl.hp.com/research/linux/perfmon/`.

[16] `www.ussg.iu.edu/hypermail/linux/kernel/0201.0/0810.html`

[17] `oss.sgi.com/projects/lockmeter`

[18] `oss.sgi.com/projects/kernprof`

[19] `lse.sourceforge.net/lockhier/bkl_rollup.html`

[20] `www.intel.com/software/products/vtune`

[21] Star-CD, `www.cd-adapco.com`.

[22] `www.spec.org`

An Implementation of HIP for Linux

Miika Komu[*]
Miika.Komu@hiit.fi

*Mika Kousa**
Mika.Kousa@hiit.fi

Janne Lundberg[†]
Janne.Lundberg@hut.fi

Catharina Candolin [‡]
Catharina.Candolin@hut.fi

Abstract

One of the main problems with IP has been its lack of security. Although IPSec and DNSSec have provided some level of security to IP, the notion of a true identity for hosts is still missing. Typically, the IP address of the host has been used as the host identity, regardless of the fact that it is nothing more than routing information. The purpose of the Host Identity Payload/Protocol (HIP) architecture is to add a cryptographically based name space, the Host Identity, to the IP protocol. The Host Identity serves as the identity of the host, whereas the IP address is merely used for routing purposes. In this paper, we describe the HIP architecture further, and present our IPv6 based implementation of HIP for Linux.

1 Introduction

The lack of security has been one of the main problems with IP. Although IPSec [8] and DNSSec [14] have provided some level of security to IP, such as data origin authentication, confidentiality, integrity, and so forth, the notion of a true identity for hosts is still missing. The IP address has typically been used both to identify the host and to provide routing information. This has led to the misuse of IP addresses for identification purposes in many security schemes. To overcome the problems related to the current use of IP addresses, the Host Identity Payload/Protocol (HIP) architecture adds a cryptographically based name space, the Host Identity, to the IP protocol. Each host (or more specifically, its networking kernel or stack) is assigned at least one Host Identity, which can be either public or anonymous. The Host Identity can be used for authentication purposes to support trust between systems, enhance mobility and dynamic IP renumbering, aid in protocol translation/transition and reduce denial-of-service attacks. Furthermore, as all of the higher protocols are bound to the Host Identity instead of the IP address, the IP address can now be used solely for routing purposes.

In this paper, we describe the HIP architecture and present our IPv6 [6] based implementation of HIP for Linux. The rest of the paper is structured as follows: in Section 2, the HIP architecture and the Host Layer Protocol is described. Section 3 describes our implementation, and Section 4 concludes the paper.

[*]Helsinki Institute for Information Technology, P.O. Box 9800, FIN-02015 HUT, Finland

[†]Laboratory for Theoretical Computer Science, Helsinki University of Technology, P.O. Box 9205, FIN-02015 HUT, Finland

[‡]Laboratory for Theoretical Computer Science, Helsinki University of Technology, P.O. Box 5400, FIN-02015 HUT, Finland

2 The HIP architecture

There are two name spaces in use in the Internet today: IP addresses and domain names. IP addresses have been used both to identify the network interface of the host and the routing direction vector. The three main problems with the current name spaces are that dynamic readdressing cannot be directly managed, anonymity is not provided in a consistent and trustable manner, and authentication for systems and datagrams is not provided.

In [10][11][12], the HIP architecture is introduced. HIP introduces a new cryptographically based name space, the Host Identity (HI), and adds a Host Layer between the network and the transport layer in the IP stack.

The modification to the IP stack is depicted in Figure 1. In the current architecture, each process is identified by a process ID (PID). The process may establish transport layer connections to other hosts (or to the host itself), and the transport layer connection is then identified using the source and destination IP addresses as well as the source and destination ports. On the IP layer, the IP address is used as the endpoint identifier, and on the MAC layer, the hardware address is used. In HIP, the transport layer is modified so that the connections are identified using the source and destination HIs as well as the source and destination ports. HIP then provides a binding between the HIs and the IP addresses, e.g. using DNS [9].

The HI is typically a cryptographic public key, which serves as the endpoint identifier of the node. Each host will have at least one HI assigned to its networking kernel or stack. The HI can be either public or anonymous. Public HIs may be stored in directories, such as DNS, in order to allow the host to be contacted by other hosts. A host may have several HIs, and it may also generate temporary (anonymous)

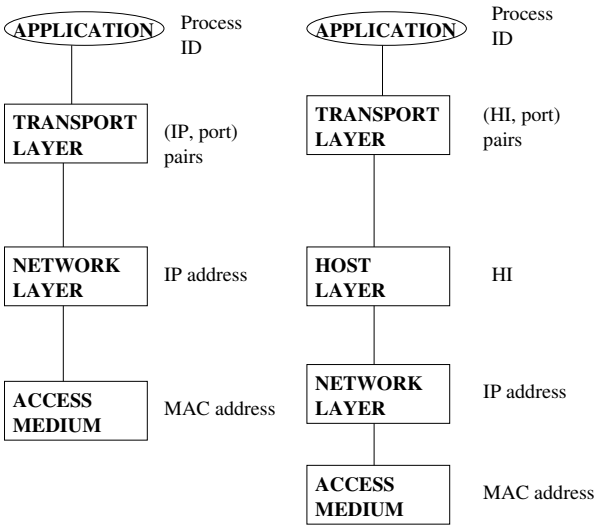


Figure 1: The current IP stack and the HIP based stack

HIs on the fly for establishing connections to other hosts. The main purpose of anonymous HIs is to provide privacy protection to the host, should the host not wish to use its public HI(s).

The HI is never directly used in any Internet protocol. It is stored in a repository, and is passed in HIP. Protocols use a 128-bit Host Identity Tag (HIT), which is a hash of the HI. Another representation of the HI is the Local Scope Identity (LSI), which has a size of 32 bits, but is local to the host. Its main purpose is to support backwards compatibility with the IPv4 API.

The main advantages of using HIT in protocols instead of the HI is that its fixed length makes protocol coding easier and also does not add as much overhead to the data packets as a public key would. It also presents a consistent format to the protocol regardless of the underlying identity technology used. HIT functions much like the SPI does in IPSec, but instead of being an arbitrary 32-bit value that identifies the Security Association for a datagram (together with the destination IP address and security protocol), HIT identifies the public key

that can validate the packet authentication.

The probability that a collision will occur is extremely small. However, should there be two public keys for one HIT, the HIT acts as a hint for the correct public key to use.

The HIP architecture basically solves the problems of dynamic readdressing, anonymity, and authentication. As the IP address no longer functions as an endpoint identifier, the problem of mobility becomes trivial, as the node may easily change its HI and IP address bindings as it moves. Anonymity is provided by temporary and anonymous HIs. Furthermore, as the name space is cryptographically based, it becomes possible to perform authentication based on the HIs. In [13], the concept of integrating security, mobility, and multi-homing based on HIP is discussed further.

2.1 The Host Layer Protocol

The Host Layer Protocol (HLP) is a signaling protocol between the communicating endpoints. The main purpose of the protocol is to perform mutual end-to-end authentication and to create IPSec ESP [7] Security Associations to be used for integrity protection and possibly also encryption. Furthermore, the protocol performs reachability verification using a simple challenge-response scheme.

The HLP protocol provides seven message types, of which four are dedicated to the base exchange. In Figure 2, the base exchange is depicted. In the first message, $I1$, the initiator I sends its own HIT and the HIT of the responder to the responder. The responder R replies with message $R1$, which contains the HITs of I and itself as well as a puzzle based challenge for I to solve. The purpose of the challenge is to make the protocol resistant to denial-of-service attacks. (Puzzle based schemes have been previously used for providing DoS protection to

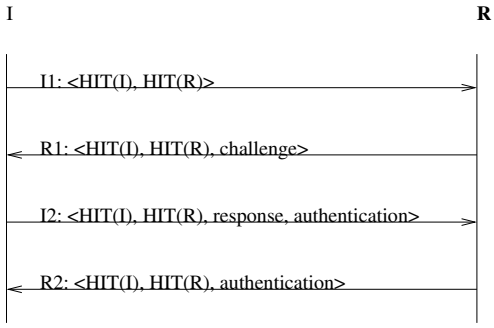


Figure 2: The base exchange of the Host Layer Protocol

both authentication [3] and encryption [5] protocols.) I solves the puzzle and sends in $I2$ the HITs of itself and R as well as the solution to the puzzle, and performs the authentication. $R2$ now commits itself to the communication, and responds with the HITs of I and itself, and performs the authentication. After this, I and R have performed the mutual authentication and established Security Associations for ESP, and can now engage in secure communications. Furthermore, reachability is verified by the fact that the protocol has more than two rounds.

If I does not have any prior information of R, it may retrieve the information from a repository, such as DNS. I sends a lookup query to the DNS server, which replies with R's address, HI, and HIT.

There are three other messages in the HLP. The HIP New SPI Packet (NES) provides the peer system with its new SPI, provides a new Diffie-Hellman key to produce new keying material, and provides any intermediate system with the mapping of the old SPI to the new. The HIP Readdress Packet (REA) allows a host to notify its partners of a change of the IP address (e.g. as a result of mobility). The HIP Bootstrap Packet (BOS) is used when the initiator is unable to learn a responders information from a repository.

3 Implementation

The protocol is implemented as a kernel module which uses a user space daemon process for some cryptographic operations, such as computation and verification of DSA signatures. Since the protocol is implemented as a kernel module, the kernel can remain as intact as possible, with only minor modifications. The modifications are backwards compatible so that normal TCP/IP connectivity without HIP can still be used. The state information required by the protocol state machine is located within the kernel module. The user space daemon acts as a slave to the kernel module and does not contain state.

The implementation is based on the Linux kernel version 2.4.18 with USAGI [2] patches. The implementation supports HIP only over IPv6.

3.1 Network Socket API

The most important design goal in the network socket API has been the transparent use of HIP by legacy applications. Thus, the legacy applications do not need any changes in their source code to utilize the benefits of HIP. On the other hand, applications that are HIP aware should be able to perform some additional tasks that will not be available to legacy applications. For example, a HIP aware application may require or deny the use of HIP. A reason to require HIP would be to benefit from the multihoming, security, and mobility features of HIP. A reason to deny the use of HIP might be to avoid the extra overhead caused by the cryptographic operations in a device with limited computing capacity.

A typical network application does not usually establish a network connection directly to an IPv6 address. Instead, the application is usually given the hostname of the peer, which has to be resolved to an IPv6 address from DNS. The connection can then be established to the IPv6 address.

When HIP is used, the network application needs additional support in the resolver for two different reasons. The first reason is that the resolver should return HITs instead of IPv6 addresses if HIP is being used transparently in a legacy application. The second reason is that a HIT to IPv6 address mapping should always be sent to the kernel as a side effect of the domain name query. Otherwise the IPv6 layer in the kernel does not have an address it can use for routing packets.

The resolver interfaces are traditionally contained in `libc`. The USAGI project has its own modified version of `libc` which is also used in the implementation. Only the `getaddrinfo` resolver interface is currently supported in the implementation for experimentation purposes.

Most of the legacy IPv6 applications, such as telnet clients and web browsers, are able to use HIP in transparent mode if they can access the HIP enabled resolver. This means that they should be relinked against the HIP patched USAGI `libc`. Firewalls, network address translators and other applications that handle raw packets may need changes in application code in order to utilize HIP.

3.2 Userspace daemon

A userspace daemon is required by the HIP module for several reasons, of which the most important reason is that the protocol requires DSA and Diffie-Hellman cryptographic algorithms which cannot easily be implemented within the kernel. Unfortunately, there are no known kernel cryptographic libraries supporting those algorithms, so those tasks have to be done in user space libraries. The HIPL im-

plementation uses the OpenSSL [1] library for user space cryptographic operations.

The daemon is used by the kernel module to perform many small operations on data. The kernel module can send queries such as *sign the given data with this DSA key* or *solve this cookie* to the daemon. The daemon calculates a response for the given query and the response either contains the answer to the query or an error message if something went wrong. Messages between the kernel module and the daemon are exchanged synchronously in a request-response fashion.

The request-response communication is implemented using a common interface in the kernel. When a request is carried out in the kernel, the current context of execution will be saved. The contents of the context depends on the operation being executed, but a minimal context defines at least a reference to a callback function. The callback function is called after the request has been served in the daemon and the daemon has sent a response message to the kernel module. The kernel module can restore its state based on the information stored in the context and then continue its execution where it left off.

The actual request-response communication is implemented in a straightforward manner. The implementation can serve only one daemon request at a time, and subsequent requests are saved into a FIFO queue. An arriving response message from the daemon triggers a new daemon request from the top of the FIFO queue.

3.3 Networking Stack

Transport layer communications are bound to HITs when HIP is used. When data is sent over the transport layer connections, packets are created and received as if they were using HITs as the source and destination addresses in the transport layer headers. As the packets are passed up and down the protocol stack, they will encounter a number of hooks that may intercept the passing packet to the HIP module for modifications. Currently, the implementation has three major entry points into the HIP module from the IPv6 stack.

IPv6 output functions. The hooks in the output functions are triggered after the packet has been built and the packet has passed IPsec ESP processing. If the packet belongs to a HIP connection, it has a HIT instead of an IPv6 address as the destination address in the IPv6 header. Such a packet will be intercepted by the HIP module. Other packets are allowed through intact.

When the HIP output functions receive a packet, the module first checks whether the packet belongs to an established HIP connection by searching its table of established connections using the source and destination HITs as the key. If an existing connection is not found, the packet is dropped and a HIP exchange is started by sending an I1 packet. On the other hand, if an existing connection is found, the source and destination HITs in the IPv6 header are replaced by the IPv6 addresses that are stored in the mapping table in the kernel. Thus, even if connections are maintained using HITs as identifiers in the transport layer, the actual packets that are sent to the network will still always contain valid IPv6 addresses.

IPv6 ESP input functions. All received packets that belong to an established HIP connection will have an ESP header. Therefore, it is only necessary to intercept HIP packets from within ESP. Packets that are received by ESP are classified to those that belong to a HIP connection and those that do not. The reverse of the output mapping is performed. The

correct mapping is located using the SPI field in the ESP header, and if a mapping is found, the source and destination addresses in the packet are replaced by the corresponding HITs before the packet is forwarded to the actual ESP processing. Again, the ESP processing only sees the HITs, and not the IPv6 addresses.

HIP protocol input. A special case is required for HIP packets that are received during the connection establishment phase. The HIP module is registered as a transport layer protocol and does not actually require a special hook for this functionality. This intercept point is used to receive I1, R1, I2 and R2 packets, as well as other HIP negotiation packets.

Also, a few other hooks are required in order to, for example, make the neighbor discovery in IPv6 work correctly with HIP. However, these hooks are not relevant for this discussion.

3.4 Collaboration of Components

This section gives an overview of the collaboration of the components through an example. Figure 3 represents an overview of the system architecture and the logical connections between the components.

For simplicity, only a minimalistic base exchange is demonstrated. For example, mobility and multihoming are not demonstrated here. The configuration in this example consists of two hosts which use legacy applications that have not been designed for HIP, and therefore the transparent mode is used. The host that starts the HIP exchange will be referred to as the initiator while the peer is known as the responder. The initiator is a host that wishes to browse a web page from another host, and the responder has a web server listening for incoming requests. A DNS server in the domain of

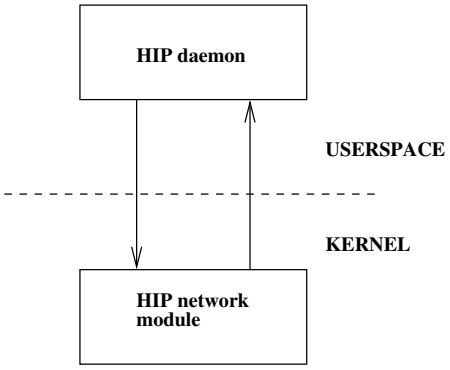


Figure 3: Collaboration of components

the responder is configured to return the IPv6 address and the HIT of the responder.

When the HIP module is loaded into the kernel, it first queries a Host Identity from the daemon. This identity will be used in all signed HIP packets that are sent by the host. The kernel module also generates a list of prebuilt R1 packets for quick sending. Finally, the module registers its hooks into the kernel. HIP connections can now be established.

When the user of the initiator host inputs the URL of the requested web page into the web browser to view the web pages in the responder's web server, the browser queries DNS for the name of the host using the resolver routine `getaddrinfo`. Since the browser is linked to the modified `libinet6`, the query is handled by the modified resolver.

The resolver queries the DNS for the responder's hostname. When the resolver receives the response from the DNS, it finds IPv6 addresses as well as HITs in the reply. Two things are done before the DNS reply is returned in a list from the resolver to the web browser. First, the resolver changes the order of the addresses in the DNS reply list before returning them to the application. The HITs of the responder are placed in the beginning of the list before the IPv6 addresses of the responder. Second, the

kernel is notified about the mapping of the HIT to the corresponding IPv6 address.

The result of the resolver call is a list containing first the HIT and then the IPv6 address of the responder. The browser is assumed to make the straightforward choice and select the first address from the list, which is a HIT in this case. The HIT is then used in the socket calls. Because the browser uses the HTTP [4] protocol which runs over TCP, the browser passes the HIT to the TCP `connect` call in the network socket API.

The `connect` call is handled by the TCP layer in the kernel. The TCP layer begins a handshake to establish a connection by generating a SYN packet to be delivered to the web server. When the SYN packet is encapsulated into an IPv6 packet in the IPv6 layer, the packet is captured by the output hook of the HIP module. The HIP module then examines the packet and discovers that the addresses in the packets are HITs instead of regular IPv6 addresses. The module also attempts to lookup a previously established HIP connection from its table of established HIP connections. The lookup fails because the connection attempt was a new one and a HIP exchange is needed to establish the new HIP connection. Since a HIP connection between the client and the server does not exist, the TCP SYN packet cannot immediately be delivered to the web server. For simplicity, the SYN packet will be dropped. Retransmissions will also be dropped until the base exchange has been completed.

To start the base exchange, the initiator sends an I1 packet to the web server. R1, I2 and R2 packets are exchanged after this. The building and parsing of each of the R1, I2, and R2 packets requires the assistance of the HIP daemon. For example, the daemon verifies the validity of the identity of the peer from DNS and creates a symmetric Diffie-Hellman key for the hosts during the base exchange.

Once the base exchange is completed, the hosts will have generated a common secret that they will be able to use to secure their communication. They will also have established IPsec Security Associations that will be used to encrypt the communication between the hosts. The TCP handshake can continue, and once it has been completed, the initiator can receive web pages from the web server at the responder. If further TCP connections need to be established between the two hosts, the HIP negotiation is not needed to be performed again, but the existing security associations are reused for the new connections.

4 Conclusion

In this paper, we described the HIP architecture, which has been designed to overcome problems mainly with respect to security, mobility, and privacy in the current Internet. HIP adds a new layer, the Host Layer, between the networking and transport layer in the IP stack, and introduces a Host Identity (HI) to serve as an end-point identifier of the host. Typically, the HI is represented by a public key. Each host will have at least one HI assigned to its networking kernel or stack. As the HI is used to identify the hosts, the IP addresses are used merely for routing purposes.

HIP defines a Host Layer Protocol to be used as a signaling protocol between end hosts. The purpose of the protocol is to perform mutual end-to-end authentication and to establish IPSec Security Associations. HLP consists of seven message types, of which four are part of the HIP base exchange.

As part of this paper, we presented our IPv6 based implementation of HIP for Linux. The Host Layer Protocol is implemented as a kernel module, which uses a user space daemon

process to perform some cryptographic operations. The advantage of our approach is that the kernel can remain as intact as possible, with only minor modifications. Furthermore, the modifications are backwards compatible so that the host is able to do networking without HIP. Our implementation is based on Linux kernel version 2.4.18 with USAGI patches.

Acknowledgments

This research is funded by the National Technology Agency of Finland (TEKES), Elisa Communications, Ericsson, Nokia, TeliaSonera, Creanor, and More Magic Software. We thank Jukka Ylitalo, Jorma Wall, Petri Jokela, and especially Pekka Nikander from Ericsson Research for fruitful cooperation and interoperability testing with their implementation of HIP for BSD. Furthermore, we are grateful for the valuable discussions we have had with Andrew McGregor and Thomas Henderson.

References

[1] Openssl: The open source toolkit for ssl/tls. `http://www.openssl.org/`.

[2] Usagi project. `http://www.linux-ipv6.org/`.

[3] T. Aura, P. Nikander, and J. Leiwo. Dos-resistant authentication with client puzzles,.

[4] T. Berners-Lee, R. Fielding, H. Frystyk, J. Gettys, and J. Mogul. Hypertext Transfer Protocol – HTTP/1.1. Technical report, Internet Engineering Task Force, January 1997. RFC 2068.

[5] Catharina Candolin, Janne Lundberg, and Pekka Nikander. Experimenting with early opportunistic key agreement. In *Proceedings of Workshop SEcurity of Communication on Internet, Internet Communication Security*, Tunis, Tunisia, September 2002.

[6] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. IETF Request for Comments 2460, December 1998.

[7] S. Kent and R. Atkinson. IP Encapsulating Security Payload (ESP). Request for Comments 2406, 1998.

[8] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. IETF Request for Comments 2401, 1998.

[9] P. Mockapetris. Domain Names — Implementation and Specification. IETF RFC 1035, 1987.

[10] R. Moskowitz. Host identity payload and protocol. Internet Draft draft-moskowitz-hip-05.txt, work in progress, 2001.

[11] R. Moskowitz. Host identity payload architecture. Internet Draft draft-ietf-moskowitz-hip-arch-02.txt, work in progress, 2001.

[12] R. Moskowitz. Host Identity Payload Implementation. Internet Draft draft-ietf-moskowitz-hip-impl-01.txt, work in progress, 2001.

[13] P. Nikander, J. Ylitalo, and J Wall. Integrating Security, Mobility, and Multi-homing in a HIP way. In *Proceedings of Network and Distributed Systems Security Symposium (NDSS'03)*, pages 87–99, San Diego, USA, February 2003.

[14] B. Wellington. Domain Name System Security (DNSSEC) Signing Authority.

IETF Request for Comments 3008,
November 2000.

Improving enterprise database performance on Intel Itanium® architecture

Ken Chen, Rohit Seth, Hubert Nueckel
Intel Corporation
Software and Solutions Group

Abstract

In this paper, we will present several operating system features that improved database performance under OLTP[1] workload significantly, such as Huge TLB[2] page to reduce DTLB[3] misses as database uses large amount of shared memory and asynchronous I/O to accommodate high amount of random I/O without introducing the overhead of many I/O processes. We will also present many other kernel optimizations that were developed by Intel, Red Hat and the Linux community that improved the scalability and performance of Linux kernel, specifically the areas are: raw vary I/O, kernel data structure footprint reduction, global io_request_lock reduction, and storage device driver optimization.

1 Introduction

Linux has been receiving a great deal of attention in the past few years. The popularity is propelled by wide range of adoption of Linux for enterprise computing. Major software vendors have been supporting their products on Linux for many years. As the enterprise software solution stack builds up everyday, it is crucial that Linux kernel development takes this opportunity to ensure that kernel provides key necessary infrastructure for enterprise application to excel. This means developing enterprise focused operating system (OS) features, improving performance by extending the scalability, and many other areas.

Relational database management systems (RDBMS) are complex server applications that solve the problems of information management. The RDBMS reliably manages large amount of data in a multi-user environment such that users can concurrently access shared data. While it is required to maintain consistent data between users, it is also required to deliver high performance. All these requirements need high-quality infrastructure provided by the operating system. Some of the examples are virtual memory management for managing vast amount of physical memory, scalable I/O subsystem, robust / high performance storage subsystem, light-weight inter-process communication, and robust / high performance networking subsystem.

Recent Linux kernel development has addressed many of the areas with a focus to provide key functionality for enterprise workloads. The rest of the paper will discuss new kernel features as well as performance enhancements in the context of database running OLTP workload.

[1] On-Line Transaction Processing
[2] Translation Lookaside Buffer
[3] Data TLB

2 Overview

On-line transaction processing refers to a class of applications that facilitates and manages transaction-oriented operation, typically for data entry and retrieval transactions in a number of industries. The basic skeleton of OLTP environment consists of multi-tier software applications that allow thousands of users to concurrently execute transactions against a database. Typically transactions are either executed on-line or queued for deferred execution and have certain characteristics on the distribution between a mixture of different types. Because of the complexity and overall execution behavior of OLTP workload, the workload characteristics can be summarized as:

- Simultaneous execution of multiple types of transactions that span a breadth of complexity
- On-line and deferred transaction execution
- Significant random disk input/output
- Transaction integrity
- Unique distribution of data access
- Contention on data access and update

From system architecture perspective, the OLTP workload exercises a breadth of system components associated with the environment. Database server application and the underlying operating system software are the key software components to provide high performance. Earlier evaluation of Linux kernel under OLTP workload revealed several hot spots or limitations from performance point of view, such as large execution time spent in low level TLB miss handling, large number of process context switch due to blocking synchronous I/O, large execution time on functions related to I/O elevator algorithm, and large execution time spent on spinning on a highly contended lock like global io_request_lock. In the following sections, we will examine how features like Huge TLB and asynchronous I/O allow database application to exploit maximum hardware capability with minimum overhead from Linux kernel and how Linux I/O subsystem is improved to reduce kernel execution time.

3 Huge TLB Support in Linux

3.1 Motivation of Huge TLB page

A TLB (Translation Lookaside Buffer) is a hardware structure for virtual-to-physical address translations that supports high performance paged virtual memory system. Typically it is a scarce resource on a processor. Operating systems always try to make best use of the limited number of available TLB entries on a system. Orthogonally, with advancement of semiconductor technology that resulting in ever growing memory capacity, it becomes more and more feasible both technically and economically to populate tens of gigabytes of memory on a server. For example, HP server rx5670 can be populated with 48 GB of memory with 1GB DIMM[4], or even 96 GB with latest 2GB DIMM.

Database server applications generally use large amounts of system memory in order to efficiently manage the actual databases that are usually much larger than system memory. It typically utilizes shared memory segments among multiple database processes. The first area in shared memory segments, usually the largest, is the database buffer cache. It holds copies of data blocks read from datafiles. A data block is the smallest unit of storage space

[4] dual in-line memory module

managed by database server. The RDBMS actively manages the data blocks in the buffer cache. When a user process requires a particular piece of data, it searches through the buffer cache. If the data is already in the cache (a cache hit), it will read the data directly from memory. Otherwise data block will be copied from datafile on disk into memory (a cache miss). It is well known that accessing data from memory is several orders of magnitude faster than accessing data from disk. Therefore in production environment, system administrator will typically allocate as much memory as possible for shared memory segments in order to improve cache hit rate by maximizing buffer cache size. However, accessing large amount of memory combined with random data access pattern of OLTP workload, it puts lots of pressure on CPU's TLB resource. For example, assuming 16K page size for Linux-IA64, a 48 GB of process memory would need 3 million TLB translations. Or to look at from hardware point of view, an Itanium 2 processor's internal TLB resource would only cover 2 MB of virtual address space with 16 KB page size.

With vast amount of memory each application process access, there is a need to make each TLB mapping as large as possible to reduce TLB pressure. Large contiguous regions in a process address space, such as contiguous data, may be mapped by using small number of large pages rather than large number of small pages. It is also important to note here that OS kernel cannot blindly pick up a larger page size for all applications because it may cause lots of fragmentation and very poor utilization of large amount of physical address space. Thus a requirement for having a separate large page size facility from the operating system becomes more and more important in terms of functionality and performance.

3.2 Design and Implementation

To support large page size for user application to utilize processor's capability, Intel worked with the Linux community to introduce a new OS feature that exposes the hardware architecture for application to benefit from using huge page size without affecting many other aspects of the OS. This new feature is called Huge TLB page. Specifically the Huge TLB support is attempting to solve the following problems:

- Increase CPU TLB coverage / Reduce data TLB miss rate
- Reduce process's page table memory requirement
- Pin data pages in physical memory

The design goal of Huge TLB interface is to expose the hardware architecture to application. Mapping the kernel, or specialized devices such as frame buffers by using large mapping is a relatively straightforward exercise. It only affects very limited portions of the operating system code. However, virtual memory implementation in Linux kernel makes the basic assumption that there is only one page size for user applications. This one size is related to MMU page size supported by a specific architecture. For example, on IA-32 this page size is 4K, and on Itanium-based system, user page size is configurable at kernel build time to be either 4K, 8K, 16K or 64K. Itanium 2 processor actually provides concurrent multiple page size support (4K, 8K, 16K, 64K, 256K, 1M, 4M, 16M, 256M, 1G and 4G). The current VM system is not suited for supporting multiple user page sizes because the knowledge of one page size is ingrained in several subsystems within the kernel. It is important to note that supporting multiple page sizes affects both architecture dependent and independent portions

of the Linux kernel. That is, a clean separation of architecture dependent and independent code in kernel is not enough to mitigate the difficulties of supporting multiple page sizes.

The allocation of Huge TLB page is performed in two phases. First a system administrator requests the kernel to reserve a set of memory in a special huge TLB page pool. The reservation of each huge TLB page is constrained that memory to be physically contiguous. Once huge TLB pages are reserved by the operating system, they can be used by application through two well defined system interfaces, either by mmap interface or through the standard System V shared memory interface. Note, application changes are required to use Huge TLB pages.

3.3 Application Benefit

To quantify the speed up of RDBMS under OLTP workload, we setup an experimental environment similar to industry standard OLTP benchmark on a Itanium 2 processor based platform.

First a baseline result is established with standard 16K page size. We then ran experiment with 256 MB page size while holding total memory in shared memroy segments constant. Throughput is then normalized to baseline. Figure 3.1 depicts the result.

We can easily see that with each incremental increase in page size used for data pages in shared memory segments, the speed up is notably at 11% overall for 256 MB huge TLB page size.

To further study how various page size speeds up the overall OLTP throughput at hardware micro-architecture level, we used Itanium-processor's hardware performance monitoring unit (PMU) to measure TLB pressure with various page size. The usage model of PMU

Figure 3.1: Relative OLTP throughput with various page size while holding database buffer cache size constant

are described in detail in several publications [1][2].

Again a baseline is established and data were collected for each page size. To measure hardware TLB pressure, we measured with metric of DTLB miss rate, or inverse of average number of data references per DTLB miss. As shown from figure 3.2, there is significant reduction in data TLB miss rate by using huge page size. For 256 MB page size, the DTLB miss rate is reduced by 65%, or inversely, the number of data references between successive TLB misses increases by 280%.

Figure 3.2: DTLB miss rate comparison

It is also interesting to observe that TLB pres-

sure for OLTP workload on Itanium 2 based system does not vary much with respect to total memory size, it is more or less a function of I/O load. For example, two experiments were conducted such that one with 16 GB database buffer cache while the other has 32 GB. The micro-architecture DTLB miss rate for both configurations are well within a couple of percentage points. This experiment points out that even at different OLTP throughput due to different size of database buffer cache, the benefit of using larger page size is equally significant. With 256 MB page size, the hardware TLB resource on Itanium 2 processor would be able to cover up to 32 GB of memory and primary source of TLB misses are shifted to data access to process's local data and task context switching.

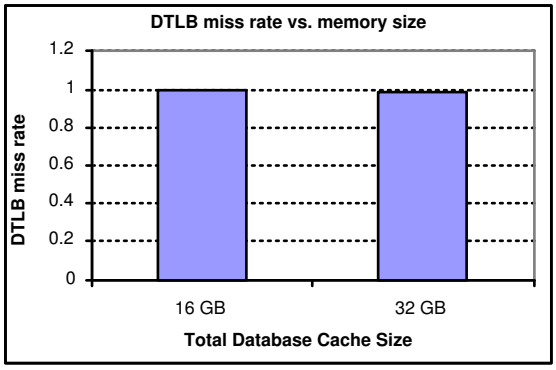


Figure 3.3: DTLB miss rate vs. memory size

A second benefit of using huge TLB feature is that the memory usage for process's page table is significantly reduced. Taking a 48 GB system as an example, if 45 GB is allocated as 180 256MB huge TLB pages, memory for page table covers that 45 GB of vma is only 1440 byte. For 100 processes that shares the 45 GB of shared memory segment, total memory for page table is 1.6 MB considering each process round up 1440 byte to one page. In the case of using normal 16 K page size, the memory requirement grows to 2250 MB (3 million entries * 8 bytes/entry * 100 processes, ignoring first and second level page table structure for simplicity). A secondary effect is that this 2.2 GB of memory reduced from page table can be better utilized by application to further increase application's performance.

A third benefit of huge TLB feature is that memory allocated for huge TLB page is pinned in physical memory and is not considered for swapping. This eliminates the chance of swapping physical pages that are being used for holding critical application data.

4 Linux I/O Subsystem

4.1 Dynamic vs. Static kiobuf allocation

Direct device access via raw devices partition improves database performance. A raw device partition is a contiguous region of a disk that can be accessed via a character device interface (/dev/raw on Linux). Such access typically bypasses the file system buffering. Since RDBMS does its own memory cache and I/O management, there is no need to have operating system to perform another level of caching and buffering. In fact, it is better to leave that task to application because it has much better information to determine optimal I/O strategy.

In a large OLTP workload configuration, due to sheer number of disk drives and the need to spread I/O load onto large number of disk drives, a database server typically opens large amount of data files where these data files reside on raw devices. Independent processes within the database server application will each open same set of data files.

The existing raw I/O code will statically allocate one kiobuf and its associated structures (mainly buffer_head structure, abbreviated as bh hereafter) upon every raw device open. There are 1024 bh allocated for each kiobuf. In a benchmark configuration, the memory re-

quirement just for the bh structure is calculated as following:

150 raw devices * 120 db processes * 1024 bh * 192 byte/bh = 3534 MB

However, since each process can have only one outstanding synchronous I/O at any given time, the active memory required for 120 processes are:

120 db processes * 1024 bh * 192 byte/bh = 24 MB

There are massive amount of memory being set aside by the bh structure and only 0.67% of them are being actively used. This large amount of under-utilized memory can be better devoted for other part of the system, for example, database buffer cache.

The cause of the issue is that each kiobuf structure is associated with a file descriptor. Although in certain cases, static bh allocation avoids the overhead of dynamic allocation, this static allocation scheme actually hurts performance for OLTP workload due to displacement of memory allocated for bh structure but otherwise can be used for database buffer cache.

To enable large number of raw devices to be opened simultaneously, we removed the static kiobuf allocation in raw_open function and at each invocation of rw_raw_dev function, kiobuf is dynamically allocated and freed for each raw I/O request. In order to reduce the prohibitive amount of overhead with dynamic allocation of all the memory arrays in kiobuf, we treat kiobuf and its associated member arrays as one entity. With the aid of constructor and destructor API provided by the kernel slab allocator, member arrays of kiobuf are allocated and initialized upon the creation of kiobuf object. Subsequent dynamic allocation would only incur one level of kmem_cache_alloc and kmem_cache_free overhead for such large data structure. With the per-CPU slab allocation area, the cost of dynamic allocation is even more affordable. The overhead of this dynamic kiobuf allocation is measured at 0.8 % for 2 KB I/O size and 0.1% for 128 KB I/O size.

It should be noted that even though the size of kiobuf structure is small (128 byte on Linux-IA64), the entire kiobuf entity is fairly large at 200KB. The per-CPU array for kiobuf slab cache should be managed pro-actively. With default parameter that calculates the per-CPU array size based on object size, there will be maximum 252 objects allocated on per-CPU array and on a 4 CPU system, this leads to 1008 kiobuf entity, or 200MB memory allocation. A small burden to the system administrator.

4.2 Variable size Block I/O

A second enhancement made to the raw device layer is to enhance the effectiveness for the raw vary I/O on Linux-IA64. The existing code restricts the sector combining to maximum size of RAWIO_BLOCKSIZE (4KB). The user pointer is also restricted to be aligned on that boundary (4K aligned). Both restrictions are sub optimal on Linux-IA64 because they reject many scenarios that can be put into speed path.

The implementation can be modified to be run time page size aware instead of hard coded constant value. The concept is to combine all sectors within a page to send down to submit_bh. For example, on a system with default page size of 16KB, a raw I/O request with 16KB size would be broken down to 4-4-4-4KB with existing code where it could be combined optimally as one 16KB request to submit_bh. The user pointer should only be restricted to sector aligned. For example, again on a system with page size of 16KB, a raw I/O request of 4 KB I/O size with user pointer

aligned on 2KB into a page would be rejected by the existing code for fast path consideration where technically it could be in the fast path. We measured the speed up varies from 10% to 280% with micro-benchmark depending on the I/O size and buffer alignment for this enhancement.

Again, using OLTP workload to measure how well the raw vary I/O and the enhancements measure up in production environment, we ran two experiments, one with and one without raw vary I/O. It was measured that raw vary I/O gives 4% performance advantage over one without for OLTP workload.

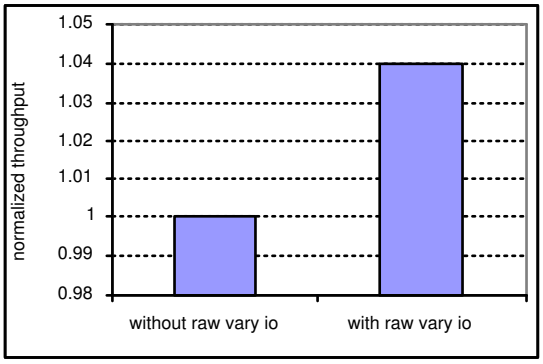


Figure 4.1: Comparison of raw vary I/O under OLTP workload

4.3 Relieve global lock contention

Another area of improvement in block I/O subsystem was the reduction of global io_request_lock usage. Much work has been done in this area [4] and the result of the work is incorporated in products released by several major Linux OS distributors. In earlier releases of Linux kernel 2.4, I/O requests are queued one at a time while holding the global lock io_request_lock. The Linux community has implemented many iterations/versions to break the global lock to per device lock. With the optimization, I/O requests are queued while holding a lock specific to the queue associated with the request. This improves concurrent I/O queuing and significantly improves I/O throughput.

5 Asynchronous I/O

5.1 History of AIO implementation

Several asynchronous I/O implementations following the POSIX standard were developed during the Linux kernel 2.3 development cycle. The implementations were either in kernel space or user space. Both of them employed an idea of an I/O queue with N number of helper threads that issues synchronous I/O to the underlying OS. However, there are several drawbacks with this approach for database application. First of all, even though the interface is asynchronous like, the I/O throughput is severely limited due to another layer of queuing. The optimum number of helper threads also depends on the characteristics of I/O subsystem and thus not flexible for wide range of production environment. A second issue is that the POSIX defined reap function aio_suspend() has a worst case of O(n) operation and tends to break down with large number of pending I/O.

5.2 A New paradigm

During the Linux 2.5 kernel development cycle, Red Hat kernel developers implemented a new AIO and its API based on the concept of completion queue [3]. Subsequently Intel worked with Red Hat to port and refined the AIO design for Linux kernel 2.4 on Linux-IA64.

The core of this kernel AIO implementation is centered around the completion queue. It introduces 5 new system calls for asynchronous operation. The core is generic that the operation is not just restricted to disk I/O, but also for network and file system I/O. The completion

queue is created by system call io_queue_init and destroyed via io_queue_release. New I/Os are submitted via io_submit and queued only if there is sufficient space in the completion queue to receive resulting event. When I/O is completed, a corresponding event is put into the complete queue and can be reaped via io_get_events. RDBMS application typically uses unbuffered I/O and combined with AIO infrastructure, disk I/Os are now being queued directly at block layer to exploit maximum concurrency for the capability of the underlying hardware devices.

5.3 AIO Evaluation and Optimization

We first turn our attention to evaluate how well does kernel asynchronous I/O performs under heavy disk I/O workload using micro-workload. The system under test has 3 fiber channel host adaptors connected to 180-disk Clariion towers. The disk towers are configured as 10 hardware RAID-0 disk drives and each RAID-0 drive has 10 raw partitions. A micro-benchmark program is then requesting AIO randomly on the 180 raw partitions with random offset (round to multiple of sector size). The I/O size is limited to 2KB and 16KB to limit the permutation of all other variables.

The micro benchmark basically throttles I/O to keep the system busy with at least N number of I/O pending at any given time. When number of pending I/O reduces to N, test program will batch next set of I/O with 'B' number of I/O in one AIO io_submit call. Completed I/O also gets reaped with each occurrence of AIO submit, i.e., program will reap approximately 'B' number of I/O in one io_get_events call.

The first experiment is to measure average CPU time spent on processing one I/O in the AIO request array. We sweep across the 'B' parameter from 32 to 1024 while holding N constant at 1000. The data was measured with pure CPU cycles spend on processing I/O excluding the wait time due to disk access latency. Figure 5.1 depicts the result.

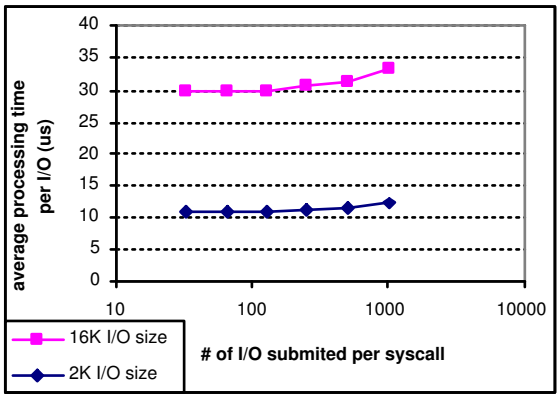


Figure 5.1: average AIO cost per I/O

For un-buffered I/O via raw device, the processing cost per AIO request in the most ideal case should be insensitive to the size of I/O. However, the large differences in the average cost between 2KB and 16 KB size in figure 4.1 indicate that there are some code path in the system break down badly with large I/O size. A kernel profiler showed that the elevator algorithm was responsible for the extra cost in the 16 KB case. It was apparent that raw vary I/O is also needed for asynchronous I/O path on raw device. Enhancements in addtion to raw vary I/O were also made in the generic AIO layer. Figure 5.2 illustrates the result of optimizations.

With raw vary I/O optimization, the cost of AIO on raw device is now quite consistent for different I/O size which matches to our expectation. The overall optimization improves 16 KB I/O size by 400% and 27% for 2 KB I/O size.

5.4 Application benefit

With all these fancy analysis done with micro-benchmark, the next question is how does

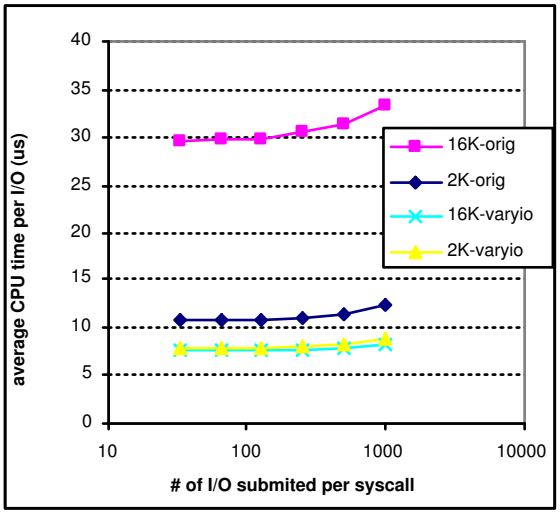


Figure 5.2: average AIO cost per I/O with optimization

AIO and the optimizations measure up in a real world production environment, like RDBMS with OLTP workload? Most I/O cache schemes employ deferred I/O operations and periodically sync up memory content with persistent data storage. A write back process is typically woken up on various conditions. One condition is database checkpoint where the process will write modified database records to persistent media in order to bring those copies of record in the persistent media current.

At high transaction rate and especially large percentage of update intensive queries in the OLTP transactions, the amount of modified database records existed in the buffer cache are high at the time when checkpointing initiates. It is essential that a write back process write those records to disks as quickly as possible to minimize amount of CPU processing time consumed on checkpoint task. Since the purpose of checkpoint is to sync-up persistent data file with content in memory, it actually has very little data dependency on when the blocks are being written, as long as database server gets notified that the writes are completed. This requirement fits perfectly with the non-blocking semantics of asynchronous I/O.

There are two ways for the writeback process to submit I/O to the OS. One is an internal RDBMS facility that distributes I/O among multiple helper processes for system that lacks the native AIO implementation. This facility is similar to I/O queue and help threads described earlier. The other is to submit I/O to the OS via native AIO interface. Again, two experiments were conducted, first with I/O helper threads configuration to establish a baseline result and second with native AIO configuration. Figure 5.3 depicts the result.

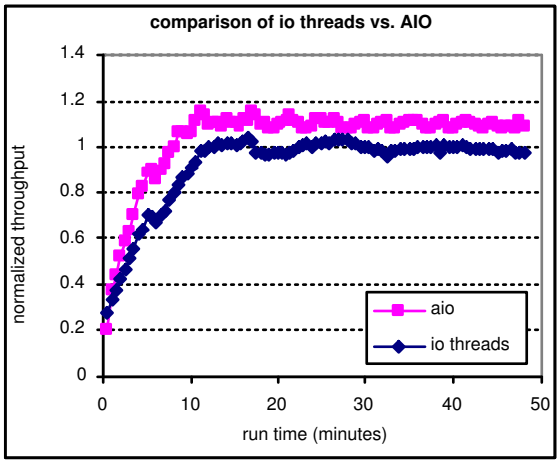


Figure 5.3: Benefit of AIO for OLTP workload

Several points worth noting here. At steady state, configuration with AIO is 10% higher in OLTP throughput compare to without AIO. It is due to combination of reduction in I/O processes' overhead and efficient OS I/O queuing and event reaping. Second note is that in the I/O helper thread configuration, system takes extra overhead in context switching between the helper threads and other active processes. Not only the helper thread takes a penalty hit with process context switch, it also puts more pressure on the CPU's data cache because more processes are actively running on the system. In the asynchronous I/O case, disk I/Os are

submitted directly to OS, thus reduces number of context switches. As illustrated from figure 5.4, the number of process context switches is reduced by 12 % with asynchronous I/O.

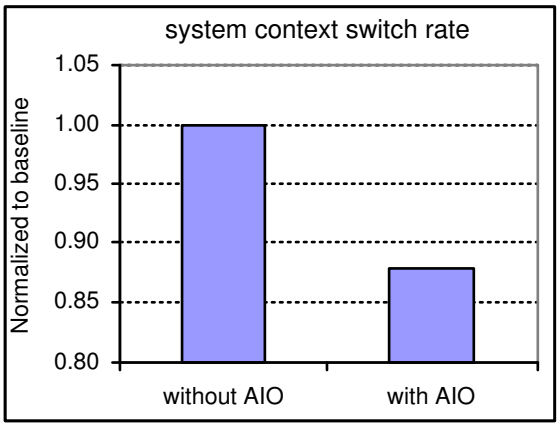


Figure 5.4: system context switch rate with and without AIO

There are many other secondary effects indirectly improving overall system performance by using AIO. System memory consumption is reduced because there aren't any I/O helper threads at all. OS scheduler will have less pressure because less number of active tasks it need to manage, and lastly inter-process communication overhead between the helper threads are eliminated. All of these translate into highly efficient scalable asynchronous I/O layer and higher OLTP throughput.

6 Storage Device Driver Optimization

While most I/O enhancements outlined in previous section are more or less transparent to storage device driver, some still do require cooperation from each individual driver to enable specific optimization. One example would be HP's smart array family of disk controllers. Since this driver hooks directly into Linux I/O block layer, it missed out all the enabling infrastructures for the raw vary I/O and the global io_request_lock optimization implemented for SCSI devices.

Both optimizations are fairly straightforward to enable. What we did was at the time of the controller's initialization, we initialize a per-controller raw vary I/O capability array and then hook that array into the blkdev_varyio defined in the block layer. To enable per device request lock, two locks are added in the controller's data structure, one lock for I/O request queue, and one for the controller itself. Locking primitives are then modified to use the corresponding request queue lock in the case of I/O queuing/dequeuing. For operations that pertain to controller, the controller lock will be used.

Other optimizations that were also actively worked on for this particular storage device driver are interrupt coalescing, 32-bit DMA command pool, and Itanium architecture specific command structure alignment.

7 Conclusions

In this paper we have outlined some of the key operating system requirements for running a high performance database on Linux. Implementations of huge TLB support and asynchronous I/O have been described along with how these features perform to expectation under OLTP workloads. The I/O subsystem for the Linux kernel 2.4 has been improved significantly to achieve high concurrency and efficiency for high demand I/O workload. Storage device driver optimizations are also shown to be equally important to materialize optimizations done at generic layer.

8 Acknowledgement

The authors of this paper would like to thank the following people who enthusiastically con-

tribute directly or indirectly to this paper, in no particular order: Asit Mallick, Arun Sharma, Tony Luck, Sunil Saxena, Mark Gross and several other groups at Intel Corporation; Red Hat kernel developers; Linux community around the world.

References

[1] *Intel Itanium Architecture Software Developer's Manual*, Volume 1-3.

[2] David Mosberger and Stephane Eranian, *ia-64 linux kernel design and implementation.*, Prentice Hall, 1st edition, 2002.

[3] Benjamin LaHaise, *An AIO Implementation and its Behavior*, Ottawa Linux Symposium proceedings 2002.

[4] Peter Wai Yee Wong, et al., *Improving Linux Block I/O for Enterprise Workloads*, Ottawa Linux Symposium proceedings 2002.

Trademarks

High Availability Data Replication

Paul Clements
SteelEye Technology, Inc.
http://www.steeleye.com/
paul.clements@steeleye.com

James E.J. Bottomley
SteelEye Technology, Inc.
http://www.steeleye.com/
james.bottomley@steeleye.com

Abstract

This paper will identify some problems that were encountered while implementing a highly available data replication solution on top of existing Linux kernel drivers. It will also discuss future plans for implementing asynchronous replication and intent logging (which are requirements for performing disaster recovery over a WAN) in the Linux kernel.

1 Introduction

The first part of this paper (Section 2) will discuss some issues in the 2.2 and 2.4 Linux kernels that had to be overcome in order to implement a replication solution using raid1 over nbd.

The second part of the paper (Section 3) will present future plans for implementing asynchronous replication and intent logging in the md and raid1 drivers.

2 Fixing the existing problems

We've done considerable work over the past 3 years testing, debugging, and finally fixing several problems in the md, raid1, and nbd drivers of the Linux kernel. We ran into several bugs in these drivers, primarily due to the fact that we're using them in an unusual fashion, with one of the underlying disk devices of the raid1 mirror being accessed over the network, via a network block device (see Figure 1). Our usage of raid1 in conjunction with nbd led to the increased occurrence of several race conditions and also caused the error handling code of the drivers to be stressed much more than a normal, internal disk only, raid1 setup.

The following is a brief summary of some of the problems we've uncovered and solved:

- eliminating md retries in order to avoid massive stalls when a device (in our case, a network device) fails
- correcting SMP locking errors and allowing an nbd connection to be cleanly aborted when problems are encountered
- fixing various bugs in the raid1 driver:
  - mistakes in the error handling code
  - incorrect SMP locking
  - IRQ enabling/disabling bugs
  - non-atomic memory allocations in critical regions
  - block number "off by one" error[1]

At the time of this writing, patches for all of these problems have been accepted into the latest releases of the mainline 2.4 and 2.5 kernel

[1] This problem was actually corrected by Neil Brown after our initial bug report to him.

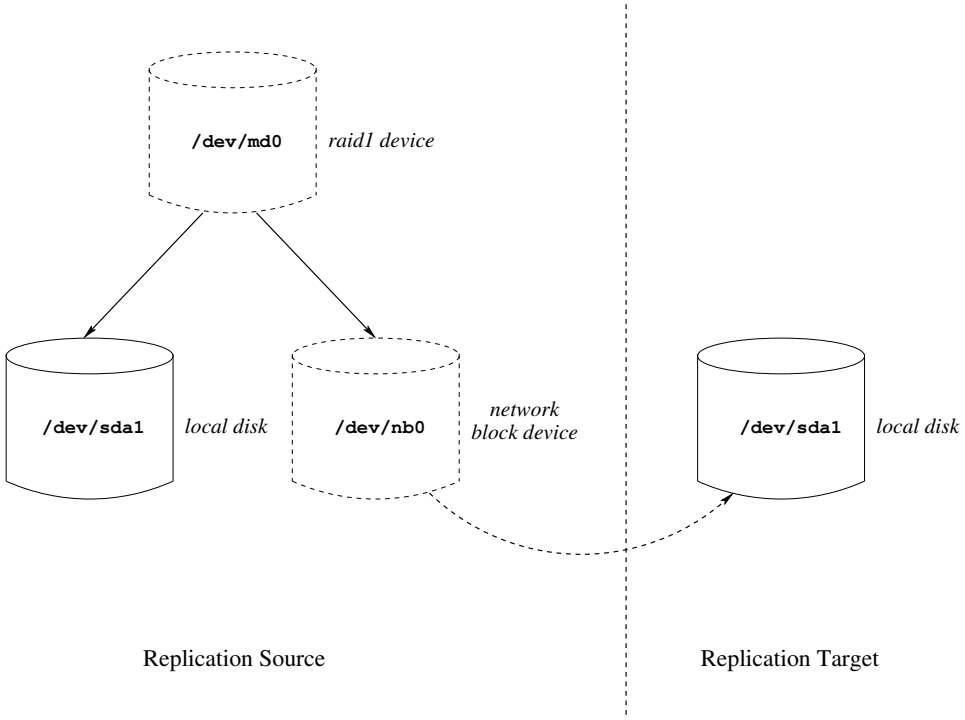


Figure 1: Data replication using raid1 over nbd

series. For information about all of SteelEye's open source patches or to download the source code for the patches, visit the SteelEye Technology Open Source Website[1].

2.1 Eliminating md retries

This was one of the first problems that we encountered back in the fall of 2000. At the time, we were working with the 2.2 Linux kernel. The md resynchronization code (`md_do_sync`) was written, at the time, to always retry any failed I/O (read or write) no less than 4096 times![2] On a network failure, this caused raid1 and nbd to spin in tight loops for several seconds, hanging the entire system. Our stopgap solution (read, hack) was to strategically insert `schedule` calls into the error handling code of those drivers.[3] Needless to say, the md resynchronization code got a major overhaul before the 2.4 kernel was released and this issue was fixed.

2.2 Allowing nbd connections to be aborted

After initially fixing a few trivial bugs in nbd having to do with missing or incorrect spin lock calls, we realized that we could not afford to wait for TCP socket timeouts when we needed to abort a network connection, or when the network went down. We needed to have the ability to terminate nbd connections at will, so that our high availability services could have complete control over the data replication process. To fix this, we unblocked `SIGKILL` during the network transmission phase of nbd so that we could send a signal from user space to terminate an nbd connection. We also needed to add code to ensure that nbd's request queue was

[2] `PAGE_SIZE`$*(1 << 1) * 2/$ `sizeof(struct buffer_head *)` (md.c, c. 2.2.16 kernel)

[3] We did not have the option of modifying md, since it is compiled into the kernel in most Linux distributions, and we did not want to be in the business of distributing entire rebuilt kernels.

cleared and all outstanding I/Os were marked as failed when a connection was terminated.

Our patches for nbd have been accepted into the mainline 2.5 kernel (c. 2.5.50) and were backported and accepted into the 2.4 kernel (c. 2.4.20-pre).

2.3 Fixing various bugs in the raid1 driver

The raid1 driver, by far, has been the biggest thorn in our side... we've made many fixes to raid1 over the past few years in order to increase its robustness. Neil Brown has simultaneously been performing a lot of cleanup and bugfix work in the md and raid1 drivers that was beneficial to our cause, as well.

The first set of problems that we ran into with raid1 was related to handling failures of the underlying devices.[4] To correct the problems, we added code to detect device failures during resync and during normal I/O operations. The additional code correctly marks the device "bad," fails all outstanding I/Os, and aborts resync activities, if necessary.

After fixing this initial set of problems, we were able to stress the raid1 driver much more heavily than we previously had been able to (without it falling over dead). Unfortunately, this heavier stress uncovered a whole raft of new problems. We were able, however, to eventually pin-point and solve each of these new problems. Many of the problems turned out to be due to some fairly common kernel programming mistakes, such as:

- **"nested" `spin_lock_irq` calls** – Failure to use the `save` and `restore` versions of the spin lock macros with nested calls (i.e., `spin_lock_irq` called while another `spin_lock_irq` is already in force) leads to the CPU flags being improperly set. This means that interrupts could be enabled at inappropriate times, causing deadlocks to occur. The rule of thumb here is that it's best to avoid nesting spin locks whenever possible, and to always use the `irqsave` and `irqrestore` versions of the macros, instead of the simple `irq` versions, if deadlocks are a concern.
- **sleeping with a spin lock held** – There were cases where the driver was doing non-atomic memory allocations or calling `schedule` with a spin lock held, which caused deadlocks to occur. To avoid the deadlocks, the code was rearranged so that a spin lock was never held while calling `schedule` and a few `kmalloc` calls were changed to use the `GFP_ATOMIC` flag rather than the `GFP_KERNEL` flag.
- **"off by one" error** – This was a simple case of differing block sizes being used in the md and raid1 drivers resulting in one of the block counts used in the resync code being shifted incorrectly. This bug caused resyncs to hang, leaving the raid device in an unusable state.

Our patches for raid1 have been accepted into the Red Hat Advanced Server 2.1 kernel (2.4.9-ac based) and an alternate version of the fixes (authored by Neil Brown) has been accepted into the mainline 2.4 kernel (c. 2.4.19-pre). The raid1 driver in the 2.5 kernel is not believed to suffer from any of the aforementioned problems.

3 Future enhancements

We are planning to enhance the md and raid1 drivers of the Linux kernel to support asynchronous data replication and intent logging.

[4] Since we use nbd underneath raid1, device "failures" are quite a common occurrence (e.g., when the network goes down).

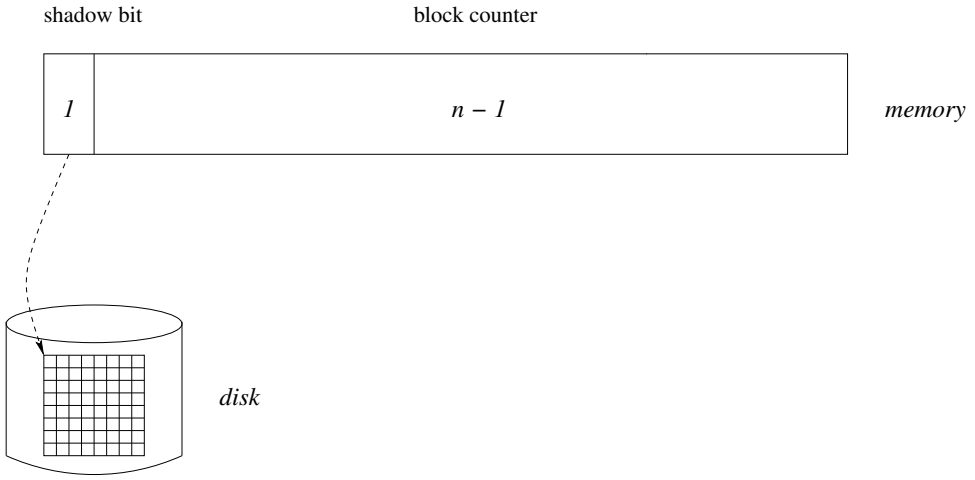


Figure 2: In-memory and on-disk bitmap layout

Our strategy for implementing these changes will be to place the bulk of the code into the md driver, in a manner that will allow all the underlying raid drivers to take advantage of it. We will also add the necessary code to raid1 to call the md driver hooks.

We plan to leverage some of the implementation and design of Peter T. Breuer's *fr1* code[2], which was recently published[3]. The *fr1* driver implements intent logging and asynchronous replication as an add-on to the raid1 driver. We will make the following, additional changes to the *fr1* code, to produce a final solution:

- disk backing for the bitmap (intent log)
- addition of a daemon to asynchronously clear the on-disk bitmap
- conversion of single bits to 16-bit block counters (to track pending writes to a given block, so as not to prematurely clear a bitmap bit on disk)
- allow rescaling of the bitmap (i.e., allow one bit to represent various block sizes—the current code is restricted to one bit per 1024-byte data block only)
- make the code fully leveragable by all the raid personality drivers
- add some additional configuration interfaces for the new features

3.1 Intent Logging

In a data replication system, an intent log is used to keep track of which data blocks are out of sync between the primary device and the backup device. An intent log is simply a bitmap, in which a set bit (1) represents a data block that is out of sync, and a cleared bit (0) represents a data block that is in sync. The use of an intent log obviates the full resync that is normally required upon recovery of an array.

3.1.1 Bitmap Layout

We will store the bitmap both in memory and on disk, in order to be able to withstand failures (or reboots) of the primary server without losing resynchronization status information.

We will use a simple, one-bit-per-block bitmap for the on-disk representation of the intent log, while the in-memory representation will be

slightly more complex. The reason for this additional complexity is the need to track pending writes, so as not to clear a bit in the bitmap until all pending writes for that data block have completed[5]. The write tracking will be handled using a 16-bit counter for each data block. One bit in the counter will actually be used as a "shadow" of the corresponding on-disk bit, reducing the usable counter size by one bit (see Figure 2). The counter will be incremented when a write begins and decremented when one has completed. Only when the counter has reached zero, can the on-disk bit be cleared.

In order to conserve RAM, the in-memory bitmap will be constructed in a two-level fashion, with memory pages being allocated and deallocated on demand (see Figure 3). This allows us to allocate only as much memory as is needed to hold the set bits in the bitmap. As a fail-safe mechanism, when a page cannot be allocated, the (pre-allocated) pointer for that page will actually be hijacked and used as a counter itself. This will allow logging to continue, albeit with less granularity,[6] during periods of extreme memory pressure.

The bitmap will also be designed so that it is possible to readjust the size of the data "chunks" that the bits represent. This will be handled by translating from the default md driver I/O block size of 1KB to the chunk size, whenever the bitmap is marked or cleared. So, with a chunk size of 64KB, for example, the I/O to 64 contiguous disk blocks will be tracked by a single bit in the on-disk bitmap (and the corresponding in-memory counter).

[5] clearing the bit prematurely could result in data corruption on the backup device if a network failure coincides

[6] On x86, with 32-bit pointers and 4KB pages, the granularity is reduced to roughly 1/1000 the normal level.

3.1.2 Bitmap Manipulation

To make use of the bitmap, we will make modifications to two areas of the raid1 driver:

1. Ordinary **write operations** will require a bitmap entry be made (and synced to disk) before the actual data is written—the bitmap entry will be cleared once the data has been written to the backup device.
2. **Resynchronization operations** will no longer involve a full resynchronization of the backup device, but rather a resync of just the "dirty" blocks (as indicated by the bitmap).

3.1.3 Write Operations

The sequence of events to write block *n* on a raid1 device with an intent log is as follows:

1. set the *n*th shadow bit in the in-memory bitmap and increment the counter for block *n* (both can be done as a single operation since the shadow bit and counter are contiguous)
2. increment the "outstanding write request" counter for the array[7] (and disallow further writes to the device if the counter has exceeded the configured limit)
3. sync the shadow bit to disk, if the on-disk bit was not already set
4. duplicate the write request, including its data buffer
5. queue the write request to the primary device

[7] This counter is really only used when the array is in asynchronous replication mode. For more details, see Section 3.2.

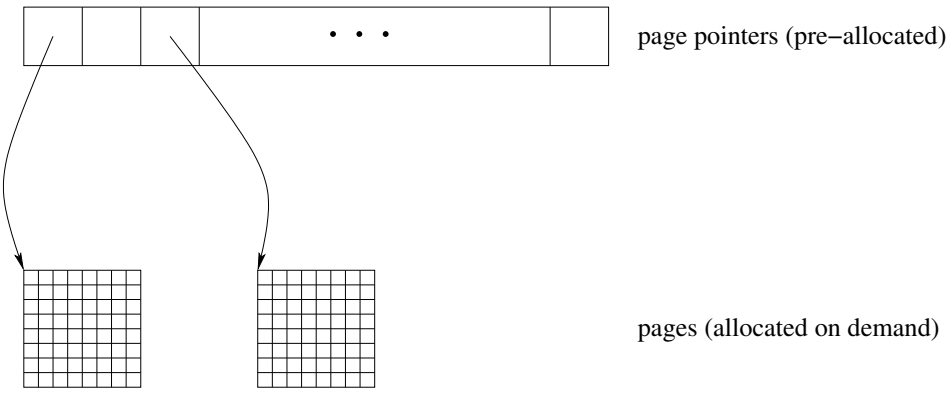


Figure 3: Two-level, demand-allocated bitmap

6. queue the duplicate request to the backup device

We then allow the writes to complete asynchronously. After each write is completed, the raid1 driver is notified with a call to its `b_end_io` callback function (`raid1_end_request`). This function is responsible for signalling the completion of I/O back to its initiator. In synchronous mode, we wait until the writes to both the primary and backup devices have completed before acknowledging the write as complete. In asynchronous mode, the write is acknowledged as soon as the data is written to the primary device.

After the write has been acknowledged, the callback function is responsible for decrementing the block counter and, if the counter's value is 0, clearing the shadow bit in the in-memory bitmap. Whenever a shadow bit is cleared, a request will also be placed in a queue to indicate that the on-disk bit needs to be cleared.

The bits in the on-disk bitmap will be cleared asynchronously, by a dedicated kernel daemon, `mdflushd`. The daemon will periodically awaken and flush all the queued updates to disk.[8] The interval at which the daemon will awaken and flush its queue will be tunable (with a default value of 5 seconds).

Clearing the bits in the on-disk bitmap in a lazy manner will help to reduce the number of disk writes, and will also ensure that any bits that happen to correspond to I/O "hotspots"[9] will simply remain dirty, rather than causing a constant stream of writes to the on-disk bitmap.

3.1.4 Resynchronization Operations

The resynchronization process of the md driver is fairly straightforward. Following recovery from a failure, the driver will attempt a complete resync of the backup device. We will modify this process slightly, so that for each data block that is to be resynchronized, we will first check the appropriate shadow bit in the in-memory bitmap and then, either:

- resync the block (if the bit is set), or
- discard the resync request and indicate success (if the bit is cleared)

Once a block has been resynced, its shadow bit will be cleared and its block counter zeroed. An update request will then be queued

[8] unless the shadow bit has been reset in the meantime, in which case the update is simply discarded and the on-disk bit is left set.

[9] areas of the disk that are continually written, such as an ext3 filesystem journal

to tell `mdflushd` that the on-disk bit should be cleared.

3.2 Asynchronous Replication

In an asynchronous replication system, write requests to a mirror device are acknowledged as soon as the data is written to the primary device in the mirror. In contrast, in a synchronous replication system, writes are not acknowledged until the data has been written to all components of the mirror. Synchronous replication works well in environments where the mirror components are local. However, when the backup device is located on a network, the write throughput of a synchronous mirror decreases as network latency increases. An asynchronous mirror does not suffer this performance degradation since a write operation can be completed without waiting for the write request and its acknowledgement to make a complete roundtrip over the network. To achieve reasonable write throughput in a WAN replication environment, an asynchronous mirror is generally employed.

3.2.1 Outstanding Write Request Limit

In an asynchronous mirror, there can be several outstanding (i.e., in-flight) write requests at any given time. In order to limit the amount of data that is out of sync on the backup device during normal mirror operation, it is necessary to keep the number of outstanding write requests fairly low. Therefore, we will place a limit on the number of outstanding write requests. However, to avoid degrading the write throughput of the mirror, this limit must be adequately high. Since the limit will need to be tuned appropriately for each environment, it will be made a user configurable parameter.[10] When the limit for outstanding writes has been exceeded, the driver will throttle writes to the mirror until another write acknowledgement returns from the remote system (i.e., the mirror will degrade to synchronous write mode). A message will be printed in the system log when this event occurs, to warn system administrators that they should adjust the relevant parameters. The outstanding write request limit will default to a reasonable value (which will be determined through testing).

3.2.2 Device Tagging

In synchronous replication mode, there is no real need to differentiate between primary and backup devices, since writes must be committed to all array components before being acknowledged. However, in asynchronous mode, the component devices of a raid1 array will need to be tagged as "primary" or "backup" to ensure that the bitmap is handled correctly, and to ensure that read requests are always satisfied from the primary device. To accomplish this, we will need an additional `/etc/raidtab` directive to enable a device to be tagged as a "backup." Devices tagged as backups will be placed into a special "write-only" mode that exists in md.

4 Conclusion

With the recent bugfix and cleanup work that has been done, and with the upcoming additional features that are in the works, the Linux kernel md driver will finally be an enterprise-class software RAID and data replication solution: robust, and capable of being used for many different applications, from simple internal disk mirroring and striping, to LAN data replication, and even disaster recovery over a

[10] To avoid overflowing the block counters in the in-memory bitmap, we will make it impossible to set this limit higher than the maximum value for those counters.

WAN.

5 Acknowledgements

We would especially like to thank Peter T. Breuer and Neil Brown for their outstanding and ongoing work in the Software RAID (md) subsystem of the Linux kernel. Without their contributions, we would not have been able to undertake such a huge endeavor.

References

[1] SteelEye Technology, Inc. *SteelEye Technology Open Source Website* `http://licensing.steeleye.com/open_source/`

[2] Peter T. Breuer. *Fast Intelligent Software RAID1 Driver* `http://www.it.uc3m.es/ptb/fr1/` `http://freshmeat.net/projects/fr1/`

[3] Peter T. Breuer, Neil Brown, Ingo Molnar, Paul Clements. *linux-raid mailing list discussions on raid1 bitmap and asynchronous writes* `http://marc.theaimsgroup.com/?l=linux-raid&b=200302` Jan-Apr 2003

Porting NSA Security Enhanced Linux to Hand-held devices

Russell Coker
russell@coker.com.au
http://www.coker.com.au/

Abstract

In the first part of this paper I will describe how I ported SE Linux to User-Mode-Linux and to the ARM CPU. I will focus on providing information that is useful to people who are porting to other platforms as well. In the second part I will describe the changes necessary to applications and security policy to run on small devices. This will be focussed on hand-held devices but can also be used for embedded applications such as router or firewall type devices, and any machine that has limited memory and storage.

1 Introduction

SE Linux offers significant benefits for security. It accomplishes this by adding another layer of security in addition to the default Unix permissions model. This is achieved by firstly assigning a *type* to every file, device, network socket, etc. Then every process has a *domain*, and the level of access permitted to a type is determined by the domain of the process that is attempting the access (in addition to the usual Unix permission checks). Domains may only be changed at process execution time. The domain may automatically be changed when a process is executed based on the type of the executable program file and the domain of the process that is executing it, or a privileged process may specify the new domain for the child process.

In addition to the use of domains and types for access control SE Linux tracks the *identity* of the user (which will be *system_u* for processes that are part of the operating system or the Unix user-name) and the role. Each *identity* will have a list of roles that it is permitted to assume, and each *role* will have a list of domains that it may use. This gives a high level of control over the actions of a user which is tracked through the system. When the user runs SUID or SGID programs the original identity will still be tracked and their privileges in the SE security scheme will not change. This is very different to the standard Unix permissions where after a SUID program runs another SUID program it's impossible to determine who ran the original process. Also of note is the fact that operations that are denied by the security policy [1] have the *identity* of the process in question logged.

I often run SE Linux demonstration machines on the Internet which provide root access to the world and an invitation to try and break the security. [2]

For a detailed description of how SE Linux works I recommend reading the paper Peter Loscocco presented at OLS in 2001 [3].

SE Linux has been shown to provide significant security benefits for little overhead on servers, desktop workstations, and laptops.

However it has not had much use in embedded devices yet.

Some people believe that SE Linux is only needed for server systems. I think that is incorrect, and I believe that in many situations laptops and hand-held devices need more protection than servers. A server will usually have a firewall protecting it, with a small number of running applications which are well maintained and easy to upgrade. Portable computers are often used in hostile environments that servers do not experience, they have no firewalls to protect them, and often they are connected to routers operated by potentially negligent or hostile organizations.

But there are two main factors that cause an increased need for security on portable devices. One is that it is usually extremely difficult and expensive to upgrade them if a new security fix is needed. This means that in commercial use portable computers tend to never have security fixes applied. Another factor is that often the person in posession of a hand-held computer is not authorised to access all the data it contains, and may even be hostile to the owner of the machine.

Naturally for a full security solution for portable computers a strong encryption system will need to be used for all persistent file systems. There are various methods of doing this, but all aspects of such encryption are outside the scope of this project and can be implemented independently.

2 Kernel Porting

The current stable series of SE Linux is based on the 2.4.x kernels and uses the Linux Security Modules (LSM) [4] interface. The current LSM interface has a single sys_security() system call that is used to multiplex all the system calls for all of the security modules. SE Linux uses 52 different system calls through this interface. Due to problems in porting the kernel code to some platforms (particularly those that have a mixed 32 and 64bit memory model) the decision was made to change the LSM interface for kernel 2.6.0. The new interface will make the code fully portable and remove the painful porting work that is currently required. However I needed to have SE Linux working with the 2.4.x kernels so I couldn't wait for kernel 2.6.0.

The main difficulty in porting the code is the system call execve_secure() which is used to specify the security context for the new process. This calls the kernel funtion do_exec() to perform the execution, and do_exec() needs a pointer to the stack, thus requiring architecture specific code in the sys_execve_secure() function. The sys_security_selinux_worker() function (which determines which SE Linux system call is desired and passes the appropriate parameters to it) calls sys_execve_secure() and therefore also needs architecture specific code, and so does the main system call sys_security_selinux().

My first port of SE Linux was to User-Mode Linux [5]. This was a practice effort for the main porting work. It is quite easy to debug kernel code under UML, and as it uses the i386 system call interface I could port the kernel code without any need to port application code.

The main architecture dependent code is in the source file `security/selinux/arch/i386/wrapper.c`, which has code to look on the stack for the contents of particular registers. This needs to be changed for platforms with different register names, and for UML which does not permit such direct access of registers.

The solution in the case of UML was to not have a wrapper function, as the *current* structure had a pointer to the stack anyway that

could be used inside the sys_execve_secure() function. So I renamed the sys_security_ selinux_worker() function to sys_security_ selinux() for the UML port and entirely removed all reference to the wrapper. Then I moved the implementation of sys_execve_ secure() into the platform specific directory and implemented a different version for each port.

This was essentially all that was required to complete the port, the core code of SE Linux was all cleanly written and could just be compiled. The only other work involved getting the Makefile's correctly configured, and adding a hook to sys_ptrace().

One thing I did differently with my port to the ARM architecture was that I removed the code to replace the system call entry. When the SE Linux kernel code loads on UML and i386 it replaces the system call with a direct call to the SE Linux code (rather than using the option for LSM to multiplex between different modules). As there is currently no support for having SE Linux be a loadable module there seems to be no benefit in this, and it seems that on ARM there will be more overhead for adding an extra level of indirection for this. So I made the SE Linux patch hard-code the SE system call into the sys-call table.

3 iPaQ Design Constraints

The CompaQ/HP iPaQ [6] computers are small hand-held devices. The most powerful iPaQ machines on sale have a 400MHz ARM based CPU that is of comparable speed to a 300MHz Intel Celeron CPU, with 64M of RAM and 48M of flash storage.

An iPaQ is not designed for memory upgrades. There are some companies that perform such upgrades, but they don't support all models, and this will void your warantee. Therefore you are stuck with a memory limit of 64M.

The flash storage in an iPaQ can only be written a limited number of times, this combined with the small amount of storage makes it impossible to use a swap space for virtual memory unless you purchase a special *sleeve* for using an external hard drive. Attaching an external hard drive such as the IBM/Hitachi *Micro Drive* is expensive and bulky. Therefore if you have a limited budget then storage expansion (for increased file storage or swap space) is not an option.

For storing files, the 32M file system can contain quite a lot. The Familiar distribution is optimised for low overheads (no documentation or man pages) and all programs are optimised for size not speed. Also the JFFS2 [7] file system used by Familiar supports several compression algorithms including the Lempel-Ziv algorithm implemented in zlib, so more than 32M of files can fit in storage.

For a system such as SE Linux to be viable on an iPaQ it has to take up a small portion of the 32M of flash storage and 64M of RAM, and not require any long CPU intensive operations.

Finally the screen of an iPaQ only has a resolution of 240x320 and the default input device is a keyboard displayed on the screen. This makes an iPaQ unsuitable for interactive tasks that involve security contexts as it takes too much typing to enter them and too much screen space to display them. As a strictly end-user device this does not cause any problems.

4 CPU Requirements

Benchmarks that were performed on SE Linux operational overheads in the past show that trivial system calls (reading from /dev/zero and writing to /dev/null) can take up to 33% longer to complete when SE Linux is running, but that

the overhead on complex operations such as compiles is so small as to be negligible [8]. The machines that were used for such tests had similar CPU power to a modern iPaQ.

One time consuming operation related to SE Linux installation is compiling the policy (which can take over a minute depending on the size of the policy and the speed of the CPU). This however is not an issue for an iPaQ as the policy takes over a megabyte of permanent storage and 5 megs of temporary file storage, as well as requiring many tools that are not normally installed (make, m4, the SE Linux policy compilation program checkpolicy, etc). The storage requirements make it impractical to compile policy on the iPaQ, and the typical use involves configuration being developed on other machines for deployment on iPaQ. So the time taken to compile the policy database is not relevant.

The only SE Linux operation which can take a lot of time that must be performed on an iPaQ is labeling the file system. The file system must be relabeled when SE Linux is first installed, and after an upgrade. On my iPaQ (H3900 with 400MHz X-Scale CPU) it takes 29.7 seconds of CPU time to label the root file system which contains 2421 files. For an operation that is only performed at installation or upgrade time 29.7 seconds is not going to cause any problems. Also the *setfiles* program that is used to label the file system could be optimised to reduce that time if it was considered to be a problem.

I conclude that for typical use of a handheld machine SE Linux only requires the CPU power of an iPaQ. In fact the CPU use is small enough that even the older iPaQ machines (which had half the CPU power) should deliver more than adequate performance.

5 Kernel Resource Use

To compare the amounts of disk space and memory I compiled three kernels. One was 2.4.19-rmk6-pxa1-hh13 with the default config for the H3900 iPaQ. One was a SE Linux version of the same kernel with the options CONFIG_SECURITY, CONFIG_SECURITY_CAPABILITIES, and CONFIG_SECURITY_SELINUX. Another was the same SE Linux kernel with development mode enabled (which slightly increases the size and memory).

For this project I have no need for the multi-level-security (MLS) functionality of SE Linux or the options for labelled networking and extended socket calls. This optional functionality would increase the kernel size. I am focussing on evaluating the choice of whether or not to use SE Linux for specific applications, once you have decided to use SE Linux you would then need to decide whether the optional functionality provides useful benefits to your use to justify the extra disk space and memory use.

The kernel binaries are 658648 bytes for a non-SE kernel, 704708 bytes for the base SE Linux kernel, and 705560 bytes for the development mode kernel. The difference between the kernel with development mode enabled and the regular one is that the development kernel allows booting without policy loaded, and booting in permissive mode (with the policy decisions not being enforced). For most development work a kernel with development mode enabled will be used, also for this test it allowed me to determine the resource consumption of SE Linux without a policy loaded.

To test the memory use of the different kernels I configured an iPaQ to not load any kernel modules. My test method was to boot the machine, login at the serial console, wait 30 seconds to make sure that all daemons have

started, and run *free* to see the amount of memory that is free. This is not entirely accurate as random factors may result in different amounts of memory usage, however this is not as significant on the Familiar distribution due to the use of *devfs* for device nodes and *tmpfs* for /var and /tmp which means that in the normal mode of operation almost nothing is written to the root file system, so two boots will be working on almost the same data.

From the results I looked at the *total* field in the results (which gives the amount of RAM that is available for user processes after the kernel has used memory in the early stages of the boot process), and the *used* field which shows how much of that has been used. The kernel message log gives a break-down of RAM that is used by the kernel for code and data in the early stages of boot, however that is not of relevance to this study only the total amount that is used matters.

The *total* memory available was reported as 63412k for the non-SE kernel, 63308k for the SE Linux kernel, and 63300k for the development mode kernel. So SE Linux takes 104k of kernel memory early in the boot process and 112k if you use the development mode option.

The memory reported as *used* varied slightly with each boot. For the vanilla kernel the value 18256k was reported in two out of four tests, with values of 18252k and 18260k also being reported. I am taking the value 18256k as the working value which I consider accurate to within 8k.

For a standard SE Linux kernel the amount reported as *used* was 19516k in three out of six tests with the values of 19532k, 19520k, and 19524k also being returned. So I consider 19516k as the working value and the accuracy to be within 16k.

For the SE Linux kernel with development mode enabled the memory *used* was 19516k in three out of four tests, and the other test was 19524k. So the difference between the development mode kernel and the regular SE Linux kernel is only 8K of kernel memory in the early stages of the boot process.

Finally I did a test of a development mode kernel with no policy loaded. The purpose of this test was to determine how much memory is used on a SE Linux kernel if the SE Linux code is not loading the policy. For this the memory reported as *used* was 18292k in three out of five tests, with the values of 18296k and 18300k also being returned.

Kernel	memory used
non-SE	18256k
SE no policy	18292k
SE with policy	19516k

So an SE Linux kernel without policy loaded uses approximately 36K more memory after boot than a non-SE kernel in addition to the 104k or 112k used in the early stages of boot.

With a small policy loaded (360 types and 23,386 rules for a policy file that is 583771 bytes in size) the memory used by the kernel is about 1224k for the policy and other SE Linux data structures. The policy could be reduced in size as there are many rules which would only apply to other systems (the sample policy is quite generic and was quickly ported to the iPaQ), although there may be other areas of functionality that are desired which would use any saved space.

So it seems that when using SE Linux the memory cost is 104k when the kernel is loaded, and a further 1260k for SE Linux memory structures and policy when the boot process is complete. The total is 1364k of non-swappable kernel memory out of 64M of total RAM in an iPaQ, this is about 2% of RAM.

All tests were done with GCC 3.2.3, a modified Linux 2.4.19, and an X-scale CPU. Different hardware, kernel version, and GCC version will give different results.

6 Porting Utilities

The main login program used on the Familiar [9] distribution is *gpe-login*, which is an *xdm* type program for a GUI login. This program had to be patched to check a configuration file and the security policy to determine the correct security context for the user and to launch their login shell in that context. The patch for this functionality made the binary take 4556 bytes more disk space in my build (29988 bytes for the non-SE build compared to 34544 bytes for the version with SE Linux support).

The largest porting task was to provide SE Linux support in Busybox [10]. Busybox provides a large number of essential utility programs that are linked into one program. Linking several programs into one reduces disk space consumption by spreading the overhead for process startup and termination code across many programs. On arm it seems that the minimum size of an executable generated by GCC 3.2.3 is 2536 bytes. In the default configuration of Familiar Busybox is used for 115 commonly used utilities, having them in one program means that the 2.5K overhead is only used once not 115 times. So approximately 285K of uncompressed disk space is saved by using busybox if the only saving is from this overhead. The amount of disk space used for initialisation and termination code would probably increase the space used by more than 80% if all the applets were compiled separately (my build of Busybox for the iPaQ is 337028 bytes).

The programs that are of most immediate note in busybox are *ls*, *ps*, *id*, and *login*. *ls* needs the ability to show the security contexts of the files, *ps* needs to show the security contexts of the running processes, and *id* needs to show the context of the current process. Also the */bin/login* applet had to be modified in the same manner as the *gpe-login* program. These changes resulted in the binary being 5600 bytes larger (337028 bytes for a non-SE version and 342628 bytes for the version with SE Linux support.

7 Busybox Wrappers for Domain Transition

In SE Linux different programs run in different security *domains*. A domain change can be brought about by using the *execve_secure()* system call, or it can come from an automatic domain transition. An example of an automatic domain transition is when the *init* process (running in the *init_t* domain) runs */sbin/getty* which has the type *getty_exec_t*, which causes an automatic transition to the domain *getty_t*. Another example is when getty runs */bin/login* which has the type *login_exec_t* and causes an automatic transition to the domain *local_login_t*. This works well for a typical Linux machine where */sbin/getty* and */bin/login* are separate programs.

When using Busybox the getty and login programs will both be sym-links to */bin/busybox* and the type of the file as used for domain transitions will be the type of */bin/busybox*, which is *bin_t*. SE Linux does not perform domain transitions based on the type of the sym-link, and it assignes security types to the Inodes not file names (so a file with multiple hard links will only have one type). This means that we can't have a single Busybox program automatically transitioning into the different domains.

There are several possible solutions to this

problem, one possible partial solution would be to have Busybox use *execve_secure()* to run copies of itself in the appropriate domain. Busybox already has similar code for determining when to change UID so that some of the Busybox applets can be effectively SETUID while others aren't. The SETUID management of Busybox requires that it be SETUID root, and involves some risk (any bug in busybox can potentially be exploited to provide root access). Providing a similar mechanism for transitioning between SE Linux security domains would have the same security problems whereby if you crack one of the Busybox applets you could then gain full access to any domain that it could transition to. This does not provide adequate security. Also it would only work for transitions between privileged domains (it would not work for transitions from unprivileged domains). I did not even bother writing a test program for this case as it is not worth considering due to a lack of security and functionality.

A better option is to split the Busybox program into smaller programs so transitions can work in the regular manner. With the current range of applets that would require one program for *getty*, one for *login*, one for *klogd*, one for *syslogd*, one for *mount* and *umount*, one for *insmod*, *rmmod*, and *modprobe*, one for *ifconfig*, one for *hwclock*, one for all the fsck type programs, one for *su*, and one for *ping*. Of course there would also be one final build of busybox with all the utility programs (ls, ps, etc) which run with no special privilege. To test how this would work I compiled Busybox with all the usual options apart from modutils, and I did a separate build with only support for modutils. The non-modutils build was 323236 bytes and the build with only modutils was 37764 bytes. This gave a total of 361000 bytes compared to 342628 bytes for a single image, so an extra 18372 bytes of disk space was required for doing such a split.

Splitting the binary in such a simple fashion would likely cost 18K for each of the eleven extra programs. If we changed the policy to have syslogd and klogd run in the same domain (and thus the same program) and have hwclock run with no special privs (IE the domain that runs it needs to have access to */dev/rtc*) then there would only be nine extra programs for a cost of approximately 162K of disk space. This disk space use could be reduced by further optimisation of some of the applets, for example in the case of *ifconfig* the code to check *argv[0]* to determine the applet name could be removed. A simple split in this manner would also make it more difficult for an attacker to make the program perform unauthorized actions. When a single program has */bin/login* functionality as well as */bin/sh* then there is potential for a buffer overflow in the login code to trigger a jump to the shell code under control of the attacker! When the shell is a separate program that can only be entered through a domain transition it is much more difficult to use an attack on the login program to gain further access to the system.

Finally if we have a single Busybox program that includes applets running in different domains we need to make some significant changes to the policy. The default policy has *assert* rules to prevent compilation of a policy that contains mistakes which may lead to security holes. For the domains *getty_t*, *klogd_t*, and *syslogd_t* there are assertions to prevent them from executing other programs without a domain transition, and to prevent those domains being entered through executing files of types other than the matching executable type (this requires that each of those domains have a separate executable type, IE they are not all the same program). Adding policy which requires removing these assertions weakens the security of the base domains and also makes the policy tree different from the default tree which has been audited by many people.

Another way of doing this which uses less disk space is to have a wrapper program such as the following:

```
#include <unistd.h>
#include <string.h>

int main(int argc, char **argv,
         char **envp) {
  /* ptr is the basename of the
   executable that is being run */
  char *ptr = strrchr(argv[0],
                      '/');
  if(!ptr)
    ptr = argv[0];
  else
    ptr++;

  /* basename must match one of
     the allowed applets,
     otherwise it's a hacking
     attempt and we exit   */
  if(strcmp(ptr, "insmod")
  && strcmp(ptr, "modprobe")
  && strcmp(ptr, "rmmod"))
    return 1;
  return execve("/bin/busybox",
                argv, envp);
}
```

This program takes 2912 bytes of disk space. The idea would be to have a copy of it named */sbin/insmod* with type *insmod_exec_t* which has symlinks */sbin/rmmod* and *modprobe* pointing to it. Then when *insmod*, *rmmod*, or *modprobe* is executed an automatic domain transition to the *insmod_t* domain will take place, and then the Busybox program will be executed in the correct context for that applet.

This option is easy to implement, one advantage is that there is no need to change the Busybox program. The fact that the entire Busybox code base is available in privileged domains is a minor weakness. Implementing this takes about 2900 bytes of disk space for each of the nine domains (or seven domains depending on whether you have separate domains for klogd and syslogd and whether you have a domain for hwclock). It will take less than 33K or 27K of disk space (depending on the number of domains). This saves about 130K over the option of having separate binaries for implementing the functionality.

A final option is to have a single program to act as a wrapper and change domains appropriately. Such a program would run in its own domain with an automatic domain transition rule to allow it to be run from all source somains. Then it would look at its parent domain and the type of the symlink to determine the domain of the child process. For example I want to have *insmod* run in domain *insmod_t* when run from *sysadm_t*. So I have an automatic transition rule to transition from *sysadm_t* to the domain for my wrapper (*bbwrap_t*). Then the wrapper determines that its parent domain is *sysadm_t*, determines that the type of the symlink for its *argv[0]* is *insmod_exec_t* and asks the kernel what domain should be entered when a process in *sysadm_t* executes a program of type *insmod_exec_t*, and the answer is *insmod_t*. So the wrapper then uses the *execve_secure()* system call to execute Busybox in the *insmod_t* domain and tell it to run the insmod applet.

I implemented a prototype program for this. For my prototype I used a configuration file to specify the domain transitions instead of asking the kernel. The resulting program was 6K in size (saving 27K of disk space over the multiple-wrapper method, and 156K of disk space over the separate programs method), although it did require some new SE Linux policy to be written which takes a small amount of disk space and kernel memory.

One problem with this method is that it allows security decisions to be made by an application

instead of the kernel. It is preferrable that only the minimum number of applications can make such security decisions. In a typical configuration of SE Linux the only such applications will be *login*, an X login program (in this case *gpe-login*), *cron* (which is not installed in Familiar), and *newrole* (the SE Linux utility for changing the security context which operates in a similar manner to *su*).

The single Busybox wrapper is more of a risk than most of these other programs. The login programs are only executed by the system and can not be run by the user with any elevated privileges which makes them less vulnerable to attack. *Newrole* is well audited and the domains it can transition to are limited by kernel to only include domains that might be used for a login process (dangerous domains such as *login_t* are not permitted).

Due to the risks involved with a single busybox wrapper, and the fact that the benefits of using 6K on disk instead of 33K are very small (and are further reduced by an increase in kernel memory for the larger policy) I conclude that it is a bad idea.

I conclude that the only viable methods of using Busybox on a SE Linux system are having separate wrapper programs for each domain to be entered (taking 33K of extra disk space and requiring minor policy changes), or having entirely separate programs compiled from the Busybox source for each domain (taking approximately 162K of extra disk space with no other problems). Also with some careful optimisation the 162K of overhead could be reduced for the option of splitting the Busybox program. If 162K of disk space can be spared (which should not be a problem with a 32M file system) then splitting Busybox is the right solution.

8 Removed Functionality

A hand-held distribution doesn't require all the features that are needed on bigger machines such as servers, desktop workstations, and laptops. Therefore we can reduce the size of the SE Linux policy and the number of support programs to save disk space and memory.

For a full SE Linux installation there are wrappers for the commands *useradd*, *userdel*, *usermod*, *groupadd*, *groupdel*, *groupmod*, *chfn*, *chsh*, and *vipw*. These can possibly be removed as there is less need for adding, deleting, or modifying users or groups on a hand-held device in the field. These programs would take 27K of disk space if they were included.

A default installation of Familiar does not include support for */etc/shadow*, and therefore there is no need for the wrapper programs for the administrator to modify users' accounts. However I think that the right solution here is to add */etc/shadow* support to Familiar rather than removing functionality from SE Linux. This will slightly increase the size of the login programs.

In a full install of SE Linux there are programs *chsid* and *chcon* to allow changing the security type of files. These are of less importance for a small device. There will be fewer types available, and the effort of typing in long names of security contexts will be unbearable on a touch-screen input device. A hand-held device has to be configured to not require changing the contexts of files, and therefore these programs can be removed.

In the Debian distribution there is support for installing packages on a live server and having the security contexts automatically assigned to the files. As iPaQ's are used in a different environment I believe that there is less need for such upgrades and such support could optionally be removed to save disk space. I have not

written the code for this yet, but I estimate it to be about 100K.

The default policy for SE Linux has separate domains for loading policy and for policy compilation. On the iPaQ we can't compile policy due to not having tools such as *m4* and *make*, so we can skip the compilation program and its policy. Also the policy for a special domain for loading new policy is not needed as the system administration domain *sysadm_t* can be used for this purpose. It is possible to even save 3500 bytes of disk space by not including the program to load the policy (a reboot will cause the new policy to take affect).

A server configuration of SE Linux (or a full workstation configuration) includes the *run_init* program to start daemons in the correct security context. On a typical install of Familiar there are only three daemons, a program to manage X logins, a daemon to manage bluetooth connections, and the PCMCIA cardmgr daemon. For restarting these daemons it should be acceptable to reboot the iPaQ, so *run_init* is not needed.

9 Disk Space and RAM Use

In the section on kernel resource usage I determined that the kernel was using 1364K of RAM for SE Linux with a 583771 byte policy comprising 23,386 rules loaded. Since the time that I performed those tests I reduced the policy to 455,422 bytes and 18,141 rules which would reduce the kernel memory use. I did not do any further tests as it is likely that I will add new functionality which uses the memory I have freed. So I can expect that 1.3M of kernel memory is taken by SE Linux.

The SE Linux policy that is loaded by the kernel takes 67K on disk when compressed. The *file_contexts* file (which specifies the security contexts of files for the initial installation and for upgrades) takes 24K. The kernel binary takes 64K more disk space for the SE Linux kernel. So the kernel code and SE Linux configuration data takes 156K of disk space (most of which is compressed data).

The program *setfiles* is needed to apply the *file_contexts* data to the file system. *Setfiles* takes 20K of disk space. The *file_contexts* file could be reduced in size to 1K if necessary to save extra disk space, but in my current implementation it can not be removed entirely. In Familiar a large number of important system directories (such as */var*) on Familiar are on a *ramfs* file system. I am using *setfiles* to label */mnt/ramfs*. So far it has not seemed beneficial to have a small *file_contexts* file for booting the system and an optional larger one for use when installing new packages or upgrading, but this is an option to save 23K. Another option would be to write a separate program that hard-codes the security contexts for the *ramfs*. It would be smaller than setfiles and not require a emphfile_contexts file, thus saving 30K or more of disk space. Currently this has not seemed worth implementing as I am still in a prototype phase, but it would not be a difficult task. Also if such a program was written then the next step would be to use a *jffs2* loop-back mount to label the root file system on a server before installation to the iPaQ (so that *setfiles* never needs to run on the iPaQ.

The patches for the *gpe-login* and *busybox* programs to provide SE Linux login support and modified *ls*, *ps*, and *id* programs cause the binaries to take a total of 10K extra disk space.

Splitting Busybox into separate programs for each domain will take an estimated 162K of disk space.

The total of this is approximately 348K of additional disk space for a minimal installation of SE Linux on an iPaQ. Adding support for */etc/shadow* and other desirable features may

increase that to as much as 450K depending on the features chosen. However if you use multiple Busybox wrappers instead of splitting Busybox then the disk space for SE Linux could be reduced to less than 213K. If you then replaced *setfiles* for the system boot labeling of the *ramfs* then it could be reduced to 190K.

10 Conclusion

Security Enhanced Linux on a hand-held device can consume less than 1.3M of RAM and less than 400K of disk space (or less than 200K if you really squeeze things). While the memory use is larger than I had hoped it is within a bearable range, and it could potentially be reduced by changing the kernel code to optimise for reduced memory use. The disk space usage is trivial and I don't think it is a concern.

I believe that the benefits of reducing repair and maintenance problems with hand-held devices that are deployed in the field through better security outweigh the disadvantage of increased memory use for many applications.

All source code and security policy code releated to this article will be on my web site [11].

References

[1] *Configuring the SELinux Policy*. Stephen D. Smalley, NAI Labs. `http://www.nsa.gov/selinux/policy2-abs.html`

[2] *Details of SE Linux test machine*, `http://www.coker.com.au/selinux/play.html`

[3] *Meeting Critical Security Objectives with Security-Enhanced Linux*. Peter A. Loscocco, NSA; Stephen D. Smalley, NAI Labs. `http://www.nsa.gov/selinux/ottawa01-abs.html`

[4] *Linux Security Modules*, `http://lsm.immunix.org/`

[5] *User-Mode Linux*, `http://sourceforge.net/projects/user-mode-linux/`

[6] *HP Site for iPaQ Information*, `http://whp-sp-orig.extweb.hp.com/country/us/eng/prodserv/handheld.html/`

[7] *Journalled Flash File System 2*, `http://sources.redhat.com/jffs2/`

[8] *Integrating Flexible Support for Security Policies into the Linux Operating System*. Peter A. Loscocco, NSA; Stephen D. Smalley, NAI Labs. `http://www.nsa.gov/selinux/freenix01-abs.html`

[9] *Familiar Linux distribution for hand-held devices*, `http://familiar.handhelds.org/`

[10] *Busybox - Swiss Army Knife of Embedded Linux*, `http://busybox.net/`

[11] *My SE Linux Web Pages*, `http://www.coker.com.au/selinux/`

Strong Cryptography in the Linux Kernel

Discussion of the past, present, and future of strong cryptography in the Linux kernel

Jean-Luc Cooke
CertainKey Inc.
jlcooke@certainkey.com

David Bryson
Tsumego Foundation
david@tsumego.com
PGP: 0x74B61620

Abstract

In 2.5, strong cryptography has been incorporated into the kernel. This inclusion was a result of several motivating factors: remove duplicated code, harmonize IPv6/IPSec, and the usual crypto-paranoia. The authors will present the history of the Cryptographic API, its current state, what kernel facilities are currently using it, which ones should be using it, plus the new future applications including:

1. Hardware and assembly crypto drivers
2. Kernel module code-signing
3. Hardware random number generation
4. Filesystem encryption, including swap space.

1 History of Cryptography inside the kernel

The Cryptographic API came about from two somewhat independent projects: the international crypto patch last maintained by Herbert Valerio Riedel and the requirments for IPv6.

The international crypto patch (or 'kerneli') was written by Alexander Kjeldaas and intended for filesystem encryption, it has grown to also optionally replace duplicated code in the UNIX random character device (/dev/*random). This functionality could not be incorporated into the main line kernel at the time because kernel.org was hosted in a nation with repressive cryptography export regulations. These regulations have since been relaxed to permit open source cryptographic software to travel freely from kernel.org's locality.

The 2.5 kernel, at branch time, did not include any built in cryptography. But with the advent of IPv6 the killer feature of kernel space cryptography has shown itself. The IPv6 specification contains a packet encryption industry standard for virtual private network (VPN) technology. The 2.5 kernel was targeted to have a full IPv6 stack–this included packet encryption. The IPv6 and kernel maintainers in their infinite wisdom (!) saw an opportunity to remove duplicated code and encouraged the kerneli.org people to play with others.

And so, strong cryptography is now at the disposal of any 2.5+ kernel hacker.

2 Why bring cryptography into our precious kernel?

Cryptography, in one form or another, has existed in the main line kernel for many versions. The introduction of the random device driver

by Theodore Ts'o integrated two well known cryptographic (digest) algorithms, MD5 and SHA-1. Other forms of cryptography were introduced with the loopback driver (also written by Theodore Ts'o) these included an XOR and DES implementation for primitive filesystem encryption.

The introduction of cryptography for filesystem encryption, coupled with the kerneli patches, allowed users to hook the loopback device up to a cipher of their choosing. Thus providing a solution for secure hard disk storage on Linux.

With the advent of IPSec the introduction of crypto into the kernel makes setting up encrypted IP connections extremely easy. Previous implementatinons have used userspace hooks and required compilcated configuration to setup properly. With IPSec being inside the kernel much of those tasks can be automated.

More advanced features for cryptography in the kernel will be explained throughout this paper.

2.1 Example Code

The use of the API is quite simple and straightforward. The following lines of code show a basic use of the MD5 hash algorithm on a scatterlist.

```
#include <linux/crypto.h>

struct scatterlist sg[2];
char result[128];
struct crypto_tfm *tfm;

tfm = crypto_alloc_tfm("md5", 0);
if (tfm == NULL)
  fail();

/* copy data into */
/* the scatterlists */
crypto_digest_init(tfm);
crypto_digest_update(tfm, &sg, 2);
crypto_digest_final(tfm, result);

crypto_free_tfm(tfm);
```

Ciphers are implemented in a similar fashion but must set a key value (naturally) before doing any encryption or decryption operations.

```
#include <linux/crypto.h>

int len;
char key[8];
char result[64];
struct crypto_tfm *tfm;
struct scatterlist sg[2];

tfm = crypto_alloc_tfm("des", 0);
if (tfm == NULL)
  fail();

/* place key data into key[] */
crypto_cipher_setkey(tfm, key, 8);

/* copy data into scatterlists */

/* do in-place encryption */
crypto_cipher_encrypt(tfm,sg[0],
         sg[0],len);
crypto_free_tfm(tfm);
```

The encryption and decryption functions are capable of doing in-place operations as well as in/out (separate source and destination) operations. This example shows in-place opertaion. By changing the encrypt line to:

```
crypto_cipher_encrypt(tfm,
       sg[0], sg[1], len);
```

the code then becomes an in/out operation.

3 Kernel module code-signing

Signing of kernel modules has been a desired addition to the kernel for a long time. Many

people have attempted to do some kind of authenticated kernel module signing/encryption but usually by the means of an external user-mode program. With the movement of the module loader into the kernel in the 2.4 series a truly secure module loader is possible. The authors would like to propose a method for trusted module loading.

To create the secure module structure we need a way of designating a module as trusted. During compile time, a token can be created for each module. The token contains two identifiers.

- Time stamp token, denoting module creation time.
- A secure hash of the module in its compiled state.

After these three tokens are created they are encrypted by an internal private key (protected by a separate password of course) bound to the kernel. The encrypted file is then stored in a file on the local disk.

Loading of the module occurs as follows.

1. A request to load module `rot13` is made by the system.
2. The kernel reads the encrypted file for module `rot13`.
3. Using the kernels public key the file is decrypted, and the tokens are placed in memory
4. A hash is computed against the file on resident disk of module `rot13` and compared against the signed token.
5. If the hashes are equal the module is trusted and code loaded into memory.

This allows for a large degree of flexibility. Anybody on the system who has access to the kernels public key can verify the validity of the modules. Plus the kernel does not need to have the private key in memory to authenticate since the public key can do the decryption. Thus reducing the time that the private key is stored in resident memory unencrypted.

However this approach can only protect a system to a point. If a malicious user is on your system and is at the stage where they can load modules (root access) this will only slow them down. Nothing prevents them from compiling a new kernel with a 'dummy' module loader that skips this check (solutions to this problem welcome!).

This system requires that the kernel contain functionality to support arbitrarily large integers and perform asymmetric cryptography operations. Currently, there is preliminary code that supports this functionality, but has yet to be formally tested or introduced to the community.

4 Cryptographic Hardware and Assembly Code

A new exciting aspect of cryptography in the kernel is the ability to use hardware based cryptographic accelerators. Many vendors offer a variety of solutions for speeding up cryptographic routines for symmetric and asymmetric operations.

The chips provide cheap, efficient, and fast cryptographic processors. These can be purchased for as little as $80.00USD and offer a considerable speedup for the algorithms they support. The proposed method of integrating hardware and assembly is to have the chip or card register its capabilities with the API.

This way the API can serve as a front end to the

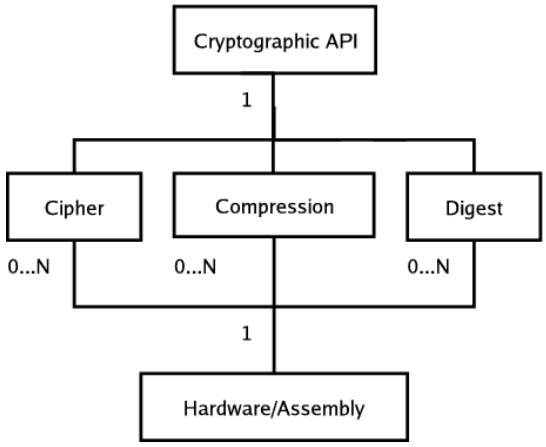


Figure 1: The proposed hardware interface model

hardware driver module. Instead of a the hardware registering "aes" it would register "aes-<chipset name>" with the API. Calls to the API can then specify which implementations of the ciphers that are desired depending on what performance is needed.

As of this writing (*May 2003*) there is also no way to query the API for the fastest method it has for computing a cipher. There is in the design stage an asynchronous system for dynamically receiving requests and distributing them to the various pieces of hardware (or software) present on the system. The OpenBSD API and cryptography sub-system is being used as a reference model.

This method would allow users of the API to send queries to a scheduler with a call similar to the current interface, but adding using the cypher name `aes_fastest` or `aes_hardware`. The scheduler then sends the requested command to a piece of hardware that is waiting for requests, and fulfills the requested hardware requirements.

Drivers that are currently finished and/or under development include:

- Hifn series processors 7751 and 7951.
- Motorola MPC190 and MPC184 series.
- Broadcom BCM582[0|1|3] series.

4.1 Hardware Random Number Generation

Most cryptographic hardware and lately some motherboards have been including a hardware random number generation chip. These are a wonderful source of generating entropy because they are both fast and produce very random data. A set of free-running oscillators usually generates the data. The oscillators frequency clocks drift relative to each other and to the internal clock of the chipset. Thus, producing randomly flipped bits in a specified 'RNG' register.

Random number generation in the kernel uses interrupts from the keyboard, mouse, hard disk, and network interfaces to create an entropy pool. This entropy pool produces the output of `/dev/random` and the less secure `/dev/urandom`.

The current interface is missing a way to add random data from an arbitrary hardware source. By using tying the random driver into the Cryptographic API the random driver can gain both extra sources of entropy and the acceleration from making its MD5 and SHA-1 functions available for hardware execution. The result would be a faster and better entropy pool for random data generation.

5 Filesystem encryption

By far, the cryptographic API has the largest user-base with filesystem encryption. Several distributions have shipped with support for filesystem loopback encryption for over a year. Let us take a moment to explore the details of

filesystem encryption. When a write or read request occurs in the kernel the information destined for a device passes through the VFS layer, then down through the device driver layer and onto the physical media.

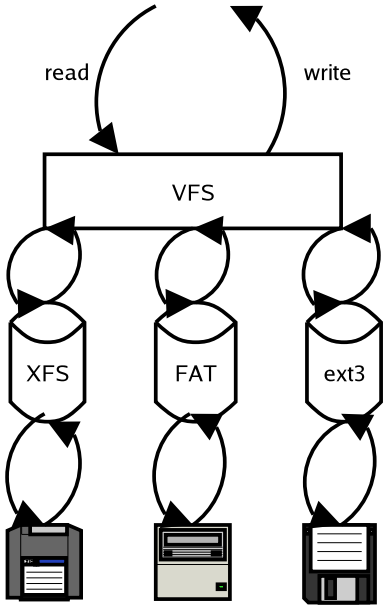


Figure 2: A diagram of the Linux VFS

Looking for a place to encrypt the data isn't easy, we could intercept the information at the VFS layer, but the result is encrypted data with plaintext metadata. Thus giving an attacker an edge, for example being able to track down your `/tmp/.SCO_source_code` directory and begin attacking the encrypted data there. The next place to intercept the plaintext data would be at the filesystem level. But writing per filesystem hooks to encrypt both filesystem data and filesystem metadata would be a nightmare to implement, not to mention a horrible design decision. So the only place left is somewhere outside of the filesystem code, but before the data is passed to the device driver for the media. Enter loopback drivers.

The loopback device driver in Linux allows us to send the data (plaintext) and metadata (plaintext) through a layer of memory copying before it is written to a device. Here is where the encryption will be done—this way all data written to the device can be encrypted instead of just the filesystem data.

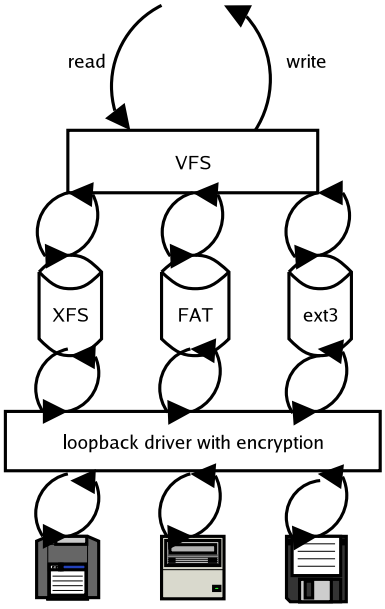


Figure 3: A diagram of the loopback encryption layer

By using the loopback driver an added level of flexibility is added. Users can have their home directories stored as large encrypted files on the primary drive. These would then be loaded via the cryptographic loopback driver upon login and unmount when the user exits all sessions.

5.1 Swap memory Encryption

Encryption of swap is a difficult problem to approach. Any system in which the filesystem data is encrypted has the chance of the data being moved out to swap memory when the OS gets low on RAM. This can easily be solved by 'locking' all memory into RAM and not allowing it to be swapped to a physical media with the `mlock()` function. However, this vastly reduces the usability of the system (Linux tends to kill processes when out of

memory). In the past, Linux has implemented encrypted swap with a loopback device running through the swap accesses. This approach works, but is slow and cumbersome to implement.

What is needed is a policy for encrypting **all** pages swapped to disk. The OpenBSD community has had a similar policy for a long time and feels that the performance loss (roughly 2.5 times longer to write a page to disk) is worth the added security.

The 2.4 series crypto (not a part of the mainline the kernel) named the "International Kernel Cryptography Patch" included a loopback encryption driver. The driver had limited features but did the job of encrypting data fairly well. In the 2.5 series driver there have been some performance improvements, like multithreaded (and SMP) reading from loopback devices, and code readability improvements.

6 Userspace Access

Access to the API is not currently possible from user space. Discussions over how to implement this have come up with a variety of proposals. The current direction is to have a device provide access to the API via ioctls.

Compatibility with the already mature OpenBSD API has been suggested. This would decrease application porting time to almost nothing.

7 Final Comments

Thus far the 2.5 Cryptographic API has been under constant development with the adding of new ciphers and functionality since its inclusion in the kernel. The API is young and has promising plans to expand, hopefully the authors of this paper have given you an adequate intro to the capabilites of the API.

References

[Bryson] David Bryson, *The Linux CryptoAPI: A Users Perspective*. 16 May 2002. `http://www.kerneli.org/howto/index.php`.

[Morris] Morris, James, and David S. Miller. *Scatterlist Cryptographic API*. 10 May. 2003. Linux Kernel Documentation linux/Documentation/crypto/api-intro.txt.

[Steve] Steve, `steve@trevithick.net`. *Re: [CryptoAPI-devel] Re: hardware crypto support for userspace ?* 11 Dec. 2002 via `http://www.kerneli.org/pipermail/cryptoapi-devel`.

[Hifn] Hifn, Inc. *7951 Data Sheet - Device Specification*. 2 June. 2001. `http://www.hifn.com/docs/a/DS-0028-02-7951.pdf`.

[Provos] Provos, Niels. *Encrypting Virtual Memory*. `http://www.openbsd.org/papers/swapencrypt.ps`.

Porting drivers to the 2.5 kernel

Jonathan Corbet
LWN.net
corbet@lwn.net

Abstract

The 2.5 development series has brought with it the usual large set of changes to the internal driver API. The end result is a kernel that is far more pleasant to program for, and a more robust and reliable system. The cost of all these changes, of course, is that kernel code—including device drivers—must be updated to work under the new regime. This paper will give an overview of what has changed in the internal kernel API, why the changes were made, and what must be done to make drivers work again. Some familiarity with kernel programming is assumed.

1 Introduction

2.5 was a busy time for kernel developers. Much work was done to make kernel code more reliable and less susceptible to common problems. There has also been a large emphasis on improved performance on high-end systems. The end result is that almost no part of the kernel—and almost no internal API—was left untouched. The degree of change varies from relatively small (for network drivers, for example) to extreme (block drivers). An awareness of these changes is helpful for anybody who is interested in how kernel development is proceeding, and crucial for anybody who must make code work with the new kernel.

This paper will start with the basic changes which affect all drivers—module loading, memory allocation, etc. Later sections will get into the more advanced topics, including the block layer, and memory management. Space constraints make it impossible to get into much detail here. A series of documents can be found at the web site listed at the end of this paper; those documents explore the topics found below in much greater depth, and will be kept current as the kernel evolves.

2 Loadable modules

The module loader was completely replaced in 2.5; the new implementation works almost entirely within the kernel. Interestingly, moving the module loader into the kernel resulted in a net reduction in kernel code. This development has forced a few changes in how modules work, however.

2.1 Hello world

The obvious place to start is the classic “hello world” program, which, in this context, is implemented as a kernel module. The 2.4 version of this module looked like:

```
#define MODULE
#include <linux/module.h>
#include <linux/kernel.h>

int init_module(void)
{
    printk(KERN_INFO "Hello, world\n");
    return 0;
}

void cleanup_module(void)
```

```
{
    printk(KERN_INFO "Goodbye cruel world\n");
}
```

One would not expect that something this simple and useless would require much in the way of changes, but, in fact, this module will not quite work in a 2.5 kernel. So what do we have to do to fix it up?

The first change is relatively insignificant; the first line:

```
#define MODULE
```

is no longer necessary, since the kernel build system defines it for you.

The biggest problem with this module, however, is that you have to explicitly declare your initialization and cleanup functions with `module_init` and `module_exit`, which are found in `<linux/init.h>`. You really should have done that for 2.4 as well, but you could get away without it as long as you used the names `init_module` and `cleanup_module`. You can still get away with it, but the new module code broke this way of doing things once, and could do so again. It's time to bite the bullet and do things right.

With these changes, "hello world" now looks like:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

static int hello_init(void) {
  printk(KERN_ALERT "Hello, world\n");
  return 0;
}

static void hello_exit(void) {
  printk(KERN_ALERT "Goodbye, cruel world\n");
}

module_init(hello_init);
module_exit(hello_exit);
```

This module will now work—the "Hello, world" message shows up in the system log file. At least, once you have succeeded in building the module properly...

2.2 Module compilation

One result of the module changes (combined with significant changed in the kernel build mechanism) is that compiling loadable modules has gotten a bit more complicated. In the 2.4 days, a makefile for an external module could be put together in just about any old way; the job of creating a loadable module was handled in a single, simple compilation step. All you really needed was a handy set of kernel headers to compile against.

With the 2.5 kernel, you still need those headers. You also, however, need a configured kernel source tree and a set of makefile rules describing how modules are built. All this is required because the new module loader needs some additional symbols defined at compilation time; because all modules must now go through a linking step (even single-file modules); and because the new modversions implementation requires a separate processing step.

One could certainly, with some effort, write a new, standalone makefile which would handle the above issues. But that solution, along with being a pain, is also brittle; as soon as the module build process changes again, the makefile will break. Eventually that process will stabilize, but, for a while, further changes are almost guaranteed.

So, now that you are convinced that you want to use the kernel build system for external modules, how is that to be done? The first step is to learn how kernel makefiles work in general; `makefiles.txt` from a recent kernel's `Documentation/kbuild` directory is recommended reading. The makefile magic needed for a simple kernel module is minimal, however. In fact, for a single-file module, a single-line makefile will suffice:

```
obj-m := module.o
```

(where `module` is replaced with the actual name of the resulting module, of course). The kernel build system, on seeing that declaration, will compile `module.o` from `module.c`, link it with `vermagic.o` from the kernel tree, and leave the result in `module.ko`, which can then be loaded into the kernel.

A multi-file module is almost as easy:

```
obj-m := module.o
module-objs := file1.o file2.o
```

In this case, `file1.c` and `file2.c` will be compiled, then linked into `module.ko`.

Of course, all this assumes that you can get the kernel build system to read and deal with your makefile. The magic command to make that happen is something like the following:

```
make -C /usr/src/linux \
        SUBDIRS=\$PWD modules
```

Where `/usr/src/linux` is the path to the source directory for the target kernel. This command causes `make` to head over to the kernel source to find the top-level makefile; it then moves back to the original directory to build the module of interest.

Of course, typing that command could get tiresome after a while. A trick posted by Gerd Knorr can make things a little easier, though. By looking for a symbol defined by the kernel build process, a makefile can determine whether it has been read directly, or by way of the kernel build system. So the following will build a module against the source for the currently running kernel:

```
ifneq ($(KERNELRELEASE),)
obj-m := module.o
else
KDIR  := /lib/modules/$(shell uname -r)/build
PWD   := $(shell pwd)
default:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
endif
```

Now a simple “make” will suffice. The makefile will be read twice; the first time it will simply invoke the kernel build system, while the actual work will get done in the second pass. A makefile written in this way is simple, and it should be robust with regard to kernel build changes.

2.3 Module parameters

The old `MODULE_PARM` macro, which used to specify parameters which can be passed to the module at load time, is no more. The new parameter declaration scheme add type safety and new functionality, but at the cost of breaking compatibility with older modules.

Modules with parameters should now include `<linux/moduleparam.h>` explicitly. Parameters are then declared with `module_param`:

```
module_param(name, type, perm);
```

Where `name` is the name of the parameter (and of the variable holding its value), `type` is its type, and `perm` is the permissions to be applied to that parameter’s sysfs entry. The `type` parameter can be one of `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `charp`, `bool` or `invbool`. That type will be verified during compilation, so it is no longer possible to create confusion by declaring module parameters with mismatched types. The plan is for module parameters to appear automatically in sysfs, but that feature had not been implemented as of 2.5.69; for now, the safest alternative is to set `perm` to zero, which means “no sysfs entry.”

If the name of the parameter as seen outside the module differs from the name of the variable

used to hold the parameter's value, a variant on `module_param` may be used:

```
module_param_named(name, value, type, perm);
```

Where `name` is the externally-visible name and `value` is the internal variable.

String parameters will normally be declared with the `charp` type; the associated variable is a `char` pointer which will be set to the parameter's value. If you need to have a string value copied directly into a `char` array, declare it as:

```
module_param_string(name, string, len, perm);
```

Usually, `len` is best specified as `sizeof(string)`.

2.4 The module use count

In 2.4 and prior kernels, modules maintained their "use count" with macros like `MOD_INC_USE_COUNT`. The use count, of course, is intended to prevent modules from being unloaded while they are being used. This method was always somewhat error prone, especially when the use count was manipulated inside the module itself. In the 2.5 kernel, reference counting is handled differently.

The only safe way to manipulate the count of references to a module is outside of the module's code. Otherwise, there will always be times when the kernel is executing within the module, but the reference count is zero. So this work has been moved outside of the modules, and life is generally easier for module authors.

Any code which wishes to call into a module (or use some other module resource) must first attempt to increment that module's reference count:

```
int try_module_get(&module);
```

It is also necessary to look at the return value; a zero return means that the try failed (perhaps the module is being unloaded), and the module should not be used.

A reference to a module can be released with `module_put()`.

Again, modules will not normally have to manage their own reference counts. The only exception may be if a module provides a reference to an internal data structure or function that is not accounted for otherwise. In that (rare) case, a module could conceivably call `try_module_get()` on itself.

2.5 Exporting symbols

In 2.5, module symbols are not exported by default. Chances are that change will cause few problems. When you get a chance, however, you can remove `EXPORT_NO_SYMBOLS` lines from your module source. Exporting no symbols is now the default, so `EXPORT_NO_SYMBOLS` is a no-op.

The 2.4 `inter_module_` functions have been deprecated as unsafe. The `symbol_get()` function exists for the cases when normal symbol linking does not work well enough. Its use requires setting up weak references at compile time, and is beyond the scope of this document.

2.6 Kernel version checking

2.4 and prior kernels would include, in each module, a string containing the version of the kernel that the module was compiled against. Normally, modules would not be loaded if the compile version failed to match the running kernel.

In 2.5, things still work mostly that way. The kernel version is loaded into a separate, "link-once" ELF section, however, rather than be-

ing a visible variable within the module itself. As a result, multi-file modules no longer need to define `__NO_VERSION__` before including `<linux/module.h>`.

The new "version magic" scheme also records other information, including the compiler version, SMP status, and preempt status; it is thus able to catch more incompatible situations than the old scheme did.

3 The device model

One of the more significant changes in the 2.5 development series is the creation of the integrated device model. The device model was originally intended to make power management tasks easier through the maintenance of a representation of the host system's hardware structure. A certain amount of mission creep has occurred, however, and the device model is now closely tied into a number of device management tasks—and other kernel functions as well.

The device model presents a bit of a steep learning curve when first encountered. The fact that the whole thing is still (as of 2.5.69) in a state of fairly serious flux doesn't help, especially considering that the documentation is, in many cases, a few revisions behind the actual code. But the underlying concepts are not *that* hard to understand, and driver programmers will benefit from a grasp of what's going on.

The fundamental task of the driver model is to maintain a set of internal data structures which reflect the architecture and state of the underlying system. Among other things, the driver model tracks:

- Which devices exist in the system, what power state they are in, what bus they are attached to, and which driver is responsible for them.
- The bus structure of the system; which buses are connected to which others (i.e., a USB controller can be plugged into a PCI bus), and which devices each bus can potentially support (along with associated drivers), and which devices actually exist.
- The device drivers known to the system, which devices they can support, and which bus type they know about.
- What *kinds* of devices ("classes") exist, and which real devices of each class are connected. The driver model can thus answer questions like "where is the mouse (or mice) on this system?" without the need to worry about how the mouse might be physically connected.
- And many other things.

Underneath it all, the driver model works by tracking system configuration changes (hardware and software) and maintaining a complex "web woven by a spider on drugs" data structure to represent it all.

Many driver programmers will be able to get away with ignoring the device model altogether; most of the gory details are handled at the bus level. There are times, however, when an understanding of what's going on can be useful. A full discussion of the device model would require a talk of its own (indeed, there are two on the OLS schedule); suffice to say, for now, that details can be found on the web site.

4 Support interfaces

Now that we know how to compile a module, it's time to look at the various other changes it

will need to be adapted for. We'll start with various low-level support interfaces used by many or most drivers (and other modules).

4.1 Memory allocation

The 2.5 development series has brought relatively few changes to the way device drivers will allocate and manage memory. In fact, most drivers should work with no changes in this regard. There are a few improvements that have been made, however, that are worth a mention. These include some changes to page allocation, and the new "mempool" interface.

4.1.1 Allocation flags

The old `<linux/malloc.h>` include file is gone; it is now necessary to include `<linux/slab.h>` instead.

The `GFP_BUFFER` allocation flag is gone (it was actually removed in 2.4.6). That will bother few people, since almost nobody used it. For reference, here is the full set of 2.5 allocation flags, from the most restrictive to the least:

`GFP_ATOMIC`: a high-priority allocation which will not sleep; this is the flag to use in interrupt handlers and other non-blocking situations.

`GFP_NOIO`: blocking is possible, but no I/O will be performed.

`GFP_NOFS`: no filesystem operations will be performed.

`GFP_KERNEL`: a regular, blocking allocation.

`GFP_USER`: a blocking allocation for user-space pages.

`GFP_HIGHUSER`: for allocating user-space pages where high memory may be used.

The `__GFP_DMA` and `__GFP_HIGHMEM` flags still exist and may be added to the above to direct an allocation to a particular memory zone. In addition, 2.5.69 added some new modifiers:

`__GFP_REPEAT`: This flag tells the page allocater to "try harder," repeating failed allocation attempts if need be. Allocations can still fail, but failure should be less likely.

`__GFP_NOFAIL`: Try even harder; allocations with this flag must not fail. Needless to say, such an allocation could take a long time to satisfy.

`__GFP_NORETRY`: Failed allocations should not be retried; instead, a failure status will be returned to the caller immediately.

The `__GFP_NOFAIL` flag is sure to be tempting to programmers who would rather not code failure paths, but that temptation should be resisted most of the time. Only allocations which truly cannot be allowed to fail should use this flag.

4.1.2 Page-level allocation

For page-level allocations, the `alloc_pages()` and `get_free_page()` functions (and variants) exist as always. They are now defined in `<linux/gfp.h>`. There are a few new ones as well. On NUMA systems, the allocator will do its best to allocate pages on the same node as the caller. To explicitly allocate pages on a different NUMA node, use:

```
struct page *
alloc_pages_node(int node_id,
                 unsigned int gfp_mask,
                 unsigned int order);
```

4.1.3 vmalloc_to_page()

Occasionally, it is necessary to find a `struct page` pointer for a page obtained from `vmalloc()`; usually this need arises in the implementation of `nopage()` methods. In the past, a driver had to walk through the page tables to find this pointer. As of 2.5.5 (and 2.4.19), however, all that is needed is a call to:

```
struct page *vmalloc_to_page(void *address);
```

This call is not a variant of `vmalloc()`—it allocates no memory. It simply returns a pointer to the `struct page` associated with an address obtained from `vmalloc()`.

4.1.4 Memory pools

Memory pools were one of the very first changes in the 2.5 series—they were added to 2.5.1 to support the new block I/O layer. The purpose of mempools is to help out in situations where a memory allocation must succeed, but sleeping is not an option. To that end, mempools pre-allocate a pool of memory and reserve it until it is needed. Mempools make life easier in some situations, but they should be used with restraint; each mempool takes a chunk of kernel memory out of circulation and raises the minimum amount of memory the kernel needs to run effectively.

A full discussion of mempools doesn't fit into this document; see the web site for details on their use.

4.2 Per-CPU variables

The 2.5 kernel makes extensive use of per-CPU data—arrays containing one object for each processor on the system. Per-CPU variables are not suitable for every task, but, in situations where they can be used, they do offer a couple of advantages:

- Per-CPU variables have fewer locking requirements since they are (normally) only accessed by a single processor.
- Restricting each processor to its own area eliminates cache line bouncing and improves performance.

Examples of per-CPU data in the 2.5 kernel include lists of buffer heads, lists of hot and cold pages, various kernel and networking statistics (which are occasionally summed together into the full system values), timer queues, and so on. There are currently no drivers using per-CPU values, but some applications (i.e., networking statistics for high-bandwidth adapters) might benefit from their use. See the web site listed at the end of this paper for a full description of how to use per-CPU data.

4.3 Timekeeping

One might be tempted to think that the basic task of keeping track of the time would not change that much from one kernel to the next. And, in fact, most kernel code which worries about times (and time intervals) will likely work unchanged in the 2.5 series. Code which gets into the details of how the kernel manages time may well need to adapt to some changes, however.

4.3.1 Internal clock frequency

One change which *shouldn't* be problematic for most code is the change in the internal clock rate on the x86 architecture. In previous kernels, `HZ` was 100; in 2.5 it has been bumped up to 1000. If your code makes any assumptions about what `HZ` really was (or, by extension, what `jiffies` really signified), you may have to make some changes now.

4.3.2 Kernel time variables

With a 1KHz clock, a 32-bit `jiffies` will overflow in just under 50 days, leading to occasional problems. So the 2.5 kernel has a new counter called `jiffies_64`. With 64 bits to work with, `jiffies_64` will not wrap around in a time frame that need concern most of us—at least until some future kernel starts using a petahertz internal clock.

For what it's worth, on most architectures, the classic, 32-bit `jiffies` variable is now just the least significant half of `jiffies_64`. Note that, on 32-bit systems, a 64-bit `jiffies` value raises concurrency issues. It is deliberately not declared as a `volatile` value (for performance reasons), so the possibility exists that code like:

```
u64 my_time = jiffies_64;
```

could get an inconsistent version of the variable, where the top and bottom halves do not match. To avoid this possibility, code accessing `jiffies_64` should use `xtime_lock`, which is the new seqlock type as of 2.5.60. In most cases, though, it will be easier to just use the convenience function provided by the kernel:

```
#include <linux/jiffies.h>
u64 my_time = get_jiffies_64();
```

Users of the internal `xtime` variable will notice a couple of similar changes. One is that `xtime`, too, is now protected by `xtime_lock` (as it is in 2.4 as of 2.4.10), so any code which plays around with disabling interrupts or such before accessing `xtime` will need to change. The best solution is probably to use:

```
struct timespec current_kernel_time(void);
```

which takes care of locking for you. `xtime` also now is a `struct timespec` rather than `struct timeval`; the difference being that the sub-second part is called `tv_nsec`, and is in nanoseconds.

4.3.3 Timers

The kernel timer interface is essentially unchanged since 2.4, with one exception. The new function:

```
void add_timer_on(struct timer_list *timer,
                  int cpu);
```

will cause the timer function to run on the given CPU with the expiration time hits.

4.3.4 Delays

The 2.5 kernel includes a new macro `ndelay()`, which delays for a given number of nanoseconds. It can be useful for interactions with hardware which insists on very short delays between operations. On most architectures, however, `ndelay(n)` is equal to `udelay(1)` for waits of less than one microsecond.

4.4 Delayed tasks and workqueues

The longstanding task queue interface was removed in 2.5.41; in its place is a new "workqueue" mechanism. Workqueues are

very similar to task queues, but there are some important differences. Among other things, each workqueue has one or more dedicated worker threads (one per CPU) associated with it. So all tasks running out of workqueues have a process context, and can thus sleep. Note that access to user space is not possible from code running out of a workqueue; there simply is no user space to access. Drivers can create their own work queues—with their own worker threads—but there is a default queue (for each processor) provided by the kernel that will work in most situations.

See the web site for a detailed discussion of workqueues.

4.5 DMA support

The direct memory access (DMA) support layer has been extensively changed in 2.5, but, in many cases, device drivers should work unaltered. For developers working on new drivers, or for those wanting to keep their code current with the latest API, there are a fair number of changes to be aware of.

The most evident change is the creation of the new generic DMA layer. A new set of generic DMA functions has been added which is intended to provide a DMA support API that is not specific to any particular bus. The new functions look much like the older PCI-based ones; changing from one API to the other is a fairly automatic job. The full set of equivalences between old and new DMA functions may be found on the web site.

There has been one significant change in the creation of scatter/gather streaming DMA mappings. The 2.4 version of `struct scatterlist` used a `char *` pointer (called `address`) for the buffer to be mapped, with a `struct page` pointer that would be used only for high memory addresses. In 2.5, the `address` pointer is gone, and all scatterlists must be built using `struct page` pointers.

Other developments of interest here include support for DAC (64-bit) PCI DMA, an interface for explicitly non-coherent mappings, and PCI pools.

5 Kernel preemption

One significant change introduced in 2.5 is the preemptible kernel. Previously, a thread running in kernel space would run until it returned to user mode or voluntarily entered the scheduler. In 2.5, if preemption is configured in, kernel code can be interrupted at (almost) any time. As a result, the number of challenges relating to concurrency in the kernel goes up. But this is actually not that big a deal for code which was written to handle SMP properly—most of the time. If you have not yet gotten around to implementing proper locking for your 2.4 driver, kernel preemption should give you yet another reason to get that job done.

The preemptible kernel means that your driver code can be preempted whenever the scheduler decides there is something more important to do. “Something more important” could include re-entering your driver code in a different thread. There is one big, important exception, however: preemption will not happen if the currently-running code is holding a spinlock. Thus, the precautions which are taken to ensure mutual exclusion in the SMP environment also work with preemption. So most (properly written) code should work correctly under preemption with no changes.

That said, code which makes use of per-CPU variables should take extra care. A per-CPU variable may be safe from access by other processors, but preemption could create races on the same processor. Code using per-CPU vari-

ables should, if it is not already holding a spinlock, disable preemption if the possibility of concurrent access exists. Usually, macros like `get_cpu_var()` should be used for this purpose.

Should it be necessary to control preemption directly (something that should happen rarely), some macros in `<linux/preempt.h>` will be helpful. A call to `preempt_disable()` will keep preemption from happening, while `preempt_enable()` will make it possible again. If you want to re-enable preemption, but don't want to get preempted immediately (perhaps because you are about to finish up and reschedule anyway), `preempt_enable_no_resched()` is what you need.

One interesting side-effect of the preemption work is that it is now much easier to tell if a particular bit of kernel code is running within some sort of critical section. A single variable in the task structure now tracks the preemption, interrupt, and softirq states. A new macro, `in_atomic()`, tests all of these states and returns a nonzero value if the kernel is running code that should complete without interruption.

6 Sleeping and waiting

Contrary to expectations, the classic functions `sleep_on()` and `interruptible_sleep_on()` were not removed in the 2.5 series. It seems that they are still needed in a few places where (1) taking them out is quite a bit of work, and (2) they are actually used in a way that is safe. Most authors of kernel code should, however, pretend that those functions no longer exist. There are very few situations in which they can be used safely, and better alternatives exist.

6.1 Safe sleeping

Most of those alternatives have been around since 2.3 or earlier. In many situations, one can use the `wait_event()` macros:

```
DECLARE_WAIT_QUEUE_HEAD(queue);
wait_event(queue, condition);
int wait_event_interruptible (queue, condition);
```

These macros work as they did in 2.4: `condition` is a boolean condition which will be tested within the macro; the wait will end when the condition evaluates true. It is worth noting that these macros have moved from `<linux/sched.h>` to `<linux/wait.h>`, which seems a more sensible place for them. There is also a new one:

```
int wait_event_interruptible_timeout(
        queue, condition, timeout);
```

which will terminate the wait if the timeout expires.

In many situations, `wait_event()` does not provide enough flexibility—often because tricky locking is involved. The longstanding "manual sleep" method can be used in these cases. In 2.5, however, a set of helper functions has been added which makes this task easier. The modern equivalent of a manual sleep looks like:

```
DECLARE_WAIT_QUEUE_HEAD(queue);
DEFINE_WAIT(wait);

while (! condition) {
    prepare_to_wait(&queue, &wait,
                    TASK_INTERRUPTIBLE);
        if (! condition)
            schedule();
    finish_wait(&queue, &wait)
}
```

Use `prepare_to_wait_exclusive()` instead when an exclusive wait is needed. Note that the new macro `DEFINE_WAIT()` is used here, rather than `DECLARE_WAITQUEUE()`. The former should be used when the wait queue entry is to be used with `prepare_to_wait()`, and should

probably *not* be used in other situations unless you understand what it is doing (which we'll get into next).

6.2 Wait queue changes

In addition to being more concise and less error prone, using `prepare_to_wait()` can yield higher performance in situations where wakeups happen frequently. This improvement is obtained by causing the process to be removed from the wait queue immediately upon wakeup; that removal keeps the process from seeing multiple wakeups if it doesn't otherwise get around to removing itself for a bit.

The automatic wait queue removal is implemented via a change in the wait queue mechanism. Each wait queue entry now includes its own "wake function," whose job it is to handle wakeups. The default wake function (which has the surprising name `default_wake_function()`), behaves in the customary way: it sets the waiting task into the `TASK_RUNNING` state and handles scheduling issues. The `DEFINE_WAIT()` macro creates a wait queue entry with a different wake function, `autoremove_wake_function()`, which automatically takes the newly-awakened task out of the queue.

And that, of course, is how `DEFINE_WAIT()` differs from `DECLARE_WAITQUEUE()`—they set different wake functions. How the semantics of the two differ is not immediately evident from their names, but that's how it goes. (The new runtime initialization function `init_wait()` differs from the older `init_waitqueue_entry()` in exactly the same way).

If need be, you can define your own wake function—though the need for that should be quite rare (the only user, currently, is the support code for the `epoll()` system calls). See the web site for details on how this is done.

One other change that most programmers won't notice: a bunch of wait queue cruft from 2.4 (two different kinds of wait queue lock, wait queue debugging) has been removed from 2.5.

6.3 Completions

Completions are a simple synchronization mechanism that is preferable to sleeping and waking up in some situations. If you have a task that must simply sleep until some process has run its course, completions can do it easily and without race conditions. They are not strictly a 2.5 feature, having been added in 2.4.7, but they merit a quick summary here.

A completion is, essentially, a one-shot flag that says "things may proceed." Working with completions requires including `<linux/completion.h>` and creating a variable of type `struct completion`. This structure may be declared and initialized statically with:

```
DECLARE_COMPLETION(my_comp);
```

A dynamic initialization would look like:

```
struct completion my_comp;
init_completion(&my_comp);
```

When your driver begins some process whose completion must be waited for, it's simply a matter of passing your completion event to `wait_for_completion()`:

```
void
wait_for_completion(struct completion *comp);
```

When some other part of your code has decided that the completion has happened, it can wake up anybody who is waiting with one of:

```
void complete(struct completion *comp);
void complete_all(struct completion *comp);
```

The first form will wake up exactly one waiting process, while the second will wake up all processes waiting for that event. Note that completions are implemented in such a way that they will work properly even if `complete()` is called before `wait_for_completion()`.

If you do not use `complete_all()`, you should be able to use a completion structure multiple times without problem. It does not hurt, however, to reinitialize the structure before each use—so long as you do it before initiating the process that will call `complete()`! The macro `INIT_COMPLETION()` can be used to quickly reinitialize a completion structure that has been fully initialized at least once.

7 Interrupt handling

The kernel's handling of device interrupts has been massively reworked in the 2.5 series. Fortunately, very few of those changes are visible to the rest of the kernel; most well-written code should "just work" under 2.5. There are, however, two important exceptions: the return type of interrupt handlers has changed, and drivers which depend on being able to globally disable interrupts will require some changes for 2.5.

7.1 Interrupt handler return values

Prior to 2.5.69, interrupt handlers returned `void`. There is, however, one useful thing that interrupt handlers can tell the kernel: whether the interrupt was something they could handle or not. If a device starts generating spurious interrupts, the kernel would like to respond by blocking interrupts from that device. If no interrupt handler for a given IRQ has been registered, the kernel knows that any interrupt on that number is spurious. When interrupt handlers exist, however, they must tell the kernel about spurious interrupts.

So, interrupt handlers now return an `irqreturn_t` value; `void` handlers will no longer compile. If your interrupt handler recognizes and handles a given interrupt, it should return `IRQ_HANDLED`. If it knows that the interrupt was not on a device it manages, it can return `IRQ_NONE` instead. The macro `IRQ_RETVAL(handled)` can also be used; `handled` should be nonzero if the handler could deal with the interrupt. The "safe" value to return, if, for some reason you are not sure, is `IRQ_HANDLED`.

7.2 Disabling interrupts

In the 2.5 kernel, it is no longer possible to globally disable interrupts. In particular, the `cli()`, `sti()`, `save_flags()`, and `restore_flags()` functions are no longer available. Disabling interrupts across all processors in the system is simply no longer done. This behavior has been strongly discouraged for some time, so most code *should* have been converted by now.

The proper way to do this fixing, of course, is to figure out exactly which resources were being protected by disabling interrupts. Those resources can then be explicitly protected with spinlocks instead. The change is usually fairly straightforward, but it does require an understanding of what is really going on.

It is still possible to disable all interrupts locally with `local_save_flags()` or `local_irq_disable()`. A single interrupt can be disabled globally with `disable_irq()`. Some of the spinlock operations also disable interrupts on the local processor, of course.

8 Asynchronous I/O

One of the key "enterprise" features added to the 2.5 kernel is asynchronous I/O (AIO). The AIO facility allows user processes to initiate multiple I/O operations without waiting for any of them to complete; the status of the operations can then be retrieved at some later time. Block and network drivers are already fully asynchronous, and thus there is nothing special that needs to be done to them to support the new asynchronous operations. Character drivers, however, have a synchronous API, and will not support AIO without some additional work. For most char drivers, there is little benefit to be gained from AIO support. In a few rare cases, however, it may be beneficial to make AIO available to your users. The web site has details on how to do that.

9 Block drivers

The first big, disruptive changes to the 2.5 kernel came from the reworking of the block I/O layer. As 2.5 heads towards its final phases, the block layer is still seeing a lot of attention. As one might guess, the result of all this work is a great many changes as seen by driver authors—or anybody else who works with block I/O. The transition may be painful for some, but it's worth it: the new block layer is easier to work with and offers much better performance than its predecessor.

Space constraints make it impossible to cover all of the block layer changes here; they could easily justify an entire paper to themselves. These changes *are* covered in detail on the web site listed at the end of the paper. Here, however, we'll have to content ourselves with an overview.

So, what has changed with the block layer?

- A great deal of old cruft is gone. For example, it is no longer necessary to work with a whole set of global arrays within block drivers. These arrays (`blk_size`, `blksize_size`, `hardsect_size`, `read_ahead`, etc.) have simply vanished.
- As part of the cruft removal, most of the `<linux/blk.h>` macros (`DEVICE_NAME`, `DEVICE_NR`, `CURRENT`, `INIT_REQUEST`, etc.) have been removed, and the file itself should go away soon. It is still possible to implement a simple request loop for straightforward devices where performance is not a big issue, but the mechanisms have changed.
- The `io_request_lock` is gone; locking is now done on a per-queue basis.
- Request queues have, in general, gotten more sophisticated. There is simple support for tagged command queueing, along with features like request barriers and queue-time device command generation.
- Buffer heads are no longer used in the block layer; they have been replaced with the new "`bio`" structure. The new representation of block I/O operations is designed for flexibility and performance; it encourages keeping large operations intact. Simple drivers can pretend that the `bio` structure does not exist, but most performance-oriented drivers—i.e., those that want to implement clustering and DMA—will need to be changed to work with `bio`s.

  One of the most significant features of the `bio` structure is that it represents I/O buffers directly with `page` structures and offsets, not in terms of kernel virtual addresses. By default, I/O buffers can be located in high memory, on the assumption that computers equipped with that much

memory will also have reasonably modern I/O controllers. Support operations have been provided for tasks like `bio` splitting and the creation of DMA scatter/gather maps.

- Sector numbers can now be 64 bits wide, making it possible to support very large block devices.

- The rudimentary `gendisk` ("generic disk") structure from 2.4 has been greatly improved in 2.5; generic disks are now used extensively throughout the block layer. The most significant change for block driver authors may be the fact that partition handling has been moved up into the block layer, and drivers no longer need know anything about partitions. That is, of course, the way things should always have been.

The end result is a fair amount of short-term pain for maintainers of block drivers. It does not take long, however, to realize that the new interface is easier to program for and far more robust.

10 Network drivers

Here's the good news for people maintaining network drivers: once you have deal with the basic changes that affect all drivers and loadable modules, your driver will likely work as it is. The API that was presented to drivers in 2.4 is essentially unchanged. There has been no gratuitous network driver breakage in 2.5.

The bad news is: a bunch of useful new stuff has been added in 2.5. If you want your driver to use the features of 2.5 to get the best performance out of your hardware, you will have to spend some time dealing with changes.

10.1 NAPI

The most significant change, perhaps, is the addition of NAPI ("New API"), which is designed to improve the performance of high-speed networking. NAPI works through **interrupt mitigation** (reducing the thousands of interrupts per second that accompany high network traffic) and **packet throttling** (keeping packets out of the kernel if they will be dropped anyway). NAPI was also backported to the 2.4.20 kernel.

Converting drivers to NAPI is essentially a two-step process:

- Your driver should no longer process incoming packets in its interrupt handler. Instead, it should disable further "packet available" interrupts and tell the networking system to begin polling the interface.

- A new `poll()` method must be created which processes all available incoming packets (up to a kernel-specified limit). A new function, `netif_receive_skb()` has been set up to accept packets from `poll()` methods.

The web site has the inevitable details.

10.2 Receiving packets in non-interrupt mode

Network drivers tend to send packets into the kernel while running in interrupt mode. There are occasions where, instead, packets will be received by a driver running in process context. There is no problem with this mode of operation, but it is possible that the networking software interrupt which performs packet processing may be delayed, reducing performance. To avoid this problems, drivers handing packets to the kernel outside of interrupt context should use:

```
int netif_rx_ni(struct sk_buff *skb);
```

instead of `netif_rx()`.

10.3 Other 2.5 features

A number of other networking features were added in 2.5. Here is a quick summary of developments that driver developers may want to be aware of.

Ethtool support. Ethtool is a utility which can perform detailed configuration of network interfaces; it can be found on the gkernel SourceForge page. This tool can be used to query network information, tweak detailed operating parameters, control message logging, and more. Supporting ethtool requires implementing the `SIOCETHTOOL ioctl()` command, along with (parts of, at least) the lengthy set of ethtool commands. See `<linux/ethtool.h>` for a list of things that can be done. Implementing the message logging control features requires checking the logging settings before each `printk()` call; there is a set of convenience macros in `<linux/netdevice.h>` which make that checking a little easier.

VLAN support. The 2.5 kernel has support for 802.1q VLAN interfaces; this support has also been working its way into 2.4, with the core being merged in 2.4.14.

TCP segmentation offloading. The TSO feature can improve performance by offloading some TCP segmentation work to the adaptor and cutting back slightly on bus bandwidth. TSO is an advanced feature that can be tricky to implement with good performance; see the `tg3` or `e1000` drivers for examples of how it's done.

11 User space access

The `kiobuf` abstraction was introduced in 2.3 as a low-level way of representing I/O buffers. Its primary use, perhaps, was to represent zero-copy I/O operations going directly to or from user space. A number of problems were found with the `kiobuf` interface, however; among other things, it forced large I/O operations to be broken down into small chunks, and it was seen as a heavyweight data structure. So, in 2.5.43, kiobufs were removed from the kernel. Direct access to user space remains possible, however, as we'll see.

The modern equivalent of `map_user_kiobuf()` is a function called `get_user_pages()`:

```
int get_user_pages(
       struct task_struct *task,
       struct mm_struct *mm,
       unsigned long start,
       int len,
       int write,
       int force,
       struct page **pages,
       struct vm_area_struct **vmas);
```

`task` is the process performing the mapping; the primary purpose of this argument is to say who gets charged for page faults incurred while mapping the pages. This parameter is almost always passed as "`current`". The memory management structure for the user's address space is passed in the `mm` parameter; it is usually `current->mm`. Note that `get_user_pages()` expects that the caller will have a read lock on `mm->mmap_sem`. The `start` and `len` parameters describe the user-buffer to be mapped; `len` is in pages. If the memory will be written to, `write` should be non-zero. The `force` flag forces read or write access, even if the current page protection would otherwise not allow that access. The `pages` array (which should be big enough to hold `len` entries) will be filled with pointers to the `page` structures for the user pages. If `vmas` is non-`NULL`, it will be filled with a pointer to the `vm_area_struct` structure containing each page.

The return value is the number of pages actually mapped, or a negative error code if something goes wrong. Assuming things worked, the user pages will be present (and locked) in memory, and can be accessed by way of the `struct page` pointers. Be aware, of course, that some or all of the pages could be in high memory.

There is no equivalent `put_user_pages()` function, so callers of `get_user_pages()` must perform the cleanup themselves. There are two things that need to be done: marking of modified pages, and releasing them from the page cache. If your device modified the user pages, the virtual memory subsystem may not know about it, and may fail to write the pages to permanent storage (or swap). That, of course, could lead to data corruption and grumpy users. The way to avoid this problem is to call:

```
int set_page_dirty_lock(struct page *page);
```

for each page in the mapping.

Finally, every mapped page must be released from the page cache, or it will stay there forever; simply pass each page structure to:

```
void put_page(struct page *page);
```

After you have released the page, of course, you should not access it again.

For a good example of how to use `get_user_pages()` in a char driver, see the definition of `sgl_map_user_pages()` in `drivers/scsi/st.c`.

12 Conclusion

This paper is drawn from the LWN.net "Porting Drivers to 2.5" series, which can be found at:

```
http://lwn.net/Articles/
    driver-porting/
```

Those articles contain much more detail than was possible to squeeze into this paper. They are also being maintained as kernel development continues; there are, beyond a doubt, things that have changed since this paper was written. If nothing else, the kernel developers will eventually figure out how they want to support larger device numbers. The driver-porting web site will have the latest information on kernel API changes.

13 Acknowledgments

Previous versions of this material have been improved by comments from Jens Axboe, Jamal Hadi Salim, Greg Kroah-Hartman, Andrew Morton, Rusty Russell, and others I have certainly forgotten. The creation of the "driver porting" series of articles was funded by LWN.net subscribers.

Class-based Prioritized Resource Control in Linux

Shailabh Nagar, Hubertus Franke, Jonghyuk Choi, Chandra Seetharaman
Scott Kaplan, Nivedita Singhvi, Vivek Kashyap, Mike Kravetz*
IBM Corp.
{nagar,frankeh,jongchoi,chandra.sekharan}@us.ibm.com
{nivedita,vivk,kravetz}@us.ibm.com
sfkaplan@cs.amherst.edu

Abstract

In Linux, control of key resources such as memory, CPU time, disk I/O bandwidth and network bandwidth is strongly tied to kernel tasks and address spaces. The kernel offers very limited support for enforcing user-specified priorities during the allocation of these resources.

In this paper, we argue that Linux kernel resource management should be based on classes rather than tasks alone and be guided by class shares rather than system utilization alone. Such class-based kernel resource management (CKRM) allows system administrators to provide differentiated service at a user or job level and prevent denial of service attacks. It also enables accurate metering of resource consumption in user and kernel mode. The paper proposes a framework for CKRM and discusses incremental modifications to kernel schedulers to implement the framework.

1 Introduction

With Linux having made rapid advances in scalability, making it the operating system of choice for enterprise servers, it is useful and timely to examine its support for resource management. Enterprise workloads typically run on two types of servers: clusters of 1-4 way SMPs and large (8-way and upward) SMPs and mainframes. In both cases, system administrators must balance the needs of the workload with the often conflicting goal of maintaining high system utilization. The balancing becomes particularly important for large SMPs as they often run multiple workloads to amortize the higher cost of the hardware.

A key aspect of multiple workloads is that they vary in the *business importance* to the server owner. To maximize the server's utility, the system administrator needs to ensure that workloads with higher business importance get a larger share of server resources. The kernel's resource schedulers need to allow some form of differentiated service to meet this goal. It is also important that the resource usage by different workloads be accounted accurately so that the customers can be billed according to their true usage rather than an average cost. Kernel support for accurate accounting of resource usage is required, especially for resource consumption in kernel mode.

Differentiated service is also useful to the desktop user. It would allow large file transfers to get reduced priority to the disk compared to disk accesses resulting from interactive com-

*on sabbatical from Amherst College, MA

mands. A kernel compile could be configured to run in the background with respect to the CPU, memory and disk, allowing a more important activity such as browsing to continue unimpeded.

The current Linux kernel (version 2.5.69 at the time of writing) lacks support for the abovementioned needs. There is limited and varying support for any kind of performance isolation in each of the major resource schedulers (CPU, network, disk I/O and memory). CPU and inbound network scheduling offer the greatest support by allowing specification of priorities. The deadline I/O scheduler [3] offers some isolation between disk reads and writes but not between users or applications and the VM subsystem has support for limiting address space size of a user. More importantly, the granularity of kernel supported service differentiation is a task (process), or infrequently the userid. It does not allow the user to define the granularity at which resources get apportioned. Finally, there is no common framework for a system administrator to specify priorities for usage of different physical resources.

The work described in this paper addresses these shortcomings. It proposes a class-based framework for prioritized resource management of all the major physical resources managed by the kernel. A class is a user-defined, dynamic grouping of tasks that have associated priorities for each physical resource. The proposed framework permits a better separation of user-specified policies from the enforcement mechanisms implemented by the kernel. Most importantly, it attempts to achieve these goals using incremental modifications of the existing mechanisms.

The paper is organized as follows. Section 2 introduces the proposed framework and the central concepts of classification, monitoring and control. Sections 3,4,5,6 explore the framework for the CPU, disk I/O, network and memory subsystems and propose the extensions necessary to implement it. Section 7 concludes with directions for future work.

2 Framework

Before describing the proposed framework, we define a few terms.

Tasks are the Linux kernel's common representation for both processes and threads. A *class* is a group of tasks. The grouping of tasks into classes is decided by the user using rules and policies.

A *classification rule*, henceforth simply called a rule, is a method by which a task can be classified into a class. Rules are defined by the system administrator, generally as part of a policy (defined later) but also individually, typically as modifications or increments to an existing policy. Attributes of a task, such as real uid, real gid, effective uid, effective gid, path name, command name and task or application tag (defined later) can be used to define rules. A rule consists of two parts: a set of attribute-value tuples (A,V) and a class C. If the rule's tuple values match the corresponding attributes of a task , then it is considered to belong to the class C1.

A *policy* is a collection of class definitions and classification rules. Only one policy is active in a system at any point of time. Policies are constructed by the system administrator and submitted to a CKRM kernel. The kernel optionally verifies the policy for consistency and activates it. The order of rules in a policy is important. Rules are applied in the order they are defined (one exception is the application tags as described in the notes below).

An *Application/Task Tag* is a user-defined attribute associated with a task. Such an attribute

is useful when tasks need to be classified and managed based on application-specific criteria. In such scenarios, an applications tasks can specify its tag to the kernel using a system call, ioctl, /proc entry etc. and trigger its classification using a rule that uses the task tag attribute. Since the task tag is opaque to the kernel, it allows applications and system administrators additional flexibility in grouping tasks.

A *Resource Manager* is the entity which determines the proportions in which resources should be allocated to classes. This could be either a human system administrator or a resource management application middleware.

With all the new terms of the framework defined, we can now describe how the framework operates to provide class-based resource management. Figure 1 illustrates the lifecycle of tasks in the proposed framework with an emphasis on the three central aspects of classification, monitoring and control.

2.1 Initialization

Sometime after system boot up, the Resource Manager commits a policy to the kernel. The policy defines the classes and it is used to classify all tasks (pre-existing and new) created and all incoming network packets. A CKRM-enabled Linux kernel also contains a default policy with a single default class to which all tasks belong. The default policy determines classification and control behaviour until the Resource Manager's policy gets loaded. New policies can be loaded at any point and override the existing policy. Such policy loads trigger reclassification and reinitialization of monitoring data and are expected to be very infrequent.

2.2 Classification

Classification refers to the association of tasks to classes and the association of resource requests to a class. The distinction is mostly irrelevant as most resource requests are initiated by a task except for incoming network packets which need to be classified before it is known which task will consume them. Classification is a continuous operation. It happens on a large scale each time a new policy is committed and all existing tasks get reclassified. At later points, classification occurs whenever (1) a new task is created e.g. fork(), exec(); (2) the attributes of a task change e.g setuid(), setgid(), application tag change (initiated by the application) and (3) explicit reclassification of a specific task by the Resource Manager. Scenarios (1) and (2) are illustrated in Figure 1.

Classification of tasks potentially allows all work initiated by the task to be associated with the class. Thus the CPU, memory and I/O requests generated by this task, or by the kernel on behalf of this task, can be monitored and regulated by the class-aware resource schedulers. Kernel-initiated resource requests which are on behalf of multiple classes e.g. a shared memory page writeout need special treatment as do application initiated requests which are processed asynchronously. Classification of incoming network connections and/or data (which are seen by the kernel before the task to which they are destined) is discussed separately in Section 5.

2.3 Monitoring

Resource usage is maintained at the class level in addition to the regular system accounting by task, user etc. The system administrator or an external control program with root privileges can obtain that information from the kernel at any time. The information can be used to assess machine utilization for billing purposes or as an input to a future decision on changing the share allotted to a class. The CKRM API provides functions to query the current usage data as shown in Figure 1.

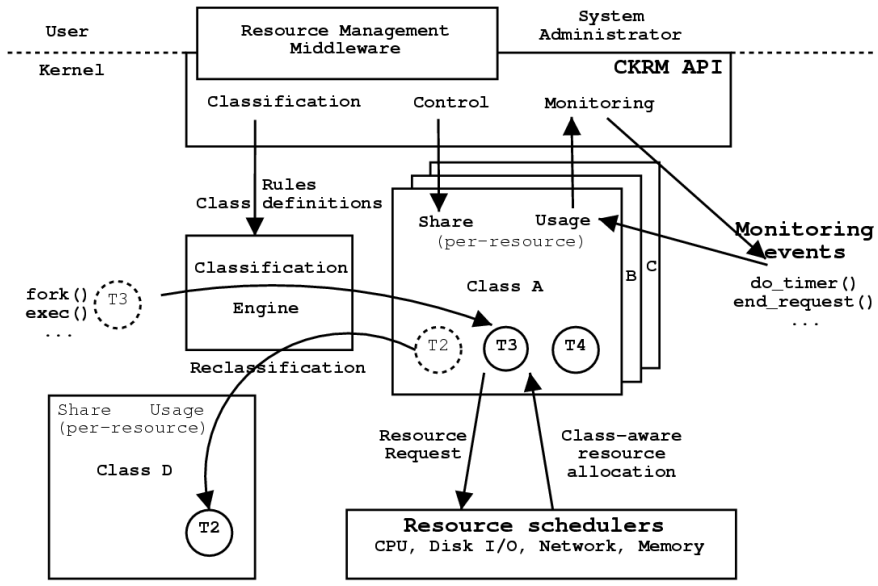


Figure 1: CKRM lifecycle

2.4 Control

The system administrator, as part of the initial policy or at a later point in time, assigns a per-resource share to each class in the system. Each class gets a separate share for CPU time, Memory pages, I/O bandwidth and incoming network I/O bandwidth. The resource schedulers try their best to respect these shares while allocating resources to a class. e.g. the CPU scheduler tries to ensure that tasks belonging to Class A with a 50% CPU share collectively get 50% of the CPU time. At the next level of the control hierarchy, the system administrator or a control program can change the shares assigned to a class based on their assessment of application progress, system utilization etc. Collectively, the setting of shares and share-based resource allocation constitute the control part of the resource management lifecycle and are shown in Figure 1. This paper concentrates on the lower level share-based resource allocation since that is done by the kernel.

The next four sections go into the details of classification, monitoring and control aspects of managing each of the four major physical resources.

3 CPU scheduling

The CPU scheduler is central to the operation of the computing platform. It decides which task to run when and how long. In general realtime and timeshared jobs are distinguished, each with different objectives. Both are realized through different scheduling disciplines implemented by the scheduler. Before addressing the share based scheduling schemes, we describe the current Linux scheduler.

3.1 Linux 2.5 Scheduler

To achieve its objectives, Linux assigns a static priority to each task that can be modified by the user through the `nice()` interface. Linux has a range of $[0 \ldots \texttt{MAX\_PRIO}]$ priority classes, of which the first `MAX_RT_PRIO` (=100) are set aside for realtime jobs and the remaining 40 priority classes are set aside for timesharing (i.e. normal) jobs, representing the $[-20 \ldots 19]$ nice value of UNIX processes. The lower the priority value, the higher the "logical" priority

of a task, i.e. its general importance. In this context we always assume the logical priority when we are talking about priority increases and decreases. Realtime tasks always have a higher priority then normal tasks.

The Linux scheduler in 2.5, a.k.a the O(1) scheduler, is a multi queue scheduler that assigns to each cpu a run queue, wherein local scheduling takes place. A per-cpu runqueue consists of two arrays of task lists, the active array and the expired array. Each array index represents a list of runnable task at their respective priority level. After executing for a period task move from the active list to the expired list to guarantee that all tasks get a chance to execute. When the active array is empty, expired and active arrays are swapped. More detail is provided further on. The scheduler simply picks the first task of the highest priority queue of the active queue for execution.

Occasionally a load balancing algorithm rebalances the runqueues to ensure that a similar level of progress is made on each cpu. Realtime issues and load balancing issues are beyond this description here, hence we concentrate on the single cpu issue for now. For more details we refer to [12]. It might also be of interest to abstract this against the Linux 2.4 based scheduler, which is described in [10].

As stated earlier, the scheduler needs to decide which task to run next and for how long. Time quantums in the kernel are defined as multiples of a system *tick*. A tick is defined by the fixed delta `(1/HZ)` of two consecutive timer interrupts. In Linux 2.5: `HZ=1000`, i.e. the interrupt routine `scheduler_tick()` is called once every msec at which time the currently executing task is charged a tick.

Besides the *static priority* (`static_prio`), each task maintains an *effective priority* (`prio`). The distinction is made in order to account for certain temporary priority bonuses or penalties based on the recent *sleep average* `sleep_avg` of a given task. The sleep average, a number in the range of $[0 \ldots \texttt{MAX\_SLEEP\_AVG} * \texttt{HZ}]$, accounts for the number of ticks a task was recently descheduled. The time (in ticks) since a task went to sleep (maintained in `sleep_timestamp`) is added on task wakeup and for every time tick consumed running, the sleep average is decremented.

The current design provisions a range of `PRIO_BONUS_RATIO=25%` $[-12.5\%..12.5\%]$ of the priority range for the sleep average. For instance a "nice=0" task has a static priority of 120. With a sleep average of 0 this task is penalized by 5 resulting in an effective priority of 125. On the other hand, if the sleep average is `MAX_SLEEP_AVG` = 10 secs, a bonus of 5 is granted leading to an effective priority of 115. The effective priority determines the priority list in the active and expired array of the run queue. A task is declared *interactive* when its effective priority exceeds its static priority by a certain level (which can only be due to its accumulating sleep average). High priority tasks reach interactivity with a much smaller sleep average than lower priority tasks.

The timeslice, defined as the maximum time a task can run without yielding to another task, is simply a linear function of the static priority of normal tasks projected into the range of $[\texttt{MIN\_TIMESLICE} \ldots \texttt{MAX\_TIMESLICE}]$. The defaults are set to $[10 \ldots 200]$ msecs. The higher the priority of a task the longer its `timeslice`. A task's initial timeslice is deducted from parents' remaining timeslice. For every timer tick the running task's timeslice is decremented. If decremented to "0", the scheduler replenishes the timeslice, recomputes the effective priority and either reenqueues the task into the active array if it is interactive or into the expired array if it is non-interactive. It then

picks the next best task from the active array. This guarantees that all others tasks will execute first before any expired task will run again. If a task is descheduled, its timeslice will not be replenished at wakeup time, however its effective priority might have changed due to any sleep time.

If all runnable tasks have expired their timeslices and have been moved to the expired list, the expired array and the active array are swapped. This makes the scheduler O(1) as it does not have to traverse a potentially large list of tasks as was needed in the 2.4 scheduler. Due to interactivity the situation can arise that the active queue continues to have runnable tasks. As a result tasks in the expired queue might get starved. To avoid this, if the longest expired task is older than `STARVATION_LIMIT`=10secs, the arrays are switched again.

3.2 CPU Share Extensions to the Linux Scheduler

We now examine the problem of extending the O(1) scheduler to allocate CPU time to classes in proportion of their CPU shares. Proportional share scheduling of CPUs has been studied in depth [7, 21, 13] but not in the context of making minimal modifications to an existing scheduler such as O(1).

Let C_i, i=$[1 \ldots N]$ be the set of N distinct classes with corresponding cpu shares S_i^{cpu} such that $\sum_{i=1}^{N} S_i^{cpu} = 1$. Let `SHARE_TIME_EPOCH` (STE) be the time interval, measured in ticks, over which the modified scheduler attempts to maintain the proportions of allocated CPU time. Further, we use $\Phi(a)$ and $\Phi(a, b)$ to denote a functions of parameters a and b.

Weighted Fair Share (WFS): In the first approach considered, henceforth called weighted fair share (WFS), a per class scheduling resource container is introduced that accounts for timeslices consumed by the tasks currently assigned to the class. Initially, the timeslice TS_i, i=$[1 \ldots N]$ of each class C_i is determined by $TS_i = S_i^{cpu} \times STE$. The timeslice allocated to a task $ts(p)$ remains the same as in O(1). Everytime a task consumes one of its own timeslice ticks, it also consumes one from the class' timeslice. When a class' ticks are exhausted, the task consuming the last tick is put into the expired array. When the scheduler picks other tasks from the same class to run, they immediately move to the expired array as well. Eventually the expired and active arrays are switched at which time all resource containers are refilled to $TS_i = S_i^{cpu} \times STE$. Since the array switch occurs as soon as the active list becomes empty, this approach is work conserving (the CPU is never idle if there are runnable tasks). A variant of this approach was initially implemented by [17] based on a patch from Rik van Riel for allocating *equal* shares to all *users* in the system.

However, WFS has some problems. If the tasks of a class are CPU bound and $\sum_{p \in C_i} ts(p) > TS_i$ then a class could exhaust its timeslice before all its tasks have had a chance to run atleast once. Therefore the lower priority tasks of the class could perpetually move from the active to expired lists without ever being granted execution time. Starvation occurs because neither the static priority (sp) nor the sleep average (sa) of the tasks is changed at any time. Hence each task's timeslice $ts(p) = \Phi(sp)$ and effective priority $ep(p) = \Phi(sp, sa)$ remain unchanged. Hence the relative priority of tasks of a class never changes (in a CPU bound situation) nor does the amount of CPU time consumed by the higher priority tasks.

To ensure a fair share for individual tasks within classes, we need to ensure that the rate of progress of a task depends on the share assigned to its class. Three approaches to achieve

this are discussed next.

Priority modifications (WFS+P): Let a *switching interval* be defined as the time interval between consecutive array switches of the modified scheduler, Δ_j be its duration and t_j and t_{j+1} be the timestamps of the switches. In the priority modifications approach to alleviating starvation in WFS, henceforth called WFS+P, we track the number of array switches se at which a task got starved due to its class' timeslice exhaustion and increase the task's effective priority based on se, i.e. $ep(p) = \Phi(sp, sa, se)$. This ensures that starving tasks eventually get their ep high enough to get a chance to run at which point se is reset. The drawback of this approach is that the increased scheduler overhead of tasks being selected for execution and moving directly to the expired list due to class timeslice exhaustion, remains unchanged.

Timeslice modifications (WFS+T1, WFS+T2): Recognizing that starvation can occur in WFS for class C_i only if $\sum_{p \in C_i} ts(p) > TS_i$, the timeslice modification approaches attempt to change one or the other side of the inequality to convert it to an equality. In WFS+T1, the left hand side of the inequality is changed by reducing the timeslices of each task of a starved class as follows. Let $exh_i = \sum_{p \in C_i} ts(p) - TS_i$ when class timeslice exhaustion occurs. At array switch time, each $ts(p)$ is multiplied by $\lambda_i = \frac{TS_i}{TS_i + exh_i}$ which results in the desired equality. WFS+T1 is slow to respond to starvation because task timeslices are recomputed in O(1) *before* they move into the expired array and not at array switch time. Hence any task timeslice changes take effect only one switching interval later i.e. two intervals beyond the one in which starvation occurred. One way to address this problem is to treat a task as having exhausted its timeslice when $ts(p)$ gets decremented to $(1 - \lambda_i \times ts(p))$ instead of 0. A bigger problem with WFS+T1 is that smaller timeslices for tasks could lead to increased context switches with potentially negative cache effects.

To avoid reducing $ts(p)$'s, WFS+T2 increases TS_i of a starving class to make$\sum_{p \in C_i} ts(p) = TS_i$ i.e. the class does not exhaust its timeslice until each of its tasks have exhausted their individual timeslices. To preserve the relative proportions between class timeslices, all other class timeslices also need to be changed appropriately. Doing so would disturb the same equality for those classes and hence WFS+T2 is not a workable approach.

Two-level scheduler: Another way to regulate CPU shares in WFS is to take tasks out of the runqueue upon timeslice exhaustion and return them to the runqueue at a rate commensurate with the share of the class. A prototype implementation of this approach was described in [17] in the context of user-based fair sharing. This approach effectively implements a two-level scheduler and is illustrated in Figure 2. A modified O(1) scheduler forms the lower level and a coarse-grain scheduler operates at the upper level, replenishing class timeslice ticks. In the modified O(1), when a task expires, it is moved into a FIFO list associated with its class instead of moving to the expired array. At a coarse-granularity determined by the upper level scheduler, the class receives new time ticks and reinserts tasks from the FIFO back into O(1)'s runqueue. Class time tick replenishment can be done for all classes at every array switch point but that violates the O(1) behaviour of the scheduler as a whole. To address this problem, [17] uses a kernel thread to replenish 8 ms worth of ticks to one class (user) every 8 ms and round robin through the classes (users). A variant of this idea is currently being explored.

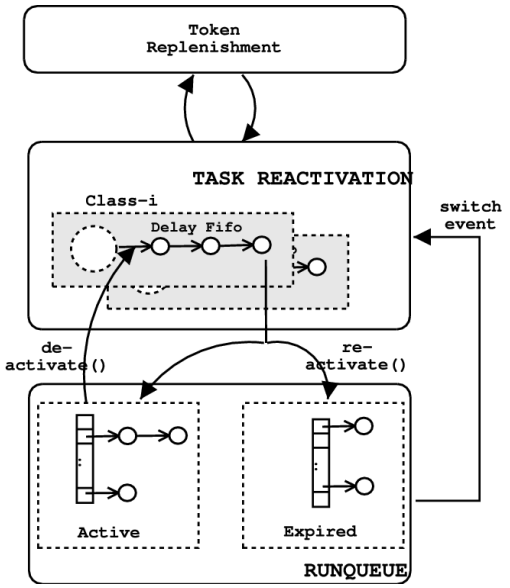


Figure 2: Proposed two-level CPU scheduler

4 Disk scheduling

The I/O scheduler in Linux forms the interface between the generic block layer and the low level device drivers. The block layer provides functions which are used by filesystems and the virtual memory manager to submit I/O requests to block devices. These requests are transformed by the I/O scheduler and made available to the low-level device drivers (henceforth only called device drivers). Device drivers consume the transformed requests and forward them, using device specific protocols, to the device controllers which perform the I/O. Since prioritized resource management seeks to regulate the use of a disk by an application, the I/O scheduler is an important kernel component that is sought to be changed. It is also possible to regulate disk usage in the kernel layers above and below the I/O scheduler. Changing the pattern of I/O load generated by filesytems or the virtual memory manager (VMM) is an important option. A less explored option is to change the way specific device drivers or even device controllers consume the I/O requests submitted to them. The latter approach is outside the scope of general kernel development and this paper.

Class-based resource management requires two fundamental changes to the traditional approach to I/O scheduling. First, I/O requests should be managed based on the priority or weight of the request submitter with disk utilization being a secondary, albeit important objective. Second, I/O requests should be associated with the class of the request submitter and not a process or task. Hence the weight associated with an I/O request should be derived from the weight of the class generating the request.

The first change is already occurring in the development of the 2.5 Linux kernel with the development of different I/O schedulers such as deadline, anticipatory, stochastic fair queueing and complete fair queueing. Consequently, the *additional* requirements imposed by the second change (scheduling by class) are relatively minor. This fits in well with our project goal of minimal changes to existing resource schedulers.

We now describe the existing Linux I/O schedulers followed by an overview of the changes being proposed.

4.1 Existing Linux I/O schedulers

The various Linux I/O schedulers can be abstracted into a generic model shown in Figure 3. I/O requests are generated by the block layer on behalf of processes accessing filesystems, processes performing raw I/O and from the virtual memory management (VMM) components of the kernel such as kswapd, pdflush etc. These producers of I/O requests call __make_request() which invokes various I/O scheduler functions such as elevator_merge_fn. The enqueuing functions' generally try to merge the newly submitted block I/O unit (bio in 2.5 kernels, buffer_head in

2.4 kernels) with previously submitted requests and sort it into one or more internal queues. Together, the internal queues form a single logical queue that is associated with each block device. At a later point, the low-level device driver calls the generic kernel function elv_next_request() to get the next request from the logical queue. elv_next_request interacts with the I/O scheduler's dequeue function elevator_next_req_fn and the latter has an opportunity to pick the appropriate request from one of the internal queues. The device driver then processes the request, converting it to scatter-gather lists and protocol-specific commands that are then sent to the device controller. As far as the I/O scheduler is concerned, the block layer is the producer of I/O requests and the device drivers are the consumers. Strictly speaking, the block layer includes the I/O scheduler but we distinguish the two for the purposes of our discussion.

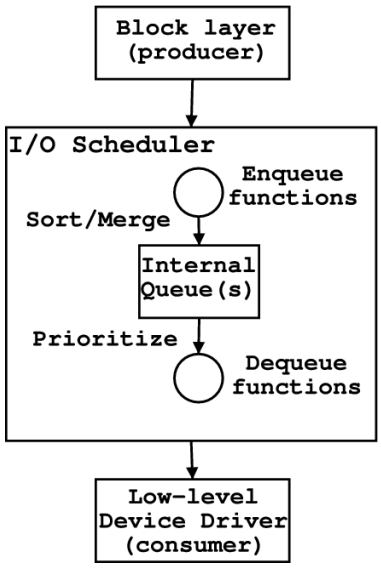


Figure 3: Abstraction of Linux I/O scheduler

Default 2.4 Linux I/O scheduler: The 2.4 Linux kernel's default I/O scheduler (elevator_linus) primarily manages disk utilization. It has a single internal queue. For each new bio, the I/O scheduler checks to see if it can be merged with an existing request. If not, a new request is placed in the internal queue sorted by the starting device block number of the request. This minimizes disk seek times if the disk processes requests in FIFO order from the queue. An aging mechanism limits the number of times an existing request can get bypassed by a newer request, preventing starvation. The dequeue function is simply a removal of requests from the head of the internal queue. Elevator_linus also has the welcome property of improving request response times *averaged* over all processes.

Deadline I/O scheduler: The 2.5 kernel's default I/O scheduler (deadline_iosched) introduces the notion of a per-request deadline which is currently used to give a higher preference to read requests. Internally, it maintains five queues. During enqueing, each request is assigned a deadline and inserted into queues sorted by starting sector (sort_list) *and* by deadline (fifo_list). Separate sort and fifo lists are maintained for read and write requests. The fifth internal queue contains requests to be handed off to the driver. During a dequeue operation, if the dispatch queue is empty, requests are moved from one of the four sort or fifo lists in batches. Thereafter, or if the dispatch queue was not empty, the head request on the dispatch queue is passed on to the driver. The logic for moving requests from the sort or fifo lists ensures that each read request is processed by its deadline without starving write requests. Disk seek times are amortized by moving a large batch of requests from the sort_list (which are likely to have few seeks as they are already sector sorted) and balancing it with a controlled number of requests from the fifo list (each of which could cause a seek since they are ordered by deadline and not sector). Thus, deadline_iosched effectively emphasizes average read request response times over disk utilization and total average request response time.

Anticipatory I/O scheduler: The anticipatory I/O scheduler [9, 4] attempts to reduce *per-process* read response times. It introduces a controlled delay in dispatching *any* new requests to the device driver, thereby allowing a process whose request just got serviced to submit a new request, potentially requiring a smaller seek. The tradeoff between reduced seeks and decreased disk utilization (due to the additional delays in dispatch) are managed using a cost-benefit calculation. anticipatory I/O scheduling method is an additional optimization that can potentially be added to any of the I/O scheduler mentioned in this paper

Complete Fair Queueing I/O scheduler: Two new I/O schedulers recently proposed in the Linux kernel community, introduce the concept of fair allocation of I/O bandwidth amongst producers of I/O requests. The Stochastic Fair Queueing (SFQ) scheduler [5] is based on an algorithm originally proposed for network scheduling [11]. It tries to apportion I/O bandwidth equally amongst all *processes* in a system using 64 internal queues and one output (dispatch) queue. During an enqueue, the process ID of the currently running process (very likely to be the I/O request producer) is used to select one of the internal queues and the request inserted in FIFO order within it. During dequeue, SFQ round-robins through the non-empty internal queues, picking requests from the head. To avoid too many seeks, one full round of requests are collected, sorted and merged into the dispatch queue. The head request of the dispatch queue is then passed to the device driver. Complete Fair Queuing is an extension of the same approach where no hash function is used. Hence each process in the system has a corresponding internal queue and can get an fair share of the I/O bandwidth (equal share if all processes generate I/O requests at the same rate). Both CFQ and SFQ manage per-process I/O bandwidth and can provide fairness at a process granularity.

Cello disk scheduler: Cello is a two-level I/O scheduler [16] that distinguishes between classes of I/O requests and allows each class to be serviced by a different policy. A coarse grain class-independent scheduler decides how many requests to service from each class. The second level class-dependent scheduler then decides which of the requests from its class should be serviced next. Each class has its own internal queue which is manipulated by the class-specific scheduler. There is one output queue common to all classes. Enqueuing into the output queue is done by the class-specific schedulers in a way that ensures individual request deadlines are met as far as possible while reducing overall seek time. Dequeuing from the output queue occurs in FIFO order as in most of the previous I/O schedulers. Cello has been shown to provide good isolation between classes as well as the ability to meet the needs of streaming media applications that have soft realtime requirements for I/O requests.

4.2 Costa: Proposed I/O scheduler

This paper proposes that a modified version of the class-independent scheduler of the Cello I/O scheduling framework can provide a low-overhead class-based I/O scheduler suitable for CKRM's goals.

The key difference between the proposed scheduler called Costa and Cello is the elimination of the class-specific I/O schedulers which may be add unnecessary overhead for CKRM's goal of I/O bandwidth regulation. Fig 4 illustrates the Costa design. When the kernel is configured for CKRM support, a new internal queue is created for each class that gets added to the kernel. Since each process is always associated with a class, I/O requests that they generate can also be tagged with the class id and used to enqueue the request in the class-

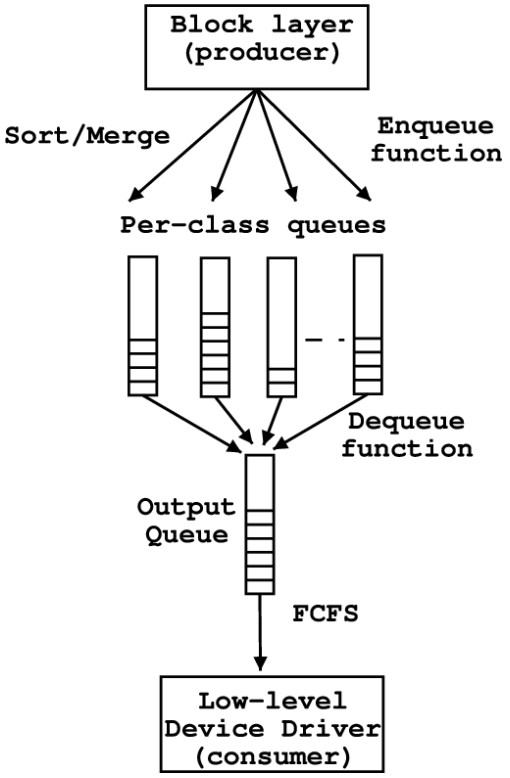


Figure 4: Proposed Costa I/O scheduler

specific internal queue. The request->class association cannot be done through a lookup of the *current* process' class alone. During dequeue, the Costa I/O scheduler picks up requests from several internal queues and sorts them into the common dispatch queue. The head request of the dispatch queue is then handed to the device driver.

The mechanism also allows internal queues to be associated with *system* classes that group I/O requests coming from important producers such as the VMM. By separating these out, Costa can give them preferential treatment for urgent VM writeout or swaps.

In addition to a weight value (which determines the fraction of I/O bandwidth that a class will receive), the internal queues could also have an associated *priority* value which determines their relative importance. At a given priority level, all queues could receive I/O bandwidth in proportion of their weights with the set of queues at a higher level always getting serviced first. Some check for preventing starvation of lower priority queues could be used similar to the ones used in deadline_iosched.

5 QoS Networking in Linux

Many research efforts have been made in networking QoS (Quality of Service) to provide quality assurance of latency, bandwidth, jitter, and loss rate. With the proliferation of multimedia and quality-sensitive business traffic, it becomes essential to provide reserved quality services (IntServ [23]) or differentiated services (DiffServ [2]) for important client traffic.

The Linux kernel has been offering a well established QoS network infrastructure for outbound bandwidth management, policy-based routing, and DiffServ. Hence, Linux is being widely used for routers, gateways, and edge servers, where network bandwidth is the primary resource to differentiate among classes.

When it comes to Linux as an end server OS, on the other hand, networking QoS has not been given as much attention because QoS is primarily governed by the system resources such as CPU, memory, and I/O and less by the network bandwidth. However, when we consider the end-to-end service quality, we also should require networking QoS in the end servers as exemplified by the fair share admission control mechanism proposed in this section.

In the rest of the section, we first briefly introduce the existing network QoS infrastructure of Linux. Then, we describe the design of the fair share admission control in Linux and preliminary performance results.

5.1 Linux Traffic Control, Netfilter, DiffServ

The Linux traffic control [8] consists of queueing disciplines (qdisc) and filters. A qdisc consists of one or more queues and a packet scheduler. It makes traffic conform to a certain profile by shaping or policing. A hierarchy of qdiscs can be constructed jointly with a class hierarchy to make different traffic classes governed by proper traffic profiles. Traffic can be attributed to different classes by the filters that match the packet header fields. The filter matching can be stopped to police traffic above a certain rate limit. A wide range of qdiscs ranging from a simple FIFO to classful CBQ or HTB are provided for outbound bandwidth management, while only one ingress qdisc is provided for inbound traffic filtering and policing [8]. The traffic control mechanism can be used in various places where bandwidth is the primary resource to control. For instance in service providers, it manages bandwidth allocation shared among different traffic flows belonging to different customers and services based on service level agreements. It also can be used in client sites to reduce the interference between upstream and downstream traffic and to enhance the response time of the interactive and urgent traffic.

Netfilter provides sophisticated filtering rules and targets. Matched packets can be accepted, denied, marked, or mangled to carry out various edge services such as firewall, dispatcher, proxy, NAT etc. Routing decisions can be made based on the netfilter markings so packets may take different routes according to their classes. The qdiscs would enable various QoS features in such edge services when used with Netfilter. Netfilter classification can be transferred for use in later qdiscs by markings or mangled packet headers.

The Differentiated Service (DiffServ) [2] provides a scalable QoS by applying per-hop behavior (PHB) collectively to aggregate traffic classes that are identified by a 6-bit code point in the IP header. Classification and conditioning are typically done at the edge of a DiffServ domain. The domain is a contiguous set of nodes compliant to a common PHB. The DiffServ PHB is supported in Linux [22]. Classes, drop precedence, code point marking, and conditioning can be implemented by qdiscs and filters. At the end servers, the code point can be marked by setting the `IP_TOS` socket option.

In the policy based networking [18], a policy agent can configure the traffic classification of edge and end servers according to a predefined filtering rules that match layer 3/4 or layer 7 information. Netfilter, qdisc, and application layer protocol engines can classify traffic for differentiated packet processing at later stages. Classifications at prior stages can be overridden by the transaction information such as URI, cookies, and user identities as they are known. It has been shown that a coordination of server and network QoS can reduce end-to-end response time of important client requests significantly by virtual isolation from the low priority traffic [15].

5.2 Prioritized Accept Queues with Proportional Share Scheduling

We present here a simple change to the existing Linux TCP accept mechanism to provide differentiated service across priority classes. Recent work in this area has introduced the concept of prioritized accept queues [19] and accept queue schedulers using adaptive proportional shares to self-managed web servers [14].

Under certain load conditions [14], the TCP accept queue of each socket becomes the bottleneck in network input processing. Normally, listening sockets fork off a child process to handle an incoming connection request. Some optimized applications such as the Apache web

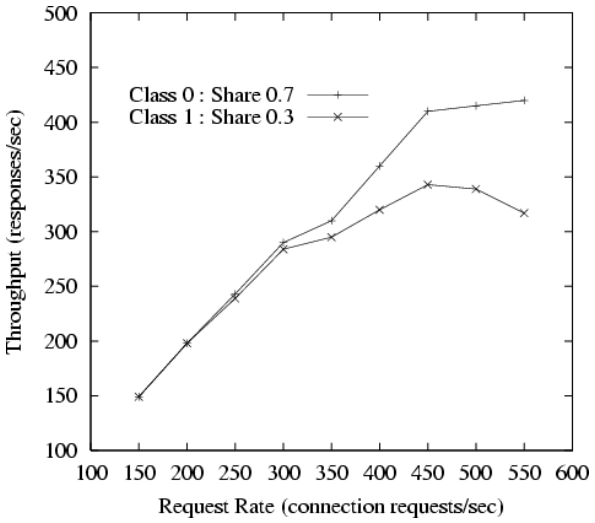


Figure 5: Proportional Accept Queue Results.

server maintain a pool of server processes to perform this task. When the number of incoming requests exceeds the number of static pool servers, additional processes are forked up to a configurable maximum. When the incoming connection request load is higher than the level that can be handled by the available server processes, requests have to wait in the accept queue until one is available.

In the typical TCP connection, the client initiates a request to connect to a server. This connection request is queued in a single global accept queue belonging to the socket associated with that server's port. Processes that perform an `accept()` call on that socket pick up the next queued connection request and process it. Thus all incoming connections to a particular TCP socket are serialized and handled in FIFO order.

We replace the existing single accept queue per socket with multiple accept queues, one for each priority class. Incoming traffic is mapped into one of the priority classes and queued on the accept queue for that priority. There are eight priority classes in the current implementation.

The accepting process schedules connection acceptance according to a simple weighted deficit round robin to pick up connection requests from each queue according to its assigned share. The share, or weight can be assigned by the sysctl interface.

In the basic priority accept queue design proposed earlier in [6], starvation of certain priority classes was a possibility as the accepting process picked up connection requests in a descending priority order. With a proportional share scheme in this paper, it is easier to avoid starvation of particular classes to give share guarantees to low priority classes.

The efficacy of the proportional accept queue mechanism is demonstrated by an experiment. In the experiment, we used Netfilter with mangle tables and `MARK` options to characterize traffic into priority classes based on source IP address. Httperfs from two client machines send requests to an Apache web server running on a single server over independent gigabit Ethernet connections. The only Apache parameter changed from the default was the maximum number of httpd threads. This was set to 50 in the experiment.

Figure 5 shows throughput of Apache for two priority classes, sharing inbound connection bandwidth by 7:3. We can see that the throughput of the priority class 0 requests is slightly higher than that of the priority class 1 requests when the load is low. As load increases, the acceptance rates to the priority classes 0 and 1 will be constrained in proportion to their relative share, which in turn determines the processing rate of the Apache web server and connection request queueing delay. Under a severe load, the priority class 0 requests are processed at a considerably higher throughput.

6 Controlling Memory

While many other system resources can be managed according to priorities or proportions, *virtual memory managers (VMM)* currently do not allow such control. The *resident set size (RSS)* of each process—the number of physical page frames allocated to that process–will determine how often that process incurs a page fault. If the RSS of each process is not somehow proportially shared or prioritized, then paging behavior can overwhelm and undermine the efforts of other resource management policies.

Consider two processes, A and B, where A is assigned a larger share than B with the CPU scheduler. If A is given too small an RSS and begins to page fault frequently, then it will not often be eligible for scheduling on the CPU. Consequently, B will be scheduled more often than A, and the VMM will have become the *de facto* CPU scheduler, thus violating of the requested scheduling proportions.

Furthermore, it is possible for most existing VMM policies to exhibit a specific kind of degenerative behavior. Once process A from the example above begins to page fault, its infrequent CPU scheduling prevents it from referencing its pages at the same rate as other, frequently scheduled processes. Therefore, its pages will become more likely to evicted, thereby reducing its RSS. The smaller RSS will *increase* the probability of further page faults. This degenerative feedback loop will cease only when some other process either exits or changes its reference behavior in a manner that reduces the competition for main memory space.

Main memory use must be controlled just as any other resource is. The RSS of each *address space*—the logical space defined either by a file or by the anonymous pages within the virtual address space of a process—must be calculated as a function of the proportions (or priorities) that are used for CPU scheduling. Because this type of memory management has received little applied or academic attention, our work in this area is still nascent. We present here the structural changes to the Linux VMM necessary for proportional/prioritized memory management; we also present previous, applicable research, as well as future research directions that will address this complex problem. While the proportional/prioritized management of memory is currently less well developed than the other resources presented in this paper, it is necessary that it be comparably developed.

6.1 Basic VMM changes

Consider a function that explicitly calculates the desired RSS for each address space—the *target RSS*—when the footprints of the active address spaces exceeds the capacity of main memory. After this function sets these targets, a system could *immediately* bring the actual RSS into alignment with these targets. However, doing so may require a substantial number of page swapping operations. Since disk operations are so slow, it is inadvisable to use aggressive, pre-emptive page swapping. Instead, a system should seek to move the the actual RSS values toward their targets in a *lazy* fashion, one page fault at a time. Until the actual RSS of address space matches its target, it can be labeled as being either in *excess* or in *deficit* of its target.

As the VMM reclaims pages, it will do so from address spaces with excess RSS values. This approach to page reclamation suggests a change in the structure of the *page lists*—specifically, the *active* and *inactive* lists that impose an order on pages[1]. The current Linux

[1] In the Linux community, these are known as the

VMM uses *global* page lists. If this approach to ordering pages in unchanged, then the VMM would have to search the page lists for pages that belong to the desired address spaces that have excess RSS values. Alternatively, one pair of page lists—*active* and *inactive*—could exist for each address space. The reclamation of pages from a specific address space would therefore require no searching.

By ordering pages separately with each address space, we also enable the VMM to more easily track reference behavior for each address space. While the information to be gathered would depend on the underlying policy that selects target RSS values, we believe that such tracking may play an important role in the development of such policies.

Note that the target RSS values would need to be recalculated periodically. While the period should be directly proportional to the *memory pressure*—some measure of the current workload's demand for main memory space—it is a topic of future research to determine what that period should be. By coupling the period for target RSS calculations to the memory pressure, we can ensure that this strategy only incurs significant computational overhead when heavy paging is occuring.

6.2 Proportionally sharing space

Waldspurger [20] describes a method of proportionally sharing the space of a system running VMWare's ESX Server between a number of virtual machines. Specifically, as with a proportional share CPU scheduler, memory shares can be assigned to each virtual machine, and Waldspurger's policy will calculate target RSS values for each virtual machine.

Proportionally sharing main memory space can result in superfluous allocations to some virtual machines. If the target RSS for some virtual machine is larger than necessary, and some of the main memory reserved for that virtual machine is rarely used[2], then its target RSS should be reduced. Waldspurger addresses this problem with a *taxation policy*. In short, this policy penalizes each virtual machine for unused portions of its main memory share by reducing its target RSS.

Application to the Linux VMM. This approach to proportionally sharing main memory could easily be generalized so that it can apply to address spaces within Linux instead of virtual machines on an ESX Server. Specifically, we must describe how shares of main memory space can be assigned to each address space. Given those shares, target RSS values can be calculated in the same manner as for virtual machines in the original research.

The taxation scheme requires that the system be able to measure the active use of pages in each address space. Waldspurger used a sampling strategy where some number of randomly selected pages for each virtual machine were access protected, forcing *minor page faults* to occur upon the first subsequent reference to those pages, and therefore giving the ESX Server an opportunity to observe those page uses. The same approach could be used within the Linux VMM, where a random sampling of pages in each address space would be access protected. Alternatively, a sampling of the pages' reference bits could be used to monitor idle memory.

Waldspurger observes that, within a normal OS, the use of access protection or reference bits, taken from virtual memory mappings, will not detect references that are a result of DMA transfers. However, since such DMA transfers

LRU lists. However, they only approximate an LRU ordering, and so we refer to them only as *page lists*.

[2] Waldspurger refers to such space as being *idle*.

are scheduled within the kernel itself, those references could also be explicitly counted with help from the DMA scheduling routines.

6.3 Future directions

The description above does not address a particular problem: *shared memory*. Space can be shared between threads or processes in a number of ways, and such space presents important problems that we must solved to achieve a complete solution.

The problem of shared spaces. The assignment of shares to an address space can be complicated when that address space is shared by processes in different process groups or service classes. One simple approach is for each address space to belong to a specific service class. In this situation, its share would be defined only by that service class, and not by the processes that share the space. Another approach would be for each shared address space to adopt the highest share value of its shared tasks or processes. In this way, an important process will not be penalized because it is sharing its space with a less important process.

Note that this problem does not apply only to memory mapped files and IPC shared memory segments, but to any shared space. For example, the threads of a multi-threaded process share a virtual address space. Similarly, when a process calls `fork()`, it creates a new virtual address space whose pages are shared with the original virtual address space using the *copy-on-write (COW)* mechanism. These shared spaces must be assigned proportional shares even though the tasks and processes using them may themselves have differing shares.

Proportionally sharing page faults. The goal of proportional share scheduling is to fairly divide the utility of a system among competing clients (e.g., processes, users, service classes). It is relatively simple to divide the utility of the CPU because that utility is *linear* and *indepedent*. One second of scheduled CPU time yields a nearly fixed number of executed instructions[3]. Therefore, each additional second of scheduled CPU time nearly yields a constant increase in the number of executed instructions. Furthermore, the utility of CPU does not depend on the computation being performed: Every process derives equal utility from each second of scheduled CPU time.

Memory, however, is a more complex resource because its utility is neither independent nor linear. Identical RSS values for two processes may yield vastly different numbers of page faults for each process. The number of page faults is dependent on the reference patterns of each process.

To see the non-linearity in the utility of memory, consider two processes, A and B. Assume that for A, an RSS of p pages will yield m misses, where $m > 0$. If that RSS were increased to $p + q$ pages, the number of misses incurred by A may take any value m' where $0 \leq m' \leq m$[4]. Changes in RSS do not imply a constant change in the number of misses suffered by an address space.

The proportial sharing of memory *space*, therefore, does not necessarily achieve the stated goal of fairly dividing the utility of a system. Consider that A should receive 75% of the system, while B should receive the remaining 25%. Dividing the main memory space by these proportions could yeild heavy page fault-

[3] We assume no delays to access memory, as memory is a separate resource from the CPU.

[4] We ignore the possibility of *Belady's anomaly*[1], in which an increase in RSS could imply an increase in page faults. While this anomaly is likely possible for any real, in-kernel page replacement policy, it is uncommon and inconsequential for real workloads.

ing for A but not for B. Note also that none of the 25% assigned to B may be idle, and so Waldspurger's taxation scheme will not reduce its RSS. Nonetheless, it may be the case that a reduction in RSS by 5% for B may increase its misses only modestly, and that an increase in RSS by 5% for A may reduce its misses drastically.

Ultimately, a system should proportionally share the utility of main memory. We consider this topic a matter of significant future work. It is not obvious how to measure online the utility of main memory for each address space, nor how to calculate target RSS values based on these measurements. Balancing the contention between fairness and throughput for virtual memory must be considered carefully, as it will be unacceptable to achieve fairness simply by forcing some address spaces to page fault more frequently. We do, however, believe that this problem can be solved, and that the utility of memory can be proportionally shared just as with other resources.

7 Conclusion and Future Work

In this paper we make a case for providing kernel support for class-based resource management that goes beyond the traditional per process or per group resource management. We introduce a framework for classifying tasks and incoming network packets into classes, monitoring their usage of physical resources and controlling the allocation of these resources by the kernel schedulers based on the shares assigned to each class. For each of four major physical resources (CPU, disk, network and memory), we discuss ways in which proportional sharing could be achieved using incremental modifications to the corresponding existing schedulers.

Much of this work is in its infancy and the ideas proposed here serve only as a starting point for future work and for discussion in the kernel community. Prototypes of some of the schedulers discussed in this paper are under development and will be made available soon.

8 Acknowledgments

We would like to thank team members from the Linux Technology Center, particularly Theodore T'so, for their valuable comments on the paper and the work on individual resource schedulers. Thanks are also in order for numerous suggestions from the members of the kernel open-source community.

References

[1] L. A. Belady. A study of replacement algorithms for virtual storage. *IBM Systems Journal*, pages 5:78–101, 1966.

[2] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. RFC 2475, Dec 1998.

[3] Jonathan Corbet. A new deadline I/O scheduler. `http://lwn.net/Articles/10874`.

[4] Jonathan Corbet. Anticipatory I/O scheduling. `http://lwn.net/Articles/21274`.

[5] Jonathan Corbet. The Continuing Development of I/O Scheduling. `http://lwn.net/Articles/21274`.

[6] IBM DeveloperWorks. Inbound connection control home page. `http://www-124.ibm.com/pub/qos/paq_index.html`.

[7] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI)*, pages 107–121, 1996.

[8] Bert Hubert. Linux Advanced Routing & Traffic Control. `http://www.lartc.org`.

[9] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *18th ACM Symposium on Operating Systems Principles*, October 2001.

[10] M. Kravetz, H. Franke, S. Nagar, and R. Ravindran. Enhancing Linux Scheduler Scalability. In *Proc. 2002 Ottawa Linux Symposium, Ottawa*, July 2001. `http://lse.sourceforge.net/scheduling/ols2001/elss.ps`.

[11] Paul E. McKenney. Stochastic Fairness Queueing. In *INFOCOM*, pages 733–740, 1990.

[12] Ingo Molnar. Goals, Design and Implementation of the new ultra-scalable O(1) scheduler. In 2.5 kernel source tree documentation (Documentation/sched-design.txt).

[13] Jason Nieh, Chris Vaill, and Hua Zhong. Virtual-time round-robin: An o(1) proportional share scheduler. In *2001 USENIX Annual Technical Conference*, June 2001.

[14] P. Pradhan, R. Tewari, S. Sahu, A. Chandra, and P. Shenoy. An Observation-based Approach Towards Self-Managing Web Servers. In *IWQoS 2002*, 2002.

[15] Quality of Service White Paper. Integrated QoS: IBM WebSphere and Cisco Can Deliver End-to-End Value. `http://www-3.ibm.com/software/webservers/edgeserver/doc/v20/QoSwhitepap%er.pdf`.

[16] Prashant J. Shenoy and Harrick M. Vin. Cello: A disk scheduling framework for bext generation operating systems. In *ACM SIGMETRICS 1998*, pages 44–55, Madison, WI, June 1998. ACM.

[17] Antonio Vargas. fairsched+O(1) process scheduler. `http://www.ussg.iu.edu/hypermail/linux/kernel/0304.0/0060.html`.

[18] Dinesh Verma, Mandis Beigi, and Raymond Jennings. Policy based SLA Management in Enterprise Networks. In *Policy Workshop*, 2001.

[19] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel Mechanisms for Service Differentiation in Overloaded Web Servers. In *2001 USENIX Annual Technical Conference*, Jun 2001.

[20] Carl A. Waldspurger. Memory resource management in {VM}ware {ESX} {S}erver. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, December 2002.

[21] Carl A. Waldspurger and William E. Weihl. Stride scheduling: Deterministic proportional- share resource management. Technical Report MIT/LCS/TM-528, 1995.

[22] Werner Almesberger Werner and Jamal Hadi Salim amd Alexye Kuznetsov. Differentiated Services on Linux. In *Globecom*, volume 1, pages 831–836, 1999.

[23] J. Wroclawski. The Use of RSVP with IETF Integrated Services. RFC 2210, Sep 1997.

Trademarks and Disclaimer

Linux Support for NUMA Hardware

Matthew Dobson, Patricia Gaughen, Michael Hohnbaum
IBM LTC, Beaverton, Oregon, USA
colpatch@us.ibm.com, gone@us.ibm.com, hohnbaum@us.ibm.com
Erich Focht
NEC HPCE, Stuttgart, Germany
efocht@hpce.nec.com

Abstract

New large CPU-count machines are being designed with non-uniform memory architecture (NUMA) characteristics. The 2.5 Linux® kernel includes many enhancements in support of NUMA machines. Data structures and macros are provided within the kernel for determining the layout of the memory and processors on the system. These enable the VM subsystem to make decisions on the optimal placement of memory for processes. This topology information is also exported to user-space via `sysfs`. In addition to items that have been incorported into the mainline Linux kernel, there are NUMA features that have been developed that continue to be supported as patchsets. These include NUMA enhancements to the scheduler, multipath I/O, and a user-level API that provides user control over the allocation of resources in respect to NUMA nodes.

1 Introduction

1.1 Non-Uniform Memory Architecture

Demand for greater computing capacity has lead to the increased use of multi-processor computers. Most multi-processor computers are considered Symmetric Multi-Processors (SMP) as each processor is equal and has equal access to all system resources (e.g., memory and I/O busses). SMP systems generally are built around a system bus that all system components are connected to, and is used to communicate between the components. As SMP systems have increased their processor count, the system bus has increasingly become a bottleneck. One solution that is gaining in use by hardware designers is Non-Uniform Memory Architecture (NUMA).

NUMA systems co-locate a subset of the system's overall processors and memory into nodes, and provide a high speed and high bandwidth interconnect between the nodes, see Figure 1. Thus there are multiple physical regions of memory, but all memory is tied together into a single cache-coherent physical address space. The resulting system has the property such that for any given region of physical memory, some processors are closer to it than other processors. Conversely, for any processor, some memory is considered local (i.e., it is close to the processor) and other memory is remote. Similar characteristics may also apply to the I/O busses—that is, I/O busses may be associated with nodes.

While the key characteristic of NUMA systems is the variable distance of portions of memory from other system components, there are nu-

merous NUMA system designs. At one end of the spectrum are designs where all nodes are symmetrical—they all contain memory, CPUs, and I/O busses. At the other end of the spectrum are systems where there are different types of nodes—the extreme case being separate CPU nodes, memory nodes, and I/O nodes. All NUMA hardware designs are characterized by regions of memory being at varying distances from other resources, thus having different access speeds.

To maximize performance on a NUMA platform, Linux must take into account the way the system resources are physically laid out. This includes information such as which CPUs are on which node, which range of physical memory is on each node, and what node an I/O bus is connected to. This type of information describes the topology of the system.

There are several challenges Linux must address to provide NUMA support. These include:

- discovery and internal representation of the system topology
- minimization of traffic over the interconnect between nodes
- localization of memory references
- fair access to locks
- I/O locality
- synchronization of time between nodes
- the location of low address memory (e.g., memory with physical address under 4 GB) all on the first node, on i386™ processor (and potentially other 32-bit processor architectures) based machines
- scheduling of processes and groups of processes on the same node.

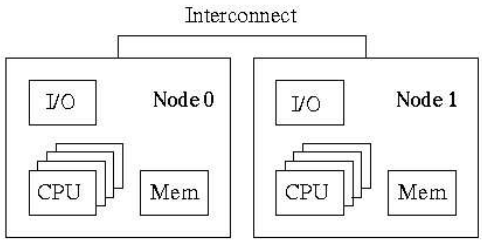


Figure 1: Simple view of NUMA system

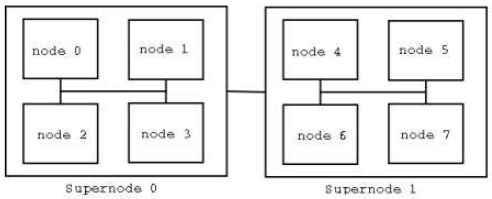


Figure 2: NUMA system with supernodes

Linux running on a NUMA system obtains optimal performance by keeping memory accesses to the closest physical memory. For example, processors benefit by accessing memory on the same node (or closest memory node), and I/O throughput gains by using memory on the same (or closest) node to the bus the I/O is going through. At the process level, it is optimal to allocate all of a process's memory from the node containing the CPU(s) the process is executing on. However, this also requires keeping the process on the same node.

This paper looks at how Linux addresses these NUMA challenges, focusing on the NUMA support that is available in the 2.5 development kernel. In addition, some discussion is included about additional NUMA support that is under development for future Linux releases.

1.2 Hardware Implementations

There are many design and implementation choices that result in a wide variety of NUMA platforms. This variety creates additional chal-

lenges for the Linux OS developer, as a single solution is desired to support the many types of NUMA hardware. This section discusses hardware implementations, and provides examples and descriptions of NUMA hardware implementations.

Types of nodes

The most common implementation of NUMA systems consists of interconnecting symmetrical nodes. In this case, the node itself is an SMP system that has some form of high speed and high bandwidth interconnect linking it to other nodes. Each node contains some number of processors, physical memory, and I/O busses. Typically, there is a node-level cache. This type of NUMA system is depicted in Figure 1.

A variant on this design is to only put the processors and memory on the main node, and then have the I/O busses separate. Another design option is to have separate nodes for processors, memory, and I/O busses which are all interconnected.

It is also possible to have nodes which contain nodes, resulting in a hierarchical NUMA design. This is depicted in Figure 2.

Types of interconnects

There is no standardization of interconnect technology. More relevant to Linux is the topology of the interconnect. NUMA machines exist that use the following interconnect topologies:

- ring topology – each node is connected to the node on either side of it. Memory access latencies can be non-symmetric; that is, accesses from node A to node B might take longer than accesses from node B to node A.

- crossbar interconnect – all nodes connect into a common crossbar.

- point to point – each node has a number of ports to connect to other nodes. The number of nodes in the system is limited to the number of connection ports plus one, and each node is directly connected to each other node.

- mesh topologies – more complex topologies that, like point to point topologies, are built upon each node having a number of connection ports. But unlike point to point topologies, there is not a direct connection between each node. Hypercube and torus topologies are examples of mesh topologies.

The topology provided by the interconnect affects the distance between nodes. This distance needs to be accounted for when Linux is making resource placement decisions.

Latency ratios

One important measurement for determining the “NUMA-ness” of a system is the latency ratio. This is the ratio between memory latency for on-node memory access versus off-node memory accesses. Depending upon the topology of the interconnect, there might be multiple off-node latencies. This latency can be used to analyze the cost of memory references to different parts of the physical address space, and thus be used to influence decisions affecting memory usage.

Specific NUMA implementations

Several hardware vendors are building NUMA machines that run the Linux operating system. This section briefly describes some of these machines, but is not an all inclusive survey of the existing implementations.

One of the earlier commercial NUMA machines is the IBM® NUMA-Q® box. This machine is based upon nodes which contain 4 processors (i386), memory and PCI busses. Each node also contains a management module to coordinate booting, monitor environmentals, and communicate with the system console. The nodes are interconnected using a ring topology. Up to 16 nodes can be connected for a maximum of 64 processors and 64 GB of memory. Remote to local memory latency ratios range from 10:1 to 20:1. Each node has a large remote cache which helps compensate for the large remote memory latencies. Much of the Linux NUMA development has been on these boxes due to their availability.

NEC builds NUMA boxes using Intel™ Itanium™ processors. The most recent system in this line is the NEC TX7. The TX7 supports up to 32 Itanium2 processors in nodes of 4 processors each. The nodes are connected by a crossbar and grouped in two supernodes of four nodes each. The crossbar provides fast access to non-local memory with low latency and high bandwidth (12.8 GB/second per node). The memory latency ratio for remote to local memory in the same supernode is 1.6:1. The remote to local memory latency ratio for outside the supernode is 2.1:1. There is no node level cache. I/O devices are connected through PCI-X busses to the crossbar interconnect and thus are all the same distance to any CPU/node.

The large IBM xSeries® boxes use Intel processors and the IBM XA-32™ chipset. This chipset provides an architecture that supports four processors, memory, PCI busses, and three interconnect ports. These interconnect ports allow point to point connection of up to 4 nodes for a 16 processor system. Also supported is a connection to an external box with additional PCI slots to increase the I/O capacity of the system. The IBM x440 is built on this architecture with Intel Xeon™ processors.

2 Linux NUMA Support

The basic infrastructure for supporting NUMA hardware has been incorporated into the Linux 2.5 development kernel. This support includes topology discovery and internal representation, memory allocation, process scheduling, and timer support. In addition there are kernel extensions in support of NUMA that are not yet included in the mainline kernel, but are being maintained as separate patchsets. These include NUMA-aware multi-path I/O, application-level directed binding of memory to nodes, and scheduler extensions.

2.1 CONFIG Options

Most NUMA support options within the kernel are enabled by the CONFIG_NUMA option. This includes the scheduler extensions, NUMA memory allocation, and topology support. There is also an option, CONFIG_DISCONTIGMEM, that is used for enabling a portion of the NUMA memory support.

2.2 Linux Architecture Support

Linux supports many types of processors and many hardware architectures. Within Linux, when reference is made to an architecture it typically refers to the processor type (e.g., i386, Power4™, alpha, etc.). Subarchitectures are used to refer to a substantially different hardware implementation of a particular architecture. Also, within an architecture there are

platforms which are an implementation of the architecture.

For example, the x440 is part of the i386 architecture within Linux. However, the x440 is also a unique subarchitecture within the i386 architecture. Another example is the IA64 architecture which has DIG-64 and HP platforms, and SGI-SN1 subarchitecture.

Throughout this paper references are made to architecture which are more correctly subarchitectures. References to architectures are meant to refer to a specific hardware implementation of a NUMA system.

3 Topology

There are performance penalties involved in accessing hardware devices (CPUs, memory, disks, network cards, etc.) that are remote to the currently executing CPU. These can be significantly reduced by having a knowledge of the system's topology, and using that information to make good scheduling, allocation, migration, and I/O decisions. Topology information is crucial to the kernel for making good decisions on a NUMA machine, but this information is also important to some user-space applications as well.

Topology information is currently used in the kernel to schedule processes and allocate memory. This has contributed to performance improvements for NUMA architectures throughout the 2.3, 2.4, and 2.5 kernel series.

3.1 Topology Elements

The topology of a system includes all hardware components that make up the system. However, for the context of this paper, topology is restricted to those physical elements which are directly affected by the NUMA characteristics of the system. These elements are nodes, processors, memory, and I/O busses. Physical components not considered here consist of the actual I/O devices which are connected into the system through the I/O busses.

CPUs provide the computing power in the system. The location of the individual CPUs in the overall topology is extremely important for scheduling decisions. The current in-kernel topology API exposes 4 CPU related functions: `cpu_to_node()`, `node_to_cpumask()`, `node_to_first_cpu()`, and `pcibus_to_cpumask()`. Information about these functions is provided in the next section. The two-node system in Figure 1, contains 8 CPUs, 4 on node 0, and 4 on node 1.

A Memory Block, or memblk for short, represents a physically contiguous piece of memory. It is typically used to represent all the memory in a particular memory bank on a particular node. A node is permitted to have either 0 or 1 memory blocks. For example, in Figure 1, we have a two-node system. Its total memory is split into two memblks, one on node 0, one on node 1. In UP/SMP systems, all the memory in the system is represented by a single memory block.

I/O bus elements represent physical I/O busses in the underlying system. These are important elements for operations like scheduling disk I/O, networking tasks, or any I/O intensive process. By utilizing knowledge about I/O locality, processes can ensure they run efficiently by constraining themselves to CPUs and memory blocks on or close to the I/O bus they utilize.

When discussing NUMA, the word node is often overloaded. For the purposes of a Linux topology discussion, a node is solely an abstract container. Nodes are not meant to represent any physical element of the underlying architecture. All elements, including nodes themselves, are children of a node in the sys-

tem. The node element is designed to be a medium through which queries can be made. For example, in Figure 1, we see a simple illustration of a two node system. Each node has 4 CPUs in it, as well as a block of memory. To find out which memory block is closest to CPU 3, a process can determine that CPU 3 is a child of node 0, and that memblk 0 is also a child of node 0. Thus, for efficiency, a process running on CPU 3 would want to be sure that its memory is allocated from memblk 0.

On some systems nodes can be nested, as seen in Figure 2. This is important with things like hyperthreading and multi-core processors becoming more available, which can be easily represented as a small node with processors, but no memory or I/O busses. Another usage of nested nodes is for hierarchical NUMA systems, such as the NEC TX7, which is built with supernodes that contain nodes.

There exists a strong possibility that there will be a need to introduce new topology elements in the future. Due to the simplicity of the design of the topology subsystem, adding new elements is a straightforward procedure. As long as there is a parent-child relationship between the new element and nodes, the new element should drop right in to the existing infrastructure.

3.2 Topology Kernel Functions

The following is a list of topology-related kernel calls that form the basis of the current topology framework. Along with the description of the call is a default return value for the non-NUMA case. Architecture specific definitions of these kernel calls are provided for each architecture that has NUMA topology support. Most architectures simply use the default asm-generic/topology.h version, usually because the architecture does not support NUMA.

- `cpu_to_node(int cpu)` – Returns the number of the node containing CPU `cpu`. For non-NUMA, defaults to 0.
- `memblk_to_node(int memblk)` – Returns the number of the node containing memory block `memblk`. For non-NUMA systems, defaults to 0.
- `parent_node(int node)` – Returns the number of the node containing node `node`. If the node number returned is `node`, `node` is a top-level node. For non-NUMA, defaults to 0.
- `node_to_cpumask(int node)` – Returns a bitmask of the cpus on Node `node`. For non-NUMA, defaults to `cpu_online_map`.
- `node_to_first_cpu(int node)` – Returns the number of the first CPU on Node `node`. For non-NUMA, defaults to the lowest-numbered CPU in the system.
- `node_to_memblk(int node)` – Returns the number of the Memory Block, if any, on Node `node`. For non-NUMA, defaults to 0.
- `pcibus_to_cpumask(int bus)` – Returns a bitmask of the CPUs closest to PCI bus `bus`. For non-NUMA, defaults to `cpu_online_map`.
- `numa_node_id()` – Returns the number of the node containing the current CPU. For non-NUMA systems, defaults to 0.

3.3 Closest Element versus Distance Matrix

The current implementation of the topology system is very helpful if the caller is looking for information relating to the closest element. This choice was made primarily because this made the code small and compact,

but also because the majority of consumers of this information simply want the closest element. The other option, and possible future method, is to use a distance matrix approach. Using this approach, each machine type would build a latency matrix representing the distance from element X to element Y. The distance matrix approach allows us much more flexibility when retrieving information, and much more complicated queries can be satisfied. We decided against this approach, however, because it was determined that the added complexity did not offer that much benefit, and likely would have few consumers. However, in hierarchical NUMA systems this type of approach is more likely to be required for optimal performance benefits.

Exporting Topology Information to User Space

Topology information is important to multi-threaded user-space applications. With a large parallel NUMA machine, threads can be coordinated across, but more importantly within, nodes. This can yield significant speedups over a standard SMP version run on the same machine. There is also a proposal to facilitate the sharing of memory regions by establishing bindings for those regions. This would allow multi-threaded applications to specify memory blocks close to the set of CPUs the group of threads is executing on, and guarantee pages faulted into specific memory regions (likely shared) come from those memory blocks.

User-space applications currently have access to NUMA topology information through `sysfs`. The information is laid out following the normal directory structure. Node directories contain CPU and memory block directories, as well as other node directories on machines that take advantage of nested nodes. Each of these directories have files in them, with those files containing various bits of information that can be read and/or written. For example, the nodes contain a `cpumap` file that contains a bitmap of CPUs on that node.

Currently I/O busses are not represented in the topology directory of `sysfs`. Adding this is a future work item.

There is currently no way for a user-space application to determine the CPU it is currently executing on. This data is inherently volatile, as it requires going into kernel-space to get it, and while returning from the kernel it is possible for the process to be switched to another CPU thus invalidating the information that is about to be returned. There are other ways to give user-space access to this information; for example, by mapping a page that is shared between user-space and kernel-space and having the kernel store the CPU that the process is currently executing on at a set location within that page.

4 Memory

This section describes some of the issues encountered during the development of 2.5 to support NUMA memory allocation, and the Linux implementation to address the issues. The purpose of this section is not to provide an in depth look at the Linux memory subsystem—there is documentation available on the net for that [1].

4.1 Discontiguous Memory Support

Each architecture needs to describe its physical layout to the kernel. This includes specifying which address ranges belong to which node, and whether there are holes in between those ranges (a hole is a physical address range for which there is no real memory). CONFIG_DISCONTIGMEM is currently used to represent a solution to some of these problems. The name of the config option is a bit of a misnomer

because the memory may not be discontiguous. In the case of the IBM x440 the memory is contiguous, except for a large hole on the first node.

The core data structure for describing the physical layout is the `pg_data_t`. This data structure currently has a 1:1 mapping to nodes. For each node in the system, there exists one `pg_data_t`. The `pg_data_t` describes the start and end of memory for the node, a pointer to the zones for the node, and related information. Support for multiple `pg_data_t`'s have been in the kernel since 2.4 (although several fixes and optimizations have occured since then). It is up to each architecture to populate these correctly for their system.

The config option, CONFIG_DISCONTIGMEM turns on the functionality for creating multiple `pg_data_t`'s. CONFIG_NUMA turns on the code (i.e. scheduling decisions, allocation decisions) that makes use of the per node `pg_data_t`'s.

Zone Normal Memory on 32-bit Systems

During the setup of memory in system initialization a special allocator is used—the bootmem allocator. This allocator only allocates memory out of what will later become ZONE_NORMAL. Once memory is setup the bootmem allocator is no longer used. The `bootmem_data_t` represents the address range used by the bootmem allocator. The `pg_data_t` for the node containing the memory has a pointer to the `bootmem_data_t` (`bdata`).

One of the early issues ran into during the development of i386 CONFIG_DISCONTIGMEM support was the idea that not all `pg_data_t`'s will have a portion of ZONE_NORMAL, or a `bootmem_data_t`. On i386, ZONE_NORMAL is limited to the first 896 Mb of physical memory because of limitations of the 32-bit architecture. So, a system populated with 1GB of RAM per node will only have a ZONE_NORMAL (and ZONE_DMA) on node 0, the rest of the nodes will only contain ZONE_HIGHMEM. Because the slab allocator only allocates memory from ZONE_NORMAL, and the kernel uses the slab allocator to allocate memory for internal data structure, most kernel related memory will be on node 0. This also means that during early boot only the first node's memory will be used by the bootmem allocator.

Two changes were made to make ZONE_NORMAL only on node 0 work: (1) a dereferencing a null pointer bug was fixed in `__alloc_pages()` that didn't check that the node had ZONE_NORMAL before using it. (2) `alloc_bootmem_node()` and friends needed to be made to only use the first node's `pg_data_t`, because the other nodes would not have a `bootmem_data_t`. Since `alloc_bootmem_node()` is architecture independent, it was important to not put arch specific requirements in the code, so the changes were made in the header files. Thus the creation of CONFIG_HAVE_ARCH_BOOTMEM_NODE.

Page to Node Translation

Finding the node a memory address belongs to is used throughout the kernel. The core routine doing the translation from address to node id, is `pfn_to_nid()`. Because it is called often, it is important that the translation is fast. On some architectures, the physical layout is such that the first 64GB of address space belongs to the first node (whether or not there really is 64GB of RAM), second 64GB for the second, and so on. This makes the algorithm for figuring out what node an address belongs to

very simple, and very fast: if the address is between 0-64GB it's node 0. But on i386 NUMA architectures that have been tested, it is not as clear. Because the memory is contiguous, there isn't a nice GB to node translation. The solution was to create a mapping of addresses to node IDs on 256MB address ranges. The map is created during memory setup and allows for fast translations.

4.2 Node Aware Memory Allocation

One of the features enabled by CONFIG_NUMA is that the system makes NUMA-aware memory allocation decisions. The current policy is when memory is allocated, the kernel tries to allocate from the local node. If that fails, the allocator will allocate from the other nodes. The exception is in the case of a system with ZONE_NORMAL only on the first node; in an i386 NUMA box for example, memory allocated from ZONE_NORMAL will only be allocated from the first node.

The allocation policies only apply to new memory. Should a process migrate across node, the memory related to the process will not be migrated. Although if the memory is swapped out, when the pages are swapped back in they will be swapped to the node the process has migrated to. One thing to keep in mind, is that migrating the memory is expensive; however, if a page is being accessed often, it would be a performance benefit to move it to the local node.

When the kernel is attempting to allocate memory, and the system is low on pages, `kswapd` will be woke to address the low memory issue. Without NUMA awareness `kswapd` may free up lots of memory by swapping pages out, but it may not make available memory local to the node that is in need of memory. The solution was to make `kswapd` per node. `kswapd` monitors the memory on the local node. When memory needs to be freed, it's freed from the local node. Rmap [3] made this change to `kswapd` possible, because of the ability to find the virtual address(es) associated with a physical address (local to the node).

4.3 Node Local Kernel Data Structures

For kernel data structures that are frequently accessed and have node specific information, it makes sense to have their data structures in node local memory. On most architectures, when the bootmem allocator is available, it is possible to allocate memory on a specific node through the use of `alloc_bootmem_node()`. However, on i386 the bootmem allocator only allocates from node 0, so `alloc_bootmem_node()` doesn't work for allocating per node and all memory is allocated from node 0. Because of the limited lifespan of the bootmem allocator, `alloc_bootmem_node()` is not a complete solution. No other generic mechanism is available at this time for allocating data structures on a per node basis. A possible solution for a generic mechanism is currently in development by Bill Irwin [2].

To work around this 32-bit architecture limitation, for the specific case of the `mem_map` and `pg_data_t`, Martin Bligh has successfully made these two types of data structures reside in node local memory. That is, these structures are located on the node for the memory that they are describing.

The first phase was to make `mem_map` per node. This was done by reserving pages at the top of the node and decrementing the size of the address space by the size of `mem_map`, and then making use of that reserved space when `mem_map` was set up. Nothing special had to be done for node 0, because it is where the bootmem allocator gets it memory for node 0's data. So, for node 0 the the normal bootmen allocator can be used. Phase two was to make

`pg_data_t` per node. This was done using the same method as for the `mem_map`.

4.4 Replication

Since kernel text is read-only on production systems, there is little downside to replicating it and placing a copy on each node. This does consume extra memory, but kernel text is relatively small and memories of NUMA machines relatively large. Kernel-text replication requires special handling from debuggers when setting breakpoints and from `/dev/mem` in cases where users are sufficiently insane to modify kernel text on the fly. In addition, kernel-text replication means that there is no longer a single "well-known" offset between a kernel-text virtual address and the corresponding physical address.

This functionality is present in some architectures (e.g., sgi-ip27) in the 2.4 kernel. Also, the IA64 discontigmem patch provides kernel text replication support for IA64. It is not likely to show up in the i386 tree because of the limitations of the architecture.

4.5 Memory Binding

As mentioned in other sections of this paper, writing code to run on a NUMA machine can require changes to take advantage of the interesting hardware configurations these machines offer. Memory Binding is one API that we feel large user-space programs will be able to use to make significant performance improvements for NUMA. The idea behind Memory Binding is that processes can selectively bind ranges of their virtual memory space to particular blocks of memory, according to different allocation policies. For example, a large database program that has many threads could bind its threads to CPUs on two nodes, and also bind a large section of its shared memory to the memory that belongs on those two nodes. By setting a policy that enforces an equal distribution of pages, the database could be sure that all its shared pages are at least on the same set of nodes as its processes, and that the memory is evenly spread across those nodes. The Memory Binding API is available as a patch from:

```
http://www-124.ibm.com/linux/
patches/?patch_id=753
http://www-124.ibm.com/linux/
patches/?patch_id=754
```

5 NUMA scheduler

5.1 Introduction

As explained in the introductory section, accessing the memory of a remote node implies taking penalties in memory access latency and bandwidth. Therefore, it is desirable to keep processes on or near the node on which their memory (or most of it) is allocated.

The old (pre 2.5) Linux scheduler wasn't aware of the NUMA structure of a machine. Processes could migrate to any CPU in the system if the CPU was less loaded. On NUMA machines with many CPUs, two scalability problems were additionally limiting the performance: the CPUs were competing for the runqueue lock and the time needed for selecting the task to be scheduled next was linearly growing with the length of the runqueue.

The scalability problems were mostly solved by the $O(1)$ scheduler[4]. Like other approaches [5, 6] it implements per CPU runqueues[1] avoiding the lock starvation problem. Additionally it implements an $O(1)$ search algorithm for the task to be scheduled next. The scalability problems for SMP machines were solved, but the $O(1)$ scheduler was not

[1] There are actually two runqueues per priority level per CPU.

NUMA-aware, either. An idle CPU could easily steal a task from the node where its memory was allocated letting it run with degraded memory performance.

5.2 NUMA scheduler approaches

The first notable Linux NUMA scheduler was the one Andrea Arcangeli made on top of the old Linux scheduler[7]. It implemented per node runqueues and scheduled across node boundaries only after failing to find an optimal CPU within the same node. Being built on top of the old Linux scheduler this approach suffered of very similar scalability limitations.

Another approach was the extension of the IBM MQ scheduler [5] to allow rescheduling only inside pools of CPUs [8]. A loadbalancing module was added which allowed periodic rebalancing across the pool boundaries.

The first NUMA scheduler on top of the $O(1)$ scheduler was designed and implemented by Erich Focht [9]. Tasks were assigned a home node at creation time (either at `fork()` or at `exec()`, selectable for each task), allocated their memory on (or near) the home node, and were attracted by the home node CPUs. The tasks were node-affine. Because the scheduler changes were too complex for inclusion into the 2.5 kernel baseline, Erich Focht, Michael Hohnbaum, Martin Bligh, and Andrew Theurer collaborated with the target to strip down and rewrite the node-affine scheduler to a slim NUMA variant acceptable for inclusion. The result was included into the 2.5.59 kernel and is described in the following section.

5.3 NUMA scheduler in the 2.5 kernel: implementation

When stripping down the node-affine scheduler, the goals were to keep the changes to the $O(1)$ scheduler as small as possible, and to add NUMA awareness by making it difficult for a task to change the node while trying to keep the node load well-balanced. This was achieved by three patches.

Initial load balancing at `exec()`

The NUMA support for the memory subsystem described in section 4 ensures that memory pages are allocated from the node on which the page-faulting task is running[2]. Normally processes allocate most of their memory right after creation; therefore, the choice of the initial node and CPU is very important for getting well-balanced nodes.

Initial load balancing implies some overhead because it involves scanning the current node loads and determining the best CPU on which the freshly created task should be scheduled. This can be done either at `fork()` or at `exec()`. Doing it at `fork()` (and `clone()`) has the advantage that multi-threaded jobs lead to a balanced machine as well. This might be desireable on machines with good latency ratios between the nodes. On the other hand, every small and short living thread picks up the initial balancing overhead, unnecessarily migrates pages to other nodes by copy on write (COW), and finds a cold instruction cache.

Doing initial balancing at `exec()` avoids the COW problem because all pages are dropped at that stage. Short-living threads which don't `exec()` benefit from a warm instruction cache. But long running memory intensive multi-threaded programs might pick up performance penalties due to the unbalanced nodes.

The implementation adds the array `static atomic_t node_nr_running[MAX_NUMNODES]` to keep track of the num-

[2] If the current node has sufficient free memory.

ber of tasks running on each node. `kernel/sched.h` is extended by three functions:

- `sched_best_cpu()`: Finds the least loaded CPU on the least loaded node using the current runqueue lengths.
- `sched_migrate_task()`: Migrates a task to a certain CPU.
- `sched_balance_exec()`: Called by `do_execve()`, it moves the current task to the least loaded CPU.

Intra-node load balancing

The `load_balance()` and `find_busiest_queue()` functions of the $O(1)$ scheduler have been modified to restrict the search for the busiest CPU to the set given by the new `cpumask` argument. In the NUMA scheduler this mask uses topology information and usually limits the search to the current node. To be precise: all calls to `load_balance()` except the one from the timer interrupt are balanced only within the current node.

Cross-node load balancing

Even with a perfect initial load balancing, a machine can easily end up with poorly balanced nodes, e.g. nodes with more running tasks than available CPUs and idle nodes. In such cases, it is preferrable to use the idle CPUs for doing real work even if the tasks running on them need to access memory from other nodes. It is better if a task runs slower on a remote node instead of waiting for the CPU on its own node. The cross-node balancing occurs periodically during the timer interrupt, with the current settings (kernel 2.5.67) this means: an idle CPU will try node-rebalancing every 5 ticks (5ms on a `HZ=1000` system); a busy CPU will do it every 20s.

There continues to be debate as to the frequency of the busy rebalance, with some believing the busy rebalance is occuring much too infrequently. It is felt that the current frequency, while showing advantages on simple benchmarks is not optimal for real world conditions.

Node rebalancing is achieved by a change in `scheduler_tick()` and three additional routines:

- `rebalance_tick()`: Decides when to balance within the node and when across the node boundaries. In the later case it will first try an intra-node rebalance.
- `balance_node()`: Calls `load_balance()` with `cpumask` set to the least-loaded node plus the current CPU.
- `find_busiest_node()` Finds busiest node and uses a geometrically decaying weight for the load measure: $load_t = load_{t-1}/2 + nr\_node\_running_t$. This flattens out sudden load peaks.

5.4 Current limitations and future developments

The NUMA scheduler currently implemented in the Linux kernel is far from being complete. The degree of NUMA-awareness of the scheduler gives clear performance boosts for "simple" load situations like parallel kernel compiles or an arbitrary but more or less constant number of similar and long running `exec`'d processes. The limitations are shown in environments with long running jobs, and in suddenly varying loads, or with long running multithreaded applications like OpenMP.

The stripped down version of the node-affine scheduler strongly reflects the influence of former IBM work [8]. Some of the useful features of [9] were lost, among them the capability of a process to remember the node on which its memory resides and to return to that node. A scheduler with such features is in production on NEC TX7 IA64 servers and shows significant benefits in production environments. Thus possible extensions of the 2.5 NUMA scheduler could be:

- An option to allow particular tasks to initially balance their children at `fork()`.
- A method of keeping track of where one task's memory is.
- A method of pushing tasks to the node where most of their memory resides.

6 Locking

In contrast to much of the other NUMA work, NUMA-aware locking is not about making a per-node lock, but rather it is about preventing lock starvation on highly-contended locks. Lock starvation occurs when the contention on a given lock is so high that by the time a CPU releases the lock, at least one other CPU on that same node is requesting it again. NUMA latencies mean that these local CPUs can acquire the lock when it becomes available faster than remote CPUs can. On some architectures, the CPUs on the node where the lock is located can monopolize the lock, completely starving CPUs on other nodes.

The best solution is to reduce lock contention, but NUMA-aware locks can be an interim fix while the locking design is reworked.

Two fair locking primitives are:

- `mcs locks` [10]

  mcs locks are queued locks. The primitive enforces fairness because requesters are queued. The queuing ensures that the local CPU does not have an advantage on getting the lock. It's first come, first served.

- `NUMA-aware locks` [11]

  NUMA-aware locks enforce fairness by using a round-robin system amongst nodes waiting for a lock. The implementation was written so that the fairness algorithm could be modified to fit the need. This means, if round robin proves inefficient, another method can be inserted.

The work in the area of NUMA-aware locking is currently not active. As previously mentioned, the Linux solution for a highly-contended lock is to break the lock up, and so far lock starvation has not been seen to be a problem. Therefore a need for a NUMA-aware lock has not been established.

7 I/O

As with many aspects of writing software to run efficiently on NUMA platforms, I/O code benefits from fine-tuning for these machines. The following section goes into more detail about: why I/O subsystems require NUMA considerations, the current state of Linux support of I/O on NUMA hardware, and where it might be going.

7.1 I/O Locality

As discussed in the topology section, on NUMA machines I/O busses are ususally spread across nodes. When scheduling I/O we attempt to ensure that the memory being used for the I/O is close to the specific I/O bus we are using. Cross-node I/O requests

suffer a performance penalty when compared to I/O requests that are constrained to a single node. Cross-node I/O travels across the node interconnect busses and has the potential to consume interconnect bandwidth, thus degrading the performance of other processes. If the memory buffers used for I/O are physically located in memory far from the I/O bus, there will also be delays for cross-node memory access. Ideally, the requesting process executes on a CPU on the node with the memory and I/O bus, thus eliminating any inter-node accesses.

7.2 Multi-Path I/O

While Multi-Path I/O, or MPIO for short, is not a new concept, it can be a particularly powerful tool on a NUMA platform. MPIO involves using multiple I/O adaptors (i.e., SCSI cards, network cards) to gain multiple paths to the underlying resource (i.e., hard disks, the network), thus increasing overall bandwidth. On SMP platforms, potential speedups due to MPIO are limited by the fact that all CPUs and memory typically share a bus, which has a maximum bandwidth. On NUMA platforms, however, different groups of CPUs, memory, and I/O busses have their own distinct busses. This allows potentially achieving larger aggregrate I/O throughput by allowing each node to independently reach its maximum bandwidth. An ideal MPIO on NUMA setup consists of an I/O card (SCSI, Network, etc.) on each node connected to every I/O device, so that no matter where the requesting process runs, or where the memory is, there is always a local route to the I/O device. With this hardware configuration, it is possible to saturate several PCI busses with data. This is even further assisted by the fact that many machines of this size will be using RAID or other MD devices, thus further increasing the potential bandwidth by using multiple disks.

There is a patch, currently against 2.5.59, that implements MPIO for the SCSI Mid-Layer in Linux. The SCSI layer is in the midst of many changes right now, some of which affect algorithms this patch was based on. This patch [12] is maintained by Patrick Mansfield, and is being discussed in vastly more detail at another presentation at OLS.

7.3 Interrupt Routing and Balancing

Interrupt handling is another area where ignoring NUMA locality issues can be costly. When dealing with interrupts, it is important that they are handled locally. Some architectures and APIC setups prevent interrupts from being handled remotely by their design, but for those that don't, we must make sure that interrupts are kept local. What this means is that if, for example, an I/O device raises an interrupt, it should be handled by a CPU on the same node as the I/O device. At the same time, we don't want every interrupt occurring on a particular node to be handled by the same CPU. Currently the Linux kernel takes advantage of the balance IRQ functionality, which changes the destination of individual IRQs to a different CPU after a certain number of ticks. This code is not aware of NUMA topology, though, and thus may sometimes make poor IRQ destination decisions. There is significant work to be done still in this area for NUMA support.

On some chipsets, IRQ balancing is provided by the hardware, for example the 460GX realted chipsets (used by the NEC TX7). This chipset provides either a fixed redirection or can redirectable within a target node based on priorities.

8 Timers

On UP systems, the processor has a time source that is easily and quickly accessible, typically implemented as a register. On SMP systems,

the processors' time source is usually synchronized as all of the processors are clocked at the same rate, and thus synchronization of the time register between processors is a straight forward task.

On NUMA systems synchronization of the processors' time source is not practical as not only does each node have its own crystal providing the clock frequency, but there tend to be minute differences in the frequencies that the processors are driven at which thus leads to time skew.

On multi-processor systems it is imperative that there is a consistent system time. Otherwise time stamps provided by different processors cannot be relied upon for ordering and if a process is dispatched on a different processor it is possible that there can be unexpected jumps (backward or forward) in time.

Ideally, the hardware provides one global time source with quick access times. Unfortunately, global time sources tend to require off-chip access and often off-node access which tend to be slow. Clock implementations are very architecture specific, with no clear leading implementation amongst the NUMA platforms. On the x440, for example, the global time source is provided by node 0 and all other nodes must go off-node to get the time.

In Linux 2.5, the i386 timer subsystem has an abstraction layer that simplifies the addition of a differnt time source provided by a specific machine architecture. For standard i386 architecture machines, the TSC is used which provides a very quick time reference. For NUMA machines, a global time source is used (e.g., on the x440 the cyclone timer).

9 Summary

Much work has been done to provide NUMA support for the Linux kernel. At this point, the basic infrastructure is in place. Performance testing has shown measureable improvements, though they tend to be widely variable dependent upon the workload and the NUMA hardware. As Linux gets used on more NUMA hardware platforms, there are bound to be additional areas exposed which will benefit from additional NUMA optimizations.

Some areas that are actively being worked on or considered for future work are:

- I/O busses in `sysfs` topology
- MPIO
- scheduler enhancements
- interrupt routing and balancing
- kernel data structure placement
- memory binding
- page migration
- timers

Legal Statement

References

[1] M. Gorman: "Understanding the Linux Virtual Memory Manager," April 2003, http://www.csn.ul.ie/~mel/projects/vm/guide/html/understand/

[2] W. Irwin, March 2003 http://marc.theaimsgroup.com/?l=linux-kernel&m=104660383911943&w=2

[3] LWN.net, "Speeding up reverse mapping" http://lwn.net/Articles/9555/

[4] Ingo Molnár, http://people.redhat.com/mingo/O(1)-scheduler/

[5] M. Kravetz, H. Franke: "Implementation of a Multi-Queue Scheduler for Linux," April 2001, http://lse.sourceforge.net/scheduling/mq1.html

[6] Davide Libenzi, October 2001, http://www.xmailserver.org/linux-patches/lnxsched.html

[7] Andrea Arcangeli, "NUMA," presentation at the UKUUG Manchester, June 2001, http://www.ukuug.org/events/linux2001/papers/html/AArcangeli-numa.html

[8] H. Franke et al, "PMQS: Scalable Linux Scheduling for High-End Servers," http://lse.sourceforge.net/scheduling/als2001/pmqs.ps

[9] Erich Focht: "Node-Affine NUMA Scheduler," Feb. 2002, http://home.arcor.de/efocht/sched

[10] John Stultz: "Nodeless MCS Lock," http://www-124.ibm.com/linux/patches/?patch_id=218

[11] "NUMA AWARE LOCKS," http://lse.sourceforge.net/numa/locking

[12] Patrick Mansfield: "SCSI Mid-Level Multi-path/port storage," http://www-124.ibm.com/storageio/multipath/scsi-multipath/index.php.

Kernel configuration and building in Linux 2.5

Kai Germaschewski
University of Iowa
kai@germaschewski.name

Sam Ravnborg
Ericsson Diax A/S
sam@ravnborg.org

Abstract

The development phase of Linux 2.5 brought substantial changes to the kernel configuration process, the actual kernel build and, in particular, implementation and building of loadable modules.

The first part of this paper will give an overview of the user-visible changes which occured in Linux 2.5, on the one hand for users which build kernels themselves, on the other hand for developers which maintain drivers or other parts of the kernel, in order to help porting to Linux 2.5/2.6.

The second part of the paper deals with the actual design and implementation of the current kbuild, showing how *GNU make* is actually flexible enough to allow for nice condensed Makefile fragments which per subdirectory describe which objects to build into the kernel or as loadable modules. The paper ends with an outlook showing possible approaches for implementing additional features.

The paper also explains the improvements in handling loadable kernel modules, including symbol versioning, and the necessary build system changes.

1 Introduction and history

Why is a kernel build system necessary at all, and why does the Linux kernel use its own special solution?

As the Linux kernel evolved from a student's terminal emulation program towards a full-featured UNIX-like kernel, changes to the way it was built became necessary and were integrated, so the kernel build system basically followed the evolutionary development of the kernel itself.

In the science world, in particular people running numerical simulations, many people consider a build system completely unnecessary, they just run

```
f77 code.f
./a.out
```

However, this approach obviously doesn't scale to large projects. To keep projects maintainable, some kind of modularization occurs the code is divided into a number of source files and as the project is growing further, a directory hierarchy is introduced which helps organizing the code even further.

During development, normally only one or a few files are edited and then the developer wants to rebuild the program, in this case the kernel *vmlinux*, be it just for compile-time checks or testing.

First of all, one does not want to enter all the commands manually for each build, so some type of script is necessary to record those commands. Next, it is actually a waste to recompile every file if only few have changed. Smart

programmers recognized this a long time ago and invented a tool called *make* which is still the most popular build tool used today. We assume in this paper that the audience is familiar with the basics of *make*.

So even Linux 0.01 came already with a Makefile which took care of building the kernel.

As time passed and Linux matured, new features were incorporated into the build system, such as

- **Automatic generation of dependency information.** *make* only handles simple dependencies like the dependency of an object file on the corresponding source automatically, other prerequisites as for example included header files need to be added to the Makefile explicitly, a task which can be (and was) automated.
- **Configurability.** As the code base for the Linux kernel expanded, a need for a user selectable configuration became apparent and was introduced before release of Linux 1.0. This system allows the user to answer questions with respect to which components are desired, and then only builds those components into the kernel.
- **Different architectures and cross–compilation.** Linux introduced support for different architectures, which means the kernel is build from a large arch-independent code base as well as some machine-specific low-level code. It is also often necessary to cross–compile the kernel, i.e. do the compilation on a different platform than it is actually run on.
- **Loadable modules.** Within Linux 1.3, support for loadable kernel modules was introduced, which again needed special support in configuration and building of those objects.

In particular the high configurability and support for loadable modules distinguish the Linux kernel from most other projects, and it thus comes as no surprise that its build system also evolved away from a standard Makefile. However, *make* is still the underlying tool used for building the kernel. In fact, the extensibility of the *GNU* version of *make* [1] in conjunction with some support scripts / C code renders it possible to meet the goals listed above.

2 A dummy's guide to kbuild

This section is addressed to users and will explain how to use the kernel build system in Linux-2.5/2.6. "Users" (as opposed to "developers") here mean people who download the linux kernel tree source, possibly apply patches and then build and install their own kernels. Of course, since kernel developers need to build and run kernels, too, this section is of relevance for them as well.

The build system is based on *GNU make*, i.e. all commands are given to *make* by invoking it as

```
make <target>
```

Contrary to many userspace packages which are using autoconf/automake, there is no preceding `./configure` necessary, the necessary configuration process is embedded into the build process.

The actual targets are in part platform-specific, for example on i386 one typical wants to build the boot image `bzImage` and modules. A list of supported targets for the platform can be obtained from `make help`.

Arch maintainers should setup their arch-specific Makefile in a way that invoking *make* without parameters will build the commonly used boot target for the architecture, for example on i386 just typing

```
make
```

will build bzImage and modules (the latter only when `CONFIG_MODULES` is selected, of course) which is what is typically needed.

If one just runs *make* after unpacking the kernel source tarball, *make* will actually just error out, asking you to configure your kernel first by running `make *config`. (In Linux-2.4 and before, it would invoke `make config` for you, but this is the wrong choice in 99% of the cases, since nobody likes answering a straight sequence of a couple of hundred questions. . .)

To generate a new kernel configuration, it is recommended to use `make menuconfig`, `make xconfig` (which uses Qt now) or `make gconfig` (uses gtk).

However in most cases, it is easier to adapt an existing kernel configuration to the current kernel than to create a new one from scratch. This is done by copying the *.config* file into the top-level directory of the source tree. kbuild will recognize that the .config file may need adaption for the current kernel source and automatically run `make oldconfig` for you, which makes sure that *.config* is consistent with the current rules and asks the user about the value of previously not existing options.

So the normal sequence for building a kernel is just

```
cp /my/old/.config .config
make
```

where one could insert a `make *config` between those two steps if a change of configuration options is desired.

The last remaining step is the installation of the newly built kernel. The procedure to install the boot image depends of course on the bootloader used.

For *lilo*, the kernel boot image *bzImage* should be copied to a certain location (typically */boot*), then */etc/lilo.conf* may need an appropriate entry and finally */sbin/lilo* must be run.

For *grub*, copying the kernel image to */boot* and possibly editing */etc/grub.conf* should suffice.

An important change is that on i386 *bzImage*/*zImage* can not be directly booted from a floppy disk anymore. Instead the targets `zdisk` and `fdimage` create a boot floppy disk and a boot disk image, respectively. Those targets now require mtools and syslinux to be installed.

Since the actual installation of the boot image varies as described above, one can give the `install` target to `make`, which will invoke a user– or distro–provided script, `~/bin/installkernel` or `/sbin/installkernel` which can be customized for the local setup.

Installing modules is simpler, just invoking `make modules_install` will do the necessary work. By default this will install into `/lib/modules/‘uname -r‘/`, though this can be customized by setting `INSTALL_MOD_PATH`, e.g. if one wants to collect the modules for transfer onto a different machine.

This is basically all knowledge which is needed to build a Linux kernel—everything else is handled automatically by the build system. Applying patches, editing files, changing configuration options or adding compiler flags—

the build system will notice the change and rebuild whatever is needed. The one exception to this rule is changing the architecture (by setting the `ARCH` variable), which needs an explicit `make distclean` to work correctly.

3 kbuild for kernel developers

3.1 kbuild in the daily work

Since developers tend to build kernels and modules a lot, the previous section of course also applies to them, in particular the fact that just running `make` will recognize all changes and rebuild whatever is necessary to generate a consistent *vmlinux* and modules.

Some additional features exist to support the development / debugging process:

- `make some/path/file.o` will rebuild the single file given, using compiler flags (e.g. `-DMODULE`) according to the current *.config*.
- `make some/path/file.i` will generate a preprocessed version of `/some/path/file.c`, again using compiler flags for the current configuration.
- `make some/path/file.s` will generate a file containing the raw assembler code for `some/path/file.[cS]`.
- `make some/path/file.lst` (little known but very useful) gives interspersed assembler code with the C source, relocated to the correct virtual address when a current *System.map* exists.

Another useful feature for the daily work, which has existed for a long time, is the ability to override the `SUBDIRS` variable on the command line, which will force *make* to only descend into the given subtree. This can be very useful for faster build times, but it bypasses some dependencies and thus does not guarantee to result in a consistent state.

So while e.g. working on the *hisax* ISDN driver, it's useful to call *make* as

```
make SUBDIRS=drivers/isdn/hisax \
     modules
```

for compile checks etc. However, before installing a new kernel and modules, the authors advise to always run a full `make bzImage/vmlinux/modules` (or otherwise, do not complain ;).

3.2 Integrating a driver

Basically each subdirectory in the Linux kernel tree contains a file called *Makefile*, which is included by *make* during the kernel build process. However, these Makefiles are different from regular Makefiles in that they normally don't have any targets or rules, but only set variables which tell the build process what should be built and the latter takes control of the actual compiling and linking.

In conjunction with the Makefile there normally exists a *Kconfig* file, these files were introduced with the configurator rewrite by Roman Zippel and replace the old *Config.in/Config.help* files used during the configuration phase of the kernel build.

This paper does not intend to elaborate on the new kernel configuration system, however the following examples will provide some basic usage guidance.

The most common case is adding a new driver which is built from a single source file.

```
config TIGON3
    tristate "Broadcom Tigon3 support"
    depends on PCI
    help
      This driver supports Broadcom Tigon3 based gigabit Ethernet cards.

      If you want to compile this driver as a module ( = code which can be
      inserted in and removed from the running kernel whenever you want),
      say M here and read <file:Documentation/modules.txt>.  This is
      recommended.  The module will be called tg3.
```

Figure 1: *Kconfig* fragment for the Tigon3 driver.

Figure 1 shows the *Kconfig* fragment for the Tigon3 driver, which defines a config option `TIGON3` (the corresponding variable will be given the name `CONFIG_TIGON3`), which is a tristate, i.e. can have the values `y`, `m`, or `n` with the obvious meanings (a config option which has been turned off, actually has the value `""`, to be correct here). The fragment `depends on PCI` states that this option is only selectable when the option `PCI` is also set (that is, if the kernel supports the PCI bus).

The Makefile fragment for the Tigon3 driver

```
obj-$(CONFIG_TIGON3) += tg3.o
```

is very short and though a little awkward at first, a very elegant way to quickly express what files are supposed to be built. The idea dates back to Micheal Elizabeth Castain's *dancing Makefiles* [2] and was globally introduced into the kernel by Linus shortly before the release of kernel 2.4.

What happens is that depending on the config option `CONFIG_TIGON3`, the value `tg3.o` is appended to either of the variables `obj-y`, `obj-m` or `obj-`.

The meaning of those special variables is as follows:

- `obj-y`. All objects listen in `obj-y` will be compiled as built-in objects, and will finally be linked into *vmlinux*.
- `obj-m`. All objects listed in `obj-m` and not not listed in `obj-y` will be compiled as modules (so they actually end up being called e.g. `tg3.ko` in 2.5/2.6).
- `obj-`. All objects listed in `obj-` and not in `obj-y` or `obj-m` will be ignored by kbuild.

Since the build system does not have any further information on `tg3.o`, it will try to build it from a source file called `tg3.c` (or an assembler source `tg3.S`, which only happens in the architecture dependent part of the kernel, though).

This is all what is needed to integrate a simple driver into the kernel build, other than of course writing the driver (`tg3.c`) itself.

It is also possible to list more than one object to be built in the Makefile statement. The Makefile line dealing with the *eepro100* driver looks like the following:

```
obj-$(CONFIG_EEPRO100) += \
        eepro100.o mii.o
```

If this driver is selected, the *mii.o* support module also needs to be compiled, which is achieved by simply appending it to the statement.

Other network drivers will, if selected, also add *mii.o* to the list of objects to be built—this is fine, the build system handles this case. It is even possible that a support module like *mii.o* got added to the list of built-in objects `obj-y` and `obj-m`—again, the build system recognizes this fact and just compiles the built-in version, which will also be usable for the drivers compiled modular.

The new *e100* driver examplifies two more features. *drivers/net/Makefile* only contains the line

```
obj-$(CONFIG_E100) += e100/
```

which tells kbuild that it should descend into the *e100/* subdirectory if the option `CONFIG_E100` is set. What to do there will then be determined by *drivers/net/e100/Makefile* (Figure 2):

The first line after the comment looks familiar, it advises the build system to build *e100.o* built-in/modular depending on the value of `CONFIG_E100`. When `CONFIG_E100` equals "m" the e100 driver is built as a module and will be named em e100.ko.

The next line then states that *e100.o* is a composite object which should be linked from the listed individual object files—these object files will automatically compiled with the appropriate flags.

As a last point, instead of using the variable `<modname>-objs` to declare the components of the module *<modname>.o*, the variant `<modname>-y` can be used, which allows for easy definition of optional parts to a composite modules, as seen in the example in Figure 3.

4 What is new in Linux-2.5/2.6's kbuild?

In this section, we describe some of the steps in the evolution of the kernel build system during the development phase of Linux 2.5. One purpose is to show how this evolution could actually be divided into small, Linus-compatible "piece-meal" patches without the famous "flag-day" patches and with only little breakage along the way.

We will also show how using the extensions provided by *GNU make* were actually exploited to provide a better build system while still using a standard tool instead of creating a specialized build solution for the kernel from scratch.

We start by comparing *drivers/isdn/Makefile* in 2.4 and 2.5 (Figure 4), where many of the improvements are easily seen. (a) shows the *Makefile* as it is present in Linux 2.4.20, and (b) shows the simpler variant present in 2.5. kbuild has been adapted incrementally to allow the more concise syntax. The following sections will describe the internals that eventually allowed for the layout seen in (b).

4.1 `O_TARGET` / linking objects in subdirectories

First of all, we start with a short description of what the kbuild interal implementation, which is hidden in the top-level *Makefile* and *scripts/Makefile.** typically does in a subdirectory: From the kbuild *Makefile* located in the subdirectory we obtain a list of what to build from the variables `obj-y` (built-in) and `obj-m` (modular) as explained in the previous section. The default target in *scripts/Makefile.build* is `__build` and the corresponding rule, shown in Figure 5, defines what work needs to be done. Important here is that we build `O_TARGET` or `L_TARGET`, respectively, when building *vmlinux* and `obj-m` when compiling modules. As opposed to 2.4, in 2.5 `O_TARGET` is a kbuild internal variable

```
#
# Makefile for the Intels E100 ethernet driver

obj-$(CONFIG_E100) += e100.o

e100-objs := e100_main.o e100_config.o e100_phy.o \
             e100_eeprom.o e100_test.o
```

Figure 2: `/drivers/net/e100/Makefile`

```
#
# Makefile for the Linux X.25 Packet layer.
#

obj-$(CONFIG_X25) += x25.o

x25-y          := af_x25.o x25_dev.o x25_facilities.o x25_in.o \
                  x25_link.o x25_out.o x25_route.o x25_subr.o \
                  x25_timer.o x25_proc.o
x25-$(CONFIG_SYSCTL)    += sysctl_net_x25.o
```

Figure 3: `net/x25/Makefile`

and needs no longer be defined in the kbuild makefiles. Except for the rare case of building an actual library, `O_TARGET` is used in the built-in case and we find the rule how to make it as

```
$(O_TARGET): $(obj-y) FORCE
     $(call if_changed,link_o_target)
```

So `O_TARGET` is linked from the objects listed in `obj-y`, which contains files locally compiled in the current directory as well as objects which are built in subdirectories by descending. In Figure 6, we see how going from the leaves to the root, the `O_TARGET` in each subdirectory (here always called *built-in.o*) accumulates the objects built below that directory until we finally end up with *vmlinux* at the root of the hierarchy containing all built-in objects generated throughout the tree (this example only shows a small fraction of the objects linked in a normal build).

```
vmlinux
|-- drivers/built-in.o
|   `-- isdn/built-in.o
|       |-- isdn.o
|       |   |-- isdn_common.o
|       |   `-- isdn_net.o
|       |
|       |-- hisax/built-in.o
|       |   |-- hisax.o
|       |   |   |-- config.o
|       |   |   `-- isdnl*.o
|       |   `-- hisax_fcpcipnp.o
|       |
|       `-- icn/built-in.o
|           `-- icn.o
|
`-- fs/built-in.o
```

Figure 6: The hierarchy for linking *vmlinux*

(a)

```
O_TARGET        := vmlinux-obj.o
export-objs     := isdn_common.o

list-multi      := isdn.o
isdn-objs       := isdn_net.o isdn_tty.o isdn_v110.o isdn_common.o
isdn-objs-$(CONFIG_ISDN_PPP)            += isdn_ppp.o
isdn-objs                               += $(isdn-objs-y)

obj-$(CONFIG_ISDN)                      += isdn.o
obj-$(CONFIG_ISDN_PPP_BSDCOMP)          += isdn_bsdcomp.o

mod-subdirs                             := hisax
subdir-$(CONFIG_ISDN_HISAX)             += hisax
subdir-$(CONFIG_ISDN_DRV_ICN)           += icn

obj-y += $(addsuffix /vmlinux-obj.o, $(subdir-y))

include $(TOPDIR)/Rules.make

isdn.o: $(isdn-objs)
        $(LD) -r -o $@ $(isdn-objs)
```

(b)

```
obj-$(CONFIG_ISDN)              += isdn.o
obj-$(CONFIG_ISDN_PPP_BSDCOMP)  += isdn_bsdcomp.o

isdn-y                          := isdn_net_lib.o isdn_fsm.o isdn_tty.o \
                                   isdn_v110.o isdn_common.o
isdn-$(CONFIG_ISDN_PPP)         += isdn_ppp.o

obj-$(CONFIG_ISDN_DRV_HISAX)    += hisax/
obj-$(CONFIG_ISDN_DRV_ICN)      += icn/
```

Figure 4: *drivers/isdn/Makefile* in (a) 2.4.20 and (b) adapted for the build system in 2.5

In Linux 2.4, the name for the object which accumulates all built-in objects in and below the current subdirectory was chosen by setting the variable `O_TARGET` in the local Makefile. In Linux-2.5, it is instead just set to *built-in.o* by the build system. This allows to get rid of the assignment of `O_TARGET` in every subdir Makefile and, more importantly, allows for further clean-up:

In kbuild-2.4, we need to explicitly add the subdirectories to descend into to the variables `subdir-y/m`, and then also add the subdir-generated built-in objects to `obj-y` so that they get linked. This is redundant and error-prone, in 2.5 it is sufficient to just add the objects generated in the subdirectories to the list of objects to be linked, and the build system will deduce from there that it needs to descend into the named subdirectories. To simplify things further, the name of the `O_TARGET` (now always being *built-in.o*) itself is left out and only the trailing slash is kept:

```
__build: $(if $(KBUILD_BUILTIN),$(O_TARGET) $(L_TARGET) $(extra-y)) \
         $(if $(KBUILD_MODULES),$(obj-m)) \
         $(subdir-ym) $(always)
        @:
```

Figure 5: The default `__build` rule from *scripts/Makefile.build*

```
obj-$(CONFIG_...HISAX) += hisax/
obj-$(CONFIG_...ICN)   += icn/
```

4.2 Multi-part modules

As can be seen from Figure 4, a number of statements were necessary for generating a multi-part module, *isdn.o* in that example. First of all, the parts constituting the module need to be declared by assigning them to the variable `isdn-objs`. This step is of course essential and was kept in 2.5.

However, it was also necessary to declare that *isdn.o* is a multi-part module by listing it in the variable `list-multi`. This information is redundant as it can be deduced by checking for the existence of `<module>-objs`, which is now done in 2.5.

Furthermore, in 2.4 a link rule has to be explicitly given for each multi-part object, which was annoying and error-prone. In the new build system, this link rule is generated by *make*, hitting just about the limits of what *GNU make* is capable of. We use a feature called "static pattern rules," and the code looks like the following:

```
cmd_link_multi-m = $(LD) ... \
  -o $@ $(link_multi_deps)

$(multi-used-m) : \
%.o: $(multi-objs-m) FORCE
  $(call if_changed,link_multi-m)
```

`multi-used-m` contains all multi-part modules we want to be built in the current directory and `multi-objs-m` contains all of the individual objects those are built of. This makes each multi-part module in the directory depend on the set of all components for all multi-part modules in that directory, which is actually too large, as it of course only is dependend its own components; however the latter is not implementable within the restrictions of *GNU make*. When doing the link, the variable `link_multi_deps` recovers the right list of components from the target `$@`, so that linker is invoked correctly.

Another interesting detail is that we here as well as in other places need to uniquify the prerequisites, so that listing a component multiple times doesn't lead to a link error. *GNU make* offers the `sort` function, which throws away duplicates, however it is unfortunately not usable for this purpose since it sorts, i.e. reorders its arguments and thus changes the link/init order. The workaround here is to use the variable `$^` which actually uniquifies the list of prerequisites exactly as needed. Finally, since `$^` lists all prerequisites which as mentioned above exceeds the list of components for the current module, we filter the uniquified list with that list of components to get the information we need.

4.3 Including `Rules.make`

Each subdirectory Makefile in kbuild-2.4 needed to include *$(TOPDIR)/Rules.make* explicitly. In 2.5, when descending into subdirectories, the build system now always calls *make* with the same Makefile, *scripts/Makefile.build*, which then again includes the local subdirectory *Makefile*, so the statement to include *Rules.make* could be dropped.

Furthermore, in 2.5. the build is still organized in a recursive way, i.e. *make* is only invoked to build the objects in the local subdirectory and other instances of *make* are spawned for the underlying directories. However, it does not actually descend into the subdirectories, it always does its work from the top-level directory and prepends the path as necessary. One of the advantages is that the output includes the correct paths, so a compiler warning will not show "*inode.c*: Warning ...", but "*fs/ext2/inode.c*: ...", which makes it easier to recognize where the problem occurs. More importantly, it allows to use relative paths throughout the build, so that paths like "BUG in `/home/kai/src/kernel/v2.5/linux-2.5.isdn/include/linux/fs.h`" are history. Renaming/moving a kernel tree will not cause spurious rebuilds due to changing paths as seen above anymore, and tools like "ccache" can work more effectively.

4.4 Objects exporting symbols

The old module symbol versioning scheme used with Linux 2.4 needed the Makefiles to declare which objects export symbols to modules, which was done by listing them in the variable `export-objs`. In 2.5, module versioning was completely redesigned, removing the need for this explicit declaration. The changes are so complex that they are rewarded their own section in this paper.

Here we conclude the comparison between a 2.4 and 2.5 subdirectory Makefile, where we have shown that all the redundant and deducible information has been removed and the necessary information is revealed to the build system in a very compact form.

Two additional important internal changes, which did not affect the subdirectory Makefile layout will be described in the following:

4.5 Compiling built-in objects and modules in a single pass, recognizing changed command line arguments

The major performance issue for the kernel build are the invocations of *make* (most of the time is of course normally spent compiling / linking, but this cost is independent of the build system used). *make* has to read the local Makefile, the general rules and all the dependencies and figure out the work to be done from there. An obvious way to optimize the performance of the build system is thus to avoid unnecessary invocations. In 2.4, *make* needs to do separate passes for modules and built-in objects and within each directory, it will even call itself again, so an about four-times performance increase is possible by just combining those invocations into a single pass.

The primary reason why kbuild-2.4 needs two passes for built-in and modules lies in its flags handling. This means that it tries to check not only whether prerequisites have changed (e.g. the C source for an object), but also if the compiler flags have changed.

This objective was achieved by generating a .<*target*>.*flags* file like the following (simplified) for each target built:

```
ifeq (-D__KERNEL__ -DMODULE
      -DEXPORT_SYMTAB,
      $(CFLAGS) -DEXPORT_SYMTAB)))
  FILES_FLAGS_UP_TO_DATE += config.o
endif
```

On a rebuild, the Makefile would read all those .*.*flags* fragments and forces all files which are not listed in `FILES_FLAGS_UP_TO_DATE` to be rebuild.

The flaw of this method is that it cannot handle differing flags for different groups of files, so *make* needs to invoked twice, once for the targets to be built-in with the normal `CFLAGS`, and again for the modular targets with `-DMODULE`

added to `CFLAGS`. In the example above it is also visible that the handling for `-DEXPORT_SYMTAB` is broken, this method can not detect when a file was added / removed from the list of files exporting symbols, since the `-DEXPORT_SYMTAB` was hardcoded on both sides of the comparison and thus useless—the only way to fix this within in the old framework would have been to invoke *make* four times, for all combinations of built-in/module and export/no-export.

A more flexible scheme to handle changing command lines within *GNU make* was created:

As an example, we present the rule which is responsible for linking built-in objects into a library in Figure 7. The actual use is pretty simple, instead of writing the command directly into the command part of the rule, it is instead assigned to the variable `cmd_link_l_target` and the build system takes care of executing the command as necessary, keeping track of changes to the command line itself.

The implementation works as follows: After executing the command, the macro `if_changed`, records the command line into the file *.<target>.cmd*. As *make* is invoked again during a rebuild, it will include those *.*.cmd* files. As it tries to decide whether to rebuild `L_TARGET`, it will find `FORCE` in the prerequisites, which actually forces it to always rerun the command part of the rule.

However, the command part of the rule now does the actual work: It checks whether any of the prerequisites changed, i.e. `$?` is non-empty or if the command line changed, which is achieved by the two `filter-out` statements. Only if either of those two conditions is met, `if_changed` expands to a command rebuilding the target, otherwise it is empty and the target will not be rebuilt.

The advantage of this method, apart from the easier use in a rule as shown above, is that all the checking is done within the context of the actual rule and not in a unrelated place later in the Makefile. This allows for the use and correct checking of *GNU make*'s per target variables, e.g.

```
modkern_cflags := $(CFLAGS_KERNEL)
$(real-objs-m) : \
  modkern_cflags := $(CFLAGS_MODULE)
```

which sets `modkern_cflags` to `$(CFLAGS_KERNEL)` by default, but to `$(CFLAGS_MODULE)` for objects listed in `$(real-objs-m)`, i.e. for objects compiled as modules. The compilation rule can then just use `$(modkern_cflags)` to get the right flags for the current object, where the mechanism described above will take care of recognizing changes and acting accordingly.

4.6 Dependencies

Between configuration and building of a kernel, the old kernel build needed the user to run "`make dep`", which served to purposes: It generated dependency information for the C source files on headers and other included files, and it generated the version checksums for exported symbols.

Both of these task have become unnecessary in 2.5, so the reliance on the user to rerun "`make dep`" as needed is gone (additionally, the system in 2.4 is broken that in some modversions cases it's not even sufficient to rerun "`make dep`", the only solution then is to do "`make distclean`" and start over).

2.4 used a small tool called *mkdep* to generate dependencies for C sources. This tools basically extracted the names of the included files out of the source, but did not actually recursively scan those includes then. So, if *foo.c* includes *foo.h*, which itself includes *bar.h*, *mkdep* would only pick up the dependency of *foo.c* on *foo.h*, but *foo.c* also needs recompiling when

```
cmd_link_l_target = rm -f $@; $(AR) $(EXTRA_ARFLAGS) rcs $@ $(obj-y)

$(L_TARGET): $(obj-y) FORCE
        $(call if_changed,link_l_target)

targets += $(L_TARGET)

[...]

if_changed = $(if $(strip $? \
                          $(filter-out $(cmd_$(1)),$(cmd_$@))\
                          $(filter-out $(cmd_$@),$(cmd_$(1)))),\
        @set -e; \
        $(cmd_$(1)); \
        echo 'cmd_$@ := $(cmd_$(1))' > $(@D)/.$(@F).cmd)
```

Figure 7: Checking for a changed command line

foo.h changes. This problem was solved in 2.4 by assuming that *foo.h* would reside in *include/** (which is mostly, but not always, true). For those files it would generate another set of dependencies, basically:

```
foo.h: bar.h
        @touch $@
```

So as *bar.h* changes, this rule will update the timestamp on *foo.h*, which will then be seen by the rule for *foo.c* and cause *foo.c* to be rebuild.

This method has several disadvantages:

- Changing the timestamp on files which have not actually been modified confuses a number of source management systems.
- It only works for header files in the *include/** subdirectories.
- As *foo.h* is changed to also include *baz.h*, the dependency information does not get updated, so a subsequent change to *baz.h* will erroneously not cause *foo.c* to be recompiled.
- Starting from a clean tree, the user has to wait for the dependency information to be created (for all files, even for entire subsystem which may not be selected in the configuration at all), even though this information is totally useless for a first build—it's only useful for deciding whether a file needs to be rebuilt.

The build system in Linux 2.5 instead uses gcc's -MD flag to generate the dependency information during the build. This flag generates the full list of all files included during the compile, so in the example above it would generate "*foo.o*: *foo.c foo.h bar.h*" (and "*baz.h*" as that gets added). This procedure is much simpler, and it gets around all the disadvantages listed above.

The only quirk which is applied similarly in 2.4 and 2.5 is related to the high configurability of the linux kernel.

Using the gcc generated list of dependencies as-is has the drawback that virtually every file in the kernel includes <*linux/config.h*> which then again includes <*linux/autoconf.h*>

If a user reruns `make *config` to change a configuration option, *linux/autoconf.h* will be regenerated. *make* will notice this and rebuild

every file which includes autconf.h, i.e. basically all files. This is correct, but extremely annoying if the user just changed some option `CONFIG_THIS_DRIVER` from n to m.

So we use the same trick that "*mkdep*" applied before. We replace the dependency on *linux/autoconf.h* by a dependency on every config option which is mentioned in any of the listed prerexquisites.

The effect is that if a user changes the `CONFIG_THIS_DRIVER` option, only the objects which (themselves, or in any of the included files) reference `CONFIG_THIS_DRIVER` will be rebuilt, which most likely is only this one driver.

5 Modules and the kernel build process

The implementation of loadable kernel modules has been substantially rewritten by Rusty Russell in the development cycle 2.5. These changes are so complex that this paper will not attempt to describe them in detail. Instead, we concentrate on the changes which were done in the build system to accomodate the new concepts.

5.1 Module symbol versions

Loadable modules need to interface with the kernel. They do this by accessing certain data structures and functions which have been marked as exported symbols in the source. That means not all global symbols in the kernel are accessible to modules, but only an explicitly exported API.

These symbols remain unresolved in the loadable module objects at build time and are then resolved at load time, either by an external program, *modutils*, in 2.4, or by an in-kernel loader in 2.5. A common problem is that Linux does not guarantee a stable binary interface to modules, in fact the binary interface often changes between releases in a stable kernel series and even depending on the configuration of the kernel. One simple example is the `struct net_device`, which embeds a `spinlock_t`. If the kernel is configured for uni-processor operation, this lock expands to nothing, so the layout of the `struct net_device changes`. When calling `register_netdev(struct net_device *)` where the in-kernel function `register_netdev()` assumes the SMP layout, though the module set up the argument in the UP layout, we have an obvious mismatch which often leads to hard to explain kernel crashes.

Other operating systems solve this problem by prescribing a stable ABI between kernel and modules, however in Linux it is preferred to not carry around binary compatibility layers and cope with unflexible interfaces, instead since the source is openly accessible, one just needs to recompile the modules so that they match the kernel.

Now, it is easily possible for users to get this wrong and we thus want a way to detect version mismatches and refuse to load the modules or at least warn. This is what "module symbol versioning" accomplishes. The basic idea is to analyze the exported symbols, including the types of the arguments for function calls and generate a checksum for a specific layout. If anything changes in the ABI, the versioning process will generate a different checksum and thus detect the mismatch. The main work in this scheme is done by the program *genksyms*, which is basically a C parser that reads a preprocessed source file and finds the definitions for the exported symbols from there.

This procedure has caused trouble in the

build system for a long time. In Linux 2.4, the "`make dep`" stage, apart from building dependency information, preprocesses all source files which export symbols (that is why they need to be specifically declared in the Makefiles) and then generates *include/linux/modversions.h* which mangles the exported symbols with the generated checksum, using the C preprocessor. The kernel will then not export the symbol `register_netdev`, but instead `register_netdev_R43d2381`. A module referencing `register_netdev` will end up with an unresolved symbol `register_netdev_R43d2381`, so loading it into the kernel will work fine. Has the module however built against a different kernel or a different configuration, the checksum has changed and any attempt to load it will result in an error about unresolved symbols.

This implementation was rather fragile, as it relies on the user to rerun "`make dep`" whenever the version information has possibly changed, and even if only one symbol changed, that basically forces a recompilation of every file. In addition, some of the optimizations made in 2.4's build system were actually broken, leading to the well-known fact that it can get into a state where not even running "`make dep`" will recover from generating inconsistent version information, and starting over from "`make mrproper/distclean`" is needed.

Module versioning is still a challenge to the build system in 2.5, the underlying reason for that is that it introduces cross-directory dependencies, which a recursive build system cannot easily handle. For example, the ISDN module *drivers/isdn/hisax/hisax.ko* uses `register_isdn()`, which is exported by *drivers/isdn/isdn_common.o*. So building *hisax.ko* needs knowledge of the checksum generated from *drivers/isdn/isdn_common.o*, but it has no way to make sure that it is up-to-date since it is located in a different subdirectory.

Module versioning is instead implemented as a two stage process, the first stage is the normal build, which also generates all the checksums. After this stage is completed, we can be sure that all checksums are up-to-date now, and then just record this up-to-date information into the modules. This is one of the reasons why modules have been renamed with a ".ko" extension: The first stage just builds the normal ".o" objects, and afterwards a postprocessing step follows, which builds ".ko" modules adding version checksums for unresolved symbols and other information.

In more detail, the following steps are executed:

- **Compiling**

  Knowledge of which source files export symbols is not required up front. As an `EXPORT_SYMBOL(foo)` is encountered, the definition of `EXPORT_SYMBOL` from *include/linux/module.h* will generate special sections with tables containg the name of the symbol, its address and its checksum. Actually, since the checksum is not known at this time, the value of the checksum is set to a symbol called `__crc_foo`. This is a trick which allows to use the linker to record the checksum even after the object file is already compiled.

  As the object file has been generated, we check it for the existance of the special section mentioned above. If it exists, the source file did export symbols and *genksyms* is run to obtain the checksums for those symbols. Finally, these checksums are entered into the object using the linker in conjunction with a small linker script.

```
$ nm drivers/isdn/i4l/isdn.ko | grep __crc
86849dd0 A __crc_isdn_ppp_register_compressor
843d2381 A __crc_isdn_ppp_unregister_compressor
66d136e2 A __crc_register_isdn
```

Figure 8: Examining the checksums for exported symbols

The checksums can easily examined at running the command shown in Figure 8.

- **Postprocessing**

  After stage one, we have the checksums for the exported symbols embedded within `vmlinux` and the modules. What is yet to be done is recording the checksums into the consumers, that is adding the checksums for unresolved symbols into the modules.

  This step was initially handled by a small shell script but is now done by a C program for performance reasons, which also handles other postprocessing needs like generating aliases.

  This program basically reads all the exported symbols and their checksums from all modules, and then scans the modules for unresolved symbols. For each unresolved symbol, an entry in a table associating the symbol string with the checksum is made, this table is output as C source *module.mod.c* and compiled and linked into the final `.ko` module object.

  Figure 9 shows an excerpt from *drivers/isdn/hisax/hisax.mod.c* which calls `register_isdn()`. The checksum obviously matches the checksum for the exported symbol in *drivers/isdn/i4l/isdn.ko*, so that the module will load without complaint.

An additional advantage of the new way of handling module version symbols, apart from being cleaner from a build system point of view, is that the actual symbols are not mangled, so it became possible to force a module load even if the checksums do not match—though the kernel will set the taint flag in these case.

The module postprocessing step, introduced mainly for the module symbol versioning, allowed for a number of additional features, i.e. module aliases / device table handling, additional version checks as well as recognition of unresolved symbols during the build stage.

6 Conclusion and Outlook

This paper presented an introduction to using the kernel build system for the Linux kernel 2.5 and 2.6 for users who want to compile their own kernels and developers working on kernel code. We also showed how in the transition from kbuild-2.4 to 2.5, features of *GNU make* could be applied to remove redundant information and allow for simpler Makefile fragments as well as a more consistent and foolproof build system.

Additionally, parts of the internal implementation have been described and an overview over changes related to the new module loader and new module versioning system has been given.

The kernel build system in 2.5 has been improved significantly, but some features remain to be implemented.

```
  static const struct modversion_info ____versions[]
__attribute__((section("__versions"))) = {
        { 0xfa7bbba7, "struct_module" },
        { 0x66d136e2, "register_isdn" },
        { 0x1a1a4f09, "__request_region" },
  [...]
```

Figure 9: Excerpt from *drivers/isdn/hisax/hisax.mod.c*, generated by the postprocessing stage

Separate source and object directories

As opposed to kernel 2.4, source files are not altered or touched during the build in 2.5 anymore, enhancing interoperability with source management systems. The next step is to allow for completely separate source and object directory trees, so that the source can be completely read-only and multiple builds at the same time from the same source are possible. The current code in 2.5 has taken preparatory steps for this feature but work is not completed yet.

Non-recursive build

It is an open question whether it is actually advisable to switch to a non-recursive build system. Obviously, distributing build information with the source files is desirable, this trend is visible in e.g. the split of the global Configure.help file into per-directory fragments which eventually were unified with the new *Kconfig* configuration info. Of course it is essential to keep the build information in the per-subdirectory Makefiles distributed as it is currently, it would be a step back to collapse it into one big file.

However this does not preclude collecting the distributed information when starting a build and generating a global Makefile, which is then used as a main stage. The advantage of this method is that it can handle cross-directory dependencies more easily, whereas the current system has to resort to a two-stage process for module post-processing. On the other hand, a global Makefile which contains also needs to incorporate dependencies for all files will use a significant amount of memory and may turn out to be problematic on low–end systems.

There are two ways to implement a global Makefile: One possibility is using *GNU make* itself, replacing the rules to actually compile / link objects by dummy routines recording the necessary actions into a global Makefile. The second possibility is, as the subdir Makefiles have a very consistent form by now, to write a specialized parser for those files and have that generate a global Makefile.

Whether switching to a non-recursive build system is worth the tradeoffs will be investigated in the Linux 2.7 development cycle.

References

[1] *GNU make* `http://www.gnu.org/software/make/make.html`

[2] Michael Elizabeth Castain: *dancing-makefiles* `http://www.kernel.org/pub/linux/kernel/projects/kbuild/dancing-makefiles-2.4.0-test10.gz`

Device discovery and power management in embedded systems

David Gibson
OzLabs, IBM Linux Technology Center
dwg@au1.ibm.com, ols2003@gibson.dropbear.id.au

Abstract

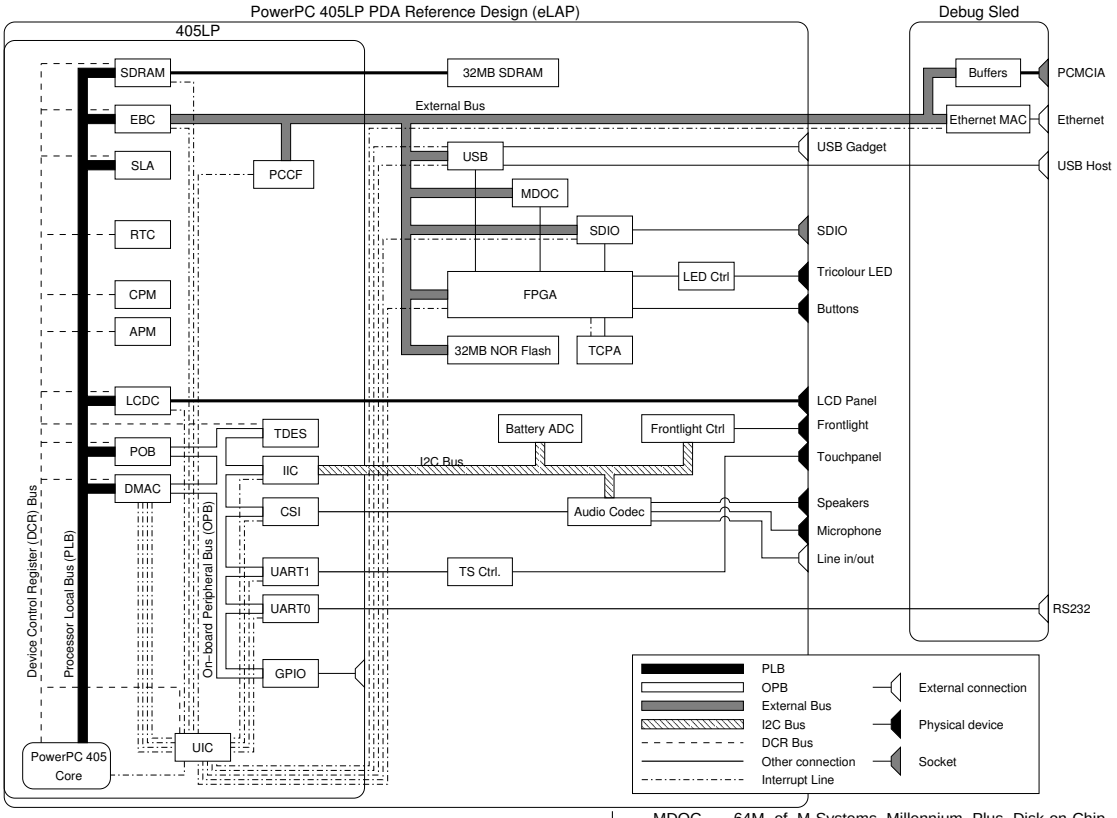
This paper covers issues in device discovery and power management in embedded Linux systems. In particular, we focus on the IBM® PowerPC® 405LP (a "system-on-chip" CPU designed for handheld applications) and IBM's PDA reference design based upon it. Peripherals in embedded systems are often connected in an ad-hoc manner and are not on a bus which can be scanned or probed. Thus the kernel must have knowledge of what devices are present built in at compile time. We examine how the new unified device model provides a clean method for representing this information, while allowing good re-use of code from machine to machine. The 405LP includes a number of novel power management features, in particular the ability to very rapidly change CPU and bus frequencies. We also examine how the device model provides a framework for representing constraints the peripherals and their interconnections place upon allowable frequencies and other information relevant to power management.

1 Introduction: the device discovery problem

Device discovery is the process the kernel and its device drivers use to determine what peripheral devices are present in a machine and how to communicate with them. Generally this means determining what IO addresses, interrupt lines and/or other bus specific addresses and resources are associated with each device.

Usually there are a few peripherals that are present in every machine of a particular type. Then there are optional devices that may or may not be installed in a particular machine. Some of these may be added or removed only from one boot to the next, and some may be hot-pluggable, added or removed while the machine is running.

The peripheral devices in an embedded machine often look very different to those in a conventional desktop or server. Even when a similar peripheral is used, differences in the way it is connected into the system can mean that it must be accessed and initialised quite differently. Many assumptions that are made about devices in a "normal" machine cannot be made in embedded machines, and the hardware and firmware of embedded machines generally provides much less assistance to the kernel for device discovery. All these things require different approaches to device discovery to be used.



APM	Power management unit, implements the CPU sleep/suspend states (NB: this is not related to the APM BIOS in PC laptops)
Audio Codec	Texas Instruments TLV320AIC23 stereo audio codec
Battery ADC	ADS7823 ADC monitoring battery voltage
CPM	Clock and Power Management unit
CSI	Codec Serial Interface (interface to audio codec devices)
DMAC	DMA controller (can DMA to/from devices on both the PLB and OPB)
EBC	External Bus Controller
Ethernet MAC	RTL8109 Ethernet controller
Frontlight Ctrl	DS1050 LCD frontlight controller
GPIO	General Purpose IO interface
IIC	I^2C bus interface (both master and slave capable)
LED Ctrl	BU8770KN tricolour LED driver
LCDC	LCD display controller

MDOC	64M of M-Systems Millennium Plus Disk-on-Chip (NAND flash with specialised controller)
PCCF	PCMCIA and Compact Flash controller
POB	PLB-to-OPB bridge
RTC	Real Time Clock
SDIO	Toshiba TC6380AF Secure Digital and SDIO interface
SDRAM	SDRAM controller
SLA	Speech Label Accelerator (hardware implementation of a particular speech recognition algorithm)
TCPA	Atmel AT97SC3201 TPM
TDES	Triple-DES accelerator
TS Ctrl	Semtech UR7HCT52_S840L touchscreen controller
UARTx	NS16550 compatible UARTs
UIC	Universal Interrupt Controller
USB	Phillips ISP1161 USB interface (includes both host controller and gadget-side interface)

Figure 1: Block diagram of the eLAP

2 The PowerPC 405LP PDA reference design

Most of the issues we discuss in this paper apply to many different embedded machines. However, for simplicity we focus on one example machine, the PowerPC 405LP PDA reference design, also known as the Embedded Linux Application Platform or eLAP. As the name suggests, this is a prototype reference design for a PDA based on the PowerPC 405LP CPU.

The IBM PowerPC 405LP is a CPU from the PowerPC 4xx family. This series of CPUs is designed for "system-on-chip" embedded applications. As the name suggests these processors are implementations of the PowerPC Architecture™, however they have some notable differences from "classic" PPC CPUs (as used in IBM pSeries™ servers and Apple workstations). The 4xx CPUs operate at much lower clock rates (and hence are cooler and cheaper), although they are in the high end by embedded standards. They have a much simpler MMU (just a software loaded TLB) and they have no

FPU. More interestingly, they include a number of peripheral devices built into the CPU die itself (hence the term "system-on-chip").

Different CPUs in the 4xx family are designed for different applications and have different collections of on-chip peripherals. The 405LP is designed for handheld, battery-powered applications. Figure 1 shows a block diagram of the eLAP, including the various built-in peripherals of the 405LP. The chip includes no less than three internal buses: the high-bandwidth Processor Local Bus (PLB) connected directly to the CPU core, the slower On-Board Peripheral bus (OPB), and the special DCR bus. The latter is used to implement Device Control Registers: rather than using normal memory-mapped IO, some of the on-chip devices use these special registers which are accessed using special machine instructions. As shown, the 405LP's peripherals include amongst other things, an LCD controller, a real-time clock, an I^2C interface and two serial ports. Other 4xx chips can include devices such as Ethernet controllers, HDLC interfaces, PCI host bridges, IDE and USB controllers. The 405LP also includes a number of novel power management features, which we'll examine in §5.

In addition to the devices within the 405LP, the eLAP includes 32MB of RAM, 32MB of NOR Flash and a number of additional peripherals, also shown in Figure 1. Most of these are connected via a minimal bus driven by the 405LP's on-chip External Bus Controller (EBC) unit. An extra debug and development sled can be attached to the eLAP, again shown in Figure 1. It includes an Ethernet controller, the physical PCMCIA slot driven by the 405LP's PCCF core and the physical connectors for the USB host port and serial port. Of course, as well as the peripherals shown, further devices can be attached via the PCMCIA and SDIO slots and the USB host interface.

3 Current approaches

3.1 Conventional machines

On normal server or workstation machines, device discovery is mostly quite straightforward.[1] Nearly all modern machines are based on PCI, which (like most modern buses) is designed so that devices can be queried and configured in a standard way. This makes it easy for the kernel to scan the PCI bus (or buses), determine what devices are present and their addresses, and pass this information to the appropriate device drivers. USB devices provide similar functionality, as do PCMCIA and ISA/PnP devices.[2]

The few remaining devices (including the PCI host bridge itself) are usually standard—to be found on all machines of this type and often nearly all machines of this architecture. They can be found at well-known addresses, so the drivers for these devices simply hardcode this information. On PCs, non-PnP ISA devices do introduce some problems. In fact they demonstrate a subset of the problems with embedded hardware that we will examine in the next section.

Many non-x86 machines make device discovery even simpler with firmware support. Open Firmware (on IBM and Apple PowerPC machines) and likewise its ancestor OpenPROM (on Sun machines) builds a tree with information about each of the peripherals on the machine. At boot time the kernel queries this information, making a copy of the device tree which can later be used by drivers to find devices. The ACPI BIOS found on recent Intel® machines provides some similar information, although neither the ACPI implementations nor Linux's use of them is very well es-

[1] Although on big servers keeping track of the devices once they're discovered can be another matter.

[2] At least in theory; many ISA/PnP implementations are buggy in practice.

tablished as yet.

3.2 Embedded weirdness

On embedded machines all the assumptions that are made on "normal" machines break down. Embedded machines can and do have arbitrarily peculiar combinations of peripherals connected in a more-or-less ad-hoc manner.

Often, many of an embedded machine's peripherals are connected via an unconventional bus which provides no facilities for systematic scanning or probing of devices. On the 405LP this is true of both the on-chip buses and the main external bus. Devices can appear essentially anywhere within the CPU's physical address space. Some times the address or other behaviour of a device is affected by a custom FPGA or other programmable logic device with its own control registers. Device interrupt lines introduce even more problems, being routed in complex and arbitrary ways that are often controlled or masked by a board-specific FPGA or CPLD. Devices can sometimes have multiple dependencies on other devices: for example on the eLAP, audio is driven by the TI codec, which is controlled and configured via the I^2C bus. However, the actual audio data is delivered to the codec via a serial connection to the on-chip Codec Serial Interface (CSI). The CSI in turn depends on the on-chip DMA controller to supply data from RAM.

Sometimes machines also have a more conventional bus such as PCI or PCMCIA, but it may have to be accessed via a bridge which is not configured in the same way as one would expect on a conventional machine. Worst of all, there are dozens or hundreds of different types of embedded machine, each with its own completely different set of devices and connections.

Under these circumstances, it is tempting to turn to each board's firmware to provide information about which devices are present. Unfortunately the firmware on most embedded machines is very primitive, providing little more than a boot loader. Usually it will provide a few useful pieces of information, such as the amount of RAM on the system or the board revision, but it certainly won't give comprehensive device information. Furthermore, what information the firmware does provide usually can't be used without already knowing something about the machine in question, since embedded firmwares are almost as varied as embedded machines themselves.

Since embedded machines can and do break any assumption one might care to make about how devices are attached, there is no magical way that the kernel can detect what devices are present. So, the only approach is to have the kernel "just know" the device setup for a particular board by building the knowledge into the kernel at compile time.

Although it is impossible to completely avoid hardwired knowledge of boards in the kernel, we do want to keep this information in as clean a way as possible. Specifically, we want to isolate the direct knowledge of board specific details to as small a section of the kernel as possible, and we want to make it easy to add the details of new boards and their peripherals.

In this paper, we generally assume that the kernel must be configured for one particular type of embedded machine, since this is the simplest case. Building a kernel which will support multiple machines is certainly possible, and most of the methods we discuss can still be applied. In this case the kernel need to include device information about all supported boards. Early in boot, the kernel will identify the machine it is running on (by some ad-hoc method), and select which information to use on that basis.

Now, we examine some of the existing meth-

ods by which embedded devices are supported in the kernel.

3.3 Hardcoded hacks

The naïve approach to handling embedded devices that aren't on a conventional bus is to treat them all like the "system" devices on a PC or other conventional machine. That is, simply hardcode knowledge of the device into the relevant device driver or into the kernel initialisation code for the machine in question. This approach is currently used for quite a number of embedded devices—unsurprisingly since, as we'll see, a comprehensive better approach has yet to be implemented.

This method has some serious shortcomings. The most obvious problems come with embedded peripherals that are similar to ones also found in conventional machines. For obvious reasons, it is normal in this case to adapt the existing conventional driver for use in the embedded machine.

Sometimes this has been done by adding `#ifdef`s to the driver for the board specific code. For example, this has been done with the `cs89x0` driver for the CrystalLAN CS8900 Ethernet chip. This chip is used on some ISA cards, but is also found on the "Beech" embedded board (another IBM reference board based on the 405LP). Apart from the fact that `#ifdef`s make the driver code ugly, this clearly causes a problem if the embedded machine can also have a normal ISA or PCMCIA version of the device: the kernel can't support both versions of the peripheral on the same machine.

Another method is to copy, then modify the existing driver to make a version specific to a particular embedded machine. The approach was taken for the `arctic_enet` driver for the Ethernet on the eLAP's debug sled. The sled's Ethernet is based on an RTL8019 chip, which is used in a number of ISA cards, as well as several other embedded machines. This method allows multiple versions of the device to be simultaneously supported, but incurs the obvious maintenance problems of having several almost-but-not-quite identical drivers present in the kernel. The situation is aggravated as more embedded machines are supported.

The fundamental problem with this approach is that there are many more types of embedded machine than there are of normal machines. Indeed there is often only one dominant type of conventional machine per architecture (PC for x86, CHRP for PowerPC, Sun server/workstation for SPARC, etc.). With the large number of different types of embedded machine, direct hardcoding quickly becomes messy: there is a lot of duplicated code, and it is inconvenient to add new machines and peripherals.

3.4 The OCP subsystem

Since we can't entirely avoid hardcoded information, the obvious approach to isolating the messiness is to encode information about devices into a data structure which is statically compiled into the kernel, but parsed to provide data to the drivers at runtime. This solution is conceptually similar to using device information from firmware except that the device tree is supplied by the kernel itself, rather than read at boot time.

The "OCP" subsystem (standing for On-Chip Peripheral) is a partial implementation of this approach. It only covers the on-chip devices on PPC 4xx chips, and is quite limited in the sorts of device information it can represent, but it iss still a substantial improvement over hardcoded drivers.

The subsystem has been through several significant rewrites before reaching its present form. The initial implementation in the `linuxppc_2_4_devel` BK tree had a number of serious design and interface problems. It was then rewritten in 2.5 based on the new Linux unified device model, and using the PCI subsystem as a reference. This version is considerably cleaner, but still contains some poorly thought out elements, in particular some things have been copied from PCI which make little sense in their new context. It has now been rewritten again by Benjamin Herrenschmidt in the `linuxppc-2.4` BK tree. This latest rewrite is conceptually similar to the 2.5 version, but considerably simpler and cleaner. This final version still needs to be forward ported to 2.5, re-introducing the integration with the unified device model.

For each CPU with OCP devices, there is a table of definitions like that in Figure 2. This is found in a C file specific to the particular CPU, along with any initialisation or support code for that CPU. The example in Figure 2 doesn't include all the 405LP's on-chip devices, since not all the drivers have been adapted to use the OCP infrastructure yet. The table consists of `ocp_def` structures, shown in Figure 3. At boot time, the OCP system scans the `core_ocp` table to produce a list of OCP devices present, making an `ocp_device` structure (also in Figure 3) for each to keep track of it at runtime.

The `vendor` and `function` fields between them identify the type of device. This mimics the vendor/function pairs used to identify PCI and USB devices. However in this case the ID values are not built into the device but are simply arbitrary values allocated in `include/asm/ocp_ids.h`. Drivers for the on-chip peripherals register themselves when loaded, using the `ocp_register_driver` function and a table of OCP device

from
`arch/ppc/platforms/ibm405lp.c`

```
struct ocp_def core_ocp[]
 __initdata = {
  { .vendor    = OCP_VENDOR_IBM,
    .function  = OCP_FUNC_OPB,
    .index     = 0,
    .irq       = OCP_IRQ_NA,
    .pm        = OCP_CPM_NA,
  },
  { .vendor    = OCP_VENDOR_IBM,
    .function  = OCP_FUNC_16550,
    .index     = 0,
    .paddr     = UART0_IO_BASE,
    .irq       = UART0_INT,
    .pm        = IBM_CPM_UART0
  },
  { .vendor    = OCP_VENDOR_IBM,
    .function  = OCP_FUNC_16550,
    .index     = 1,
    .paddr     = UART1_IO_BASE,
    .irq       = UART1_INT,
    .pm        = IBM_CPM_UART1
  },
  { .vendor    = OCP_VENDOR_IBM,
    .function  = OCP_FUNC_IIC,
    .paddr     = IIC0_BASE,
    .irq       = IIC0_IRQ,
    .pm        = IBM_CPM_IIC0
  },
  { .vendor    = OCP_VENDOR_IBM,
    .function  = OCP_FUNC_GPIO,
    .paddr     = GPIO0_BASE,
    .irq       = OCP_IRQ_NA,
    .pm        = IBM_CPM_GPIO0
  },
  { .vendor    = OCP_VENDOR_INVALID
  }
};
```

Figure 2: OCP device table for 405LP

from `include/asm/ocp.h`

```
struct ocp_def {
  unsigned int   vendor;
  unsigned int   function;
  int            index;
  phys_addr_t    paddr;
  int            irq;
  unsigned long  pm;
  void           *additions;
};

struct ocp_device {
  struct list_head     link;
  char                 name[80];
  const struct ocp_def *def;
  void                 *drvdata;
  struct ocp_driver    *driver;
  u32                  current_state;
};
```

Figure 3: OCP device structure

from `drivers/i2c/i2c-ibm_iic.c`

```
static struct ocp_device_id
 ibm_iic_ids[] = {
  { .vendor = OCP_ANY_ID,
    .function = OCP_FUNC_IIC },
  { .vendor = OCP_VENDOR_INVALID }
};

static struct ocp_driver
 ibm_iic_driver = {
   .name          = "iic",
   .id_table      = ibm_iic_ids,
   .probe         = iic_probe,
   .remove        = iic_remove,
};

⋮

  ocp_register_driver(
      &ibm_iic_driver);
```

Figure 4: OCP driver registration (for the IIC driver)

IDs like that shown in Figure 4. This again is analogous to a PCI driver. The OCP subsystem matches the driver against the list of OCP devices, calling the driver's probe routine for each relevant device.

The `index` field is used to distinguish between multiple devices of the same type. The `paddr` and `irq` fields, unsurprisingly, give the device's physical base address (on PPC all IO is memory-mapped) and its IRQ line. The `pm` field is used for power management, we'll look at it in §5.3. Finally, the `additions` field is a hack used to supply extra device-specific information. It is not needed for any of the devices on the 405LP but it is used on some other chips: for example, some 4xx CPUs, such as the 405GP and NPe405H include one or more Ethernet MAC controller (EMAC) units. These make use of a specialised DMA controller known as the Memory Access Layer (MAL). The `additions` field is used to identify which MAL channels are associated with each EMAC—another piece of information that the kernel has to "just know."

The 2.5 version of the OCP system is integrated with the unified device model. At bootup the OCP code registers an OCP `bus_type` and one instance of it—all the OCP devices are registered as devices on this bus.[3] The `ocp_device` and `ocp_driver` structures become wrappers around the device model's `device` and `device_driver` structures.

4 Future approaches

As yet, there is no really convenient and comprehensive way of dealing with embedded "unprobeable" peripherals. The OCP system is

[3] This ignores the distinction between the two on-chip buses, PLB and OPB. We can get away with this because the POB is always enabled and has a fixed configuration, so in practice we can ignore the distinction for the purposes of device discovery.

probably the closest thing to such a system, but it has significant limitations: it only covers PPC 4xx on-chip devices, and its data structure is a flat table so it cannot represent peripherals behind a bus-to-bus bridge or other more complex interconnections.

The fact that some peripherals are built into the CPU chip is interesting from a hardware point of view. However, for the purposes of device discovery, there is little inherent difference between on-chip devices, and devices which are on a separate chip, but which still can't be straightforwardly scanned or probed. It seems worthwhile, then, to extend the idea of the OCP system to cover embedded devices more generally.

The Linux unified device model provides the obvious place to represent information about these devices at runtime: it already provides the code for matching devices to drivers and its tree structure allows multiple buses with configurable bridges between them to be represented.

It is not immediately clear how to represent everything that's needed in the device tree, though. While for many devices the physical address and IRQ number is all the information that is needed, some devices have multiple IRQs and/or IO windows, at discontinuous addresses. Some buses require different resource addresses: for example many of the 4xx on-chip devices need DCR numbers, and I^2C devices need I^2C addresses rather than physical IO addresses. Hence, devices on different buses are likely to need different wrappers around `struct device` providing different address information.

Even less obvious is how to represent devices with multiple connections, such as the audio codec on the eLAP, connected both the the I^2C bus and to the CSI.[4] As yet, the device model does not have a clear way to represent this.

Another as yet unanswered question is how the device information should be represented in the kernel image. In fact, we gain a little flexibility if this information is removed from the kernel proper by having it as a blob of data which is passed to the kernel by the bootstrap loader (the shim between the firmware bootloader and the kernel proper which handles decompressing the kernel and moving it to the correct address in memory). This has the advantage that on those machines which do have a reasonable sophisticated firmware or bootloader, such as PPCBoot/U-Boot (see [7]), the device information can be taken from there.

from `include/asm/bootinfo.h`

```
struct bi_record {
   unsigned long tag;
   unsigned long size;
   unsigned long data[0];
};

#define BI_FIRST              0x1010
#define BI_LAST               0x1011
#define BI_CMD_LINE           0x1012
#define BI_BOOTLOADER_ID      0x1013
#define BI_INITRD             0x1014
#define BI_SYSMAP             0x1015
#define BI_MACHTYPE           0x1016
#define BI_MEMSIZE            0x1017
#define BI_BOARD_INFO         0x1018
```

Figure 5: Boot info records

On PPC systems, there already exists a flexible method of passing data from the bootstrap to the kernel proper through "boot info records." The bootstrap passes to the kernel a list of `bi_record` structures, shown in Figure 5. Each

[4] Note that this is a different problem to multipath IO. That deals with the case where a device can be accessed by any of several routes, here we have devices that requires several connections simultaneously.

`bi_record` is a blob of data with a length and tag, the internal format of the information being determined by the tag (some tag values are also shown in Figure 5). Currently this method is used for passing information such as the size of memory and the board and bootloader versions. This system could be extended to pass an entire set of device information to the kernel (there is no reason a particular `bi_record` couldn't contain a list of further `bi_record`s, giving a tree structure).

A related question is how to represent the device information in the kernel source. The simplest approach would be to directly include the data structure used to represent the information at boot time. However, that's likely to be quite inconvenient to edit and extend, especially if the format includes length fields (like `bi_record`s) or internal pointers. It might be worthwhile, then, to create a *device tree compiler*: a program used during the kernel build to take a text file describing the device layout and generate code or data to be included in the kernel image.

5 Power management

In a battery powered device such as the eLAP, it is clearly important to minimise power consumption. The most obvious way to do this is to power down sections of the system when they are not in use. Obviously, this means that the kernel needs to know when a device is in use, including when it is in use indirectly because another device relies on it.

The topics of power management and device discovery are therefore related: device discovery is about providing exactly the sort of information about the interconnection of devices that effective power management requires.

As yet the integration of power management techniques with detailed device information is very much a work-in-progress, even on conventional systems and doubly so on embedded machines. So, we can only give an overview here of what the major issues are: most of the hard cases remain to be investigated, let alone implemented.

Again we will use the eLAP as our example: the 405LP's power management features introduce some new (and largely unsolved) problems in providing device information for power management. So, we first examine these features, then in §5.2, §5.3 and §5.4 we examine several different methods of reducing power consumption and the device information problems they introduce.

5.1 eLAP power management features

The 405LP CPU is designed especially for low power operation and as such it has some novel and interesting power management features. Most of the on-chip peripherals can be powered on and off under software control. Some also provide more detailed power control to allow power savings when only parts of the peripheral are in use, or when it is in use intermittently. It also allows for several methods of saving the CPU state while shutting down the chip as a whole (i.e., "sleep" modes).

More interestingly, the 405LP includes a clock generation core that allows the clock frequency of the CPU core and also the PLB, OPB and EBC buses to be altered dynamically. The ratios between the CPU and various bus frequencies are not fixed, so the chip can be adjusted differently for IO versus compute performance. While a number of different CPUs allow the frequency to be changed while running, the 405LP can change frequency exceptionally quickly (microseconds) which enables new power management techniques based on dynamically adjusting frequency based on workload and idle periods. The 405LP can also op-

erate at a variety of different voltages, which can provide much greater power savings than just adjusting the frequency (power consumption varies roughly linearly with frequency and cubicly with voltage, maximum frequency is roughly linear with voltage). Another novel feature of the 405LP is that it can continue to operate, albeit slowly in some cases, while a voltage transition is in progress.

As well as supporting the 405LP's features, the eLAP board has some extra power management features of its own. A number of the on-board devices, such as the audio codec, include the ability to power down some or all of their operations while not in use. Other devices, such as the USB and SDIO chips can be powered down under the control of an FPGA register.

5.2 Static power management

We use the term static power management for the process of suspending or sleeping a machine, i.e. saving the machine's state while turning most or all of the machine off, then restoring the state when the machine is powered on again. This of course is normal in everyday laptops, and handling this for embedded machines is not a great deal different.

Embedded devices do introduce some extra complexities, though. On PC laptops, the BIOS (either APM or ACPI) provides some support to the kernel on how to properly suspend the machine (indeed, in the APM case the BIOS handles most of the work itself). Embedded machines, on the other hand, usually require the kernel to know how to suspend and resume the machine directly. For example the suspend code for the eLAP knows how to use the 405LP's features to save the CPU state, how to configure the RAM to enter self-refresh mode, how to use the board's FPGA registers to turn off the board, and how to rebuild the state when the machine is resumed.

In addition, static power management on all machines requires knowledge of what devices' dependencies on each other are, so they can be shut down and later restarted in the correct order—this was one of the major motivations for the creation of the unified device model. Hence, all the complexities of obtaining detailed device information for embedded systems impact on static power management. However static power management doesn't really add further difficulties beyond those we have already discussed for device discovery.

5.3 Peripheral power management

We use the term peripheral power management to refer to disabling and powering down peripheral devices when they are not in use. This is often relatively straightforward, since it can be handled directly by the driver for the device in question. This also delegates the question of when the device is "in use" to the driver. Sometimes it is sufficient to enable power to the device when it is open, and disable it when closed, other times more fine-grained control of the power is desirable, e.g. to take advantage of idle periods.

When the device depends on other devices being enabled, the situation is a little more complex. However the driver will generally know what the other devices are and their drivers, so it is usually quite simple to create an ad-hoc interface whereby one driver can ask the other to enable the device it requires.

Difficulties do arise where power to one device is controlled by another: for example the 4xx on-chip devices are controlled by a central clock and power management unit (CPM). Similarly many boards have devices which are powered on and off by board-specific FPGA registers.

The 4xx CPM has a simple interface, allowing the OCP subsystem to support peripheral power management quite easily. The `pm` field in the OCP device definitions (see Figures 2 & 3) is a mask describing which bit in the CPM registers controls power to this peripheral. Drivers can then use functions from the OCP subsystem to switch the device on and off.

Obviously, for peripherals that are unique to a particular board it is also easy for the driver to directly control power to the device. So far however, little work has been done on the general case where common devices may be powered on and off by board specific controls.

5.4 Dynamic power management

Dynamic power management (DPM) refers to dynamically adjusting CPU frequencies and voltage during operation. This approach is quite new, at least in Linux, and very much under development. The details of the motivations and approaches to dynamic power management are outside the scope of this paper, for more information see [4]. IBM and MontaVista software are collaborating on further development in this area.

DPM does introduces some new problems related to device information. To work properly, devices in operation may have to impose constraints on what frequency or other settings are allowed. For example, a device may require a certain amount of bus bandwidth, and hence impose a minimum bus frequency, or it may require interrupts to be handled without too high a latency, and hence impose a minimum CPU frequency.

These constraint details are somewhat like the basic information about device interconnection that we have already examined, but clearly require even more detailed information about the devices. Since these constraints may well depend on details of the hardware interconnects, this is yet more information which the kernel must "just know."

Again, the unified device model provides an obvious framework in which to represent this information. [4] discusses some methods for setting constraints within drivers. A general approach to representing constraints in a way that is easily extensible to new boards is yet to be implemented, and is likely to take considerable further investigation and development.

References

[1] IBM Corporation, *PowerPC® 405LP Embedded Processor User's Manual*, Preliminary, 2002.

[2] IBM Corporation, *PowerPC® 405GP Embedded Processor User's Manual*, Seventh Preliminary Edition, 2000.

[3] IBM Corporation, *PowerNP NPe405H Network Processor User's Manual*, Preliminary, 2002.

[4] IBM and MontaVista Software, *Dynamic Power Management for Embedded Systems*, Version 1.0, `http://www.research.ibm.com/arl/projects/papers/DPM_V1.1.pdf`, 2002.

[5] `linuxppc_2_4_devel` kernel tree, `bk://ppc@ppc.bkbits.net/linuxppc_2_4_devel`.

[6] `linuxppc-2.4` kernel tree, `bk://ppc@ppc.bkbits.net/linuxppc-2.4`.

[7] PPCBoot homepage, `http://ppcboot.sourceforge.net/`.

About the author

David Gibson is an employee of the IBM Linux Technology Center, working from Canberra, Australia. Most recently he has been working on board and device bringup for Linux on embedded PowerPC machines, along with various bits of kernel infrastructure for cleanly supporting PowerPC 4xx and other system-on-chip CPUs. He is also the author and maintainer of the orinoco driver for Prism II based 802.11b NICs. In the past he has worked on ramfs (as included in the -ac kernel tree), and "esky," a userspace implementation of checkpoint/resume.

Legal Statement

Gnumeric

Using GNOME to go up against MS Office

Jody Goldberg
jody@gnome.org

Abstract

MS Office is popular for a reason. Microsoft and its massive user base have kicked it hard, and polished the roughest edges along the way. The hidden gotcha is that MS now holds your data hostage. Their applications define if your data can be read, and how you can manipulate it. The Gnumeric project began as a way to ensure the GNOME platform could support the requirements of a major application. It has evolved into the core of a spreadsheet platform that we hope will grow past the limitations of MS Excel. Gnumeric has taught us a lot about spreadsheets and, for the purpose of this talk, about what types of capabilities MS has put into its libraries and applications to provide the UI that people are familiar with. I'd like to discuss the tools (available and unwritten) necessary to produce a competitor and eventually a replacement.

1 Introduction

History

- Aug 17 1997 GNOME Created
- July 2 1998 Gnumeric Created
- Dec 31 2001 version 1.0.0 released
- Aug ?? 2003 version 1.2.0 released

Miguel de Icaza began work on Gnumeric in July 1998 with the stated purpose of building a large, real world, application to validate the GNOME libraries. Without much background in spreadsheets he persevered, learning as he went, writing maintainable code, and testing such core libraries as gnome-canvas, gnome-print, libgsf, and Bonobo.

Status

At this writing Gnumeric now has basic support for 100% of the spreadsheet functions shipped with Excel and can write MS Office 95 through XP, and read all versions from Excel 2 upwards. It covers most major features, but is still lacking pivot tables, and conditional formats which are planned for the 1.3 release cycle.

2 Container File Format

The first step in interoperability with MS Office is to read and write its files. MS Office 95 was an epochal event for the suite. Before then each application had its own format. As of Office 95 it shares a single container format used to wrap each application's native representation. This is convenient for embedding because it allows a user to transfer the entire content of a document, including all of its components as a single file.

2.1 OLE2, The MS solution

The wrapper format used throughout MS office is called *OLE2*. That title actually encompasses the container and a sub-format used to store metadata. GNOME had no support for reading or writing either one. For Gnumeric Michael Meeks used earlier work in laola by Arturo Tena and Caolan McNamara, and scraps of documentation to create libole2. In the last few years the set of available documentation has improved greatly. Apache's POIFS project has generated some nice coherent write-ups for their Java implementation. Additionally, although the Chicago project has not produced any code, they googled extremely well, and collected enough documentation to illuminate the remaining corners.

OLE2 is quite literally a filesystem in a file. It has a directory tree for the offsets and file allocation tables to store the layout of the blocks. Unfortunately for libole2 it also has a meta file allocation table, which libole2 could not export correctly. As a result, after approximately 6.8 MB of data, the library and all of its derivatives produced invalid results. Given the new documentation, we're written libgsf (the GNOME Structured File library) to solve that. The new code has been quite stable, and now that libole2 has been deprecated it has made its way into koffice, abiword, and several other applications. Including potentially OOo via the WordPerfect library.

2.2 The Future

- Format should be easy to create and manipulate without special tools.
- Existing filemagic type sniffing should be able to ID the documents as documents, not their container type
- Support for 'filesystem in a file' structured files to facilitate embedding and split data from metadata.
- Space & time efficient for reading and generation
- Portable to as many platforms as feasible
- signing
- encryption
- embedded object handling
- possible dual/multiple version support
- meta data

Things we do not need

Transaction support is probably overkill. Given the general size of documents, it seems simpler to just generate the entire file each time it is saved.

In-place update (rewrite one element without touching the rest) is also probably unnecessary. This is useful for things like presentation programs with large data blobs (images, sound) and relatively little content, it is also conceivably useful in situations where an external app is editing just metadata. However, the implementation costs for these are high in comparison to available manpower.

At the time of this writing, OOo's container format is the leading contender. There are discussions at this time to potentially adopt the container format (not the underlying xml content).

3 File Format

3.1 XLS

Like a Russian doll, parsing the wrapper format is just the beginning. Within the OLE2

files are sub-files called Book or Workbook (depending on version) that are in the *BIFF* format with several different variants. There is actually some reasonably good documentation for this due to some MS greed (tm) and hard work on the part of the OOo xls filter team. BIFF is fairly similar in structure to earlier lotus formats, and thanks to microsoft padding one of its books (The Excel 97 Developers Kit) with their internal file format docs we know a fair amount about how things are structured. Contrary to the common Slashdot wisdom the hard part is not in knowing the broad details of the records. The devil is in the details, implementers need to understand *what* Microsoft means for each flag and have a strict superset of the corresponding application to avoid information being lost when round tripping data to a proprietary format.

An odd truth of open-source spreadsheets is that they generally interoperate better via xls than their native formats. E.g., Gnumeric can read OpenCalc, but not write it, and OpenCalc has no support for Gnumeric at all. This speaks volumes for the ubiquitous nature of the Microsoft formats. The resource expenditure to support xls fully limits the amount of development time for other formats.

Within the BIFF records is nested yet another format to store the details of the expressions, which is also reasonably documented.

3.2 Escher

A major change in Office97 was the addition of a shared drawing layer for all of the applications. This allows you to draw an org chart in Excel and paste it into Powerpoint. Its high level format was reasonably documented on the web until that was pulled a few years ago. This content is nested within BIFF records within OLE2 records. Escher is a format in the classic 'specification by implementation' genre. Although both OOo and Gnumeric have decent parsers for the records themselves, parsing the content is tricky. One rarely knows what attributes correspond to visible properties. Exporting escher is even more complex. Both Gnumeric and OOo appear to have adopted a monkey see, monkey do approach to work around Microsoft's reluctance to use 'conventional' values for flags requiring things like 0xAAAAFFFF for true for some of the boolean flags.

3.3 WMF/EMF

Users expect their word/clip art to appear faithfully. That content is usually stored using a set of drawing primitives in a series of MetaFile formats (Windows and Extended). The primitives are slightly higher level than a serialized set of X protocol requests. There in an existing rasterizer for wmf in libwmf, but it can not currently handle emf. OOo has a parser for both wmf and emf, but it is tightly coupled to the OO platform and of marginal quality. Proper handling of this is still an open question. There have been discussions with the maintaers of libwmf, and libEMF to combine efforts to fill this niche, but nothing concrete has materialized as yet.

3.4 Security

Unfortunately each element of MS Office handles this slightly differently, and approaches vary from version to version. There is no secure notion of authentication within OLE2, or xls. Microsoft apparently assumes that is handled at a higher level. Within xls there are 3 main forms of encryption. The first is sheet level protection that is little more than an XOR of the records with a 16-byte hash of the password. This is tissue-thin, and can be supported trivially. Workbook level encryption is significantly more secure and uses md5 hashed

passwords and rc4 to encrypt the BIFF record content. Workbook level protection is rather strange. It uses the secure workbook level encryption, with a hard coded password. Gnumeric is the only open source application that can handle all three.

3.5 VBA

This is still largely uncharted territory for open source applications. MS Office appears to store the VBA code in at least 3 formats.

1. Compressed source code. libgsf, libole2, and openoffice can all decompress the code with varying degrees of accuracy. What none of us knows is how to locate the offset to the start of the compressed stream. OO and libole2 both have kludges in place to guess, but neither is reliable. There is clearly documentation on the subject available to the anti-virus manufacturers, but its licensing precludes its use in open source libraries. This is the most likely route to support importing VBA in the near term. It is not immediately obvious that source code is what we really want, because it requires a lexer, parser, and libraries to back it up—a rather significant amount of work. There is some hope that the Mono project and its emulation of the .Net API will provide support for this.
2. P-Code. A preparsed set of tokens to be interpreted by the VB engine. The format of this has no open documentation, although it is fairly amenable to parsing. On the positive side this is the holy grail in many ways. Having pre-parsed code removes the need for a lexer and parser, and allows us to map the content to more modern languages such as python. The down side is that the p-code comes in many variants, and depends on versioning.
3. S-Code. Like P-Code, but different in some unspecified way. No known parsers or documentation exists.

4 Expansion

In addition to their core functionality, office applications are expandable. Organizations have some limited ability to customize their installations, and third party developers can use them as a development platform for their niche applications. From tasks as simple as workflow macros, up to massive extensions such as FEA's @Risk, or DeltaGraph, people are extending MS Excel.

4.1 XLLs & XLMs

Early versions of Excel offered 2 forms of extension. The first was a kludge that grafted a pseudo-procedural language into the functional format of a spreadsheet, called XLM. This is no longer widely used.

There is also the opportunity to load an XLL, a DLL shared library with special entry points, directly into the process' address space. Although its interface is only partially documented, somewhat byzantine, and deprecated, this is the most popular form of extension. The primary benefit is that a developer's code can be written in C/C++, and compiled to link the external libraries fairly easily.

To the best of my knowledge there are no open source or proprietary applications that support XLMs or XLLs other than MS Excel. This hinders transition. XLMs could potentially be supported, but the limited remaining user base does not warrant the expense. XLLs might be feasible under win32, but due to the nature of the interface, would probably require WINE under *nix.

4.2 VBA & OLE

With Office97 came a unification of embedding and scripting via VBA and the OLE component model. VBA as a language was a significant improvement over XLM and quickly supplanted it. OLE was more of a mixed blessing. The increased complexity of the interface required Dev Studio wizards to generate the wrapper code, which was fairly unmaintainble. As a result most installations fell back on VBA external declaration support to add new capabilities. This worked well for tasks amenable to scripting, but was painful when linking to external analytics. Adding a new spreadsheet function involved writing it in C, then creating a dummy wrapper in VBA that links to it.

4.3 Gnumeric

Worksheet function management is an area where Gnumeric is well ahead of MS Office. Adding new functions to Gnumeric is trivial. An abstract interface for loading modules has been implemented for shared libraries (via glib's g_module utilities), and python. Coupled with an xml-based configuration mechanism and just-in-time loading, the vast majority of the worksheet functions are in plugins.

Less well defined are the scripting interfaces. Building on GNOME's strong set of language bindings there have also been experiments in scripting in guile, perl, CORBA, VB, and python. The only clear result thus far has been that the scripting interface is fairly language-agnostic. Defining a clear and coherent api is on the short list of extensions to be made during the 1.3–to–2.0 development cycle.

5 Preferences

Storing user preferences is another area where GNOME technology has the advantage over its Windows counterpart. GConf attempts to learn the lessons of the Windows registry while learning from its failures. By storing content in several distinct user readable xml files, gconf offers the convenience of a global structured storage, while retaining the flexibility in the face of file errors or corruption not found in the more monolithic Windows registry. Work remains though. There is still some thought necessary to implement lockdown features, and to address logical paths (HOME, PREFIX, etc.).

6 GUI Toolkit

Over time, the initial separation between gnome libraries as extensions to gtk have been largely removed. With the addition of pango to handle advanced text, and extensions to Gtk's rendering model that produced the foocanvas, gtk+ now supports the primary display needs of Gnumeric. Coupled with libglade for easy maintenance and configurability, it is relatively painless to produce extremely usable dialogs. There are, however, a few lingering issues to address.

Configurable UI

The current gtk+ api for menus and toolbars makes no distinction between the actions and their layout. Applications are forced to hard-code their menu/toolbar layouts in order to modify them. This removes the ability of a user to reorganize things. The limitations of the gtk+ path-based API prompted the creation of GNOME_UI app helpers, which simplified creation and added stock items to improve consistency between GNOME applications. However, it did nothing to solve the issue of hard-coded layouts. In an effort to solve the problem of merging menus and toolbars for components, Bonobo made an attempt to solve the layout problems by separating the layout from the actions. Unfortunately, the API was pro-

duced without enough review, and it was insufficent for large applications. The hopefully final rendition is now in its evaluation phase in libegg menu/toolbar. This code allows effective management of different action groups, and the creation of new action types such as combos and accelerators. The main remaining question is how to store a user's edits. KDE has long had support for this sort of editing, they don't appear to have a good solution to storing the edits as yet.

File Selector

The gtk+ file selector has long been a source of ridicule and disgust. It is functional, but too barebones for a modern desktop. The fact that it has no solid support for network addresses, or histories has greatly hindered the adoption of gnome-vfs. There have been several write-ups and Owen Taylor has apparently completed a replacement version that will be included in gtk+-2.4.

7 Spreadsheet-specific Functionality

Spreadsheets are an ideal testing ground for all those obscure datastructures we all learned back in school. In most instances there is not enough data to make using something esoteric worthwhile. With an apparent size of 256 $\times$ 64k (Gnumeric can scale considerably larger), it is very easy to quickly operate on significant swaths of data.

As an example, Gnumeric uses an asymetric quadtree to store style information. This allows us to easily handle someone doing a "Select all, Bold" (explode kspread). It also supports "Select all minus one row and one col" (explode MS Excel, and OpenCalc).

8 Acknowledgements

The Gnumeric development team. You know who you are, as do the CVS logs.

Ximian employees for their continuing personal contributions to Gnumeric.

9 References

`http://www.gnome.org/projects/gnumeric`

DMA Hints on IA64/PARISC

Optimizing DMA performance for HP Chip sets

Grant Grundler
Hewlett Packard
grundler@cup.hp.com

Abstract

Modern IO subsystems implement complex DMA transaction parameters, called DMA hints, which are not explicitly supported by the Linux DMA API. This paper investigates benefits of using non-default DMA hints and thus whether such hints should be abstracted into the DMA API. My conclusion is the implementation (ZX1) investigated does not warrant changing the DMA API. Other implementations need to be compared before proposing any changes.

HP PA-RISC (Astro[1]/Elroy[2]) and IA64 IO Controllers (ZX1) both support several types of DMA hints and both are commercially available. My primary interest was the ability to prefetch cache lines for PCI devices. The benefit is same as for CPU: bring the data closer to the consumer. But to my surprise, cache line prefetching is not the most important hint since default prefetching works well for all devices. Relaxing the PCI ordering rules turns out to be more important since firmware can't know when it's safe to do so.

Updated versions of this paper will be available from `http://iou.parisc-linux.org/ols2003/`

1 Introduction

DMA performance seems like such an obvious thing. Drivers just need to tell the device where to fetch something from memory, poke it, and life is good. Unfortunately, those days are over.

Modern SMP servers require multiple levels of bridges in order to support PCI-X Bandwidth (peak burst rate 133MHz/64-bits). In order to work well with CPUs and memory controllers, IO Devices participate in the CPU Cache Coherency protocols. They also need to minimize the number transactions used and use the appropriate type of transaction in order to optimally utilize available bandwidth.

Throughout this paper (and even in the title!) I use the word *Hint* which implies an "informational only" parameter. This isn't strictly accurate. Some platforms depend on certain parameters for correct operation. I.e. incorrect results may occur for some combinations of DMA hints. The DMA hints discussed in this paper *should* always provide correct results though I've crashed the ZX1 with some hints as noted.

And I recycled the ZX1 block diagram used in my 2002 OLS talk, "Porting Drivers to ZX1."[3] The diagram is useful to understand the routing of data between PCI devices, Memory, and CPU. [4]

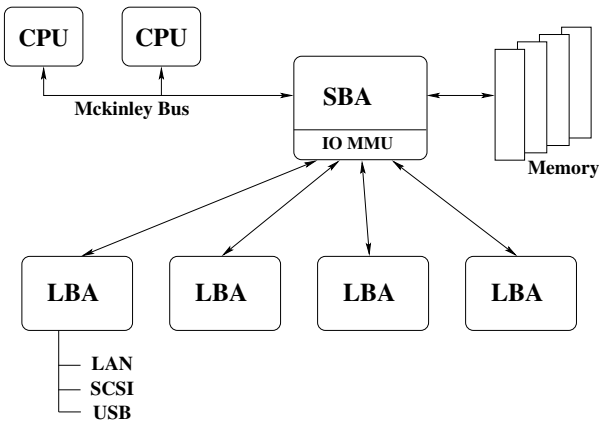


Figure 1: HP ZX1 Block Diagram

2 Overview of DMA

The following sections introduce some of the key concepts relating to Direct Memory Access.

2.1 Consistent vs. Streaming DMA Mappings

The Linux DMA mapping interface differentiates between two memory access patterns. A short summary of `DMA-mapping.txt`[5] follows.

Consistent DMA mappings are intended for data which is concurrently accessed by both CPU and PCI device(s) (i.e. Host RAM base device control structures like mailbox rings). Key feature is updates (writes) from either must be visible to the other based on PCI ordering rules. In short, fairly strict R/W ordering rules and transactions are typically less than a cache line in length.

Streaming DMA mappings are intended for memory regions exclusively accessed by the PCI device(s). This "exclusive" access begins when a host memory region is mapped and ends when the same region is unmapped.

The Streaming DMA interface provides two explicit hints: DMA direction and DMA length. From the length, we know the block size on which the DMA will terminate. But as noted in the introduction, DMA direction is required for correct operation on some platforms—but not ZX1 or PARISC IOMMUs.[1] The ZX1 *System Bus Adapter* (aka SBA; See `sba_iommu.c`) code does optionally use direction to optimize VM bits. Other hints regarding PCI Ordering compliance and DMA Read data consumption rate are not specified.

2.2 PCI DMA

A single DMA operation is fairly straight forward at the PCI bus level. The PCI Device asks the PCI arbiter for bus ownership. The Arbiter eventually grants the PCI device ownership of the bus. PCI device accepts ownership and sends the target address (possible two cycles worth for 64-bit addressing) followed by data. The transaction ends when either the PCI Controller asks the device or the PCI device volunteers to give ownership.

PCI supports several different types of Commands. Here are the ones relating to DMA along with summaries of their PCI Local bus definition:

- **Memory Read** command is used to read data from an agent mapped in the Memory Address Space. The target is free to do an anticipatory read for this command only if it can guarantee that such a read will have no side effects. Furthermore, the target must ensure the coherency (which includes ordering) of any data retained in temporary buffers after this PCI transaction is completed.
- **Memory Read Line** command is seman-

[1] PARISC platforms without IOMMU do require R/W direction hint.

tically identical to the Memory Read command. Use of MRL indicates the intention to read a full cache line of data.

- **Memory Read Multiple** command is semantically identical to the Memory Read command. Use of MRM indicates the intention to read more than one cache line of data before disconnecting. MRM is intended to be used with bulk sequential data transfers where the memory system (and the requesting master) might gain some performance advantage by sequentially reading ahead one or more additional cache line(s) when a software transparent buffer is available for temporary storage.

- **Memory Write** command is used to write data to an agent mapped in the Memory Address Space. When the target returns "ready," it has assumed responsibility for the coherency (which includes ordering) of the subject data.

- **Memory Write and Invalidate** command is semantically identical to the Memory Write command. The difference is MWI requires the device to write at least one complete cache line and the Host Cache controller can invalidate existing contents without having to send the contents (just reassign ownership) to the IO or Memory Controller. This avoids unnecessary cycles on the Front Side Bus for DMA writes. MWI requires `CACHELINE_SIZE` register in the device configuration space to (a) be implemented and (b) programmed by BIOS or PCI Initialization code to a suitable value.

The number of bytes transferred is constrained by the transfer type (MR, MRL, MRM, MW, MWI) and `LATENCY_TIMER` value. The `LATENCY_TIMER` is described briefly later in this paper and by the PCI Local Bus Specification.

Memory Writes are typically the simpler case from a software performance perspective. DMA Writes are buffered by the chip set and routed to the memory controller[6] at whatever rate the internal interconnect supports. Data throughput is typically limited by the PCI bus controller[7] or memory controller.

Reads are more complicated because the memory controller latency is harder to hide. All data handling systems (like disk IO) deal with this problem by "Read Ahead" (a.k.a. prefetching) or caching. However, large caches like those implemented in a CPU are expensive in many ways. And large caches don't help much since the read and write access patterns for IO devices typically aren't for the same cache lines repeatedly. Or in the case of shared data, the IO device competes at times with the CPU for the cache line.

Successive requests for bulk data can be prefetched by the I/O Controller. I was expecting prefetching to make a big difference for PCI devices. But the individual devices I tested (except 53c1010 Consistent DMA) did not perform better or worse for different levels of prefetching. Like disk IO, the effectiveness of the prefetching really depends on the data access pattern. I suspect this is because the PCI devices are designed to work well with out any prefetching and buffer enough data to keep the IO device from stalling.

2.3 DMA on PCI-X

PCI-X obsoletes much of what I was trying to accomplish in this paper. Cache line prefetching hint only applies to PCI. But three new PCI-X-only features are of interest:

- *Attributes* are part of the PCI-X command.

- *Split Transactions* replace the goofy "retry forever" schemes used when read data is not available.

- *Burst Transactions* replace MRM, MRL, and MWI.

2.3.1 Command Attributes

The PCI-X Specification[8] (drafts are available for free) has more details about command attributes in section *2.3 PCI-X Command Encoding*. I'll try to summarize below. Two attributes are currently defined for PCI-X commands.

Relaxed Ordering is also described in Section 4.1. In a nutshell, ignoring the PCI ordering rules regarding Programmed I/O (CPU R/W) and DMA (PCI R/W) yields measurable performance gains without sacrificing correct operation. The handful of drivers I've used under PARISC-Linux and IA64-Linux run just fine with outbound data[2] ordering rules relaxed.

Note the PCI-X spec doesn't require the PCI-X device to use Relaxed Ordering attribute when the Relaxed Ordering bit is set in the command register. ZX1 chip can override the Command Attribute for Relaxed Ordering behavior. And ZX1 chip set only implements this optimization for outbound data flow (PIO Write/DMA Read return). The PCI-X spec defines optimizations in both directions of data flow.

No Snoop attribute isn't relevant for most IO devices. Most Linux drivers expect DMA transactions under control of DMA mapping services to be coherent. "No Snoop" means the host driver guarantees the latest copy of a cache line is in the memory controller. And it implies the chip set can perform better if it doesn't have to Snoop. My understanding is non-coherent transaction are interesting for graphics devices, not the LAN/Storage devices I work with.

[2] Inbound data reordering causes Bad Things™ to happen. Discussed on ia64-linux mailing list.

2.3.2 Split Transactions

Split Transactions just means the request for information and the completion (reply) of that request are separate transactions on the bus. This is a good thing for several reasons:

- Up to 32 transactions can be pending at the same time. Only 5 bits are defined in the *Tag* field of the PCI-X command. But the exact number of outstanding transactions supported depends on chip set implementation. This is identical in concept to *Tagged Queues* defined for SCSI protocol.

- More efficient: eliminates the need to poll (aka "retry forever") when the PCI Controller disconnects in the middle of a (e.g.) read transaction.

- Acts more like a Memory bus rather than an IO bus. Thus, it's easier for HW designers to route transactions across a larger fabric.

2.3.3 Burst Transactions

Memory Read Block (MRB) and **Memory Write Block** (MWB) are replacements for MRL/MRM and MWI respectively. The key difference between the above PCI and PCI-X commands is addition of a *Byte Count* field. By making the transfer length visible to the PCI-X controller, the chipset can prefetch cache lines appropriately. This is significant since not involving driver writers for 4 or 5 different OSs to program DMA hint bits is a good thing.

3 DMA Parameters

Two additional parameters defined by PCI Local Bus specification affect DMA behavior. Both reside in the PCI device configuration space header: `LATENCY_TIMER` and `CACHELINE_SIZE`.

3.1 `LATENCY_TIMER`

`LATENCY_TIMER` constrains how long the PCI device will burst DMA before volunteering to give up the PCI bus. It should be long enough to transfer several cache lines of data if the device is capable. While this is a "tunable" parameter, I didn't feel it was necessary to experiment with this value since `LATENCY_TIMER` is a pretty well understood and described else where.

3.2 `CACHELINE_SIZE`

`CACHELINE_SIZE` is only required by PCI MWI command. `CACHELINE_SIZE` has to be the IO cache line size and not the CPU size. Typically this is either the line size of the memory controller or the line size of outer most CPU cache (e.g. L2 or L3 Cache). The CPU can use smaller lines for first and/or second level caches.

Since firmware is expected to program `CACHELINE_SIZE` with the appropriate value, it's a not really either a parameter nor tunable.

4 HP ZX1 DMA Hint bits

The HP ZX1 chip set implements several bits for DMA Hints. None of these are supported by the current Linux DMA mapping API. My original goal was to propose extensions to the Linux DMA mapping API. But I've concluded it's not absolutely necessary and I don't know which hints are of interest to other chip sets. Hopefully this paper will precipitate more discussion and comparison of chips and capabilities.

The primary reason it's not absolutely necessary is IA64 firmware is compensating for an ignorant OS. Firmware errs on the overly aggressive side in setting the cache line prefetching (for simple, single unit tests) and errs conservatively on the correctness case (Relaxed Ordering is disabled). Though it's not optimal, ZX1 IOMMU code could blindly set Relaxed Ordering hint bit (i.e. not enforce ordering) and some hacks can take care of the others.

And because PCI-X obsoletes the key PCI-only hint, I can't argue HP needs them. My pet architecture (PA-RISC) would benefit since HW shipped to date only supports PCI—but that's not of commercial interest. And PA-RISC alone is not a justification for extensions to the interface.

Even if I had HW descriptions for other IOMMUs, it would be a lot of work to independently abstract the DMA hints. Understanding platform IOMMU support well enough to abstract what's important is non-trivial. And since I'm fundamentally lazy (or "good at optimizing" as Bdale Garbee puts it), I'll pass for now.

Lastly, assuming hints are chip set specific (since no one has abstracted them), introducing hints for each chip set is the path to hell for driver writers. A relatively small number of people (per OS) understand how one IOMMU on one platform works. Trying to get a broader audience to understand several platform chip sets is unrealistic. Been there, done that.

4.1 Relaxed Ordering

Relaxed Ordering tells the HW it can ignore one PCI ordering rule. PCI-X specification offers this optimization in each direction (not just outbound) with its definition of Relaxed Ordering. HP's chip set is sufficiently well implemented in the inbound (DMA writes) path that the inbound optimization isn't helpful and thus not implemented.

HP also calls this optimization "PIOW/DMAR Ordering." The cryptic acronym means "Programmed IO Writes/DMA Reads" Ordering. Setting this hint indicates the driver and device don't depend on ordering of DMA Read returns and PIO Writes for correct operation. This hint allows DMA read returns to bypass PIO Writes in order to prevent an in-progress DMA burst from disconnecting on the PCI bus and force retries.

I didn't have any expectations for this hint. Mostly because of my ignorance when I started this investigation.

4.2 Read Current

Read Current transactions gets the most recent copy of cache line data *without changing the cache line state*. The key thing is the CPU can keep cache line ownership. By not giving up ownership, the CPU can continue to modify cache line contents without having to fight with the IO Controller (ping pong) for the cache line. However, the copy of the cache line is not maintained and can become stale. It should be consumed immediately (for some finite definition of *NOW*; parents will understand). Thus it's most useful when data is leaving the cache coherency domain (i.e. DMA reads).

Most driver writers will not need (or want) to worry about Read Current hint. First, different chip-sets have minor variations in implementation which may in fact still ping-pong the cache line. Secondly, Read Current hint has no effect on chip sets which already implement DMA reads by issuing Read Current transactions. And third, to date, none of the PCI devices that interest me obviously benefited by explicitly setting or clearing this hint bit on the platforms I've tested.

Read Current is implemented on both PARISC Runway[9] and McKinley bus.

4.3 Cache line Prefetching (PCI Only)

Cache line Prefetching for IO devices serves the same purpose as prefetching does for CPUs: avoid stalling by bringing data closer before it's needed. The amount of prefetch needed is a function of the device's *data consumption rate* and the actual memory controller latency.

For example, if the memory controller can deliver a cache line in 120ns and the device can consume a cache line in 120ns (8 * 15ns), we need to prefetch 1 cache line at any given moment in time. In other words, 2 cache lines of data will be in flight at any given moment in time. But as system workload increases, the average memory controller latency usually goes up too. It might really take 200 or 300ns to deliver a cache line. We need to compromise and pick the number of cache lines to prefetch so things work OK under worst case but perform optimally in the "expected work load" range.

ZX1 chip set can deliver a 128 byte cache line in about 110 ns[10] PCI devices can only consume 128 bytes every 240 ns (16 * 15ns) at best. The PCI device probably stalls less than 1/3 of the time waiting for data. This is substantially different than for the PARISC chip set which can only deliver a 64 byte cache line in about 180 ns.[11]

PCI-X mode of operation does NOT support cache line prefetching hints. It's not necessary because with split transactions, the device can have much more IO outstanding and in effect, perform its own prefetching.

4.4 DMA Block Size (PCI Only)

DMA Block Size tells the chip set when to stop prefetching. Prefetching will continue up to the block size boundary and resume when the first cache line of the next block is requested.

This hint also does not apply to PCI-X busses. It's not necessary because the PCI-X *Burst Transactions* specify the number of bytes being transferred and the chip set (or OS code) doesn't have to guess when to stop prefetching.

I didn't expect block size to matter much in single unit testing. It would be interesting to know how much it matters when testing a fully loaded system.

5 Case Study: BCM5701 (PCI)

In 2002, around the same time HP ZX1 products became available, HP started shipping Tigon3 NICs (designed and tested by HP). The BCM5701 NIC supported by HP-UX is shipped operating in PCI mode.

Test used is:

```
/opt/netperf/netperf -l 60 \
-H 10.0.30.0 -t UDP_STREAM -- \
-m 1024 -s 131072 -S 131072
```

UDP_STREAM is useful for testing output if the host networking stack only sends what the NIC can consume. I'm told this is the case for Linux. And while some applications really do run on top of UDP, I also ran TCP_STREAM test to get an idea of the workloads I'm familiar with.

Client (HP RX2600, 1GHz) was running 2.4.20-em19 + tg3 v1.5 over the built-in BCM5701.
Server (HP RX2600, 900MHz) was running the same kernel, same built-in BCM5701.

NICs were connected via cross-over cable and set to either 1500 or 9000 bytes MTU.

Firmware sets the default PCI Command Hint to 3 cache lines prefetch, Relaxed Ordering disabled, 4k block size, Read Current enabled.

I then varied the DMA Hints on the *Client* who was sending packets. While this sounds backwards, it's the netperf point of view. We want to observe the netperf "client" send performance.

5.1 Relaxed Ordering

Relaxed Ordering Hint is (on) enforced by default. I've turned it off selectively for MR, MRL and MRM PCI transactions in Table 1. Runs with 2.4.20+tg3 v1.5 only showed about 4% improvement with 1k messages.

Previous experience with UDP_STREAM testing on RHAS 2.1 (IA64, e.25?, using tg3 v0.99) demonstrated nearly 10% performance improvement with 1080 byte messages. Without ordering enforced, netperf reported 862 Mb/s[3] vs. around 775 Mb/s when ordering was enforced (default behavior).

TCP Stream test showed a smaller, but similar, sensitivity to this parameter. Clearing Relaxed Ordering hint in the MRM hint for Streaming DMA resulted in 778 Mb/s (vs. ~758 normally) using 1024 byte message and 1500 byte MTU.

[3] Or 848 Mb/s when the netperf client ran on the same CPU as the one interrupts were directed at.

PCI Cmd	Consistent	Streaming
NONE	759.61	758.74
MR	759.09	760.67
MRL	759.99	759.70
MRM	759.68	797.19
ALL	758.99	800.65

Table 1: BCM5701 UDP_STREAM Relaxed Ordering, 1k Msg

5.2 Cache line Prefetching

Neither TCP nor UDP showed any statistically significant differences as I varied the cache line prefetching for MR, MRL, or MRM commands. This was true for both 1k (1500 byte MTU) and 8k (9000 byte MTU) message sizes.

I suspect this is primarily because I'm measuring what the card is buffering, not PCI bus utilization. The card's ability to buffer is not affected by how inefficient the PCI bus is used. Unfortunately, I don't have the tools to measure PCI Bus utilization on the RX2600.

Again, like learning PCI-X obsoletes cache line prefetching, this is a disappointing but useful result.

5.3 DMA Block Size

Unlike cache line prefetching, I didn't expect much difference between the various block sizes. Once I knew prefetching makes no difference for the BCM5701, the fact that varying Block size hint also makes no difference was no surprise.

5.4 Read Current

Disabling Read Current Hint for Consistent mappings will crash the system. It's not clear to me why. I talked with the HW designers and it is clearly not an expected result due to how the read/write paths are implemented.

I wasn't expecting any measurable performance difference with (vs. without) Read Current for Streaming DMA. And in fact, I didn't see any.

6 Case Study: BCM5704S (PCI-X)

Since I only have one BCM5704S,[4] I ended up connecting both ports of the BCM5704S (tg3 v1.5) to the 82546EB (e1000 4.4.12-k1) in the other machine.

It's worth mentioning the BCM5704S sits behind an IBM PCI-X to PCI-X bridge:

```
...
+-[80]-+-01.0-[81]--+-04.0  QLA2312
|      |            +-04.1  QLA2312
|      |            +-06.0  BCM5704S
|      |            \-06.1  BCM5704S
|      \-1e.0     PCI Bus Controller
...
```

The bridge plays a bigger role in performance than people expect. For grins, I sent 4k messages out the client through both BCM5704S ports at the same time using default hints. Throughput was 990 $\pm$ 0.1 Mb/s for each port (total 1980 Mb/s). vmstat reports ~25% CPU (dual CPU systems) on the server (e1000 driver) and about 33% on the client (tg3 driver). Why was throughput so good? The IBM PCI-X bridge is prefetching data for the chip and also supports split transactions. The prefetching caused some heartburn for the IOMMU code since the IBM bridge ended up prefetching past page boundaries on early prototypes. Changes were made to the ZX1 PCI

[4] HP has no plans for productizing anything with BCM5704 on IA64 at this time. It happens to work and provides a nice comparison to the BCM5701 case study (PCI-X vs. PCI).

PCI Cmd	Consistent	Streaming
ALL	949.35	950.93
MR	952.97	951.79
MRL	952.64	952.66
MRM	953.97	952.58
NONE	951.19	945.10

Table 2: BCM5704 TCP_STREAM 8k Msg

Msg Size	UDP TX	UDP RX
1024	937.77	492.27
1024	937.76	477.22
4096	983.70	959.55
4096	983.70	959.59
8192	991.80	991.80
8192	991.80	991.80

Table 3: e1000 v4.3.15 UDP (Mb/s)

Bus Controller (aka LBA) to stop the prefetching behavior.

6.1 Relaxed Ordering

With 4k messages, running TCP_STREAM gave consistent results around 990 $\pm$ 0.1 Mb/s. This wasn't the case for 8k messages. I'm not sure why since the MTU should have been 9k when running the tests. Table 2 is included for your amusement only.

It's irritating I don't know why the results are lower than with 4k messages or what's causing the variability. For the record, UDP_STREAM test was able to send 984.18 $\pm$ 0.01 Mb/s using 4k messages and 992.04 $\pm$ 0.01 Mb/s for 8k messages.

7 Case Study: 82546EB (PCI-X)

This is Intel's "4th Generation Gigabit MAC design with fully integrated, physical-layer circuitry to provide two standard IEEE 802.3 Ethernet interfaces. . . ."[5] Same setup as with the BCM5701 except an Intel add-on NIC is in both netperf client and server. Both NICs are configured to use 9000 byte MTU.

Table 3 shows results for the driver Intel offered on their web site download area: e1000 v4.3.15 driver. But as Table 3 shows, TCP results varied from 639 to 660 Mb/s (1k messages) and got worse (540-565 Mb/s) for 8k messages. UDP results for smaller messages were very poor as well. Something is clearly wrong.

Msg Size	TCP	UDP TX	UDP RX
1024	976.75	937.76	501.25
4096	978.16	983.72	983.70
8192	907.69	991.80	991.80

Table 4: e1000 v4.4.12 TCP/UDP (Mb/s)

In contrast, TCP Streaming performance for v4.4.12-k1 e1000 driver was quite good. With default hints, both ports combined could send about 1770 Mb/s using 8k message.

7.1 Relaxed Ordering

Disabling ordering enforcement did not change performance in any statistically significant way. In fact, UDP results were identical to Table 4 except for slightly higher UDP RX result. TCP results also showed the same 70 Mb/s drop for 8192 byte messages. And combined port throughput stayed around 1770 Mb/s for 8k message size.

For grins, combined throughput with 4k message size achieved 1936 $\pm$ 1 Mb/s. Definitely an impressive result given both ports are shar-

[5] HP has no plans for productizing 82546EB on IA64 at this time. 82546EB only happens to work under Linux because e1000 driver uses I/O Port space. This chip has serious bugs when using MMIO space to access registers.

	Consistent		Streaming	
PCI Cmd	Order	Current	Order	Current
MR	223	NA	221	220
MRL	220	NA	221	214
MRM	220	NA	221	208
NONE[6]	222	NA	219	217

Table 5: 53c1010 Ordering/Current Hints (MB/s)

ing the PCI-X bus.

7.2 Read Current

Clearing Read Current bit for either Consistent or Streaming DMA resulted in a slight drop (890 Mb/s) for TCP Streaming test compared to Table 4. I'm suspicious of this result because afterwards, I could consistently only get 960 Mb/s (8 Mb/s less) for TCP Streaming using 4K messages.

I didn't run UDP tests for Read Current Hint.

8 Case Study: LSI 53c1010 (PCI)

LSI's 53c1010 (Ultra3 LVD) is pretty widely used along with 53c896 (Ultra2 LVD). Both are driven by the sym53c8xx_2 SCSI driver.

Since parallel SCSI busses are not duplex, testing this was fairly straightforward. I setup a MD RAID0 across both channels (alternating disks) with 10 odd-ball Ultra3 disks (9, 18, 36GB, mix of vendors). Then ran:

```
dd if=/dev/zero of=/dev/md4 \
    bs=64k count=200000
```

I learned later that running RAID0 was not such a good idea. More on this in the u320 (53c1030) case study.

Consistent DMA				
Prefetch Depth	0	1	2	3
MR	223	219	219	224
MRL	224	223	221	224
MRM	104	168	220	223
Block Size	512	1024	2048	4096
MR	223	223	222	223
MRL	220	218	218	221
MRM	220	221	219	220
Streaming DMA				
Prefetch Depth	0	1	2	3
MR	218	214	223	219
MRL	221	223	216	219
MRM	216	221	214	220
Block Size	512	1024	2048	4096
MR	216	220	208	219
MRL	215	221	221	222
MRM	218	210	224	223

Table 6: 53c1010 Cache line Prefetching, MB/s

8.1 Relaxed Ordering and Read Current

I've globbed both Relaxed Ordering & Read Current into Table 5 only because they are both boolean values. Differences of less than 3 MB/s are probably not significant.

Disabling Read Current for Streaming DMA clearly reduces performance for MRL and MRM transactions. I thought the "NONE" (217 MB/s) result is a weighted average of all three types of transactions but that is a logical fallacy. This result can't be better than the worst case unless some other interaction is taking place.

8.2 Cache line Prefetching

Of particular interest in Table 6 is the extent Consistent MRM prefetching affects throughput. I guessed this is because the 53c1010

[6] Well, this should really be "ALL" for Relaxed Ordering hint since all the bits are set.

"scripts" are kept in host memory (but cached locally) and all IO grinds to a halt when an uncached portion of script is not available. James Bottomley suggested the entire script fit in onboard RAM and was loaded under Host CPU control at init time. If true, then the scripts themselves are sequentially fetching control data and getting hurt badly by not having the control data available immediately.

9 Case Study: LSI 53c1030 (PCI-X)

Using the same methods (and the same u160 disk drives) as for 53c1010 didn't work. The results varied from 160 MB/s to 185 MB/s regardless of hint settings. I expected at least equivalent performance to the 53c1010 and suspect whatever is causing the variability is also limiting performance.

Trying a different method suggested by James Bottomley led to an interesting result. He was appalled I was using RAID0 because of issues with MD layer not coalescing IO requests again at the disk level. But using a 64k chunk, aka stride, I thought would provide big enough blocks.

To avoid RAID0, James suggested checking if multiple copies of `sg_dd` would (one per disk) would work. Well, I'd like to see multiple IOs outstanding per spindle. And fortunately `sgp_dd` man page suggests exactly that. Nice.

While the advantage of this method is it bypasses lots of kernel code related to buffer cache, the drawback is it also bypasses all the statistics gathering in the kernel. Neither vmstat nor iostat sees any of this disk activity. The solution is to measure the throughput of each disk individually (23 to 48MB/s) and then adjust the number of blocks transfered such that all 10 disks finished their sgp_dd process within about 1 second of each other. Then `date +%s` could time the cumulative I/O. Add up how much data each sgp_dd copied and divide by total time. This worked better than I expected and Table 7 shows how consistent that data was. The accuracy of the data is $\pm$ 1 second of 153 second (average, 266666 KB/s) run times. In retrospect, clearly a better method than using RAID0 and suggests roughly the performance RAID0 should be getting.

PCI Cmd	Consistent	Streaming
ALL	268421	266666
MR	266666	266666
MRL	266666	264935
MRM	268421	266666
NONE	264935	266666

Table 7: 53c1030 Relaxed Ordering, KB/s

The bad news is that despite contortions to collect reliable data, neither Read Current nor Relaxed Ordering hints made a statistical difference for the configuration I had. I still wonder if I misunderstand what the hint bits mean in the context of PCI-X. But I couldn't find anything in the chip set documentation to indicate otherwise. I worry the ZX1 chip set might "allow" (logical *And*) the ZX1 DMA Hint and PCI-X command attribute bits vs. "forcing on" (logical *Or*). My expectation was the latter based on documentation.

10 Case Study: qla2312 (PCI-X)

The qla2312 is a Qlogic PCI-X, dual port, 2Gb/s FC chip. Qlogic sells this chip for both dual port and single port FC HBAs. A single port is theoretically capable of 2 Gb/s (about 200MB/s) output and input (full duplex). The dual port HBA is theoretical capable 800 MB/s throughput. I tested the qla2312 in two configurations: with IBM PCI-X bridge and again

without (thinking the PCI-X bridge was substantially impacting results).

I used the same methodology as for the 53c1030 Case Study with one of the two ports. Unfortunately, I didn't get a second DS2405 enclosure until much too late. And then I found out the second FC port on the card with the PCI-X Board was disfunctional. I was only able to run a few tests through both ports on a QLA2342 FC HBA (uses qla2312 chip).

The qla2312 HBA was running in PCI-X mode with Firmware version 3.01.18. Same 2.4.20-em19 kernel as before with qla2300 v6.04.00 driver.

10.1 Outbound vs. Inbound IO

To cut to the chase, setting Relaxed Ordering or disabling Read Current hints did not affect performance. With 8 disks, `sgp_dd` was consistently writing 190 $\pm$ 1 MB/s. Two things might have contributed to this result: No inbound load was saturating IO path or PCI-X to PCI-X bridge was "hiding" the effect.

Alone, `sgp_dd` inbound (IO reads) workload would get 198 MB/s consistently. Combined with the same outbound (IO writes) workload as above, the inbound rate drops to about 145 MB/s and the outbound workload hovers around 51 MB/s $\pm$ 1 MB/s.[7] Again, Relaxed Ordering and Read Current Hints made no difference.

Switching to the other RX2600 (900MHz) which had a qla2312 connected to the same set of disks, I reproduced the 198412 KB/s on the inbound-only workload as well. Bidirectional throughput was about the same: 143 MB/s in and 53 MB/s out (9% CPU utilization).

Having spent several days on this, I started to doubt this HBA was operating in full duplex mode despite all the marketing literature making such a claim. Scrounging through the 300 line `qla2x00_nvram_config()` function suggests full duplex mode is intentionally disabled:

```
  ...
/*
 * Setup driver firmware options.
 */
icb->firmware_options.enable_full_duplex = 0;
icb->firmware_options.enable_target_mode = 0;
  ...
```

Setting `enable_full_duplex` to 1 did not help.

10.2 Dual Port

I tried the same `sgp_dd` workload on both ports. Unfortunately, the 7 disks in the DS2405 I was loaned were ST336605FC (10k RPM) and not ST336753FC (15K RPM). This meant I had to compensate by adjusting the amount of data written to various disks again.

The bottom line is varying Read Current and Relaxed Ordering hints didn't matter for this workload. The outbound `sgp_dd` tasks managed 370 MB/s consistently.[8]

10.3 Summary of Lessons learned

The quote about the journey being more important than the destination comes to mind. Several things learned on this journey:

1. Firmware teams will compensate for stupid OSs. In this case performance

[7] Given the 3:1 bias of inbound:outbound throughput, I tried 6:7 (inbound:outbound) and 5:8 disks—yielded basically the same results.

[8] I can't help but wonder if I've got some piece of the puzzle wrong. But I've reviewed everything several times and if something is wrong, it's not obvious to me. I'll update the paper if I learn otherwise.

gains aren't what I expected because firmware was already setting aggressive cache line prefetching. On fully loaded systems performance could be worse . . . but haven't measured that yet. However, Firmware couldn't use *unsafe hints* (e.g. Relaxed Ordering).

2. IO Card Vendors will compensate for stupid chip sets. It didn't initially occur to me high performance IO cards would buffer IO in order to compensate. But the tradeoff is latency.

3. PCI-X is a different bus protocol compared to PCI, not just a speedup. The differences in bus protocol obsoleted the key thing I was hoping to measure (cache line prefetching for DMA Reads).

4. Don't start by testing an adaptive driver. An adaptive driver will adjust its operating parameters after a period of time to optimize for the given workload. I wasted time trying to figure out why my tg3 performance measurements varied in unpredictable ways. Adding "sleep 30" between scripted test runs helped solve that problem.

5. The major weakness of this paper is methodology. I didn't know what I was measuring until I started investigating why I didn't get expected results. I need a PCI/PCI-X logic analyzer which can accurately measure the bus utilization. I believe HP has several such analyzers on site; they just won't fit in the RX2600. I would have to chop open the sheet metal so IO cards could stick out. I'm not willing to do that because airflow would be, uhm, dramatically altered.

10.4 Future Work

Several things come to mind that are still outstanding:

1. **Test fully loaded systems** The busier the memory controller is, the higher the latency memory fetches will be (2x-4x). We don't want to waste memory bandwidth (prefetching too much) or IO bandwidth (prefetch too little). Just enough to compensate for average latency.

2. **PARISC** implementation only supports PCI. Memory controller latencies are slower as is the IO MMU. It should benefit more from DMA Hints than IA64 does. I will update this paper (and remove this "Future work" item) with PARISC results when I have them.

3. **PCI-X DMA Hints** Not as much to do here but still worth exploring. Understand how different chip sets implement PCI-X DMA support.

4. **PCI/PCI-X Logic Analyzer** Perhaps in the future I can get access to an RX5670 with logic analyzer card installed and rerun the tests. Logistically it's non-trivial since RX5670 is not a machine I can walk around with under my arm.

5. **More 2Gb/s FC disks** would be useful. Need to figure out how to stress input and output at the same time. Maybe stripe across both controllers, two RAID0 md devices; one for reading and the other for writing.

10.5 And thanks to. . .

A fair number of people contributed to this paper. They provided support, ideas, or reviewed content. In no particular order:

Alan C. Meyer, James Bottomley, Erin Handgen, Thomas Bogendörfer, Kevin Carson, Stephane Eranian, David Mosberger, Alex Williamson, Dave Miller, Joe Cowan, Fred Worley, Mike Krause, Matthew Wilcox.

My apologies if I omitted other contributors.

References

[1] http://ftp.parisc-linux.org/docs/astro_intro.ps

[2] http://ftp.parisc-linux.org/docs/elroy_ers.ps

[3] http://iou.parisc-linux.org/ols2002/

[4] http://www.hp.com/products1/itanium/chipset/index.html

[5] http://cvs.parisc-linux.org/*checkout*/linux/Documentation/DMA-mapping.txt?rev=HEAD&content-type=text/plain

[6] http://h21007.www2.hp.com/dspp/files/unprotected/linux/zx1-mio.pdf

[7] http://h21007.www2.hp.com/dspp/files/unprotected/linux/zx1-ioa-mercury_ers.pdf

[8] http://www.pcisig.com/

[9] http://ftp.parisc-linux.org/docs/astro_runway.ps

[10] http://www.hp.com/products1/itanium/performance/architecture/lmbench.html

[11] http://lists.parisc-linux.org/pipermail/parisc-linux/2002-March/015966.html

A 2.5 Page Clustering Implementation

William Lee Irwin III
IBM Linux Technology Center
wli@holomorphy.com | wlirwin@us.ibm.com

Abstract

Page clustering is a form of "large pages" that increases the kernel's minimum allocation unit for physical memory (base page size). There are several good reasons to do this. One is a form of prefaulting accomplished by instantiating groups of PTE's mapping a given base page. Another is a constant factor reduction of the number of objects the kernel must traverse in order to manipulate a given collection of pages. The increase in `PAGE_SIZE` also implies an increase in `PAGE_CACHE_SIZE`, which enables the use of filesystems with larger blocksizes. Last, but not least, the constant factor reduction of lowmem consumed by `mem_map` is crucial for the performance of 64GB i386 machines.

Page clustering has a number of technical challenges involved in a 2.5 counterpart of the 2.4.7 implementation. First, highpte poses unusual difficulties, as neither sub-`PAGE_SIZE` highmem allocations nor sub-`PAGE_SIZE` kmapping were supported in the original implementation. Rmap also poses challenges, as it makes direct assumptions about PTE's being of size `PAGE_SIZE`. Finally, arch code above all makes many assumptions about `PAGE_SIZE`'s relationship to the area mapped by PTE's, particularly in arch support and VM initialization code.

In summary, the author will describe the problems that arose during his implementation of page clustering for 2.5 along with their solutions, for an audience of kernel programmers.

1 What is page clustering?

1.1 Background

Memory present in a system is described by physical addresses. Most (if not all) modern machines are byte addressable, but the MMU usually operates at a lower "resolution," and its finest resolution is what page clustering refers to as `MMUPAGE_SIZE`. When the MMU's translations are set up, be they in hardware-interpreted data structures or in software-programmable TLB's, they refer to "page frames" of that size or larger, which effectively are a unit of measurement for memory. Similarly one may refer to virtual memory in those units, and arbitrary relationships between virtual page frames and physical page frames are constructible with a combination of hardware and software translation tables.

Demand-paged virtual memory systems, when a task takes a TLB miss not resolvable via the kernel's software translation tables (which may be interpreted directly by hardware) are then faced with the task of finding a physical page frame to back a virtual page frame with.

Without page clustering, the kernel maintains a data structure, the `struct page`, representing each physical page frame, and another, the page table entry, to represent each virtual page frame. When not constrained by hardware, the

kernel is free to make ridiculous choices of structures for the page tables. For instance, each virtual page frame could (in principle) be represented by a node in a binary search tree or a linked list. Linux® uses radix trees as mandated by hardware on i386 on all architectures, which are somewhat more efficient than various choices, though some architectures natively use other structures such as inverted page tables for them.

The page tables are assisted by a binary search tree of virtual extents representing either extents of files or zero-filled regions, whose nodes are called `vma`'s. The physical page is chosen so as to arrange virtual contiguity of file pages in tandem with file offset contiguity, or otherwise to fetch largely arbitrary pages and zero them out before mapping them. When the relationship of a pagetable entry to a physical page frame and its corresponding `struct page` data structure, is restricted to within a single `vma` it is a 1:1 relationship.

A system for tracking memory in use and not in use is built around this relationship, and so the `MMUPAGE_SIZE` became the allocation unit for memory. `PAGE_SIZE` is used to simultaneously refer to the notion of the MMU's finest granularity and the memory allocator's finest granularity in preexisting Linux ® code.

1.2 How page clustering differs

First, one should observe that if the MMU's finest granularity is `MMUPAGE_SIZE`, one may simulate an MMU with a granularity of any power of two multiple (`PAGE_MMUCOUNT`) by simply instantiating `PAGE_MMUCOUNT` PTE's at a time, and making each `struct page` refer to `PAGE_MMUCOUNT` contiguous and aligned native page frames. Also, if the pagetables are not constrained by hardware, one can easily alter their structure to only have one PTE for each `PAGE_SIZE` instead of `MMUPAGE_SIZE` and by so doing reduce their space consumption. Additionally, if the MMU supports translations of size `PAGE_SIZE` one can simply perform one TLB insertion (or PTE instantiation if hardware-interpreted) for each `PAGE_SIZE` area mapped by the pagetables.

One could say this is a "weak form" of page clustering. It has the undesirable side-effect of breaking binary compatibility and hence not being transparent, but has several advantages. The port of Linux ® to the IA64 processor already uses this low code impact technique for performance reasons, as it reduces TLB misses and the overhead of manipulating large collections of pages by a factor of `PAGE_MMUCOUNT`. Some performance benefits for I/O are possible, as physical contiguity is better preserved so larger scatter/gather lists are possible, though this is offset by a larger cost of preparing buffers for small I/O transactions. BSD's VAX port did it this way.

The binary incompatibility inherent in the above approach makes it unsuitable for practical deployment on systems with significant preexisting userbases. For instance, ELF executables are linked in ways mandating differing protections within what could potentially be a single `PAGE_SIZE` virtual region, and mmap() is often performed at offsets that are not `PAGE_SIZE`-aligned or in lengths divisible by `PAGE_SIZE`. To address the `mmap()` granularity issue, the 1:1 relationship between virtual page frames and accounting structures for physical memory must be extended to `PAGE_MMUCOUNT`:1. There is also a very invasive audit required to enforce the newly introduced distinction between `MMUPAGE_SIZE` and `PAGE_SIZE` by programming dimensional analysis into various address and index calculations. This could be called the "strong form" of page clustering.

The solution, in high-level terms, essentially has two cases for userspace. The first, which is easier, is file-backed memory. The unit of memory cacheing file contents is `PAGE_CACHE_SIZE`, which (for 2.5) is identical to `PAGE_SIZE`. An index in units of `PAGE_SIZE` is usable for recovering the struct page representing the area of the file that would need to be faulted in. However, to preserve mapping semantics one must also recover an offset into the area represented by the `struct page` in units of `MMUPAGE_SIZE`. The second case is anonymous memory, which is not forced to be simultaneously virtually and physically contiguous by virtue of its contents. Userspace demands one `MMUPAGE_SIZE` unit of memory but receives `PAGE_SIZE` unit of memory, and so to prevent very noticeable amounts of waste, one scans nearby PTE's for other virtual pages anonymizing faults, that is, write faults on COW file pages or on the zero page, could be taken on. These are candidate pages for copying (the zero page is special cased to use faster zeroing algorithms on most architectures). The anonymizing case results in a complex relationship between the virtual pages in a process and the anonymous page.

In summary, page clustering divorces the kernel's internal allocation unit, or the size of an area represented by a struct page, from the notion of the MMU's mapping granularity with the constraint that the allocation unit be larger.

1.3 Why page clustering?

Page clustering introduces several advantages. The first is that by using a larger unit for cacheing file pages, one can support filesystems with larger block sizes. The second is that the additional physical contiguity introduced by the larger allocation unit allows one to construct larger scatter gather lists for I/O (again with the proviso about preparing write buffers). The third is that the number of objects in various collections of pages is reduced for a linear speedup of the algorithms. The fourth is that the page faults may be batched, reducing the page fault rate.

The fifth, which is the primary reason why this project to resurrect the 2.4.7 page clustering patch was carried out, is largely specific to i386 PAE, though possibly also applicable to 32-bit kernels running on large memory 64-bit machines. `sizeof(struct page)/PAGE_SIZE` is the constant of proportionality for the fraction of memory consumed by the `struct page`'s required to account for all the physical memory in the system. On 32-bit systems with extended addressing or when the kernel runs in 32-bit mode, this is irrespective of virtualspace and the total memory consumed may be larger than kernel virtualspace. For instance, with a fully-populated 40-bit physical address space, a 32-bit virtualspace, a 4KB `PAGE_SIZE`, and a 64B `sizeof(struct page)`, the coremap is 16GB in size, which is infeasible to simultaneously map. Page clustering reduces this space overhead by a factor of `PAGE_MMUCOUNT`, which is arbitrary (within the constraints of the quality of implementation), and so renders the coremap's space overhead $O(1)$ with respect to physical memory.

2 Implementation

2.1 Early boot

The issues encountered in early boot were largely simple, but widespread. Early boot debugging was done on a 16 processor NUMA-Q ® with 16GB RAM. First, pagetables and various fragments of memory that were formerly assumed to be 4KB but described with `PAGE_SIZE` needed to be updated, including mappings for the IO-APIC, numerous pagetables, and structures like the idle threads' stacks. Then memory detection required various kinds

of dimensional analysis to properly calculate coremap indices from page frame numbers and vice-versa.

Numerous index calculations and indexing operations into the coremap were broken. They had no counterpart in 2.4.x, but didn't require much thought to correct:

```
#define pfn_to_page(pfn)  (mem_map + (pfn))
#define page_to_pfn(page) \
    ((unsigned long)((page) − mem_map))
#define pfn_valid(pfn)    ((pfn) < max_mapnr)
```

became

```
#define pfn_to_page(pfn)    \
        (&mem_map[(pfn)/PAGE_MMUCOUNT])
#define page_to_mapnr(page) \
    ((unsigned long)((page) − mem_map))
#define page_to_pfn(page)   \
    (PAGE_MMUCOUNT∗page_to_mapnr(page))
#define pfn_valid(pfn)      \
    ((pfn) < max_mapnr∗PAGE_MMUCOUNT)
```

and so on.

Of the issues, `kmap_pte` and `pkmap_page_table` were particularly troublesome; to get booting, they were removed in favor of walking kernel pagetables, but are to be reinstated in the near future. The issue was that they were allocated 4KB at a time using the bootmem allocator, but were assumed to point at contiguous pagetables capable of mapping the entire permanent kmap and atomic kmap arenas, which had grown to where they required multiple pagetables each.

An unusual issue arose from maintaining an 8KB stack size while raising `PAGE_SIZE` to arbitrary sizes. `fork_init()` first received a divide by zero from the following code fragment:

```
/∗ create a slab on which task_structs can be allocated ∗/
task_struct_cachep =
    kmem_cache_create("task_struct",
        sizeof(struct task_struct),0,
        SLAB_MUST_HWCACHE_ALIGN, NULL, NULL);
if (!task_struct_cachep)
    panic("fork_init(): cannot create
task_struct SLAB cache");

/∗
 ∗ The default maximum number of threads is set to a safe
 ∗ value: the thread structures can take up at most half
 ∗ of memory.
 ∗/
max_threads = mempages /
(THREAD_SIZE/PAGE_SIZE) / 8;
```

This was clearly due to `THREAD_SIZE/PAGE_SIZE` vanishing. But then unusual errors arose for unclear reasons. As it turned out, I'd changed kernel stacks to be slab allocated, but, the kernel stacks were so small compared to `PAGE_SIZE` they used on-slab slab management and so failed to be 8KB-aligned. Changing the threshold to use off-slab slab management for objects larger than `MMUPAGE_SIZE` in addition to the other criteria sufficed, with zero runtime impact on the `PAGE_SIZE == MMUPAGE_SIZE` case.

Next, the placement of vmallocspace relative to fixmapspace and the overrunning of vmallocspace by fixmapspace for unusually large values of `PAGE_SIZE` became issues. This was resolved by some painful compile-time mechanics to shove the kmap and permanent kmap windows into vmallocspace, dynamically size them with respect to `PAGE_SIZE`, and make poor guesses in assembly as to the boundaries between vmallocspace and `ZONE_NORMAL`. The boundaries out to be safe because the assumptions were not truly used apart from an indirect reference to them via `MAXMEM`. Specifically:

```
#define VMALLOC_END
(FIXADDR_START−2∗MMUPAGE_SIZE)

#define __VMALLOC_START \
    (VMALLOC_END − VMALLOC_RESERVE  \
        − 2∗MMUPAGE_SIZE)
#define VMALLOC_START \
(high_memory                              \
     ?  max(__VMALLOC_START,              \
            (unsigned long)high_memory) \
     :  __VMALLOC_START                   \
)
#define __MAXMEM \
((VMALLOC_START − 2∗MMUPAGE_SIZE  \
        − __PAGE_OFFSET)          \
        & LARGE_PAGE_MASK)
#define MAXMEM \
__pa((VMALLOC_START−2∗MMUPAGE_SIZE) \
    & LARGE_PAGE_MASK)
```

This is actually a side effect of a design decision which makes the virtualspace layout change dynamically with PAGE_SIZE. That is, virtual mapping windows raise an issue now that the area callers want to map is usually PAGE_SIZE in size, and to make it the size they expect, fixmapspace must grow. There is an alternative design possible, which is to keep fixmapspace a fixed size or at most some fixed size, and to have "sliding windows into a partial page." Using such an API would proceed something like the following:

This is somewhat more invasive, but more efficient with respect to virtualspace. As PAGE_SIZE grows in a 32-bit environment with progressively more extended physical memory, such measures become progressively more prudent. However, the need to take such measures may be significantly mitigated by eliminating the permanent kmap pool in combination with using per-cpu pagetables for the kmap windows, as the typical targets needing the largest PAGE_SIZE values have a maximum of 32 or 64 cpus. Eliminating the permanent kmap pool has the additional advantage of preventing deadlocks caused by a number of tasks each attempting to acquire multiple permanent kmaps but acquiring fewer than desired by the time the pool is exhausted.

```
int k;
void ∗old, ∗new;
old = kmap_atomic_start(old_page, KM_USER0);
new = kmap_atomic_start(new_page, KM_USER1);
for (k = 0; k < PAGE_KMAP_COUNT; ++k) {
    memcpy(new, old, KMAP_SIZE);
    old = kmap_atomic(old_page, KM_USER0, k);
    new = kmap_atomic(new_page, KM_USER1, k);
}
kmap_atomic_end(old_page, old, KM_USER0);
kmap_atomic_end(new_page, new, KM_USER1);
```

2.2 coremap initialization

The coremap initialization is worthy of its own discussion. First, in order to satisfy the early boot setup code, the coremap must be laid out so it maps PAGE_SIZE units of memory to properly aligned positions in the coremap. Simultaneously, most (if not all) of the calculations are done with pfn's so various bits of dimensional analysis must be programmed in.

First, zone->zone_start_pfn and zone->spanned_pages need to be treated consistently. Then, zones_sizes[] needs to be converted to pass PAGE_SIZE units and free_area_init_core() fixed up to increment its pfn counter by PAGE_MMUCOUNT. bad_range() must then be adjusted for unit conversion before doing its bounds checks. Finally, the page allocator need not keep so many orders around to satisfy allocations of a given size, so use MAX_ORDER - PAGE_MMUSHIFT instead of MAX_ORDER.

Bootmem also needed large adjustments; they

were largely done to make its own internal accounting based on `MMUPAGE_SIZE` and then interface it with the page allocator which has a `PAGE_SIZE` granularity. This ended up being rather invasive.

2.3 Kernel pagetables

`vmalloc()` usage was too widespread to undergo a full audit for space conservation. The choice was between using `PAGE_SIZE` or `MMUPAGE_SIZE` as the unit of `vmalloc()` mapping and allocation, and I chose `MMUPAGE_SIZE`. This is transparent to userspace, so either would be legitimate, but on i386 PAE vmallocspace is too constrained to take internal fragmentation hits. This meant that instead of a full audit for space conservation, a full audit for misuse of `PAGE_SIZE` is needed. This turned out not to be very problematic at all, as few drivers needed the conversion, and those that did had mild failure modes, failing only to probe.

`page_table_range_init()` and relatives greatly disliked the change of units and the ambiguous location of `kmap_pte` and `pkmap_page_table`. It proved infeasible rapidly bring up the system while preserving them intact, so they were removed and the code greatly simplified at the expense of a very large diff.

2.4 Process pagetables

User pagetable manipulations consisted largely of straightforward substitutions in pagetable code. Of course, something was missed. It appeared that an unusual binary compatibility bug arose with respect to shared libraries that was very difficult to trigger. The cause of this was that there was only one caller of `pte_modify()` in the core VM in a corner case of `mprotect()`. This passed a first pass of inspection because `_PAGE_CHG_MASK` didn't trip grep, but it turned out to rely on `PTE_MASK`, which by virtue of the macro indirection, also slipped past grep. After over a week of chasing it, the substitution that slipped through my fingers was finally carried out.

The next "interesting" binary compatibility bug was that core dumps were corrupted. The `get_user_pages()` calling convention had become lossy. It was returning `struct page`'s to refer to the areas mapped by PTE's, and it along with `follow_page()` was the only area of the kernel exibiting this particular kind of confusion. The solution was to return pfn's and not `struct page`'s, and was highly successful. Badari Pulavarty assisted in implementing the portion relevant to direct I/O.

The most interesting bug of all was actually the first, which prevented userspace from running at all. `/sbin/init` would be stuck in a loop somewhere in userspace, and it could only very rarely be caught in the kernel. What eventually had to be done to track down the issue was to log all page faults. What was eventually discovered was that pid 1 has a special status in the kernel, and loops when taking invalid faults instead of being delivered SIGSEGV. After some poking around, it became evident they were always anonymous pagefaults.

So at first, the workaround was to fragment anonymous pages. But this could only be temporary in order to meet the performance goals. The issue was resurrected when it came time to attempt to fully utilize anonymous pages for performance reasons. It took some time to come around to examining the contents of the purportedly zeroed memory, but eventually divining the page pointed to by the PTE taking the fault, which pointed to the zero page. And the fact a nonzero address was being fetched from the zero page prompted the examination of its contents. By an unusual

coincidence the author had been implementing the GDT setup for an i386 executive earlier that day, and noticed a very clear resemblance to the contents of the supposed zero page. Very shortly thereafter it was discovered that the `empty_zero_page[]` used on i386 as backing memory for the zero page was a 4KB array followed immediately by the kernel's GDT. The bug was resolved by using a custom-allocated and zeroed page instead of the `struct page` tracking `empty_zero_page[]`.

Finally, userspace pagetables required fixups in order to prevent extremely wasteful fragmentation. The code turned out to be somewhat hairy, as it required reference counting pagetable pages and some scanning of PMD entries in an aligned `PAGE_MMUCOUNT`-sized group. Furthermore, in order to interoperate with highpte, significantly more complex definitions of `pte_offset_map()`, `pmd_populate()` and relatives were required.

```
#define pte_offset_map(dir, address)    \
((pte_t *)                              \
  kmap_atomic(pmd_page(*(dir)),KM_PTE0) \
 + (PTRS_PER_PTE                        \
    * ((pmd_val(*(dir))/MMUPAGE_SIZE)   \
       % PAGE_MMUCOUNT)                 \
 + pte_index(address))                  \
)
```

`pmd_populate()` became too large to paste here because it had to deal with several issues to recover from partial unmappings of the `PAGE_MMUCOUNT` PMD group and PTE page refcounting. For the wary, it collapses to its prior size when `PAGE_MMUCOUNT == 1`.

2.5 file-backed memory

Handling userspace faulting semantics for file-backed memory was actually trivial. The most unsophisticated fault handling scheme imaginable suffices.

There was a small issue with `sys_remap_file_pages()` where the populate methods used the `install_page()` API internally to perform the dirty work of walking the pagetables down to the PTE to edit, and as it referred to the location to map by the `struct page`, lost the offset into the page to map. This was trivially corrected with an additional argument with the offset.

2.6 Swap-backed memory

Swap faults are not truly worth optimizing with pagetable scanning; they don't fragment like freshly zeroed anonymous pages because the swapcache is an effective lookup structure and userspace can fetch things just fine. Instead they are faulted in one by one, and that simplified things at least temporarily while the scanning code wasn't in place.

The organization of the swap map differs from the 2.4.7 patch, which created a swap map entry for each `MMUPAGE_SIZE` piece of a page, and so had to account for reference counts on the page held by multiple swap entries. The 2.5 page clustering implementation instead uses a single swap map entry for every `PAGE_SIZE`-sized page, and so simplified swap reference count semantics, reduces the vmalloc-space consumption of the swap map by a factor of `PAGE_MMUCOUNT`, and reduces the search space for swapoff. Some differences there are also visible with the encoding of `swp_entry_t`'s, which directly play with swap map indices and offsets into pages in various points throughout the core VM where beforehand they didn't need to..

2.7 anonymizing faults

There is a problem to solve caused by the fact that a process faulting on anonymous memory requests `MMUPAGE_SIZE` bytes of memory but is granted `PAGE_SIZE` bytes of memory. Again, there is more than one way to deal with this.

The first, not used here, is to maintain a one `PAGE_SIZE` area as a "ready list" and service anonymous faults until it's exhausted.

The second is to speculatively prefault neighboring anonymous pages in order to utilize the entire anonymous page. Scanning neighboring PTE's for zero-mapped or COW pages (i.e. to be anonymized). This has the potential to reduce the fault rate for some loads at the cost of not guaranteeing full utilization. Initial indications appear to be that even heuristics that appear relatively weak in comparison to those of the 2.4 patch suffice.

The logic is relatively complex, and some additional complexity as compared to the 2.4.x code was added by simultaneously scanning PTE's both upward and downward. Some additional code is required to cross `vma` boundaries and detect whether a given page is anonymous or COW. Crossing pagetable page boundaries was not implemented, for the basic reason that `PAGE_MMUCOUNT*PMD_SIZE` is enough virtualspace to scan to mitigate most of the fragmentation, and also to remain future-compatible with pagetable sharing, which is somewhat adverse to crossing pagetable pages. Additionally, totally unbounded scanning could result in some overhead.

When the scanning code is done, what it has done is assembled a vector of pfn's for all the mmupages it has to copy, and they are by no means contiguous. In order not to be grossly TLB-inefficient, an interface is provided to map vectors of pfn's, `kmap_atomic_sg()`. The use of it is obvious, as it maps each component of the pfn vector to a virtually contiguous `PAGE_SIZE` virtual area in its corresponding piece of the virtual page, and the only non-straight-line code in copying is checking for the zero page.

2.8 I/O

I/O by and large had relatively simple issues. The i386 PCI DMA API had some address calculations in need of minor substitutions, and the block layer was largely immune to the whole affair apart from direct I/O and SCSI ioctl's using `get_user_pages()`. An unfortunate limitation exists in that the block layer is incapable of dealing with `512*q->max_sectors < PAGE_SIZE`. I didn't produce a fix for this, as it's a somewhat obscure condition that can only occur when particularly crippled devices meet particularly large value of `PAGE_SIZE`. I feel that it should eventually be handled as part of the implementation.

IDE had a small issue in that its PRD tables were sized in terms of `PAGE_SIZE`, which it appears to expect not to vary from 4KB. AGP also had an unusual issue involving mapping its aperture. But most drivers that failed simply performed a `vmalloc()` or `ioremap()` of an area sized proportionally to `PAGE_SIZE` during initialization and failed to probe, which was harmless apart from failing to provide functionality (i.e. no data corruption) and very easy to correct. The starfire ethernet adapter fell in this category.

3 Trademarks

This work represents the view of the author and does not necessarily represent the view of IBM.

IBM is a trademark of International Business Ma-

chines Corporation.

Linux® is a trademark of Linus Torvalds.

Other company, product or service names may be trademarks or service marks of others.

Ugly Ducklings

Resurrecting unmaintained code

Dave Jones
SuSE Labs
davej@suse.de

Abstract

Throughout the development of the 2.5 kernel, a number of drivers and pieces of infrastructure that had been left to stagnate finally got a long overdue cleanup. In some cases, code that hadn't been touched for several years got overhauled. Each time another area got the cleanup treatment, patterns started to emerge.

This paper attempts to document some of these patterns so that hopefully by keeping them in mind, future driver authors don't fall into the pitfalls that some of these have fixed up such as over-abstracting, and massive duplication. By way of examples, it covers several areas that got cleaned up in the 2.5 series, but focuses on the bulk of the work the paper author did on the agpgart driver.

1 Introduction

It has been a long-standing philosophy that bending an existing driver to work on a new piece of hardware is much favoured over a new implementation which ends up with 99% the same code as the old. The typical life-cycle of a driver is as follows.

- Driver is written for hardware vendors A's new widget
- Vendor B makes a compatible widget.
- Id's for Vendor B's product get added to the driver
- (Repeat for several other vendors/other register compatible widgets)
- Slightly different widgets start appearing, which are still mostly compatible. Driver starts to take on new form where it needs to special case certain widgets in different code paths.
- Repeat for several more new widgets.
- Driver is now 100K+ of spaghetti.
- Original driver maintainer moves on to new project, leaving driver in current state.
- New widget Id's get added.
- Most people too scared to change too much of the code in fear of subtly breaking support for other widgets.

2 Cleanups overview

Throughout the development of the 2.5 kernel, a number of drivers and pieces of infrastructure that had been left to stagnate finally got a long overdue cleanup. Each time another area got the cleanup treatment, patterns started to emerge.

2.1 The splitting up of multiple instances

The "support all hardware all in the same driver `.c` file" approach is flawed. If you want to change how vendor B's products work, you shouldn't be touching any code for other vendors devices. With hundreds or thousands of lines of irrelevant code, it's also a pain to navigate your way around the source. By splitting the driver into multiple vendor `.c` files, you also start to notice patterns such as "this function is duplicated in all vendor files, so belongs in a generic `.c` file." Sometimes, however, things go the other way. In 2.4, we have several separate RNG (Random Number Generator) drivers. Jeff Garzik found that by merging all of these to the same file, lots of code duplication got removed. Whilst the number of RNG drivers is quite low, if there comes a day when the driver supports many more, it may make sense to abstract them back out into separate files again.

2.2 Reorganising directory structures

The whole idea of directories is to keep similar things together. One simple cleanup that happened in 2.5 was the introduction of the `drivers/char/watchdog` directory. Previously, `drivers/char` contained over 200 `.c` and `.h` files. By introducing the watchdog subdirectory, you can instantly find all relevant drivers. Useful when you have to make changes that affect all watchdog drivers (as was the case in 2.4 when a security bug had been copied through all drivers).

2.3 Simplification of abstraction layers

Sometimes, after introducing support for multiple widgets to a driver, people over-abstract. A prime example of this was the agpgart cache flush routine, which ended up calling through 4–5 function pointers before it actually got to do anything useful.

2.4 Moving to new APIs to decrease LOC

With code lying dormant and unmaintained for several years, it tends to miss the opportunity to take advantage of easier or faster ways of calling kernel-supplied functionality. As helper functions get continually added, the number of lines of code needed to be duplicated in drivers goes down.

3 Case study #1: IA32 CPU setup routines

This started out life as `arch/i386/kernel/setup.c`, and in 2.4, currently stands at 84KB of code which handles setting up of the CPU in terms of working around errata, enabling CPU specific features, and doing some detective work such as finding out the cache sizes. Initially supporting Intel CPUs, the clones started to follow. Today it supports dozens of different types of x86 CPU, from 10 different vendors. In 2.5, Patrick Mochel split this file up into per-vendor support files, and a few generic files. `arch/i386/kernel/cpu/` is a much simpler place to navigate, and is a lot nicer to hack on than its predecessor as a result.

4 Case study #2: IA32 MTRR driver

This monster has been around a few years, and it shows. 71KB of monolithic code, with multiple implementations built on top of each other. Each time, with the abstraction layer bent and twisted into something new. Initially supporting generic Intel MTRRs, it was bent into shape to deal with AMD K6's variants. Then Cyrix's ARR's. Then a myriad

of other clones which did things slightly different. Again, chopping this into per-vendor pieces makes things a lot simpler, and reduces the chance of breaking one vendor when fixing something for another (which has been the case in the past on more than one occasion).

5 Case study #3: Bluesmoke

Bluesmoke is the IA32 machine check exception handling support. As usual, it first only supported Intel P5 and P6 CPUs. Over time, things were changed to support AMD processors, Intel Pentium 4, IDT Winchips, and some additional features such as background checking. This all started to blow up the file size, and it became a pain to find your way around a file with a half dozen similarly named functions. Time for the split-up treatment. 2.5 now has 7 separate C files for the implementations, with a central 'generic' file which calls the specific per-vendor/model implementations.

Whilst it's theoretically possible that you could hack the Makefiles now to only build in (for example) the Intel code if you don't own any non-Intel parts, the added complexity and reduction in functionality for the net-gain of just a few KB of object wasn't deemed worth it. A bigger challenge which would benefit from this change came in the form of the final case study.

6 Case study #4: AGPGART

6.1 History of AGPGART

AGP support was added to Linux back in 1999. Subsequent updates were somewhat infrequent. The bulk of the code never really changed much. Each update just added PCI ID's of new devices, or occasionally a new agpgart implementation when things were just too different to the existing agpgarts.

6.2 How I got involved

During 2002, I was asked by SuSE to implement AGP support for the AMD x86-64. Thinking this would be easy basing assumptions on what I'd previously seen happen to agpgart (thinking it would be just adding some new PCI idents or the likes), I (foolishly?) agreed to do it. Shortly afterwards, I discovered the GART I was writing support for was unlike anything Linux currently supported. Firstly, the north-bridge was on-CPU, which meant on an SMP box, there would be more than one of them, and they would have to be kept coherent with each other. Secondly, it was the first GART to support version 3.0 of the AGP standard. Whilst this is backwards compatible for the most part, there are some additional features that need to be taken care of (such as the transfer speed selector working in completely different ways to how it did in previous versions of the standard). This was quite a lot to take on board, so I started staring at the 134KB of agpgart back-end code (there's also 25KB of front-end code).

6.3 More problems...

Getting up to speed on a driver of this size, which supports over 50 different AGP chipsets, is not a task that happens overnight. Lots of those implementations are either the same, or very similar, but it still leaves around a dozen or so separate code paths. Now to find out which one is most similar to the GART I'm writing for. I eventually gave up trying to find one similar enough, and just started from scratch. My mails for "help" to the original maintainer of agpgart went to `/dev/null`, which meant I had to figure out how a lot of it worked, the hard way. After finally getting things working, I had decided that enough was enough, and for 2.5, I was going to give this code a major overhaul.

6.4 How things were cleaned up

- As usual, first things first, split `drivers/char/agp/agpgart_be.c` (134KB) into lots of smaller source files. One per chipset vendor. This was an instant cleanup, which had no problems being merged. Shortly afterwards, Greg Kroah-Hartman converted the chipset probing routines over partially to some of the 'new' PCI API, killing off a bunch more useless, ugly code.

- With everything in per-vendor files now, things were a lot cleaner, but there was still some real bad mess that needed cleaning. The `agpgart_be.c` file still existed, which acted as a generic part which had all the bits to call the routines in the per-vendor files. One particular ugly that stuck out was the 350-line struct that matched known PCI IDs to init routines. The redundancy in this struct was really bad.

```
static struct {
  unsigned short device_id;
  unsigned short vendor_id;
  enum chipset_type chipset;
  const char *vendor_name;
  const char *chipset_name;
  int (*chipset_setup) (struct
      pci_dev *pdev);
} agp_bridge_info[]
  __initdata = {
\#ifdef CONFIG_AGP_ALI
  { PCI_DEVICE_ID_AL_M1541_0,
    PCI_VENDOR_ID_AL,
    ALI_M1541,
    "ALi",
    "M1541",
    ali_generic_setup },
... (Continue for dozens
     more entries) ...
```

  With this wasteful struct, if 20 out of those 50 entries are for Intel GARTs, we duplicate the vendor ID, vendor name string, and in a lot of cases, the setup routine too. This was cleaned up in several steps.

  - Split the structs out from `agpgart_be.c` to `$vendor.h` (I.e., move all the ALi entries to `ali.h`, AMD entries to `amd.h`, etc.)
  - Remove all duplication from each of these structs.
  - Replace the duplication with a 'header struct' containing the vendor ID, vendor name string, and a pointer to the remaining data.
  - Replace the struct in `agpgart_be.c` with a struct that points to the various split out structures in the `$vendor.h` files.

6.5 The "new" PCI API

Somewhat pleased with myself, I mailed off the changes to Linus, who told me to start again, this time using the pci_driver functionality. As GregKH had done part of the work here already, it wasn't actually that much work to bend what I had already into shape. This did, however, bring about a big change over 2.4's agpgart. With each of the per-chipset drivers now containing a pci_driver struct, they worked independently of the agpgart core as stand-alone modules. I wasn't initially happy with this, but Linus liked it, so it stayed that way. It did, however, mean a rewrite in module locking was needed, which nicely coincided with Rusty Russell rewriting how module locking worked.

6.6 Maintainership

By this point, I had completely gutted the way the agpgart backends worked. I felt I had made significant enough change to adopt the code,

and make an entry for myself in the MAINTAINERS file. Which was probably my second biggest mistake so far. Within just a few days of doing so, my mailbox was flooded with bug reports, stagnant patches, thank-you's, and insults. One thing that I hadn't anticipated was just how far-reaching this code was. Not only did I now have to follow and understand what was going on in the agpgart code, but also found myself digging further into DRI to follow its interaction with AGPGART. Subsequently, even parts of XFree86 came under scrutiny, and even FreeBSD (which interestingly did the 'separate-file-per-vendor' thing from Day One) to see just how much I could or couldn't change without breaking things too much from a userspace point of view.

6.7 Taking AGPGART forward: AGP 3.0 support

After getting on top of the various patches, and fixing the various problems the new code brought about, AGPGART had been dragged kicking and screaming into something that resembled a modern driver. Well, almost. I then moved on to start tackling the next big thing for agpgart: generic AGP 3.0 support. Matthew Tolentino from Intel had come up with a patch for Intel's AGP3.0 chipset (the I7505), and had re-implemented a bunch of code that I had written for the x86-64. After factoring out the common parts, this got to a state where things looked just fine.

6.8 Return of the previous maintainer

Just when things were beginning to go quiet (apart from additional AGP3 GARTs turning up needing implementing), Jeff Hartmann, the original maintainer of the 2.4 AGPGART, reappeared with a 130KB patch against the original 2.4 code. It offered various functionality, supporting AGP3, and cleaning up a lot of code in the process. In a lot of other ways, however, it was a huge step backwards. Splitting Jeff's huge patch into smaller pieces was a massive job. Bits of it went in, and Linus rejected a bunch of them, but there was worse to come (more diffs). At the time of writing, Jeff's outstanding diffs vs. 2.5.59 is around 380KB. A lot of this is unlikely to be merged before 2.6 without considerable rewriting.

6.9 Useless abstractions

Furthering the cleanup mantra, agpgart code has been described in many ways by many people (including *shit* by Linus himself). Pre-cleanup, however, my pet-name for this monster was "abstraction hell." As an excellent HOWNOTTO in abstraction, here's how agpgart used to flush the cache.

- At strategic parts of the code there are `CACHE_FLUSH();` calls.
- `CACHE_FLUSH` turns out to be a macro which expands to `agp_bridge.cache_flush`
- `agp_bridge.cache_flush` in 99% of cases, points to `global_cache_flush`. The remaining case could have been special-cased in `global_cache_flush`.
- On SMP, `global_cache_flush` is a define for `smp_flush_cache`. On UP, it's a define for `flush_cache`.
- `smp_flush_cache` just does an `smp_call_function` on `flush_cache`.
- Finally, `flush_cache` does a "wbinvd" on IA32/X86_64, "mb" on IA64, or `#error`s on anything else.

7 Future directions

7.1 AGPGART

There is still a lot of work to be done on AGPGART. All the work so far concentrated on the back end (which is where all the chipset magic happens).

- The front-end of the driver (ioctl interface, etc.) is almost as crufty, and needs a lot of work to rid it of silly things like open-coded list handling routines instead of using the generic `list.h` routines. (Yet more proof that duplicating functionality is a bad thing: it gets its double-linked list implementation horribly wrong.)
- More work on making the AGP3.0 support transparent.
- Inevitably more support for additional chipsets.
- Multiple AGP bridge support.
- `sysfs` migration to get away from the horrible ioctl interface. This will unfortunately make the Linux AGPGART completely incompatible with the FreeBSD implementation. The only people this causes concern for are XFree86 developers, who have to support an additional interface.

7.2 Other kernel work

- APIC drivers. The IA32 APIC code is quite horrible, and quite fragile. It supports a lot of different types of setup, from lots of different generic PCs, to the weird and wonderful bigger machines like NUMA-Q, Summit, and more. The x86 sub-architecture support cleaned up some of this by introducing the possibility for each sub-arch to implement their own APIC code, but it hasn't really improved readability or maintainability of the APIC code to any great length.
- Watchdogs. Small scale cleanup occurred already in 2.5, which was to just group all the watchdog drivers from `drivers/char` into a new subdirectory called imaginatively `watchdog/`. A lot of these drivers are duplicating lots of code, sometimes subtly differently, when they should be using the same code. For 2.7 a nice cleanup would be to abstract out the generic parts of this to a layer above the watchdog drivers in a similar way to what happened with AGPGART. In 2.4 there was a security hole which meant every single watchdog driver needed to be audited and fixed. By moving all this functionality out of the drivers, this could have been fixed in a single place.

8 Summary

- Split out multiple implementations to their own files unless they are small and/or similar enough to the existing implementation.
- Don't re-implement code unnecessarily, even if you think you may need something extra that the generic code doesn't give you. Build on top of the generic code rather than re-implementing.
- Use modern interfaces where possible. This isn't always easy if you want your driver to compile on earlier kernel versions as well (especially true for out-of-tree drivers).
- Before abstracting something out, think about why you actually need it abstracted. What will the callers of the abstraction do in the common case?

- Directories are there to keep similar things together. Use them. (Obviously, only when they make sense; a directory for 2–3 drivers is perhaps going too far).

- Don't disappear for four years and reappear with a 380KB patch against the last code you maintained. You may find that a lot has changed whilst you were gone, and merging will be a *nightmare*—especially if you didn't keep individual per-change changesets.

udev – A Userspace Implementation of devfs

Greg Kroah-Hartman[*]
IBM Corp.
Linux Technology Center
greg@kroah.com, gregkh@us.ibm.com

Abstract

Starting with the 2.5 kernel, all physical and virtual devices in a system are visible to userspace in a hierarchal fashion through sysfs. /sbin/hotplug provides a notification to userspace when any device is added or removed from the system. Using these two features, a userspace implementation of a dynamic /dev is now possible that can provide a very flexible device naming policy.

This paper will discuss udev, a program that replaces the functionality of devfs (only providing /dev entries for devices that are in the system at any moment in time), and allows for features that were previously not able to be done through devfs alone, such as:

- Persistent naming for devices when they move around the device tree.
- Notification of external systems of device changes.
- A flexible device naming scheme.
- Allow the kernel to use dynamic major and minor numbers
- Move all naming policy out of the kernel.

This paper will describe why such a userspace program is superior to a kernel based devfs, and detail the design decisions that went into its creation. The paper will also describe how udev works, how to write plugins that extend the functionality of it (different naming schemes, etc.), and different trade offs that were made in order to provide a working system.

1 Introduction

The /dev directory on a Linux machine is where all of the device files for the system should be located.[2] A device file is how a user program can access a specific hardware device or function. For example, the device file /dev/hda is traditionally used to represent the first IDE drive in the system. The name hda corresponds to both a major and a minor number, which is used by the kernel to determine what hardware device to talk to. Currently a very wide range of names that match up to different major and minor numbers have been defined.

All major and minor numbers are assigned a name that matches up with a type of device. This allocation is done by The Linux Assigned Names And Numbers Authority (LANANA)[4] and the current device list can be always be found on their web site at http://www.lanana.org/docs/device-list/devices.txt

[*]This work represents the view of the author and does not necessarily represent the view of IBM.

As Linux gains support for new kinds of devices, they need to be assigned a major and minor number range in order for the user to be able to access them through the `/dev` directory (one alternative to this is to provide access through a filesystem [3]). In the kernel versions 2.4 and earlier, the valid range of major numbers was 1-255 and minor numbers was 1-255. Because of this limited range, a freeze was placed on allocating new major and minor numbers during the 2.3 development cycle. This freeze has since been lifted, and the 2.6 kernel should see an increase in the range of major and minor numbers available for use.

2 Problems with current scheme

2.1 What `/dev` entry is which device

When the kernel finds a new piece of hardware, it typically assigns the next major/minor pair for that kind of hardware to the device. So, on boot, the first USB printer found would be assigned the major number 180 and minor number 0 which is referenced in `/dev` as `/dev/usb/lp0`. The second USB printer would be assigned major number 180 and minor number 1 which is referenced in `/dev` as `/dev/usb/lp1`. If the user rearranges the USB topology, perhaps adding a USB hub in order to support more USB devices in the system, the USB probing order of the printers might change the next time the computer is booted, reversing the assignment of the different minor number to the two printers.

This same situation holds true for almost any kind of device that can be removed or added while the computer is powered up. With the advent of PCI hotplug enabled systems, and hot-pluggable busses like IEEE1394, USB, and CardBus, almost all devices have this problem.

With the advent of the `sysfs` filesystem in the 2.5 kernel, the problem of determining which device minor is assigned to which physical device is now much easier to determine. For a system with two different USB printers plugged into it, the `sysfs` `/sys/class/usb` directory tree can look like Figure 1. Within the individual USB device directories pointed to by the `lp0/device` and `lp1/device` symbolic links, a lot of USB specific information can be determined, such as the manufacturer of the device, and the (hopefully unique) serial number.

As can be seen by the serial files in Figure 1, the `/dev/usb/lp0` device file is associated with the USB printer with serial number `HXOLL0012202323480`, and the `/dev/usb/lp1` device file is associated with the USB printer with serial number `W09090207101241330`.

If these printers are moved around, by placing them both behind a USB hub, they might get renamed, as they are probed in a different order on startup.

In Figure 2, `/dev/usb/lp0` is assigned to the USB printer with the serial number `W09090207101241330` due to this different probing order.

`sysfs` now enables a user to determine which device has been assigned by the kernel to which device file. This is a very powerful association that has not been previously easily available. However, a user generally does not care that `/dev/usb/lp0` and `/dev/usb/lp1` are now reversed and should be changed in some configuration file somewhere, they just want to always be able to print to the proper printer, no matter where it is in the USB device tree.

```
/sys/class/usb/
|-- lp0
|   |-- dev
|   |-- device -> ../../../devices/pci0/00:09.0/usb1/1-1/1-1:0
|   `-- driver -> ../../../bus/usb/drivers/usblp
`-- lp1
    |-- dev
    |-- device -> ../../../devices/pci0/00:0d.0/usb3/3-1/3-1:0
    `-- driver -> ../../../bus/usb/drivers/usblp

$ cat /sys/class/usb/lp0/device/serial
HXOLL0012202323480
$ cat /sys/class/usb/lp1/device/serial
W09090207101241330
```

Figure 1: Two USB printers plugged into different USB busses

```
$ tree /sys/class/usb/
/sys/class/usb/
|-- lp0
|   |-- dev
|   |-- device -> ../../../devices/pci0/00:09.0/usb1/1-1/1-1.1/1-1.1:0
|   `-- driver -> ../../../bus/usb/drivers/usblp
`-- lp1
    |-- dev
    |-- device -> ../../../devices/pci0/00:09.0/usb1/1-1/1-1.4/1-1.4:0
    `-- driver -> ../../../bus/usb/drivers/usblp

$ cat /sys/class/usb/lp0/device/serial
W09090207101241330
$ cat /sys/class/usb/lp1/device/serial
HXOLL0012202323480
```

Figure 2: Same USB printers plugged into a USB hub

2.2 Not enough numbers

The current range of allowed major and minor numbers is 8 bits (0-255). Currently there are very few major numbers left for new character devices, and about half the number of major numbers available for block devices (block and character devices can use the same numbers, but the kernel treats them separately, giving the whole range for both types of devices.) This seems like a lot of free numbers, but there are users who want to use a very large number of disks all at the same time, for which the 8 bit scheme is too small. A common goal of some companies is to connect about 4,000 disks to a single system. For every disk, the kernel reserves 16 minor numbers, due to the possibility of there being up to 16 partitions on every disk. So 4,000 disks would require 64,000 different device files, needing at least 250 major num-

bers to handle all of them. This would work in the 8 bit scheme, as they all would fit, but, the majority of the current major numbers are already reserved. The disk subsystem can not just steal major numbers from other subsystems very easily. And if it did so, userspace would have to know which previously reserved major numbers are now being used by the SCSI disk subsystem.

Because of this limitation, a lot of people are pushing for an increase in the size of the major and minor number range, which would be required to be able to support more than 4,000 disks (some users talk of connecting 10,000 disks at once.) It looks like this work will go into the 2.6 kernel; however, the large problem remains of how to notify userspace which major and minor numbers are being used.

Even if the major number range is increased, the requirement of reserving major number ranges for different types of subsystems is still present. It still requires an external naming authority, and possibly we could run out of ranges sometime in the far future. If the kernel were to switch to a dynamic method of allocating major and minor numbers, for only the devices that were currently connected to the system, this authority would no longer be needed, and the range of numbers would never run out. The biggest problem with dynamic allocation is again, userspace has no idea which devices are assigned to which major and minor numbers.

A few kernel subsystems currently do allocate minor numbers dynamically. The USB to Serial subsystem has been doing this since the 2.2 kernel series, with great success. The biggest problem still is the user does not know what device is assigned to what number, and has to rely on looking in a kernel log to make that determination. With `sysfs` in the 2.5 kernel, this is much easier.

2.3 `/dev` is too big

Not all device files in the `/dev` directory of most distributions match up to a physical device that is currently connected to the computer at that time. Instead, the `/dev` directory is created when the operating system is initialized on the machine, populating the `/dev` directory with all known possible names. On a machine running Red Hat 9, the `/dev` directory holds over 18 thousand different entries. This large number of different entries soon becomes very unwieldy for users to try to determine exactly what devices are currently present.

Because of the large numbers of device files in the `/dev` directory, a number of operating systems have moved over to having the kernel itself manage the `/dev` directory, as the kernel always knows exactly what devices are present in the system. It does this by creating a ram based filesystem called `devfs`. Linux also has this option, and it has become popular over time with a number of different distributions (the Gentoo distribution being one of the more notable ones.)

2.4 `devfs`

A number of other Unix-like operating systems have solved a lot of the previously mentioned problems by using a kernel-based `devfs` filesystem. Linux also has a `devfs` filesystem, and for a number of people, this solves their immediate needs. However, the Linux-based `devfs` implementation still has a number of problems unsolved.

`devfs` does only show exactly what devices are currently in the system at any point in time, solving the "`/dev` is too big" issue. However, the names used by `devfs` are not the names that the LANANA authority has issued. Because of this, switching between a `devfs` system, and a static `/dev` system is a bit dif-

ficult, due to the number of different configuration files that need to be modified. The `devfs` authors have tried to address this problem and have provided some compatibility layers to emulate the `/dev` names.

Even with `devfs` running in compatibility mode, the Linux kernel is imposing a set naming policy on userspace. It is saying that the first IDE drive is going to be called `/dev/hda` or `/dev/ide/hd/c0b0t0u0` and there is nothing that a user can do about this. Generally, the Linux kernel developers do not like forcing any policies on userspace, when it can be helped. This naming policy should be moved out of the kernel, so that the kernel driver developers can focus not on naming arguments (of which the `devfs` naming arguments consumed many man years of time). In short, the kernel should not care what a user wants to call a device, but if `devfs` is used, this is not possible.

`devfs` also does not allow devices to be bound to major and minor numbers dynamically. The current `devfs` implementation still uses the same major and minor numbers that are assigned by LANANA. `devfs` can be modified to do dynamic allocation; however, no one has done so yet.

`devfs` also forces all of the device names and the naming database into kernel memory. Kernel memory can not be swapped out, and is always resident. For very large amounts of devices (like the previously mentioned 4,000 disks), the overhead of keeping all of the device names in kernel memory is not unsubstantial. During some testing of a wider major number range, one developer ran into memory starvation issues on a 32 bit Intel processor, just with a static `/dev` system. Add the overhead of 4,000 different disk names and structures to manage those names, and even less memory would be available for user programs to use.

3 `udev`'s goals

So, in light of all of the previously mentioned problems, the `udev` project was started. Its goals are the following:

- Run in userspace
- Create a dynamic `/dev`.
- Provide consistent device naming, if wanted.
- Provide a userspace API to access info about current system devices.

The first item, "run in userspace," is easily done by harnessing the fact that `/sbin/hotplug` generates an event for every device that is added or removed from the system, combined with the ability of `sysfs` to show all needed information about all devices.

The rest of the goals enable the `udev` project to be split into three separate subsystems:

1. `namedev` – handles all device naming
2. `libsysfs` – a standard library for accessing device information on the system.
3. `udev` – dynamic replacement for `/dev`

3.1 `namedev`

Due to the need for different naming schemes for devices, the device naming portion of udev has been moved into its own subsystem. This was done to move the naming policy decision out of the udev binary, allowing pluggable naming schemes to be developed by different groups. This device naming subsystem, namedev, presents a standard interface that udev can call to name a specific device.

With the initial releases of udev, the namedev logic is still provided in a few source files that get linked into the udev binary. There is currently only one naming scheme implemented, the one specified by LANANA[4]. This scheme is quite simple, as generally the sysfs representation of the device uses the same name, and will be suitable for the majority of current Linux users.

As the current kernel devfs provides a competing naming schema from LANANA, there has been some interest in providing a module that contains this, but this is currently unavailable due to lack of interest by the primary developers.

Part of the goal for the udev project is to provide a way for users to name devices based on a set of policies. The current version of namedev provides the user with a five step sequence for determining the name of a given device. These steps are consulted in order, and if the device's name can be determined at any step, that name is used. The existing steps are as follows:

1. label or serial number
2. bus device number
3. topology on bus
4. replace name
5. kernel name

In the first step, the device that is added to the system is checked to see if it has a unique identifier, based on that type of device. For example, on USB devices, the USB serial number is checked; for SCSI devices, the UUID is checked; for block devices, the filesystem label is checked. If this matches a identifier provided by the user (in a configuration file), the resulting name (again specified in the configuration file) is used.

The second step checks on the device's bus number. For a lot of busses, this number generally does not change over time, and all bus numbers are guaranteed to be unique at any one point in time in the system. A good example of this is PCI bus numbers, which rarely change on the majority of systems (however, BIOS upgrades, or hotplug PCI controllers, can renumber the PCI bus number the next time the machine is booted.) Again, if the bus number matches an identifier provided by the user, the resulting name is assigned to the device.

The third step checks the position of the device on the bus. For example, a USB device can be described as residing in the 3rd hub port of the hub plugged into the first port on the root hub. This topology will not change, unless the user physically moves the devices around, and is independent of any bus numbering changes that might occur between reboots of a machine. If the topology position on the bus matches the position provided by the user, the requested name is assigned to the device.

The fourth step is a simple string replacement. If the kernel name for a device matches the name specified here, the requested new name will be used in its place. This is useful for devices that users always know will have the same kernel name, but wish to name something different.

The fifth step is the catch-all step. If none of the previous steps have provided a name for this device, the default kernel name will be used for this device. For the majority of devices in a system, this is the rule that will be used, as it matches the way devices are named on a Linux system without devfs or udev.

Figure 3 shows an example namedev configuration file. This configuration file shows how the four different ways of overriding the default kernel naming scheme can be changed. The first two entries show how to specify a se-

```
# USB Epson printer to be called lp_epson
LABEL, BUS="usb", serial="HXOLL0012202323480", NAME="lp_epson"

# USB HP printer to be called lp_hp,
LABEL, BUS="usb", serial="W09090207101241330", NAME="lp_hp"

# sound card with PCI bus id 00:0b.0 to be the first sound card
NUMBER, BUS="pci", id="00:0b.0", NAME="dsp"

# sound card with PCI bus id 00:07.1 to be the second sound card
NUMBER, BUS="pci", id="00:07.1", NAME="dsp1"

# USB mouse plugged into the third port of the first hub to be
# called mouse0
TOPOLOGY, BUS="usb", place="1.3", NAME="mouse0"

# USB tablet plugged into the second port of the second hub to be
# called mouse1
TOPOLOGY, BUS="usb", place="2.2", NAME="mouse1"

# ttyUSB1 should always be called visor
REPLACE, KERNEL="ttyUSB1", NAME="visor"
```

Figure 3: Example `namedev` configuration file

rial number of a device to control what that device should be named. The third and fourth entries show how to override the bus probing order and name a device based on the specific bus id. The fifth and sixth entries show how the USB topology can be used to specify a device name, and the seventh entry shows how to do a simple name substitution.

3.2 `libsysfs`

There is a need for a common API to access device information in `sysfs` by a number of varied programs, not just the `udev` project. The device naming subsystem and the `udev` subsystem need to query a wide range of device information from a `sysfs` represented device. Instead of duplicating this logic around in different projects, splitting this logic of `sysfs` calls into a separate library that will sit on top of `sysfs` makes more sense. `sysfs` representations of different devices are not standard (PCI devices have different attributes from USB devices, etc.) so this is another reason for creating a common and standard library interface for querying device information.

Right now the current `udev` codebase is using an initial version of `libsysfs`, and the `libsysfs` codebase is under active development.

3.3 `udev`

The `udev` program will be responsible for talking to both the `namedev` and `libsysfs` libraries to accomplish the device naming policy that has been specified. The `udev` program is run whenever `/sbin/hotplug` is called by the kernel. It does this by adding a symlink to itself in the `/etc/hotplug.d/default` directory, which is searched by the `/sbin/hotplug` multiplexer script.

The `/sbin/hotplug` invocation by the ker-

nel exports a lot of device specific information on what action just happened (add or remove), what device type the action took place for (USB, PCI, etc.), and what device in the `sysfs` tree did the action. `udev` takes this information, calls `namedev` to determine the name it should give for this device (or the name that has already been given to this device if it is a remove event). If this is a new device that has been added, `udev` uses `libsysfs` to determine the major and minor number that should be used for the device file for this device, and then creates the device file in the `/dev` directory with the proper name and major/minor number. If this is a device that has been removed, then the device file in the `/dev` directory that had previously been created for this device will be removed.

4 Enhancements

There are a number of different enhancements that different users have asked for, that can be added to the existing `udev` implementation.

A lot of userspace programs want to be notified when a new device has been added or removed from the system. Gnome and KDE both want to add a new icon if a disk has been added, or possibly launch a sync program if a USB Palm device has been attached. The D-BUS project[1] has been created to help provide a simple way for applications to talk to one another using messages. It has been proposed that the `udev` program create a D-BUS message after it has created or removed a device file, so that any listening applications can act upon this event.

Currently, `namedev` uses a very simple configuration file, creating a simple ram based database that it uses to store all current device information, and device naming rules. It has been proposed that this database (if it can even really be called such a thing), be moved to a real, backing-store type database, in order to store a persistent view of the system, or to provide a more complex naming scheme for `udev` to use.

5 Thanks

The author would like to thank Daniel Stekloff of IBM who has helped shape the design of `udev` in many ways. Without his perseverance, `udev` might not even be working. He also provided the initial design documents for how `udev` could be split up into different pieces, allowing pluggable naming schemes and has been instrumental in the development of `libsysfs`. Also, without Pat Mochel's `sysfs` and driver model core, `udev` would not even have been possible to implement. The author is indebted to him for undertaking what most thought as an impossible task, and for allowing others to easily build on his common framework, allowing all users to see the "web woven by a spider on drugs"[5] that the kernel keeps track of.

6 Legal Statement

References

[1] D-BUS project. `http://www.freedesktop.org/software/dbus/`.

[2] Linux Filesystem Hierarchy Standard. `http://www.pathname.com/fhs/2.2/`.

[3] Greg Kroah-Hartman. Putting a filesystem into a device driver. In *Linux.conf.au*, Perth, Australia, January 2003.

[4] The Linux Assigned Names And Numbers Authority. `http://www.lanana.org/`.

[5] Linux Weekly News. `http://lwn.net/Articles/31185/`.

Reliable NAS from Dirt Cheap Commodity Hardware

Benjamin C.R. LaHaise
bcrl@kvack.org

Abstract

To many of us, the integrity of our data is paramount. Unfortunately, the most cost effective commodity hardware available tends to omit important features like ECC memory, or only provides it at a substantial cost premium. To address this, an approach that makes use of concepts taken from NAS and clustering, combined with a novel trick to obtain a CRC on data blocks from existing ethernet NICs is used. The resulting code is built on top of the existing Linux Network Block Device, network drivers and async io. Details ranging from design considerations to performance tuning and implementation provide an interesting story on how to attain a much easier, and cheaper, reliable storage solution.

1 Introduction

Anyone who uses computers can tell you that they will inevitably break. From software bugs, to hardware errata, or just plain old age, the components of systems will always fail at some point in their deployed lifespan. Hardware vendors try to address various subsets of these failures, but frequently only in high end systems. Even in the case of hardware specifically designed to cope with a common mode of failure, RAID controllers are lost when a failed SCSI device locks the bus.

Over the last few years, low end hardware has come to provide capabilities well beyond the needs of many server systems. Relatively cheap IDE drives cost less than a third the amount for a comperable SCSI device. As an end user, that difference in the price performance ratio raises more than a few eyebrows. Several companies sell products into this niche, yet there are a number of failure modes which their devices fail to address. Data corruption is still possible when disks make use of non-error correcting memory, or firmware bugs exist (what body of software is bug-free?). So what is a user that wishes to make use of the price point of commodity hardware to do if their data is truely valuable?

Take one kernel hacker, random bits of hardware, several discussions over beer at conferences, this problem, and mix. The result is `netmd.o`.

2 What is Netmd?

Netmd at its core is a hybrid between the Linux RAID implementation (known as md) and the network block device (known as nbd), and is quite similar to DRBD. DRBD allows users to mirror disks over a network, yet it runs under the assumption that the underlying hardware is reliable and will detect any errors which corrupt user data. Netmd changes this by making use of an end to end CRC on sectors to check if any of the BEs have unknowingly corrupted read or write data. All told, netmd is much eas-

ier to use than either drbd or nbd while providing guarantees about data integrity.

3 Design Criteria

The front end system runs the netmd kernel module. To provide the greatest likelyhood of the software being correct, simplicity is strong goal for the resulting code.

suffered from a simple single bit error, or more gross errors resulting in so called fractured blocks (ie, the case where the sector number itself is corrupt and the wrong data is read back from disk). The basic assumption netmd makes is that your front end hardware is reliable and the bulk of errors will come from the back end systems which house all of the disks. A front end system running NetMD looks very much like an NBD, but provides reliability guarantees.

The simplest configuration for NetMD (see Figure 3) consists of a front end system and three back end systems networked via a switch or hub. A minimum of three back ends are required for NetMD to establish quorum and provide the ability to hot swap one back end at a time. Additional BEs can be added to improve the performance and reliability of a NetMD setup (such as in Figure 3).

Unlike nbd, which operates over TCP, NetMD instead operates over UDP to take advantage of the packet nature of the underlying network. This allows writes to be optimized using multicast to deliver data to all BEs simultaneously. BEs may also snoop read requests to rapidly reply if the data is still available in its local cache.

In fact, the main data integrity feature of NetMD comes from the ethernet packets which are used to transmit read and write requests over the network. All ethernet frames are protected from transmission errors by a trailing 32 bit CRC. Many ethernet cards are able to record the CRC of an incoming packet. By compensating for the header of each packet, the data CRC can be extracted at low CPU cost.

Improving the Linux Test Project with Kernel Code Coverage Analysis

Paul Larson
IBM Linux Technology Center

Nigel Hinds, Rajan Ravindran, Hubertus Franke
IBM Thomas J. Watson Research Center

{plars, nhinds, rajancr, frankeh}@us.ibm.com

Abstract

Coverage analysis measures how much of the target code is run during a test and is a useful mechanism for evaluating the effectiveness of a system test or benchmark. In order to improve the quality of the Linux® kernel, we are utilizing GCOV, a test coverage program which is part of GNU CC, to show how much of the kernel code is being exercised by test suites such as the Linux Test Project. This paper will discuss the issues that make code coverage analysis a complicated task and how those issues are being addressed. We will describe tools that have been developed to facilitate analysis and how these tools can be utilized for kernel, as well as application code coverage analysis.

1 Introduction

The Linux Test Project (LTP) is a test suite for testing the Linux kernel. Over the years, the LTP has become a focal point for Linux testing and Linux test tools. Just as the Linux kernel is constantly under development, the LTP test suite must also be constantly updated, modified, and improved upon to keep up with the changes occurring in the kernel. As the Linux Test Project test suite matures, its developers constantly look for ways to improve its usefulness. Some of the considerations currently being addressed are what areas of the Linux kernel test development efforts should focus on, and proving that tests added to the LTP are testing more than what was there before. In order to gather data to support decisions for these questions, analysis should be performed to show what areas of the kernel are being executed by any given test.

The design of a process for doing this analysis took the following features into consideration:

1. Use existing tools as much as possible
2. Take a snapshot of coverage data at any time
3. Clear coverage counters before a test execution so just the coverage from the test could be isolated
4. Provide output that looks nice and is easy to read and compare
5. Show coverage percentages at any level of directory or file

6. Show execution counts for every instrumented line of every file

This paper is outlined as follows: Section 2 provides a general description of code coverage analysis. Sections 3 and 4 discuss GCOV and other code coverage tools. Section 5 describes Linux kernel modifications necessary to provide kernel code coverage. Test results are presented in Section 6. Section 7 presents the LCOV toolkit for processing and displaying GCOV results. Section 8 describes how the results of kernel code coverage analysis can be used to improve the Linux Test Project. Future work planned on this project is discussed in Section 9. Finally, conclusions are presented in Section 10.

2 How code coverage analysis works

Before examining possible methods for the task of kernel code coverage analysis, it is important to first discuss some general concepts of code coverage analysis.

Code coverage analysis shows what percentage of an application has been executed by the test process. The metrics derived from code coverage analysis can be used to measure the effectiveness of the test process [Perry].

Statement coverage analysis [Cornett] breaks the code down into basic blocks that exist between branches. By design, it is clear that if any line in a basic block is executed a given number of times, then every instruction within the basic block would have been executed the same number of times. This provides for a convenient way of showing how many times each line of code in a program has been executed.

There are, however, a number of deficiencies in this approach. The following piece of code exhibits one of the main issues with statement coverage analysis:

```
int *iptr = NULL;
if(conditional)
     iptr = &i;
*iptr = j*10;
```

The above code may show 100% code coverage but will obviously fail miserably if the conditional is ever false.

To deal with situations such as the one demonstrated above, branch coverage analysis [Marick] is required. Rather than focusing on the basic blocks and the lines of code executed, branch coverage looks at possible paths a conditional can take and how often each path through the conditional is taken. This will account for problems such as those described above because it will clearly show that the conditional was never evaluated to be false. Another advantage of branch coverage analysis is that knowledge of how many times each line of code was executed can be derived by knowing how many times each conditional was evaluated for each possible outcome.

Although branch coverage analysis provides a more accurate view of code coverage than statement coverage, it is still not perfect. To gain a better understanding of the path taken through the entire conditional, the outcome of each test in a complex conditional must be determined. For instance:

```
struct somestructure* p = NULL;

if(i == 0 || (j == 1 && p->j == 10))
    printf("got here\n");
```

If i is ever non-zero when j is 1, then the NULL pointer p will be dereferenced in the last part of the conditional. There are paths through this branch in both the positive and negative case

that will never expose the potential segmentation fault.

Branch coverage would show how many times the entire conditional was evaluated to true or false, but it would not show the outcome of each test that led to the decision.

There are many other types of coverage for dealing with a variety of corner cases. However, statement coverage and branch coverage are the two types that will be focused on in this paper since they are the only ones supported by GCOV.

3 Methods considered

This section discusses some of the more popular techniques for collecting statement and branch coverage data in a running system.

3.1 Estimation

At a most basic level, estimating techniques could be used to collect code coverage within a given program. Obviously, this method has no redeeming value at all with regards to accuracy or reproducibility. The advantage to this method, and the only reason it is mentioned, is that it requires the least amount of effort. Given a developer with sufficient knowledge of the code, a reasonable estimate of code coverage may be possible to estimate in a short amount of time.

3.2 Profiling

Performance profilers are well known, easy to use, and can be used to track coverage; however these uses are not the purpose for which performance profilers were intended. Performance profilers use statistical profiling rather than exact profiling.

Readprofile is a simple statistical profiler that stores the kernel PC value into a scaled histogram buffer on every timer tick. Readprofile profiles only the kernel image, not user space or kernel modules. It is also incapable of profiling code where interrupts are disabled.

Oprofile, another statistical profiler, leverages the hardware performance counters of the CPU to enable the profiling of a wide variety of interesting statistics which can also be used for basic time-spent profiling. Oprofile is capable of profiling hardware and software interrupt handlers, the kernel, shared libraries, and applications.

All of these statistical profilers can be used indirectly to gather coverage information, but because they approximate event distribution through periodic sampling via an interrupt, they cannot be considered accurate for true coverage analysis.

3.3 User-Mode Linux

User-mode Linux is an implementation of the Linux kernel that runs completely in user mode. Because it runs as a user mode application, the possibility exists to utilize analysis tools that were normally intended to run against applications. These tools would not normally work on kernel code, but User-mode Linux makes it possible. To make use of this feature in User-mode Linux, the kernel needs to be configured to compile the kernel with the necessary flags to turn on coverage tracking. To do this, the “Enable GCOV support” option must be selected under “Kernel Hacking” from the configuration menu.

When GCOV support is turned on for User-mode Linux, the necessary files for using GCOV are created, and coverage is tracked from boot until shutdown. However, intermediate results cannot be gathered, and counters

cannot be reset during the run. So the resulting coverage information will represent the boot and shutdown process and everything executed during the UML session. It is not possible to directly gather just the coverage results of a single test on the kernel.

3.4 GCOV-Kernel Patch

In early 2002, Hubertus Franke and Rajan Ravindran published a patch to the Linux kernel that allows the user space GCOV tool to be used on a real, running kernel. This patch meets the requirement to use existing tools. It also provides the ability to view a snapshot of coverage data at any time. One potential issue with this approach is that it requires the use of a kernel patch that must be maintained and updated for the latest Linux kernel before coverage data can be gathered for that kernel.

The GCOV-kernel patch and GCOV user mode tool were chosen to capture and analyze code coverage data. The other methods listed above were considered. However, the GCOV-kernel patch was chosen because it allowed platform specific analysis as well as the ability to easily isolate the coverage provided by a single test.

4 GCOV Coverage Tool

Having decided to use the GCOV tool for LTP, we first describe how GCOV works in user space applications and then derive what modifications need to be made to the kernel to support that functionality.

GCOV works in four phases:

1. Code instrumentation during compilation
2. Data collection during code execution
3. Data extraction at program exit time
4. Coverage analysis and presentation post-mortem (GCOV program)

In order to turn on code coverage information, GCC must be used to compile the program with the flags "`-fprofile-arcs -ftest-coverage`". When a file is compiled with GCC and code coverage is requested, the compiler instruments the code to count program arcs. The GCC compiler begins by constructing a program flow graph for each function. Optimization is performed to minimize the number of arcs that must be counted. Then a subset of program basic blocks are identified for instrumentation. With the counts from this subset GCOV can reconstruct program arcs and line execution counts. Each instrumented basic block is assigned an ID, `blockno`. GCC allocates a basic block counter vector `counts` that is indexed by the basic block ID. Within each basic block the compiler generates code to increment its related `counts`[`blockno`] entry. GCC also allocates a data-object *struct bb* `bbobj` to identify the name of the compiled file, the size of the `counts` vector and a pointer to the vector. GCC further creates a constructor function `_GLOBAL_.I.FirstfunctionnameGCOV` that invokes a global function `__bb_init_func(struct bb*)` with the `bbobj` passed as an argument. A pointer to the constructor function is placed into the ".ctors" section of the object code. The linker will collocate all constructor function pointers that exist in the various *.o files (including from other programming paradigms (e.g. C++)) into a single ".ctors" section of the final executable. The instrumentation and transformation of code is illustrated in Figure 1.

In order to relate the various source code line information with the program flow and the counter vector, GCC also generates two output files for each `sourcefile.c` compiled:

file1.c

```
→ static ulong counts[numbbs];
→ static struct bbobj =
→       { numbbs, &counts,"file1.c"};
→ static void _GLOBAL_.I.fooBarGCOV()
→       { __bb_init_func(&bbobj); }

void fooBar(void)
{
→  counts[i]++;
   <bb-i>
   if (condition) {
→     counts[j]++;
      <bb-j>
   } else {
      <bb-k>
   }
}

→ SECTION(".ctors"')
→ { &_GLOBAL_.I.fooBarGCOV }
```

"→" indicates the lines of code inserted by the compiler and <*bb-x*> denotes the x-th basic block of code in this file and italic/bold code is added by the GCC compiler. Notice that the arc from the *if* statement into bb-k can be derived by subtracting counts[j] from counts[i].

Figure 1: Code modification example

(i) `sourcefile.bb`, which contains a list of source files (including headers) and functions within those files and line numbers corresponding to basic blocks in the source file, and (ii) `sourcefile.bbg` contains a list of the program flow arcs for each function which in combination with the `*.bb` file enables GCOV to reconstruct the program flow.

The glibc (C runtime library) linked with the executable provides the glue and initialization invocation and is found in the `libgcc2.c`. More specifically: it provides the `__bb_init_func(struct bb*)` function that links an object passed as an argument to a global `bbobj` list `bb_head`. The runtime library also invokes all function pointers found in the ".ctors" section, which will result in all `bbobj`s being linked to the `bb_head` list, as part of the `_start` wrapper function.

At runtime, the counter vector entries are incremented every time an instrumented basic block is entered. At program exit time, the GCOV enabled main wrapper function walks the `bb_head` list and for each `bbobj` encountered, it creates a file `sourcefile.da` (note the source file name was stored in the structure) and populates the file with the size of the vector and the counters of the vector itself.

In the post mortem phase, the GCOV utility integrates and relates the information of the *.bbg, *.bb, and the *.da to produce the *.gcov files containing per line coverage output as shown in Figure 2. For instrumented lines that are executed at least once, GCOV prefaces the text with the total number of times the line was executed. For instrumented lines that were never executed, the text is prefaced by the string ######. For any lines of code that were not instrumented, such as comments, nothing is added to the line by GCOV.

5 GCOV Kernel Support

The GCOV tool does not work as-is in conjunction with the kernel. There are several reasons for this. Though the compilation of the kernel files also generates the `bbobj`s constructor, and the ".ctors" section, the resulting kernel has no wrapper function to call the constructors in .ctors. Second, the kernel never exits such that the *.da files can be created. One of the initial overriding requirements was to not modify the GCOV tool and the compiler.

example.c

```
       int main() {
    1    int i;

    1    printf("starting example\n");
   11    for(i=0;i<10;i++) {
   10      printf("Counter is at %d\n",i);
   10    }

         /* this is a comment */

    1    if(i==1000) {
######     printf("This line should "
                "never run\n");
######   }
    1    return 0;
    1  }
```

Figure 2: GCOV output

The first problem was solved by explicitly defining a ".ctors" section in the kernel code. The new symbol __CTOR_LIST__ is located at the beginning of this section and is made globally visible to the linker. When the kernel image is linked, __CTOR_LIST__ has the array of constructor function pointers. This entire array is delimited by the ctors_start = &__CTOR_LIST__ ; ctors_end = &__DTOR_LIST__ variables, where __DTOR_LIST__ is the starting address of the ".dtors" section, which in turn contains all destructor functions. By default the counter vectors are placed in zero-filled memory and do not need to be explicitly initialized. The data collection starts immediately with the OS boot. However, the constructors are not called at initialization time but deferred until the data is accessed (see below).

The second problem is the generation of the *.da files. This problem is solved by the introduction of a /proc/gcov file system that can be accessed from "user space" at any time. The initialization, creation, and access of the /proc/gcov filesystem is provided as a loadable module "gcov-proc" in order to minimize modifications to the kernel source code itself. When the module is loaded, the constructor array is called and the `bbobj`s are linked together. After that the `bb_head` list is traversed and a `/proc/gcov` filesystem entry is created for each `bbobj` encountered. It is accomplished as follows: the kernel source tree's basename (e.g. /usr/src/linux-2.5.xx) is stripped from the filename indicated in the `bbobj`. The program extension (e.g. *.c) of the resulting relative file path (e.g. arch/i386/kernel/mm.c) is changed to (*.da) to form the path name of the entry. Next, this pathname is traversed under the /proc/gcov filesystem and, for newly encountered levels (directory), a directory node is created. This process is accomplished by maintaining an internal tree data structure that links `bbobj`s and `proc_entries` together. Because GCOV requires all *.c, *.bb, *.bbg, and *.da files to be located in the same directory, three symbolic links for the *.{c | bb | bbg } files are created in their appropriate kernel source path. This is shown in Figure 3.

Reading from the *.da file gives access to the underlying vector of the corresponding file in the format expected by the GCOV tool. Though the data is accessed on a per file basis, a /proc/gcov/vmlinux file to dump the entire state of all vectors is provided. However, reading the full data from /proc/gcov/vmlinux would require a change of GCOV. Resetting the counter values for the entire kernel is supported through writing to the /proc/gcov/vmlinux file using `'echo "0" > /proc/gcov/vmlinux'`. Individual file counters can be reset as well by writing to its respective *.da file under the /proc/gcov filesystem.

5.1 Dynamic Module Support

Dynamic modules are an important feature that reduces the size of a running kernel by compiling many device drivers separately and only loading code into the kernel when a specific

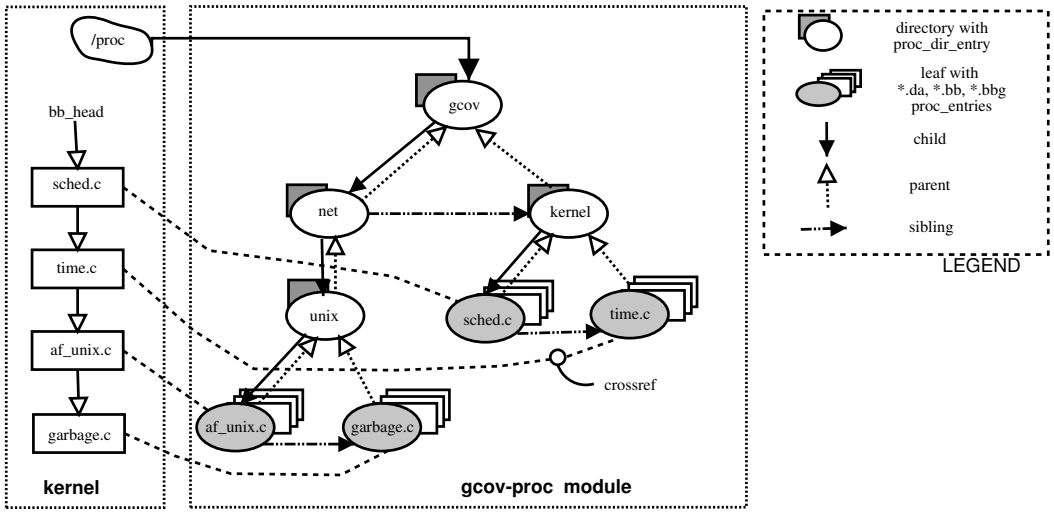


Figure 3: Data structures maintained by the gcov-proc loadable module

driver is needed. Like the kernel, the module compile process inserts the coverage code making counter collection automatic through code execution. In order to access the GCOV results through /proc/, the ".ctors" section of the module code needs to be executed during the dynamic load. As stated above, the GCOV enabled kernel locates the array of constructor routine function pointers starting from `ctors_start` to `ctors_end`. Due to the dynamic nature of a module, its constructors are not present in this section. Instead, to obtain the constructor address of the dynamic module, two more variables were added to "struct module" (ie. ctors start and end) the data structure which is passed along by the modutils to the kernel each time a module is loaded. When the dynamic module is loaded, the kernel invokes the constructor which results in the linkage of the `bbobj` to the tail of the `bb_head`. If the gcov-proc module has already been loaded, then the /proc/gcov filesystem entries are added. If the gcov-proc module is not loaded, their creation is deferred until the gcov-proc is loaded. Once when gcov-proc is loaded, the /proc/gcov filesystem is created by traversing the `bb_head` list.

Dynamic modules will create their proc file system under /proc/gcov/module/{path to the dynamic module source}. A module's .da and symbolic links *.{c | bb | bbg} will be automatically removed at the time of module unloading.

Up to Linux 2.5.48, all the section address manipulation and module loading part are performed by modutils. Accordingly we modified the modutils source to store the constructor address of the module which is loaded and pass that to the kernel as an argument. Starting with Linux 2.5.48, Rusty Russell implemented an in-kernel module loader, which allows the kernel to obtain the constructor instead of requiring modifications to the modutils.

5.2 SMP Issues

One of the first concerns with this approach was data accuracy on symmetric multiprocessors (SMPs). SMP environments introduce the possibility of update races leading to inaccurate GCOV counter data. The machine level increment operation used to update the GCOV counter is not atomic. In CISC and RISC architectures such as Intel® and PowerPC®,

memory increment is effectively implemented as load, add, & store. In SMP environments, this can result in race conditions where multiple readers will load the same original value, then each will perform the addition and store the same new value. Even if the operation is atomic on a uniprocessor, caching on SMPs can lead to unpredictable results. For example, in the case of three readers, each would read the value n, increment and store the value n+1. After all three readers were complete the value would be n+1. The correct value, after serializing the readers, should be n+3. This problem exists on SMPs and multi-threaded environments using GCOV in kernel or user space. Newer versions of GCC are experimenting with solutions to this issue for user space by duplicating the counter array for each thread. The viability of this approach is questionable in the kernel where GCOV would consume about 2 megabytes per process. We discuss this issue further in the Future Work section (10).

A test was constructed to observe the anomaly. A normally unused system call was identified and was used so the exact expected value of the coverage count would be known. Two user space threads running on different CPUs made a total of 100K calls to the `sethostname` system call. This test was performed 35 times and the coverage count was captured. On average, the coverage count for the lines in the `sethostname` kernel code was 3% less than the total number of times the system call was made.

The simplest solution was to ensure that counter addition was implemented as atomic operations across all CPUs. This approach was tested on the x86 architecture where a `LOCK` instruction prefix was available. The LOCK prefix uses the cache coherency mechanism, or locks the system bus, to ensure that the original operation is atomic system-wide. An Assembler pre-processor was constructed to search the assembly code for GCOV counter operands (LPBX2 labels). When found, the LOCK prefix would be added to these instructions. A GCC machine dependent *addition* operation was also constructed for x86 to accomplish the same task.

The results of the sethostname test on the new locking gcov kernel (2.5.50-lock-gcov) showed no difference between measured coverage and expected coverage.

5.3 Data Capture Accuracy

Another effect on data accuracy is that capturing GCOV data is not instantaneous. When we start capturing the data (first item on bb_head) it may have value $\overline{v}_{orig}$. By the time the last item on bb_head is read, the first item may now have value $\overline{v}_{new}$. Even if no other work is being performed, reminiscent of the Heisenberg effect, the act of capturing the GCOV data modifies the data itself. In Section 8 we observed that 14% of the kernel is covered just from running the data capture. So 14% coverage is the lowest result we would expect to see from running any test.

5.4 GCOV-Kernel Change Summary

In summary, the following changes were made to the linux kernel:

(a) modified head.S to include symbols which identify the start and end of the .ctors section

(b) introduced a new file kernel/gcov.c where the `__bb_init_func(void)` function is defined

(c) added two options to the configure file, (i) to enable GCOV support and (ii) enable the GCOV compilation for the entire kernel. If the latter is not desired, then it is

the responsibility of the developer to explicitly modify the Makefile to specify the file(s). For this purpose we introduced GCOV_FLAGS as a global macro.

(d) Modified kernel/module.c to deal with dynamic modules.

(e) An assembler pre-processor was written to address issues with inaccurate counts on SMP systems

Everything else is kept in the gcov-proc module.

6 GCOV patch Evaluation

This section summarizes several experiments we conducted to evaluate the GCOV-kernel implementation. Unless otherwise stated all tests were conducted on a generic hardware node with a two-way SMP, 400 MHz Pentium® II, 256 MB RAM. The Red Hat 7.1 distribution (with 2.5.50 Linux kernel) for a workstation was running the typical software (e.g. Inetd, Postfix), but remained lightly loaded. The GCC 2.96 compiler was used.

The total number of counters in all the .da files produced by GCOV was calculated for 2.5.50. 336,660 counters * 4 bytes/counter $\Rightarrow$ 1.3 Mbytes of counter space in the kernel[1]. The average number of counters per file is 718. *drivers/scsi/sym53c8xx.c* had the most lines instrumented with 4,115. And *include/linux/prefetch.h* was one of 28 files that had only one line instrumented.

6.1 LOCK Overhead

A micro-benchmark was performed to estimate the overhead of using the LOCK prefix. From two separate loops, multiple increment and locked increment instructions were called. The runtime for the loops were measured and an estimate of the average time per instruction was calculated. The increment instruction took 0.015μs, and the locked increment instruction took 0.103μs. This represents a 586% instruction runtime increase when using the LOCK prefix. At first this might seem like a frightening and unacceptable overhead. However, if the actual effect on sethostname runtime is measured, it is 19.5μs per system call with locking and 13.9μs per call without locking. This results in a more acceptable 40% increase. However, further testing is required to ensure results do not vary depending on lock contention.

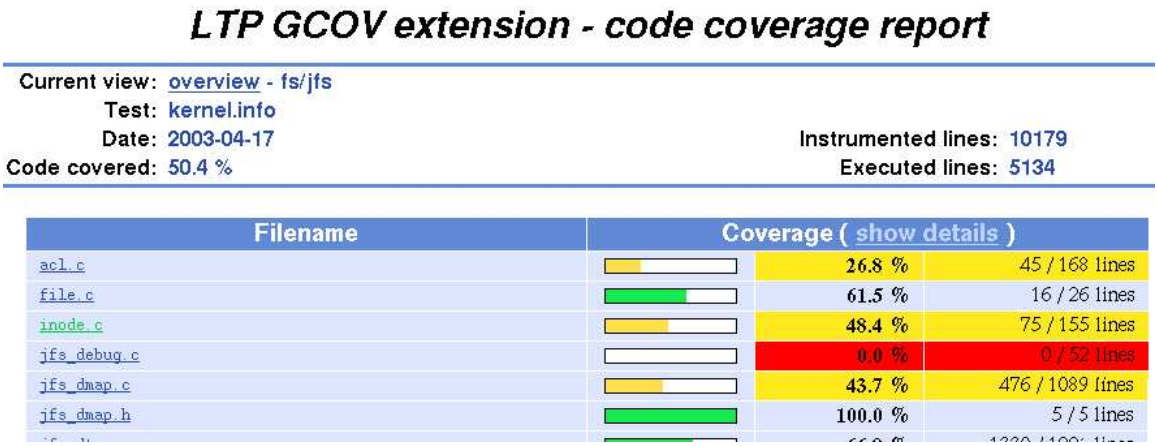
[1] in GCC 3.1 and higher counters are 8 bytes.

Test kernel	runtime (sec)
2.5.50	596.5
2.5.50-gcov	613.0
2.5.50-gcov-lock	614.0

Table 1: Results: System and User times (in seconds) for 2.5.50 kernel compiles using three test kernels.

6.2 Performance Results

Finally, a larger scale GCOV test was performed by comparing compile times for the Linux kernel. Specifically each test kernel was booted and the time required to compile (execute *make bzImage*) the Linux 2.5.50 kernel was measured. Table 1 shows runtimes (in seconds) as measured by */bin/time* for the three kernels tested. These results show a 3% overhead for using the GCOV kernel and a very small additional overhead from using the locking GCOV kernel.



LTP GCOV extension - code coverage report

Current view: overview - fs/jfs
Test: kernel.info
Date: 2003-04-17
Code covered: 50.4 %
Instrumented lines: 10179
Executed lines: 5134

Filename	Coverage (show details)		
acl.c		26.8 %	45 / 168 lines
file.c		61.5 %	16 / 26 lines
inode.c		48.4 %	75 / 155 lines
jfs_debug.c		0.0 %	0 / 52 lines
jfs_dmap.c		43.7 %	476 / 1089 lines
jfs_dmap.h		100.0 %	5 / 5 lines

Figure 4: Partial LCOV output for fs/jfs

```
                :
                : int init_module(void)
        1       : {
        1       :         int i;
                :
       11       :         for(i=0; i<10; i++) {
       10       :                 printk("This line should be executed 10 times\n");
       10       :                 printk("The value of i is %d\n",i);
                :
                :                 /* Comments are not even considered */
       10       :                 if (i == 100) {
        0       :                         printk("This line should have no coverage\n");
                :                 }
                :         }
                : }
```

Figure 5: Example LCOV output for a file

7 Using GCOV and LCOV to generate output

Once the .da files exist, GCOV is capable of analyzing the files and producing coverage output. To see the coverage data for a user space program, one may simply run 'gcov program.c' where program.c is the source file. The .da file must also exist in the same directory.

The raw output from GCOV is useful, but not very readable. In order to produce nicer looking output, a utility called LCOV was written. LCOV automates the process of extracting the coverage data using GCOV and producing HTML results based on that data [Lcovsite].

The LCOV tool first calls GCOV to generate the coverage data, then calls a script called geninfo to collect that data and produce a .info file to represent the coverage data. The .info file is a plaintext file with the following format:

```
TN: [test name]
SF: [path and filename of the \
     source file]
FN: [function start line number],\
     [function name]
DA: [line number],[execution count]
LH: [number of lines with count > 0]
LF: [number of instrumented lines]
end_of_record
```

Once the .info file has been created, the genhtml script may be used to create html output for all of the coverage data collected. The genhtml script will generate both output at a directory level as illustrated in Figure 4 and output

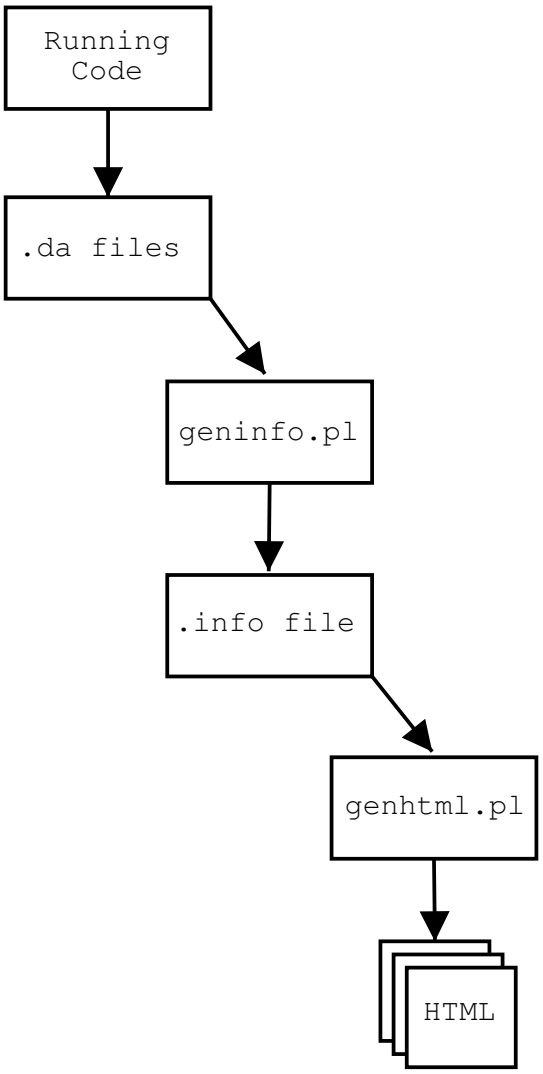


Figure 6: lcov flow

on a file level as illustrated in Figure 5. The basic process flow of the LCOV tool is illustrated in Figure 6. The commands used to generate kernel results are as follows:

1. Load the gcov-proc kernel module.
2. `lcov -zerocounters`
   This resets the counters, essentially calling 'echo "0" > /proc/gcov/vmlinux'
3. Run the test.
4. `lcov -c -o kerneltest.info`
   This produces the .info file described above.
5. `genhtml -o [outputdir] kerneltest.info`
   The final step produces HTML output based on the data collected in step 3.

The original intent of the LCOV tool was for kernel coverage analysis, so it contains some features, such as resetting the counters through /proc, that are only for use with the gcov-proc module. However, the LCOV tool can also be used for analyzing the coverage of user space applications, and producing HTML output based on the data collected. The process for using LCOV with applications is mostly the same. In order for any coverage analysis to take place, the application must be compiled with GCC using the `-fprofile-arcs` and `-ftest-coverage` flags. The application should be executed from the directory it was compiled in, and for GCOV to work on user space programs, the application must exit normally. The process for using LCOV with an application is as follows:

1. Run the program following your test procedure, and exit.
2. `lcov -directory [dirname] -c -o application.info`

3. `genhtml -o [outputdir] application.info`

LCOV can also be used to manipulate .info files. Information describing coverage data about files in a .info file may be extracted or removed. Running `lcov -extract file.info PATTERN` will show just the data in the .info file matching the given pattern. Running `lcov -remove file.info PATTERN` will show all data in the given .info file except for the records matching the pattern, thus removing that record. Data contained in .info files may also be combined by running `lcov -a file1.info -a file2.info -a file3.info... -o output.info` producing a single file (output.info) containing the data from all files passed to lcov with the `-a` option.

8 Applying results to test development

The real benefit of code coverage analysis is that it can be used to analyze and improve the coverage provided by a test suite. In this case, the test suite being improved is the Linux Test Project. 100% code coverage is necessary (but not sufficient) for complete functional testing. It is an important goal of the LTP to improve coverage by identifying untested regions of kernel code. This section provides a measure of LTP coverage and how the GCOV-kernel can improve future versions of the LTP suite.

Results were gathered by running the LTP under a GCOV enabled kernel. There are a few things that must be taken into consideration.

It is obviously not possible to cover all of the code. This is especially true for architecture and driver specific code. Careful consideration and planning must be done to create a kernel configuration that will accurately describe which kernel features are to be tested. This usually means turning on all or most things and turning off drivers that do not exist on the system under test.

ID	Test	KC	DC	TC
LTP	LTP	35.1	0.0	100.0
CP	Capture	14.4	0.3	97.7
MK	Mkbench	17.0	0.0	99.9
LM	Lmbench	22.0	0.3	98.5
SJ	SPECjAppServer	23.6	1.2	95.1
X	Xserver	24.7	1.7	93.0
XMLS	X+MK+LM+SJ	29.4	3.1	89.5
B	Boot	40.1	16.2	59.5

Table 2: Benchmarks: KC: kernel coverage; DC: delta coverage; TC: test coverage

In order to take an accurate measurement for the whole test suite, a script should be written to perform the following steps:

1. Load the gcov-proc kernel module
2. ‘echo “0” > /proc/gcov/vmlinux’ to reset the counters
3. Execute a script to run the test suite with desired options
4. Gather the results using LCOV

Using a script to do this will minimize variance caused by unnecessary user interaction. Resetting the counters is important to isolating just the coverage that was caused by running the test suite.

Alternatively, a single test program can be analyzed using the same method. This is mostly useful when validating that a new test has added coverage in an area of the kernel that was previously uncovered.

Coverage tests were performed on a IBM® Netfinity 8500R 8-way x 700 MHz SMP run-

ning the 2.5.50 Linux kernel. LTP is a test suite for validating the Linux kernel. The version of LTP used in this test was ltp-20030404. LTP hits 47,172 of the 134,396 lines of kernel code instrumented by GCOV-kernel (35%)[2]. This figure alone suggests significant room for LTP improvement. However, to get an accurate picture of LTP usefulness it is also important to make some attempt to categorize the kernel code. This way we can determine how much of the *important* kernel code is being covered by LTP. For the purposes of this work we have collected coverage information from a set of benchmarks. This set is intended to represent one reasonable set of important kernel code. The code is important because it is used by our set of popular benchmarks. Table 2 shows the coverage for LTP and our suite of tests. *Boot* captures the kernel coverage during the boot process. *Xserver* captures the kernel coverage for starting the Xserver. *Capture* is the coverage resulting from a counter reset immediately followed by a counter capture. The column *KC (kernel coverage)* specifies the percentage of kernel code covered by the test. *DC (delta coverage)* is the percent of kernel code covered by the test, but not covered by LTP.

Although LTP coverage is not yet complete, we see that it exercises all but 3.1% of the kernel code used by the combined test XMLS. Another useful view of the data is the code covered by LTP divided by the code covered by the test, *percent of test coverage covered by LTP*. The column *TC (test coverage)* shows these results. For example, LTP covered (hit) 99.9% of the kernel code covered (used) by mkbench. This figure is particularly useful because it does not depend on the amount of kernel code instrumented. LTP provides almost 90% coverage of the important code used by our combined test, XMLS. This result shows very good LTP coverage for the given test suite. *Boot* should be considered separately because much of the code used during boot is never used again once the operating system is running. But even here, LTP covers 60% of the code used during the boot process.

Another benefit of code coverage analysis in the Linux Test Project is that it provides a method of measuring the improvement of the LTP over time. Coverage data can be generated for each version of LTP released and compared against previous versions. Great care must be taken to ensure stable test conditions for comparison. The same machine and same kernel config options are used each time, however the most recent Linux kernel and the most recent version of LTP is used. Changing both is necessary to prove that the code coverage provided by LTP is increasing with respect to the changes that are occurring in the kernel itself. Since features, and thus code, are more often added to the kernel than removed, this means that tests must constantly be added to the LTP in order to improve the effectiveness of LTP as a test suite.

9 Future Work

Obviously, the tools and methods described in this paper are useful in their current state. However, there is some functionality that is still required. LCOV should be modified to process and present branch coverage data currently produced by GCOV. This information would give a better picture of kernel coverage. Many other enhancements are currently being designed or developed against the existing set of tools.

It would be very useful to know kernel coverage for a specific process or group of processes. This could be done by introducing a level of in-

[2]Results are based on one run of the LTP test suite. Although the number of times any one line is hit can vary randomly, we would expect much smaller variance when considering hits greater than zero.

direction, so process-specific counters can be accessed without changing the code. This approach could utilize Thread Local Store (TLS) which is gaining popularity[Drepper]. TLS uses an entry in the Global Descriptor Table and a free segment register as the selector to provide separate initialized and uninitialized global data for each thread. GCC would be modified to reference the TLS segment register when accessing global variables. For a GCOV kernel, the storage would have to be in kernel space (Thread Kernel Storage) The challenge here is to efficiently provide the indirection. Even if the cost of accessing TLS/TKS is low, it is probably infeasible to gather coverage data for all processes running on a system. The storage required would be considerable. So, a mechanism for controlling counter replication would be necessary.

Some parts of the kernel code are rarely used. For example, *init* code is used once and removed from the running image after the kernel boots. Also, a percentage of the kernel code is devoted to fault management and is often not tested. The result of this rarely used code is to reduce the practically achievable coverage. Categorizing the kernel code could isolate this code, provide a more realistic view of LTP results, and help focus test development on important areas in the kernel. The kernel code could be divided into at least three categories. Init and error path code being two separate categories. The rest of the kernel code could be considered mainline code. Coverage would then be relative to the three categories. A challenge to some of this categorization work would be to automate code identification, so new kernels could be quickly processed. In Section 8 we attempted to define *important* kernel code based on lines hit by some popular benchmarks. This approach could also be pursued as a technique for categorizing kernel code.

Finally, a nice feature to have with the system would be the ability to quickly identify the coverage specific to a particular patch. This could be accomplished with a modified version of the *diff* program that is GCOV output format aware. Such a system would perform normal code coverage on a patched piece of code, then do a reversal of the patch against GCOV output to see the number of times each line of code in the patch was hit. This would allow a patch developer to see how much of a patch was being tested by the current test tools and, if necessary, develop additional tests to prove the validity of the patch.

10 Conclusions

This paper described the design and implementation of a new toolkit for analyzing Linux kernel code coverage. The GCOV-kernel patch implements GCOV functionality in the kernel and modules, and provides access to results data through a /proc/gcov filesystem. LCOV processes the results and creates user friendly HTML output. The test results show minimal overhead when using the basic GCOV-kernel. The overhead of an SMP-safe extension can be avoided when precise counter data is not necessary. This paper also described how this toolkit is being used to measure and improve the effectiveness of a test suite such as the Linux Test Project.

This system is already sufficient for use in producing data that will expose the areas of the kernel that must be targeted when writing new tests. It will then help to ensure that the new tests written for the Linux Test Project are successful in providing coverage for those sections of code that were not covered before. The actual implementation of the methods and tools described above as a means of improving the Linux Test Project is just beginning as of the time this paper was written. However, several

tests such as mmap09, as well as LVM and device mapper test suites have been written based solely on coverage analysis. Many other tests are currently in development.

References

[Cornett] *Code Coverage Analysis* `http://www.bullseye.com/coverage.html` (2002)

[Drepper] Urlich Drepper, *ELF Handling For Thread Local Storage*. `http://people.redhat.com/drepper/tls.pdf` (2003)

[Lcovsite] *LTP Gcov Extension* `http://ltp.sourceforge.net/coverage/lcov.php` (2002)

[Marick] Brian Marick, *The Craft of Software Testing*, Prentice Hall, 1995.

[Perry] William E. Perry, *Effective Methods for Software Testing, Second Edition*, Wiley Computer Publishing, 2000.

Effective HPC hardware management and Failure prediction strategy using IPMI

Richard Libby
rml@hpc.intel.com

Abstract

Intelligent Power Management Interface (IPMI) defines common interfaces to "intelligent" hardware used to monitor a server's physical health characteristics, such as temperature, voltage, fans, power supplies and chassis. These capabilities provide information that enables system management, recovery, and asset tracking which help drive down the total cost of ownership (TCO) and increase reliability in today's HPC market. The new interfaces in IPMI v1.5 facilitate the management of rack-mounted HPC servers and systems in remote environment over serial, modem and LAN connections. New capabilities combined with the remote management functionality allow HPC IT managers to manage their servers and systems, regardless of system health, power state or supported communication media. IPMI compliant servers essentially eliminate the need for external hardware to perform the same function, thus saving costs. This paper will introduce the specification, the benefits of IPMI with respect to HPC and other clusters and how it could be used to generate alarms to a monitoring system before hardware failures become severe enough to cause cluster failure.

1 Introduction

If we were to look at what hinders large scale cluster deployments, only a handful of barriers come to mind. There is, of course, always the barrier of limited bandwidth, be it in the front side bus, the interconnect or any other form of I/O. Others include space, power, cooling, monitoring and management. This paper will focus on monitoring, management and predictive analysis: essentially overall HPC health and 'sickness prevention.' Intelligent Power Management Interface (hereafter referred to as IPMI) is an abstracted hardware layer that provides power control, alerting, sensor monitoring, Field Replaceable Unit (FRU) storage, Sensor Data Record (SDR) storage, customization, configuration all through multiple methods of secure access. The total cost of a hardware and software package has come down from US$30/node to sub-US$10, making management a viable solution to reduce the total cost of ownership (TCO).

The IPMI specification itself is a mere 450 pages, and the other supporting documents (four of them [Reference section: C, D, F & H], plus 23 referenced documents or RFCs) make for a good, long weekend of reading. This paper is a brief summary on how to use IPMI to get the most out managing large scale HPC deployments, and be able to use it to predict failures before they happen, thereby reducing the TCO of the entire system. The assumption is made in this paper that IPMI is implemented with strict consideration to the specifications.

2 Power Control

Hardware management of large scale HPC deployments today is quite possible if you have access to many people, infinite resources, or a magic wand. Since most of us have either none of these (in the case of the magic wand), or limited amounts of them (in the case of many people and infinite amounts of money), managing large scale deployments of anything is quite a chore. What do we really mean by 'managing?' Most think of managing a cluster as isolated systems: hardware and software. I would submit that it is a combination of both. If hardware fails, so does software. If software fails, hardware becomes a really expensive room heater. So, management becomes the ability to control software by looking at hardware as one failure point, or have the software control the hardware if thresholds increase beyond set limits. These control methods include powering off a server, powering on a server, resetting a server, and obtaining status of the power condition.

Resetting the server from a remote location seems trivial to most. "Why not just execute a command such a remote 'init 6' (Soft reset)?" or something similar depending on the operating system some might ask. Ah, but what happens if the operating system is hung? Oh, just send one of your people down and hit the reset button. In the case of the TeraGrid project, that may not be possible since there are at least four sites where the HPC might be at. So, a system reset function has gone from being a trivial action to a not-so-trivial action. Using IPMI, a system reset is trivial with a mere command.

Powering on and off a system is not quite as trivial to achieve. Sure it is, you say. You go out and purchase a network power switch, and spend about US$500.00 in the process. Ok, so you *can* achieve power control this way, but not cost effectively in a large scale HPC deployment. Let's say for example, that at best you could control eight machines with one network power switch. At US$63 per outlet, 256 outlets would cost you US$16k, 512 outlets would cost US$32k and 1000 outlets would cost US$63K. This adds only a little to the overall cost, but hey, if it is offered with an IPMI compliant server, you could use this additional money to purchase more servers. There's more on powering on and off, but it will be covered in another section.

In addition to being able to reset, power on and off, having the ability to query the server power status is also valuable. Instead of having to force power off (if on and vice versa), potentially losing a job or data, you can query to see if the server's power is on or off. This seems rather trivial at times, but in some instances when there is no OS available and the server is not in any proximity to you, it is nice to have this feature.

Lastly, one nice control feature is telling the UID light what to do, and how long to do it for. This is the little blue light found on the front and rear of most servers that turns on to identify it from other servers in cases of large scale HPC build-outs.

3 Alerting

Power control is essential, but from a predictive failure analysis standpoint, alerting is the driving force of determining if any one point of the HPC is suffering. Think of the alerting component as pain identification. This identification breaks down into event filtering, trapping (via SNMP), and policy management. Some aspects will not be mentioned fully, as they will be covered in other sections.

Without going into too much depth, when the Baseboard Management Controller (BMC, a.k.a. the service processor) observes an event

that occurs (see sensor section for types of events that might be interesting), it logs it into non-volatile space into a Server Event Log (SEL). Events generated either externally or internally are run through a filter within the BMC that can perform actions based on policies (more on policies later). These types of events might indicate impending danger of catastrophic events on any one machine, or in more serious cases, could indicate rack-wide implications, such as a fire. The ability to filter events combined with policy management gives the HPC administrator a powerful tool in predicting impending failure.

One of the keys to being able to predict (and later analyze) whether there is an impending failure, is knowing about it. The trapping component forwards on messages via SNMP to a "health master" that could look at the event (and compare it against a set of policies), determine if it was critical enough to ruin jobs on that node, and take appropriate action(s).

Policies can be set up to read either discrete or threshold-based values and define an action based on the type of event it is. For example, if a server in your HPC had a fan failure, the most likely thing you would see from an end user point of view is to hear the other fans ramp up. That action is an event that was driven by the BMC through a policy that was set either by a factory setting, or specified by an administrator programmatically. Once the other fans were ramped up to compensate for the missing fan, alerts can be trapped and sent via any number of methods to the "health master," which in turn, can determine (possibly by its own set of policies) if the event warrants moving data or jobs from one machine to a spare. Data can be recorded and analyzed later for recurring trends or even external environment conditions that might be causing failures. One example of this might be as follows: One rack might be sitting directly under a cooling source. The racks adjoining it would read higher ambient chassis temperature, and show skewed temperature data, but with the information, a stealthy sysadmin could correlate the HVAC 'on' times with a drop in ambient temperature inside the rack that gets hit with a blast of cool air. Although this condition in and of itself is not bad, one set of servers in the HPC might run faster due to being slightly cooler and cause other bottlenecks.

4 Common Sensor Model

So, what can IPMI in its present state touch and feel? Presently, it includes voltage, temperature, fan speeds, and presence.

Servers based on IPMI give the ability to monitor voltages on the baseboard and power supply. In cases where there is a lot of solar spot activity, this data could be useful. For example, too many voltage changes, and memory might start to have single bit errors, and too many of those might generate a multi-bit error that is not recoverable. From an industrial and commercial perspective, if the server suffered this, jobs would either be lost, or potentially have silent data corruption. Having the ability to monitor and log this data could be useful to see if there were external environment conditions that would impact server stability. In this particular example, a sysadmin might want to spec out Telco grade hardware that runs on DC power, thereby eliminating the external condition and increasing reliability.

With the tight thermal tolerances these days, a fan failure in server chassis can result in a catastrophic loss, especially in the smaller form factors (i.e. 1U). Having the ability to monitor fan speeds can greatly increase the chance of keeping jobs that are running on one member of the HPC and moving them to another, provided the event deems it worthy of

doing so. This does not just include chassis fans, but can be expanded to include processor, memory, and hard drive bay fans.

Temperature is a key aspect to monitor for predicting failure in an HPC. Logging events, using stats to show trends, and reacting to generated alerts are the ways to decrease a sysadmin's possibilities of downtime. It will also increase the ability to determine external environment conditions that would affect HPC performance, since heat is a critical aspect.

The last aspect of the common sensor model is the ability to detect presence. At times it would be nice to detect if the server chassis lid was opened. The system might log an entry into the SEL based on chassis intrusion. This might be useful information later on to determine causes of ambient temperature changes, or even to determine what time the processors and memory disappeared out of the best server.

5 Access Methods

Ok, so we are able to monitor all these cool things on our server, but how does the sysadmin or developer *really* get to them? There are multiple methods to get the information you need from an IPMI-based server. They include KCS, LAN, serial, modem, I2C, IPMB and ICMB. None of these methods will be discussed in too much detail here due to space and scope constraints.

One of the most common methods to get to the BMC is through an interface called Keyboard Controller Style (KCS). For IPMI laymen, this means a driver that is loaded and controlled through the OS. This is the preferred method to access the Intelligent Power Management Bus (IPMB), but certainly not the only one, and certainly in cases where the OS has gone south for the winter.

If the OS does go south, another nice method is using the Remote Management Control Protocol (RMCP) over the LAN. IPMI-based servers have 3.3 volts standby power that provides the BMC with enough juice to live on. With the cost of network interface cards (NIC) coming down, and many board manufacturers placing them on their server boards, this becomes the avenue on which to send UDP packets directed to port 623 that can issue IPMI commands. This becomes a close second to the KCS in that it replaces legacy serial concentration servers, allowing the sysadmin or IT manager to spend their US$50K (for every 40 ports and US$1.2M for 1000 servers) elsewhere on the HPC.

Other methods of access include direct serial using a serial concentration server. The downside to this interface is the cost for the serial concentration server. One nice feature worth mentioning in a Linux environment is Serial Over LAN (SOL). This allows the HPC sysadmin to enable serial redirect and have it piped over the network interface, thereby eliminating the need for a Keyboard-Video-Mouse controller (KVM). Costs are exorbitant and cabling becomes a nightmare for large scale HPC deployments. (Other features include private management bus (private I2C bus), and ability to add above board remote management cards.)

One nice feature of IPMI is the ability to forward information on to other IPMI compliant servers. Essentially, each IPMI server can become a proxy to relay messages to another IPMB through the Intelligent Chassis Management Bus (ICMB). A sysadmin or network architect could design an ICMB network (really an RS-845 multiport serial bus) that would allow up to 64 nodes per network. Management packets could be routed over this network instead of tying up the LAN channel, or in cases when the LAN was down, routing over ICMB. RS-845 also allows broadcasts, so the

sysadmin could invoke a broadcast to a bank of servers. Imagine it, a command that said: "Hello, rack of servers, shut down."

6 Authentication

All the methods of access are excellent to have, but how are they protected against intrusion? This section briefly discusses the channel privilege levels and encryption methods used to authenticate to the BMC.

There are four privilege levels to the BMC as defined by the IPMI specification: callback, user, operator, and administrative.

Callback is essentially irrelevant these days. It calls only a predefined number via a modem. It also only supports enough commands to initiate a callback. If the user is not at that location, then callback is not highly useful.

One nice channel privilege is called user. It would allow subsystems to monitor HPCs without being intrusive, and potentially lethal from a job perspective. Think of this privilege as 'sensor snag' only-essentially read-only or only benign commands are allowed. From a predictive analysis perspective, this level would give alerting mechanisms with the peace of mind that power control could be assigned to alternate members of the HPC.

Occasionally though, job scalability demands delegation of authority, but not total relinquishing of control. In these cases, the operator privilege level should be chosen. It has all commands available to administration, except configuration commands that can change the behavior of the out-of-band interfaces.

And then you have full rights-administrative privilege level. This privilege allows for full control over an IPMI-based server. In simple "health master" implementations, this would be the preferred privilege level to use. One quick word of caution: an administrative privilege level can even disable the channel they are coming in over.

Encryption is tied closely to authentication. There are basically three main points to remember on IPMI encryption. The first is that passwords can be sent clear-text, so from a security standpoint, it is something to consider. The second is that whatever the user initiates as an encryption algorithm is what will be returned, provided it falls into the third category. That is that there are generally two supported encryption algorithms, MD2 and MD5.

7 Field Replaceable Units (FRU)

An enterprise-class system will typically have FRU information for each major system board (e.g. processor board, memory board, I/O board, etc.). The FRU data can include information such as serial number, part number, model, and asset tag. What is this really good for in a failure prediction situation? Well, the "health master" could be programmed such that it could detect that a remote server's part is going to have a failure (let's just say overheating CPU), look up the part in a parts database, and alert the sysadmin which part they need to take with them to service the unit, what the unit serial number is, and possibly when it was last serviced. It would minimize downtime. By the way, FRU information can even be available when the system is powered down. Another useful example of FRUs being used is automated remote inventory. "IPMI does not seek to replace other FRU or inventory data mechanisms, such as those provided by SM BIOS, and PCI Vital Product Data. Rather, IPMI FRU information is typically used to complement that information or to provide information access out-of-band or under 'system down' conditions."

8 Sensor Data Records

IPMI was created with extensibility and scalability in mind. Unfortunately, with that capability comes a myriad of different components that can be monitored and controlled in different fashions. Sensor Data Records (SDRs) provide system management software the ability to retrieve information from the platform, and automatically configure itself to meet the capabilities of the platform-essentially an abstraction layer.

SDRs exist primarily to describe to server management software what a sensor configuration should look like, and to tell software to pay special attention to certain sub functions. SDRs are mostly not available to end users, but could potentially, especially if purchasing a board and chassis separately. For the most part, SDRs are made specifically for certain configurations of board/chassis combinations.

SDRs also define thresholds and actions based on those thresholds. For example, there might be an upper critical threshold on a board thermal sensor, and that might be tied via an SDR to an action of ramping up fans if excessive heat was detected. From a failure prediction standpoint, the "health master" could detect these changes, and ramp fans accordingly. This is just one small case, but you can see that the permutations get quite large with more servers in your HPC. What kind of sensor types are there? Two main categories exist-analog or digital, or fan, voltage, temperature. SDRs tell the BMC what the sensor type is in order to know how to process it (i.e. analog fan situation: you actually get a voltage back when polling it. The software will need to convert it via a mathematical function (a slope function: logarithmic, square root, quadratic, sin and many more defined in the IPMI spec.)

Realistically, the types of information that SDRs can store configuration on are: CPU sensors, chassis intrusion, power supply monitoring, fan speeds, fan presence, board voltages, board temperatures, bus errors, memory errors, and even possibly ASF progress codes (where the subsystem is in coming up-essentially a POST code). "Sensor Data Records are kept in a single, centralized non-volatile storage area that is managed by the BMC. This storage is called the Sensor Data Record Repository (SDR Repository). Implementing the SDR Repository via the BMC provides a mechanism that allows SDRs to be retrieved via 'out-of-band' interfaces, such as the ICMB, a Remote Management Card, or other device connected to the IPMB. Like most Intelligent Platform Management features, this allows SDR information to be obtained independent of the main processors, BIOS, system management software, and the OS."

9 System Event Log

Every IPMI compliant server has a System Event Log (SEL) which is a centralized, non-volatile repository for all events generated. Think of this not only as a BMC journal. Any authenticated user can enter a SEL entry.

Common events that might be stored are reboots, processors offline, memory (both single and multiple bit) errors, sensors that go beyond set thresholds, PCI parity errors (PERR), Non-maskable interrupts (NMI), and many more. The "health master" could periodically poll select system event logs from the HPC, and determine if there are potential problems that could be coming down the wire, and mitigate a response to those problems.

Alerting is done off the SEL as well. When an event is written to the SEL, it is checked against policies and threshold values, and performs actions based on those policies. Too

much of a health conscious sysadmin could put in place tighter thresholds, and decrease the possibility of downtime, while at the same time possibly risking more time checking out false alarms. A truly health conscious sysadmin would determine standard thresholds to work with safely, while keeping in mind that a false alarm here and there might keep them on their toes and possibly prevent a catastrophic failure on a node (or more in some cases).

Clearing the SEL is possible as well, in fact, once a SEL is queried and data is stored on the "health master," it is advisable to clear the SEL. The main reason for this is that if the SEL fills up, new entries are dropped. This could cause a sysadmin to miss critical events that could lead either to extensive downtime, or a catastrophe.

10 Configuration

IPMI gives such flexibility that many aspects can be configured to fit the end-user's needs.

These parameters include: BMC policies, LAN configuration (such as static or dynamic IP address, mask, and gateway), privilege levels, alerting functions (such as whom to forward alerts to), sensor polling rates, etc.

Imagine for a moment that your "health master" detects that it one of the nodes in the HPC is going down due to some failure, and you are able to retrieve the critical data off the hard drive. When you bring up a spare node in its place, you can dump the hard drive data back down, right? Sure, but what about BMC configuration? This is where IPMI shines through if implemented properly. With most IPMI compliant servers today, a sysadmin can retrieve the BMC information. But stuffing it back onto another machine is a little trickier, especially if it is over the LAN interface.

11 Questions Users Might Ask

A few questions might remain in the reader's mind still after this brief whetting. One such question might be, "What is the relationship or difference between IPMI and Alert Standard Forum (ASF)?" "While somewhat of an oversimplification, ASF may be considered to be scoped for 'desktop/mobile' class systems, and IPMI for 'servers' where the additional IPMI capabilities such as event logging, multiple users, remote authentication, multiple transports, management extension busses, sensor access, etc., are valued. However there are no restrictions in either specification as to the class of system that the specification can be used. [(i.e.)] you can use IPMI for desktop and mobile systems and ASF for servers if the level of manageability fits your requirements."

Another might be, I don't have IPMI capable servers today, can I add in an IPMI card? The short answer to that question is possibly, but it depends on your server base board. Provided it has an interface to IPMB, the possibilities increase. But, the best thing to do is to weigh the costs associated with not having IPMI (as mentioned earlier-no KVM, network power switch, serial concentration server) and see what benefits it can bring to predict an impending failure before it happens.

12 Summary

The key to using IPMI for failure prediction analysis is the polling and listening software, and how intelligent it is at analyzing the data to make predictions as to where problems lie.

As most HPC sysadmins know, one large scale cluster deployment barrier is management, monitoring, fault isolation and failure prediction. IPMI enables a sysadmin a great way to reduce the TCO of HPCs by monitor-

ing overall HPC health and 'prevent sickness' by providing an abstracted hardware layer that provides power control, alerting, sensor monitoring, Field Replaceable Unit (FRU) storage, Sensory Data Record (SDR) storage, customization, configuration all through multiple methods of secure access-all through software.

13 Call to Action

When writing RFQs, make sure to include IPMI as a required feature in order to reduce TCO and increase manageability in HPC deployments.

14 References

A. Alert Standard Format v1.0 Specification, ©2001, Distributed Management Task Force. `http://www.dmtf.org`

B. The I2C Bus And How To Use It, ©1995, Philips Semiconductors. This document provides the timing and electrical specifications for I2C busses.

C. Intelligent Chassis Management Bus Bridge Specification v1.0, rev. 1.2, ©2000 Intel Corporation. Provides the electrical, transport protocol, and specific command specifications for the ICMB and information on the creation of management controllers that connect to the ICMB. `http://developer.intel.com /design/servers/ipmi`

D. Intelligent Platform Management Bus Communications Protocol Specification v1.0, ©1998 Intel Corporation, Hewlett-Packard Company, NEC Corporation, and Dell Computer Corporation. This document provides the electrical, transport protocol, and specific command specifications for the IPMB. `http://developer.intel.com /design/servers/ipmi`

E. Intelligent Power Management Interface Specification v1.5; rev. 1.1, ©2002 Intel Corporation, Hewlett-Packard Company, NEC Corporation, and Dell Computer Corporation. `http://developer.intel.com /design/servers/ipmi`

F. IPMI Platform Event Trap Format Specification v1.0, ©1998, Intel Corporation, Hewlett-Packard Company, NEC Corporation, and Dell Computer Corporation. This document specifies a common format for SNMP Traps for platform events.

G. Proposal for Callback Control Protocol (CBCP), draft-ietf-pppext-callback-cp-02.txt, N. Gidwani, Microsoft, July 19, 1994. As of this writing, the specification is available via the Microsoft Corporation web site: `http://www.microsoft.com`

H. Platform Management FRU Information Storage Definition v1.0, ©1999 Intel Corporation, Hewlett-Packard Company, NEC Corporation, and Dell Computer Corporation. Provides the field definitions and format of Field Replaceable Unit (FRU) information. `http://developer.intel.com /design/servers/ipmi`

I. RFC 1319, The MD2 Message-Digest Algorithm, B. Kaliski, RSA Laboratories, April 1992.

J. RFC 1321, The MD5 Message-Digest Algorithm, R. Rivest, MIT Laboratory for Computer Science and RSA Data Security, Inc. April, 1992.

K. System Management BIOS Specification, Version 2.3.1, ©1997, 1999 American Megatrends Inc., Award Software International, Compaq Computer Corporation, Dell Computer Corporation, Hewlett-Packard Company, Intel Corporation, International Business Machines Corporation, Phoenix Technologies Limited, and SystemSoft Corporation.

L. System Management Bus (SMBus) Specification, Version 2.0, ©2000, Duracell Inc., Fujitsu Personal Systems Inc., Intel Corporation, Linear Technology Corporation, Maxim Integrated Products, Mitsubishi Electric Corporation, Moltech Power Systems, PowerSmart Inc., Toshiba Battery Co., Ltd., Unitrode Corporation, USAR Systems.

M. The TeraGrid Project, National Center for Supercomputing Applications, San Diego Supercomputer Center, Argonne National Laboratory and Pittsburgh Supercomputing Center; http://www.teragrid.org

N. Wired for Management Baseline Version 2.0 Release, ©1998, Intel Corporation. Attachment A, UUIDs and GUIDs, provides information specifying the formatting of the IPMI Device GUID and FRU GUID and the System Management BIOS (SM BIOS) UUID unique IDs.

15 Glossary of Terms

BMC Baseboard Management Controller

CMOS In terms of this specification, this describes the PC-AT compatible region of battery-backed 128 bytes of memory, which normally resides on the baseboard.

Diagnostic Interrupt A non-maskable interrupt or signal for generating diagnostic traces and ‘core dumps’ from the operating system. Typically NMI on IA-32 systems, and an INIT on Itanium®-based systems.

EFI Extensible Firmware Interface. A new model for the interface between operating systems and platform firmware. The interface consists of data tables that contain platform-related information, plus boot and runtime service calls that are available to the operating system and its loader. Together, these provide a standard environment for booting an operating system and running pre-boot applications.

FRB Fault Resilient Booting. A term used to describe system features and algorithms that improve the likelihood of the detection of, and recovery from, processor failures in a multiprocessor system.

FRU Field Replaceable Unit. A module or component which will typically be replaced in its entirety as part of a field service repair operation.

Hard Reset A reset event in the system that initializes all components and invalidates caches.

HPC High Performance Computing, commonly computationally intense.

I2C Inter-Integrated Circuit bus. A multi-master, 2-wire, serial bus used as the basis for the Intelligent Platform Management Bus.

ICMB Intelligent Chassis Management Bus. A serial, differential bus designed for IPMI messaging between host and peripheral chassis. Refer to [ICMB] for more information.

I/O Input / Output. Typically refers to I/O subsystems such as PCI (and variants), memory, and CPU buses.

IPM Intelligent Platform Management.

IPMB Intelligent Platform Management Bus. Name for the architecture, protocol, and implementation of a special bus that interconnects the baseboard and chassis electronics and provides a communications media for system platform management information. The bus is built on I2C and provides a communications path between ’management controllers’ such as the BMC, FPC, HSC, PBC, and PSC.

NMI Non-maskable Interrupt. The highest priority interrupt in the system, after SMI. This interrupt has traditionally been used to notify the operating system fatal system hardware error conditions, such as parity errors and unrecoverable bus errors. It is also used as a Diagnostic Interrupt for generating diagnostic traces and ‘core dumps’ from the operating system.

MD2 RSA Data Security, Inc. MD2 Message-Digest Algorithm. An algorithm for forming a 128-bit digital signature for a set of input data.

MD5 RSA Data Security, Inc. MD5 Message-Digest Algorithm. An algorithm for forming a 128-bit digital signature for a set of input data. Improved over earlier algorithms such as MD2.

PEF Platform Event Filtering. The name of the collection of IPMI interfaces in the IPMI v1.5 specification that define how an IPMI Event can be compared against 'filter table' entries that, when matched, trigger a selectable action such as a system reset, power off, alert, etc.

PERR Parity Error. A signal on the PCI bus that indicates a parity error on the bus.

PET Platform Event Trap. A specific format of SNMP Trap used for system management alerting. Used for IPMI Alerting as well as alerts using the ASF specification. The trap format is defined in the PET specification. See [PET] and [ASF] for more information.

POST Power On Self Test.

RFQ Request for Quote.

SDR Sensor Data Record. A data record that provides platform management sensor type, locations, event generation, and access information.

SEL System Event Log. A non-volatile storage area and associated interfaces for storing system platform event information for later retrieval.

SERR System Error. A signal on the PCI bus that indicates a 'fatal' error on the bus.

Soft Reset A reset event in the system which forces CPUs to execute from the boot address, but does not change the state of any caches or peripheral devices.

TCO Total Cost of Ownership. All of the possible costs involved in the purchase, installation, management, support and use of the IT infrastructure within an organization throughout a product's life cycle, from acquisition to disposal.

Interactive Kernel Performance

Kernel Performance in Desktop and Real-time Applications

Robert Love
MontaVista Software
rml@tech9.net, http://tech9.net/rml

Abstract

The 2.5 development kernel introduced multiple changes intent on improving the interactive performance of Linux. Unfortunately, the term "interactive performance" is rather vague and lacks proper metrics with which to measure. Instead, we can focus on five key elements:

- fairness
- scheduling latency
- interrupt latency
- process scheduler decisions
- I/O scheduler decisions

In short, these attributes help constitute the feel of Linux on the desktop and the performance of Linux in real-time applications. As Linux rapidly gains market share both on the desktop and in embedded solutions, quick system response is growing in importance. This paper will discuss these attributes and their effect on interactive performance.

Then, this paper will look at the responses to these issues introduced in the 2.5 development kernel:

- O(1) scheduler
- Anticipatory/Deadline I/O Scheduler
- Preemptive Kernel
- Improved Core Kernel Algorithms

Along the way, we will look at the current interactive performance of 2.5.

1 Introduction

Interactive performance is important to a wide selection of computing classes: desktop, multimedia, gamer, embedded, and real-time. These application types benefit from quick system response and deterministic or bounded behavior. They are generally characterized as having explicit timing constraints and are often I/O-bound. The range of applications represented in these classes, however, varies greatly—word processors, video players, Quake, cell phone interfaces, and data acquisition systems are all very different applications. But they all demand specific response times (although to varying degrees) to various stimuli (whether its the user at the console or data from a device) and all of these applications find good interactive performance important.

But what *is* interactive performance? How can we say whether a kernel has good interactive performance or not? For real-time applications, it is easy: "do we meet our timing constraints or not?" For multimedia applications, the task is harder but still possible: for example, "does our audio or video skip?" For

desktop users, the job is even harder. How does one express the interactive performance of their text editor or mailer? Worse, how does one *quantify* such performance? All too often, interactive performance is judged by the seat of one's pants. While actually perceiving good interactive performance is an important part of their actually existing good interactive performance, a qualitative experience is not regimented enough for extensive study and risks suffering from the placebo effect.

For the purpose of this paper, we break down the vague term "interactive performance" and define five attributes of a kernel which benefit the aforementioned types of applications: fairness, scheduling latency, interrupt latency, process scheduler decisions, and I/O scheduler decisions.

We then look at four major additions to the 2.5 kernel which improve these qualities: the O(1) scheduler, the deadline and anticipatory I/O schedulers, kernel preemption, and improved kernel algorithms.

2 Interactive Performance

2.1 Fairness

Fairness describes the ability of tasks to all make not only forward progress but to do so relatively evenly. If a given task fails to make any forward progress, we say the task is starved. Starvation is the worst example of a lack of fairness, but any situation in which some tasks make a relatively greater percentage of progress than other tasks lacks fairness.

Fairness is often a hard attribute to justify maintaining because it is often a tradeoff between overall global performance and localized performance. For example, in an effort to provide maximum disk throughput, the 2.4 block I/O scheduler may starve older requests in order to continue processing newer requests at the current disk head position. This minimizes seeks and thus provides maximum overall disk throughput—at the expense of fairness to all requests.

Since the starved task may be interactive or otherwise timing-sensitive, ensuring fairness to all tasks (or at least all tasks of a given importance) is a very important quality of good interactive performance. Improving fairness throughout the kernel is one of the biggest changes made during the 2.5 development kernel.

2.2 Scheduling Latency

Scheduling latency is the delay between a task waking up (becoming runnable) and actually running. Assuming the task is of a sufficiently high priority, this delay should be quite small: an interrupt (or other event) occurs which wakes the task up, the scheduler is invoked to select a new task and selects the newly woken up task, and the task is executed. Poor scheduling latency leads to unmet timing requirements in real-time applications, perceptible lag in application response in desktop applications, and dropped frames or skipped audio in multimedia applications.

Both maximum and average scheduling latency is important, and both need to be minimized for superior interactive performance. Nearly all applications benefit from minimal average scheduling latency, and Linux provides exceptionally good average-case performance. Worst-case performance is a different issue: it is an annoyance to desktop users when, for example, heavy disk I/O or odd VM operations cause their text editor to go catatonic. If the event is relatively rare enough, however, they may overlook it. Both real-time and multimedia applications, however, require a specific bound on worst-case scheduling la-

tencies to ensure functionality.

The preemptive kernel and improved algorithms (reducing lock hold time or reducing the algorithmic upper bound) result in a reduction in both average and worst-case scheduling latency in 2.5.

2.3 Interrupt Latency

Interrupt latency is the delay between a hardware device generating an interrupt and an interrupt handler running and processing the interrupt. High interrupt latency leads to poor system response as the actions of hardware are not readily perceived by the kernel.

Interrupt latency in Linux is basically a function of interrupt off time—the time in which the local interrupt system is disabled. This only occurs inside the kernel and only for short periods of time reflecting critical regions which must execute without risk of an interrupt handler running. Linux has always had comparatively small interrupt latencies—on modern machines, less than 100 microseconds.

Consequently, reducing interrupt latency was not a primary goal of 2.5, although it undoubtedly occurred as lock hold times were reduced and the global kernel lock (`cli()`) was finally removed.

2.4 Process Scheduler Decisions

The behavior and decisions made by the process scheduler (the subsystem of the kernel that divides the resource of CPU time among runnable processes) are important to maintaining good interactive performance. This should go without saying: poor decisions can lead to starvation and poor algorithms can lead to scheduling latency. The process scheduler also enforces static priorities and can issue dynamic priorities based on a rule-set or heuristic.

The process scheduler in 2.5 provides deterministic scheduling latency via an `O(1)` algorithm and a new interactivity estimator which issues priority bonuses for interactive tasks and priority penalties for CPU-hogging tasks.

2.5 I/O Scheduler Decisions

I/O scheduler (the subsystem of the kernel that divides the resource of disk I/O among block I/O requests) decisions strongly affect fairness. The primary goal of any I/O scheduler is to minimize seeks; this is done by merging and sorting requests. This maximizing of global throughput can directly lead to localized fairness issues. Request starvation, particularly of read requests, can lead to long application delays.

Two new I/O schedulers available for 2.5, the deadline and the anticipatory I/O schedulers, prevent request starvation by attempting to dispatch I/O requests before a configurable deadline has elapsed.

3 Process Scheduler

3.1 Introduction

The process scheduler plays an important role in interactive performance. The Linux scheduler offers three different scheduling policies, one for normal tasks and two for real-time tasks. For normal tasks, each task is assigned a priority by the user (the nice value). Each task is assigned a chunk of the processor's time (a timeslice). Tasks with a higher priority run prior to tasks with a lower priority; tasks at the same priority are round-robined amongst themselves. In this manner, the scheduler prefers tasks with a higher priority but ensures fairness to those tasks at the same priority.

The kernel supports two types of real-time

tasks, first-in first-out (FIFO) real-time tasks and round-robin (RR) real-time tasks. Both are assigned a static priority. FIFO tasks run until they voluntarily relinquish the processor. Tasks at a higher priority run prior to tasks at a lower priority; tasks at the same priority are round-robined amongst themselves. RR tasks are assigned a timeslice and run until they exhaust their timeslice. Once all RR tasks of a given priority level exhaust their timeslice, the timeslices are refilled and they continue running. RR tasks at a higher priority run before tasks at a lower priority. Since real-time tasks can be scheduled unfairly, they are expected to have a sane design which properly utilizes the system.

All scheduling in Linux is done preemptively, except FIFO tasks which run until completition. New in 2.5, preemption of tasks can now occur inside the kernel.

For 2.5, the process scheduler was rewritten. The new scheduler, dubbed the O(1) scheduler, features constant-time algorithms, per-processor runqueues, and a new interactivity estimator. The Linux scheduling policy, however, is unchanged.

3.2 Interactivity Estimator

The 2.5 scheduler includes an interactivity estimator [mingo1] which dynamically scales a task's static priority (nice value) based on its interactivity. Interactive tasks receive a priority bonus while tasks which excessively hog the CPU receive a penalty. Tasks in some theoretical neutral position (neither interactive nor hoggish) receive neither a bonus nor a penalty. By default, up to five priority values are added or removed to reflect the degree of the bonus or the penalty; note this corresponds to 25% of the full -20 to 19 nice value range.

Interactivity is estimated using a running sleep average. The idea is that interactive tasks are I/O bound. They spend much of their time waiting for user interaction or some other event to occur. Tasks which spend much of their time sleeping are thus interactive; tasks which spend much of their time running (continually exhausting their timeslice) are CPU hogs. These rules are surprisingly simple; indeed, they are essentially the definitions of I/O-bound and CPU-bound. The fact the heuristic is basically following the definition lends credibility to the estimator.

The heuristic determines the actual bonus or penalty based on the ratio of the task's actual sleep average against a constant "maximum" sleep average. The closer the task is to the maximum, the more of the five bonus priority levels it can receive. Conversely, the closer the task is to the negation of the maximum sleep average, the larger its penalty is.

The results of the interactivity estimator are apparent:

```
USER   NI   PRI   %CPU   STAT   COMMAND
rml    0    15    0.0    S      vim
rml    0    18    0.4    S      bash
rml    0    25    91.7   R      infloop
```

First, note the kernel priority values (the `PRI` column) correspond to a mapping of the default nice value of zero to the value 20. The lowest priority nice value, 19, is priority 39. The highest priority nice value, -20, is zero. Thus, lower priority values are higher in priority. A text editor, vim, has received the full negative five priority bonus. Since it initially had a nice value of zero, it now has a priority of 15. Conversely, a program executing an infinite loop received the full positive five priority penalty; it now has a priority of 25. Bash, which is basically interactive but performs computation in scripts, has received a smaller bonus and now has a priority of 18.

Higher priority (that is, lower priority val-

ued) tasks are scheduled prior to lower priority (higher priority valued) tasks. They also receive a larger timeslice. This implies that interactive tasks are usually runnable; they are scheduled first and generally have plenty of timeslice with which to run. This ensures that the text editor is capable of responding to a keypress instantly, even if the system is under load.

The interactivity estimator does not apply to real-time tasks, which occupy a fixed priority in a higher priority range than any normal task. The estimator benefits interactive desktop programs, such as a text editor or mailer.

3.3 Reinsertion of Interactive Tasks

All process schedulers implement a mechanism of recalculating and refilling process timeslices. In the most rudimentary of schedulers, this work occurs when all processes have exhausted their timeslice and then the timeslice and priority of each process is recalculated and reassigned. Scheduling then continues as before, until again all processes exhaust their timeslice and this work repeats.

The O(1) scheduler implements an `O(1)` algorithm for timeslice recalculation and refilling. Instead of performing a large `O(n)` recalculation when all processes exhaust their timeslice, the O(1) scheduler implements two arrays of tasks, the active array and the expired array. When a task exhausts its timeslice, it is moved to the expired array and its timeslice is refilled. When the active array is empty, the two arrays are switched (via a simple pointer swap) and the scheduler begins executing tasks out of the new active array. This algorithm guarantees a deterministic and constant-time solution to timeslice recalculation.

Another benefit of this approach is it provides a simple prevention to processor starvation of interactive tasks. In the 2.4 scheduler, when a task exhausts its timeslice it does not have a chance to run again until the remaining tasks also exhaust their timeslice and timeslices are globally recalculated. This allows starvation of the task, which might lead to perceptable delays. To prevent this, the 2.5 scheduler will reinsert interactive tasks into the active array when they expire their timeslice. *How* interactive a task need be in order to be reinserted into the active array and not expired depends on the task's priority. To prevent indefinite starvation of non-interactive tasks in the expired array, interactive tasks are only reinserted into the active array so long as the tasks on the expired array have run recently.

3.4 Finegrained timeslice distribution

A final behavioral change in the O(1) scheduler is a more finegrained timeslice distribution and calculation [mingo2]. Currently, this behavior is only present in the 2.5-mm tree but will likely be merged into the mainline 2.5 tree soon.

Normally, tasks are round-robined with other tasks of the same priority (tasks with a higher priority are run earlier and tasks with a lower priority will run later). When they exhaust their timeslice, their priority is recalculated (taking into effect the interactivity estimator) and they are either placed on the expired list or inserted into the back of the queue for their priority level.

This leads to two problems. First, the scheduler may give an interactive task a large timeslice in order to ensure it is always runnable. This is good, but it also results in a long timeslice that may prevent other tasks from running. Second, since priority is recalculated only when a task exhausts its timeslice, a task with a large timeslice may go some time without a priority recalculation. The task's behavior may change

in this time, reversing whether or not the task is deemed interactive. Recognizing this, the scheduler was modified to split timeslices into small pieces—by default, 20ms chunks. Tasks do not receive any less timeslice, instead a task of equal priority may preempt the running task every 20ms. The task is then requeued to the end of the list for its priority and it continues to run round robin with other tasks at its priority level. In addition to this finer distribution of timeslices, the task's priority is recalculated every 20ms as well.

3.5 An `O(1)` Algorithm

An important property of real-time systems is deterministic behavior. Time-sensitive applications demand consistent behavior that they can understand a priori. Critical algorithms, therefore, need to operate in constant time or at least within predefined bounds.

It is important that scheduling behavior (especially process selection and process wake up) operate deterministically, as time-sensitive applications demand minimal latency from wake up to process selection to actual execution. If a scheduling algorithm is dependent on the total number of processes (or runnable processes) even a high priority task cannot make an assumption about scheduling latency. Worse, with a sufficiently large number of processes, the time required to wake up and schedule a task may be far larger than acceptable (e.g., your mp3 may skip or the nuclear power plant may meltdown).

By introducing `O(1)`—constant time—algorithms for all scheduler functions, the O(1) scheduler offers not only deterministic but constant scheduler performance. The scheduler can wake up a task, select it to run, and execute it in the same amount of time regardless of whether there are five or five hundred thousand processes on the system. More so, since the O(1) scheduler has exceptionally quick `O(1)` algorithms, scheduling latency may be reduced for a given `n` over previous scheduling algorithms. Thus, the 2.5 scheduler offers deterministic, constant, and (perhaps) reduced scheduling latency over previous Linux kernel schedulers.

4 I/O Scheduler

4.1 Introduction

The primary job of any I/O scheduler (sometimes called an elevator) is to merge adjacent requests together (to minimize requests) and to sort incoming requests seek-wise along the disk (to minimize disk head movement). Reducing the number of requests and minimizing disk head movement is critical for overall disk throughput. Disk seeks (moving the disk head from one sector to another) are very slow. If the number and distance of seeks are minimized by reordering requests, disk transfer rates are kept closer to their theoretical maximum.

In the interest of global throughput, however, I/O scheduler decisions can introduce local fairness problems. Sorting requests can lead to the starvation of requests that are not near other requests on the disk. If a heavy writeout is underway, the incoming write requests are inserted near each other in the request queue and dealt with quickly, minimizing seeks. A request to a far-off sector may not receive attention for some time. This request starvation is detrimental to system response as it is unfair. Request starvation is a shortcoming in the 2.4 I/O scheduler.

The general issue of request starvation leads to a more specific case of starvation, writes-starving-reads. Write operations can usually occur whenever the I/O scheduler wishes to commit them, asynchronous with respect to the

submitting application or filesystem. Read operations, however, almost always involve a process waiting for the read to complete—that is, read requests are usually synchronous with respect to the submitting application or filesystem. Because system response is largely unaffected by write latency (the time required to commit a write) but is strongly affected by read latency (the time required to commit a read), prioritizing reads over writes will prevent write requests from starving read requests and increase the responsiveness of the system.

Unfortunately, minimizing seeks and preventing unfairness from request starvation are largely conflicting goals. With a proper solution, however, the fairness issues are resolvable without a large drop in global disk throughput.

4.2 Request Starvation

To prevent starvation of requests, a new I/O scheduler, the deadline I/O scheduler, was introduced [axboe1, axboe2]. The deadline I/O scheduler works by assigning tasks an expiration time and trying to ensure (although not guaranteeing) that requests are dispatched before they expire.

The 2.4 I/O scheduler [arcangeli] implements a single queue, which is sorted ascendingly by sector. Requests are either merged with adjacent requests or sorted into the proper location in the queue; requests are appended to the tail if they have no proper insertion point. The I/O scheduler then dispatches requests as the block devices request them from the head of the queue.

The deadline I/O scheduler augments this sorted queue with two more queues, a first-in first-out (FIFO) queue of read requests and a FIFO queue of write requests. Each request in the FIFO queues is assigned an expiration time. By default, this is 500 milliseconds for read requests and 5 seconds for write requests. When a request is submitted to the deadline I/O scheduler, it is added to both the sorted queue and the appropriate FIFO queue. In the case of the sorted queue, the request is merged or otherwise inserted sector-wise where it fits. In the case of the FIFO queues, the request is assigned an expiration value and placed at the tail of the queue.

Normally, the deadline I/O scheduler services requests from the sorted queue, to minimize seeks. If a request expires at the head of either FIFO queue (the requests at the head are the oldest), however, the scheduler stops dispatching items from the sorted queue and begins dispatching from the FIFO queues. This behavior ensures that, in general, seeks are minimized and thus global throughput is maximized. Fairness is maintained, however, as the I/O scheduler attempts to dispatch requests within the specified expiration time. The deadline I/O scheduler provides an upper bound on request latency—ensuring fairness—at the expense of a small degradation in overall throughput.

4.3 Writes-Starving-Reads

Usually, read operations are synchronous while writes operations are asynchronous. Basically, when an application issues a read request, it cannot continue until the operation completes and the application is given the requested value. The completion of write operations, on the other hand, usually has no bearing on the progress of the application. Aside from worrying about power failures, an application is unconcerned as to whether a write commits to disk in one second or five minutes. In fact, most applications are probably unaware if the data is ever committed! Conversely, an application usually needs the results of a read operation and will block until the data is returned. Worse, read requests are often issued en masse and each read is dependent on the previous.

The application or filesystem will not submit read request `N` until read request `N-1` completes.

In the 2.4 I/O scheduler, read and write requests are treated equal. The 2.4 I/O scheduler tries to minimize seeks by sorting requests on insert. If a request is issued that is between (seek-wise) two other requests in the queue, it is inserted there. If there is no suitable place to insert the request (perhaps because no other operations are occurring to the same area of the disk), the request is appended to the end of the queue. Consequently, something like

```
cat * > /dev/null
```

where there is even only a moderate number of files in the current directory results in hundreds of dependent read requests. If a heavy write is underway, each individual read request will be inserted at the tail of the queue. Assuming the queue can hold a maximum of about one second's worth of requests, each individual read request takes a second to reach the head of the queue. That is, the heavy write operation continually keeps the queue full with write operations to some part of the disk. When the read request is submitted, there is no suitable insertion point so it is appended to the tail of the queue. After a second, the read is finally at the head of the queue, and it is dispatched. This repeats for each and every individual read. Since each read is dependent on the next, the requests are issued in serial. Thus the previous `cat` takes hundreds of seconds to complete in 2.4 when the system is also under write pressure.

Recognizing that the asynchrony and interdependency of read operations highlights their much stronger latency requirements over writes, various patches were introduced [akpm1] to solve the problem. Acknowledging that appending reads to the tail of the queue is detrimental to performance, these modifications insert reads (failing a proper insertion elsewhere) near the head of the queue. This drastically improves application performance—more than ten-fold improvements—as it prevents writes from starving reads.

The deadline I/O scheduler, the current default I/O scheduler in 2.5, addresses this issue as well. The deadline I/O scheduler provides a separate (generally much smaller) expiration timeout for read requests. Consequently, the I/O scheduler tries to submit reads requests within a rather short period, ignoring write requests that may be adjacent to the disk head's current location or that have been waiting longer. This prevents the starvation of reads.

Unfortunately, not all is well. While the deadline I/O scheduler solves the read latency problem, the increased attention to read requests results in a seek storm. For each submitted read request, any pending writes are delayed, the disk seeks to the location of the reads and performs the operation, and then it seeks back and continues with the writes. This results in two seeks for each read request (or group of adjacent read requests) that are issued during write operations.

Compounding the problem, reads are issued in groups of dependent requests, as discussed. Not long after seeking back and continue the writes, another read request comes in and the whole mess is repeated.

The goal of a research interest in I/O schedulers, anticipatory I/O scheduling [iyer], is to prevent this seek storm. When an application submits a read request, it is handled within the usual expiration period, as usual. After the request is submitted, however, the I/O scheduler does not immediately return to handling any pending write requests. Instead, it does nothing at all for a few milliseconds (the actual value

is configurable; it defaults to 6ms). In those few milliseconds, there is a good chance the application will submit another read request. If any read request is issued to adjacent areas of the disk, the I/O scheduler immediately handles them. In this case, the I/O scheduler prevented another pair of seeks. It is important to note that the few milliseconds spent waiting is well worth the prevention of the seeks—this is the point of anticipatory I/O scheduling. If a request is not issued in time, however, the I/O scheduler times out and returns to processing any write requests. In that case, the anticipatory I/O scheduler loses and we lost a few milliseconds.

The key is properly anticipating the actions of applications and the filesystem. If the I/O scheduler can predict the actions of an application a sufficiently large enough percentage of the time, it can successfully limit seeks (which are terrible to disk performance) and still provide low read latency and high write throughput. A version of the deadline I/O scheduler, the anticipatory scheduler [piggin], is available in 2.5-mm which supports anticipatory I/O scheduling. The anticipatory I/O scheduler implements per-process statistics to raise the percentage of correct anticipations.

The results are very satisfactory. Under a streaming write, such as

```
while true; do
  dd if=/dev/zero of=file bs=1M
done
```

a simple read of a 200MB file completes in 45 seconds on 2.4.20, 40 seconds on 2.5.68-mm2 with the deadline I/O scheduler, and 4.6 seconds on 2.5 with the anticipatory I/O scheduler. In 2.4, the streaming write results in terrible starvation for the read requests. The anticipatory I/O scheduler results in nearly a ten-fold improvement in read throughput.

In 2.4, the effect of a streaming read upon a series of many small individual reads is also devastating. Perform a streaming read via:

```
while true
do
    cat big-file > /dev/null
done
```

and measure how long a read of every file in the current kernel tree takes:

```
    find . -type f -exec \
        cat '{}' ';' > /dev/null
```

2.4.20 required 30 minutes and 28 seconds, 2.5.68-mm2 with the deadline I/O scheduler required 3 minutes and 30 seconds, and 2.5.68-mm2 with the anticipatory I/O scheduler required a mere 15 seconds. That is a 121-times improvement from 2.4 to 2.5.68-mm2 with the anticipatory I/O scheduler.

How much damage does this benefit to read latency do to global throughput, though? It is clear that read throughput is improved, but at what cost to write requests and global throughput? Consider the inverse, under a streaming read such as:

```
while true
do
    cat file > /dev/null
done
```

A simple write and sync of a 200MB file takes 7.5 seconds on 2.4.20, 8.9 seconds on 2.5.68-mm2 with the deadline I/O scheduler, and 13.1 seconds on 2.5.68-mm2 with the anticipatory I/O scheduler. The 2.5 I/O schedulers are slower, but not overly so (certainly not to the degree read latency is decreased). This test does not show global throughput, though,

just write throughput in the presence of heavy reads. Since the streaming read above may operate much quicker, global throughput is often largely unchanged.

The anticipatory I/O scheduler is currently in the 2.5-mm tree. It is expected that it will be merged into the mainline 2.5 tree before 2.6.

5 Preemptive Kernel

5.1 Introduction

The addition of kernel preemption in 2.5 provides significantly lowered average scheduling latency and a modest reduction in worst-case scheduling latency. More importantly, introducing a preemptive kernel installs the initial framework for further lowering scheduling latency by allowing developers to tackle specific locks as the root of scheduling latency as opposed to entire kernel call chains. Conveniently, reducing lock hold time is also a goal for large SMP machines

5.2 Design of a Preemptive Kernel

Evolving an existing non-preemptive kernel into a preemptive kernel is nontrivial; the task is greatly simplified, however, if the kernel is already safe against reentrancy and concurrecny. Therefore, in the case of the Linux kernel, the safety provided by existing SMP locking primitives were leveraged to provide a similar protection from kernel preemptions. SMP spin locks were modified to disable kernel preemption in a nested fashion; after `n` spin locks, kernel preemption is not again enabled until the `n`-th unlock.

The `ret_from_intr` path (the architecture-dependent assembly which returns control from the interrupt handler to the interrupted code) was then modified to allow preemption even if returning to kernel mode. Thus, a task woken up in an interrupt handler (a common occurrence) can then run at the earliest possible moment, as soon as the interrupt handler returns. Consequently, a high priority task will preempt a lower priority task, even if the lower priority task is executing inside the kernel.

The preemption does not occur on return from interrupt, of course, if the interrupted task holds a lock. In that case, the pending preemption will occur as soon as all locks are released—again at the earliest possible moment.

5.3 Improved Core Kernel Algorithms

Changes to core kernel algorithms (primarily in the VM and VFS primarily) were made to improve fairness, provide a better bound on time complexity (and thus a bound on scheduling latency), and reduce lock hold time to take advantage of kernel preemption and reduce latency.

Some of the most important changes were to fix fairness issues in the VM, in code paths such as the page allocator. These changes prevent VM pressure caused by one process from unfairly affecting VM performance of other processes.

Many small changes were made to kernel functions in known high latency code paths in the kernel. These changes involved modifying the algorithm to have a minimized or fixed bound on time complexity and to reduce lock hold so as to allow kernel preemption sooner.

5.4 Reducing Scheduling Latency

Measurements of scheduling latency are highly dependent on both machine and workload (workload being a crucial element—one workload may show no perceptible scheduling la-

tency while another may introduce horrid scheduling latencies). Nonetheless, worst-case scheduling latencies of under 500 microseconds are commonly observed in 2.5.

Even on a 2.4 kernel patched with the preemptive kernel (which undoubtedly does not benefit from some of the algorithmic improvements in 2.5), a recent whitepaper [williams] noted a five-fold improvement in worst-case scheduling latency and a 1.6-time improvement in average case scheduling latency. A long-term test, part of the same whitepaper, testing the kernel for over 12 hours (to exercise many high scheduling latency paths) showed a reduction from over 200 milliseconds worst-case latency in the period to 1.5 milliseconds with a combination of the preemptive kernel and the low-latency patch [akpm2]. This drastic reduction in worst-case latency over a long period with a complex workload demonstrates the ability of the preemptive kernel and optimal algorithms to provide both excellent average and worst-case scheduling latency.

One useful benchmark is the Audio Latency Benchmark [sbenno], which simulates keeping an audio buffer full under various loads. A test of a 2.4 kernel vs. a 2.4 preemptive kernel shows a reduction in worst-case scheduling latency from 17.6 milliseconds to 1.5 milliseconds [rml]. The same test on 2.5.68 yields a maximum scheduling latency of 0.4 milliseconds.

On a modern machine, scheduling latency is low enough to prevent any perceptible stalls during typical desktop computing and multimedia work.

Further, the 2.5 kernel provides a base sufficient for guaranteeing sub one millisecond worst-case latency for demanded embedded and real-time computing needs.

6 Acknowledgments

I would like to thank the OLS program committee for providing the opportunity to write this paper and MontaVista Software for providing the means by which I work on the kernel.

Andrew Morton deserves credit for an abnormally large amount of the interactivity work which went into the 2.5 kernel. Jens Axboe was the primary developer of the deadline scheduler. Nick Piggin was the primary developer of the anticipatory scheduler, which is based on the deadline scheduler. Ingo Molnar was the primary developer of the O(1) scheduler. Various others played significant roles in the design and implementation of other related kernel bits.

References

[akpm1] Andrew Morton, *Patch: read-latency2*, `http://www.zip.com.au/~akpm/linux/patches/2.4/2.4.19-pre5/read-latency2.patch`.

[akpm2] Andrew Morton, *Patch: low-latency*, `http://www.zipworld.com.au/~akpm/linux/schedlat.html`.

[arcangeli] Andrea Arcangeli and Jens Axboe, *Source: 2.4 Elevator* , `linux/drivers/block/elevator.c`.

[axboe1] Jens Axboe, *Email: [PATCH] deadline io scheduler*, `http://www.cs.helsinki.fi/linux/linux-kernel/2002-38/0912.html`.

[axboe2] Jens Axboe, *Source: Deadline I/O Scheduler*, `linux/drivers/block/deadline-iosched.c`.

[iyer] S. Iyer and P. Druschel, *Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O*. ACM Symposium on Operating System Principals (SOSP), 2001.

[mingo1] Ingo Molnar, *Source: O(1) scheduler*, `linux/kernel/drivers/sched.c`.

[mingo2] Ingo Molnar, *Patch: sched-2.5.64-D3*, `http://www.kernel.org/pub/linux/kernel/people/akpm/patches/2.5/2.5.68/2.5.68-mm2/broken-out/sched-2.5.64-D3.patch`.

[piggin] Nick Piggin and Jens Axboe, *Source: Anticipatory I/O Scheduler*, `linux/drivers/block/as-iosched.c`.

[rml] Robert Love, *Lowering Latency in Linux: Introducing a Preemptible Kernel*, Linux Journal (June 2002).

[sbenno] Benno Senoner, *Audio Latency Benchmark*, `http://www.gardena.net/benno/linux/audio/`.

[williams] Clark Williams, *Linux Scheduler Latency*, Whitepaper, Red Hat, Inc., 2002.

Machine Check Recovery for Linux on Itanium® Processors

Tony Luck
Intel Corporation
Software and Solutions Group
tony.luck@intel.com

Abstract

The Itanium[1] processor architecture provides a machine check abort mechanism for reporting and recovering from a variety of hardware errors that may be detected by the processor or chip set. Simple errors such as single bit ECC may be corrected transparently to the operating system by hardware and firmware, but more complex errors where data has been lost require OS intervention. In cases where the OS can reconstruct the lost data, then execution can continue transparently to the application layer, otherwise the OS may decide to sacrifice affected user processes to allow the system to continue. This paper describes how Linux can recover from TLB errors without affecting applications, and also how Linux can recover from certain memory errors at the expense of terminating user processes.

1 Introduction

Server systems are not just about increased speed and capacity. They must also provide better reliability than their desktop and mobile cousins. The Intel Itanium architecture includes a machine check architecture that provides the mechanism to detect, contain and in many cases correct processor and platform errors.

[1]Itanium is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries. Other names and brands may be claimed as the property of others.

2 Source of errors

There are several sources of errors within a computer system:

1. Electrical supply line fluctuations

   Can be mitigated with a high quality power supply with surge suppression capability.

2. Static electricity

   Effects can be lessened by good enclosure design and special static reducing floor covering.

3. Heat

   Reduced by good thermal design of system enclosure and air-conditioning of the computer room. Hardware may also detect excess temperature and automatically switch to mode where less power is dissipated (e.g. a lower clock speed and/or voltage, or a reduction in the number of available functional units for retiring instructions).

4. Interaction with high energy particles due to radioactive decay

   Can be reduced by careful selection of the materials used to build the system (e.g. use of ultra pure dopants consisting of only stable isotopes), and addition of shielding.

5. Interaction with high energy particles from cosmic ray showers in the earth's atmosphere

   Reduce by shielding (locate computer room in the basement) and avoiding high altitude locations for computer systems (intensity in Denver, altitude 5280 feet, is over four times higher than at sea level locations).

Why is this important? Other aspects of hardware are getting more reliable, but as feature size is reduced they become more susceptible to particles (as lower energy particles are capable of flipping bits). Software is getting more reliable too: we cannot just blame crashes on the OS. Clusters of computers multiply the error rates. A mean time between failure of several years for a processor isn't too bad of a problem if you only have one processor, when you build a cluster of several thousand processors, then you have a big problem.

For each of the error sources listed above I have suggested methods by which the error rate may be reduced, but it may be impractical or too expensive to reduce all of these errors to insignificant levels, hence computer systems must be designed to detect, isolate and recover from errors when they do occur.

3 Itanium Machine Check Architecture

The Itanium machine check architecture provides a framework in which diverse types of errors can be handled in a logical and consistent way.

3.1 Error severity

Errors are divided into three categories:

1. Corrected errors are those that are repaired by hardware or by firmware. In either case an interrupt (CMCI[2] for processor errors, CPEI[3] for platform errors) may be raised for logging purposes.

   Examples of this type of error are a correctable single-bit ECC error in the processor cache or a correctable single-bit ECC error on the system bus.

2. Recoverable errors involve some loss of state. They require operating system intervention to determine whether it is possible for the system to continue operation.

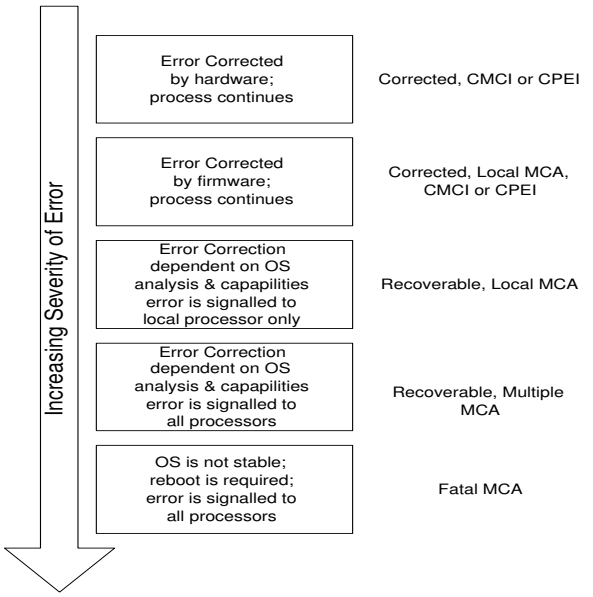
   An example of a recoverable error is one where incorrect data is about to be passed to a processor register (e.g. from a load from memory with a multi-bit ECC error).

3. Fatal errors cannot be corrected. A system reboot is required.

   On this kind of error, the processor generates a signal that is broadcast on the system bus (called BINIT#) that causes the processor to discard all in-flight transactions to prevent error propagation. The error is fatal because there is no way to recover the state of the discarded bus transaction, hence the need for a system reboot.

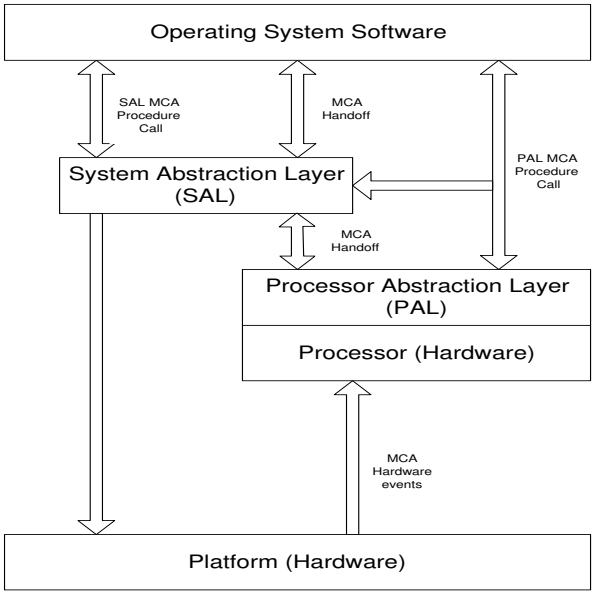
   An example of a fatal error is a processor time-out (when the processor has not retired any instructions after a certain time period).

[2] Corrected Machine Check Interrupt
[3] Corrected Platform Error Interrupt



3.2 Control flow

This diagram shows how the hardware, processor, processor abstraction layer, system abstraction layer and operating system interact to handle machine checks:



At the hardware level, hardware redundancy (parity and ECC) is used to detect and possibly correct errors during program execution. In some correctable cases the hardware may simply fix the error on the fly and raise a corrected error interrupt to allow logging of the event. In other cases the processor will save all machine state and pass control to the PAL (Processor Abstraction Layer) code at the PAL_MCA entry point for analysis and further processing by firmware. Different members of the Itanium processor family may make different choices about whether to fix errors in hardware or pass responsibility up to firmware.

Entry to the PAL is made in physical mode with caches disabled. This provides the option for the PAL to handle some types of errors detected in the cache, which is useful since on chip caches make up so much of the die area of modern processors that they are statistically one of the most likely forms of errors (from the processor itself…memory errors from a large array of DIMMS are probably the most likely system-wide source of errors). If the error is corrected at this point, then execution can be resumed, but again an interrupt is raised to allow the error to be logged.

Next layer in the firmware stack is the SAL (System Abstraction Layer) which is responsible for components outside of the processor itself (e.g. chip set, memory, I/O bus bridges etc.). The processor is still executing in physical mode (MMU disabled), we may have enabled the caches by this point (depending on whether the SAL machine check entry point is located in cacheable or uncacheable memory). SAL code examines the details of the error and determines whether it can fix it without causing loss or corruption of any data. As above, if the error is fixed, then execution resumes with a pended interrupt to log the details.

At the highest level is the operating system. Errors that have not been corrected at lower levels, but which leave the processor in an internally consistent state, are still recoverable. These can be passed to the operating system if it is interested in trying to recover. An op-

erating system indicates it is capable and willing to handle machine check errors by registering an entry point with the SAL using the SAL_SET_VECTORS call. Note that the operating system entry point must be a physical address because it is possible that the error to be handled is in the memory management H/W, hence the MMU must still be disabled when the SAL transfers control to the operating system entry point. The physical entry point means that the code to service machine check abort must either be position independent, or at least very aware of relocation issues since it will not be executing at the kernel's linked address. Also the SAL_SET_VECTORS call allows the operating system to provide the length and a simple byte checksum of the code so that the SAL may validate that the routine has not been corrupted.

4 Reporting corrected errors

As mentioned above, errors that are corrected by hardware, PAL, or SAL may be reported to the operating system by means of a CMCI (Corrected Machine Check Interrupt) for errors inside the processor, or a CPEI (Corrected Platform Error Interrupt) for system errors outside of the processor. The operating system may choose to disable these interrupts and periodically poll the SAL to see if any errors have been corrected. It might do this to avoid being swamped by corrected error interrupts (e.g. a stuck data line causing hard single-bit memory errors across a wide range of physical addresses, we would like to ensure that the operating system can continue to make forward progress).

Whether the operating system takes interrupts or polls, once a corrected error record is found the OS can retrieve the whole record, parse it to find any useful information, and then output details to its own log. As a final step in error reporting the operating system must request the SAL to clear the error record from non-volatile memory (to ensure that space is available for future errors to be logged).

User level tools could be written to analyze the operating system logs to check for patterns that may be predictive of future hard errors in components that generate a high level of soft errors.

5 Poisoned memory

Another feature of the machine check architecture is the concept of "poisoned memory." This allows a platform the option of deferring error processing in some circumstances. Suppose a modified cache line is being written back to memory when an uncorrectable error is detected in the data contained in the cache line. The current execution state of the processor probably has no connection with this data, so signaling a machine check abort at this time may be an over-reaction to the situation. Instead, the data can be written to memory together with an indication that it is corrupt (the "poison" flag, typically indicated with bad ECC bits). A regular CMC interrupt is then raised, and the OS is allowed to examine the situation later. Deferring the error in this case may be useful, because it is not certain that the corrupted data will ever be needed (e.g., the page to which the cache line belongs may already have been freed by the operating system, or the word that was corrupted in the cache line may have only been present in the cache because of false sharing).

Note that in the "read" case data poisoning does not apply, the processor will immediately begin MCA processing.

If the operating system writer is concerned that the poisoned data may be consumed before the interrupt is processed, there is an option to promote CMCI to MCA to allow immediate ac-

tion to be taken (though this option applies to all CMCI, not just to those caused by poison data).

6 Operating system examples

Here are some example cases of recoverable errors where the operating system can intervene to recover from an error that has been detected and reported using the above mechanism.

6.1 TLB translation register error

The Translation lookaside buffer in the Itanium processor is not only divided into separate structures for instruction and data access, it is also conceptually divided into two types of entries.

1. Translation cache entries can freely be replaced as new mappings are added.
2. Translation register entries are locked into the TLB. These are used to "pin" translations for critical regions to ensure that a TLB miss will never occur for a virtual address mapped by a locked entry.

Errors in translation cache can be trivially handled by H/W or PAL by simply discarding corrupted entries from the cache. This will only affect performance, if the discarded entry is needed, the processor will simply reload using the normal TLB miss execution path. However, if an error is detected in one of the translation registers, then the fault cannot be handled by F/W, PAL or SAL, since the entry cannot just be dropped and the firmware does not have enough information to reconstruct the damaged entry. So the error is propagated up to the OS which needs to reload the errant register. The ia64 Linux implementation uses the following TR registers:

ITR(0) maps one large kernel page[4] as the kernel text (code)

DTR(0) maps one large kernel page as the kernel data

ITR(1) maps one granule[5] for PAL

DTR(1) maps one page[6] for per-cpu area

DTR(2) maps one granule for kernel stack

The first four of these are loaded during kernel initialization, and are never changed, so it is a simple matter to add code to save the correct values for these registers in memory, so that the MCA handler can reload when needed. The last, mapping the kernel stack, can potentially be reloaded on every context switch (actual reloads occur when the task structure for the new process is in a different large kernel page). The Linux kernel uses one of the supervisor mode registers (ar.k4) to keep track of which large kernel page is currently mapped, and this register can be used to reconstruct the DTR(2) value during the MCA handler.

Although the SAL error record generated for an error in a translation register provides information on which register(s) have errors, it would require a large amount of code to retrieve and parse the error record to determine exactly which registers need to be reloaded. All this code would have to run in physical mode (remember that SAL passes control to the operating system MCA entry point in physical mode, and it is not possible to transition to virtual mode for this particular error, since we know that one or more of the translation registers are corrupt). The simple solution to this issue is to have the MCA TLB recovery code purge and reload all of the TR registers in the

[4] Kernel is always mapped with a single 64MB page

[5] another type of large kernel page, configurable as either 16MB or 64MB

[6] PAGE_SIZE on 2.4, 64k on 2.5

physical mode code. Then we can safely transition to virtual mode to retrieve and examine the record from the SAL error log to report the actual register(s) that were affected by the error.

6.2 Multi bit ECC error in memory

In this case some data has been irretrievably lost, so the operating system cannot escape from this situation unscathed. The basic strategy for the OS is to identify the address of the memory that reported the error. If the memory is owned by the kernel, this will currently be reported as a fatal error by the Linux MCA handler and the OS will reboot (to be strictly accurate the Linux MCA handler will return to SAL with a request to reboot, and the error will be reported by Linux after the reboot when the SAL error record is retrieved from NVRAM). If the memory is allocated to one or more user processes, the processes can be sacrificed to allow the system to continue running (just as the OOM killer will terminate processes when Linux runs out of memory). Life is rarely that simple. In this case a complication is that machine checks are not reported synchronously to the instructions that trigger them, they may be deferred for a long time (in the worst case until the values are consumed). The processor precisely identifies the location at which the fault is detected, but does not provide information about the point at which the fault occurred. The processor does not automatically [7] raise machine checks across privilege transitions from user to kernel mode and vice versa, so it is possible that an error caused in one privilege state will be reported in the other state. Easy cases:

a) fault triggered in user mode is reported in user mode—kill process

b) fault triggered in kernel mode is reported in kernel mode—system reboot

Harder cases:

c) fault triggered in kernel mode is reported in user mode—this is a very subtle case. At first sight it appears that our error handler cannot do the right thing. This case is indistinguishable from case ‘a’ above, since we only have precise information about where the error was detected. But if the error occured in some kernel data, just killing the process is not the correct action. It would leave the kernel running with a corrupted data structure! However, we are saved by the fact that the error is detected when the user process tries to consume the data, and we know that only a buggy kernel would leak details of a kernel data structure to user mode. So we can assume that any such error that happened in kernel mode and was detected in user mode must have occurred when the kernel was restoring registers that belonged to the user process. Thus killing the process is sufficient to contain the error.

d) fault triggered in user mode is reported in kernel mode—sadly this will cause a reboot because it isn’t possible to distinguish this from case ‘b’ above (though it might be possible to eliminate many of these cases by a special case for faults reported during the code that saves user registers).

6.3 PCI errors

The Itanium processor family machine check architecture provides a framework for reporting platform errors, such as PCI bus errors. From an OS perspective, these may be far more complex to handle. Issues are:

[7] There is a PAL call PAL_MC_DRAIN to do this, but it would be a major performance issue to use this on every transition

1. We would like a framework that is minimally invasive to the existing driver model.

2. Linux is just starting to get support for hot-add and hot-remove of devices, but it is a big step from there to support surprise removal of devices when errors occur.

3. Even in the case of transient errors, recovery may be complicated by the firmware on the card, which typically has been written with an expectation that the system will reboot after an error.

7 Acknowledgments

Thanks to all the people at Intel who spent time reviewing this paper and providing invaluable feedback.

References

[Menyhárt] Z. Menyhárt and D. Song, *OS Machine Check Recovery on Itanium Architecture-base Platforms*, Intel Developer Forum, Fall 2002

[Ziegler] J.F. Ziegler, *Terrestrial cosmic ray intensities*, IBM Journal of Research and Development, Volume 42, Number 1, 1998

[SDV] Intel, *Intel Itanium Architecture Software Developer's Manual, Volume 1–3*

[EHG] Intel, *Itanium Processor Family Error Handling Guide*, August 2001

[SAL] Intel, *Itanium Processor Family System Abstraction Layer (SAL) Specification*, November 2002

Low-level Optimizations in the PowerPC Linux Kernels

Paul Mackerras
IBM Linux Technology Center OzLabs
paulus@au1.ibm.com

Abstract

We examine three low-level optimizations in the Linux® kernel for 32-bit and 64-bit PowerPC®, relating to cache flushing, memory copying, and PTE (page table entry) management. Benchmarking and profiling were used to identify areas where optimizations could be performed and to identify whether the optimizations actually improved performance. The cache flushing and memory copying optimizations improved performance significantly, whilst the PTE management optimization did not.

1 Introduction

The optimizations presented in this paper represent some of the results of the continuing effort to make the Linux kernel run better and faster on PowerPC processors, both 32-bit and 64-bit. The optimizations here are low-level optimizations that are specific to PowerPC processors, aimed at decreasing the overhead of some of the fundamental operations relating to maintaining the consistency of the instruction cache with memory, copying memory, and managing page table entries.

Subsequent sections present measurements of performance using benchmarking and kernel profiling. Benchmarking is the process of measuring performance by running a specific program or set of programs to measure how quickly certain operations are performed. Two different benchmarks of different styles are used here:

- LMBench™ is a micro-benchmark suite originally written by Larry McVoy. It measures the speed of a broad range of individual kernel operations such as forking processes, reading and writing data to/from disk, transferring data over a socket, etc. See `http://www.bitmover.com/lmbench/` for details.
- For an application-level benchmark to measure the overall speed of a process that involves a range of kernel activities, we use the process of compiling the Linux kernel, and measure the time taken using the `time(1)` command. We used the same source tree (from Linux version 2.5.25) and configuration for all tests so that the results are comparable with each other. A kernel compilation tends to exercise a range of kernel functions including forking processes, starting new processes, reading and writing files, mapping in pages of memory on demand, and so on.

Profiling measures the time spent in individual kernel procedures while the kernel is performing some tasks. The form of profiling used in

this paper is that where a periodic interrupt is used to obtain a statistical sample of the total time spent executing each instruction in the kernel. When the periodic interrupt is taken, the handler examines the instruction pointer where the interrupt occurred, uses that to index into an array, and increments that array element. Over time this builds up a histogram of where the kernel is spending its time. A post-processing tool converts that histogram into a total count for each procedure in the kernel.

This can be a very powerful tool for analysing kernel performance provided that its limitations are kept in mind. First, because the data is a statistical sample, it can be quite noisy. Secondly, it does not measure the execution time for code that runs with interrupts disabled (unless some kind of non-maskable interrupt can be used). Instead, time spent with interrupts disabled tends to get attributed to the point where interrupts are re-enabled.

Three machines were used for the measurements reported here:

1. An Apple® PowerBook® G3 laptop with a 400MHz PowerPC 750™ processor, separate 32kB level 1 data and instruction caches, unified 1MB level 2 data cache, and 192MB of RAM. This is a 32-bit machine.

2. An IBM® pSeries™ model 650 computer with eight 1.45GHz IBM POWER4+™ processors, 32kB level 1 data cache and 64kB level 1 instruction cache per processor, 1.5MB level 2 cache per 2 processors, and 8GB of RAM. This is a 64-bit machine.

3. An IBM "Walnut" embedded evaluation board with a 200MHz IBM PowerPC 405GP processor, 8kB level 1 data cache, 16kB level 1 instruction cache and 128MB of RAM. This is a 32-bit machine.

2 Cache flushing optimizations

In the PowerPC architecture, the instruction cache is not required to snoop changes to the contents of memory, either by stores from this or another CPU, or by DMA from an I/O device. Instead, software is required to maintain the coherency of the instruction cache, using the `dcbst`—data cache block store instruction and the `icbi`—instruction cache block invalidate instruction. These instructions can be executed at user level, and self-modifying code is required to use them after it has written instructions to memory before those instructions are executed.

The kernel uses these instructions extensively to make sure that pages that are mapped into a user process's address space can be executed safely. User code assumes that the instruction cache is consistent with memory for pages that are supplied by the kernel on demand. Thus it is the kernel's responsibility to perform the cache flushing instructions on a page of memory before mapping it into a process's address space if there is a possibility that the instruction cache is incoherent with memory for that page. If this is not done properly, the symptom is usually that the process will get a segmentation violation or illegal instruction exception, since it is not executing the instructions that it should.

Note that almost all PowerPC implementations have caches that are effectively physically addressed—usually virtually indexed, physically tagged, set associative, with the set size no greater than the page size, so no aliasing occurs. The IBM POWER4™ processor has a virtually indexed direct-mapped instruction cache, but obviates the potential problems that this could cause by having the `icbi` instruction clear all 16 cache blocks where a given block of memory could be cached. There are also some embedded PowerPC implementa-

tions that have virtually indexed and tagged instruction caches, and these require quite different cache management and are not considered in this paper.

2.1 Initial implementation

The Linux generic virtual memory (VM) system provides a number of hooks that architecture code can define in order to do architecture-specific cache and TLB management where necessary. One of these is called `flush_page_to_ram`, and it is called at several points in the VM code when pages are mapped into a user process's address space (e.g., `do_no_page`, `do_anonymous_page` etc.). In older kernels the `flush_page_to_ram` hook was used on PowerPC to perform the cache flush on the page in order to ensure that the instruction cache was consistent for the page. This reliably ensured that the instruction cache did not contain stale data for the pages that processes see, but had a considerable performance penalty. The "Original" column of Table 1 shows the results of a kernel profile on a kernel which uses `flush_page_to_ram` to ensure instruction cache coherency. These results were obtained on the 400MHz G3 PowerBook machine compiling a kernel (3 times over). The kernel spends more time in `flush_dcache_icache`, which performs the flushing function of `flush_page_to_ram`, than any other kernel procedure. Clearly `flush_page_to_ram` is a good candidate for optimization.

2.2 Optimized implementation

A large part of the reason why the kernel is spending so much time in `flush_dcache_icache` is that it is doing unnecessary flushes. If the same program is executed many times in different processes, the kernel will call `flush_page_to_ram` on each page of the program executable each time it is mapped into a process's address space. However, once the flush has been done, the instruction cache is consistent for that page (provided that the page is not modified), and the flush doesn't need to be done when the page is subsequently mapped into other processes' address spaces.

Procedure	Original	Optimized
`flush_dcache_icache`	6763	2974
`ppc6xx_idle`	2238	2468
`do_page_fault`	857	667
`copy_page`	537	390
`clear_page`	523	509
`copy_tofrom_user`	356	299
`do_no_page`	231	129
`add_hash_page`	220	92
`flush_hash_page`	195	191
`do_anonymous_page`	194	224

Table 1: Kernel profiles before and after cache-flush optimization

A solution to this problem was suggested by David Miller. He suggested using a bit, called `PG_arch_1`, in the `flags` field of the `page_struct` structure for each page, to indicate whether the instruction is consistent with memory for the page. This bit is cleared when the page is allocated. We use this to indicate that the instruction cache may be inconsistent. When the flush is done on the page, we set the bit, indicating that the instruction cache is now consistent for the page. Subsequently, if the bit is already set, the flush does not need to be done.

David Miller also requested that we use the `flush_dcache_page` and `update_mmu_cache` hooks rather than `flush_page_to_ram`, since more information is provided to the architecture code in the calls to `flush_dcache_page` and `update_mmu_cache`, and `flush_page_to_ram` is deprecated. The VM system calls `flush_dcache_`

`page` when a page which may be mapped into a process's address space is modified by the kernel. `update_mmu_cache` is called when a page is mapped into a process's address space. In our implementation, `flush_dcache_page` clears the `PG_arch_1` bit, and `update_mmu_cache` does the flush (by calling `flush_dcache_icache`) if the `PG_arch_1` bit is clear, and then sets it.

The results are shown in the "Optimized" column in Table 1. Clearly the time spent flushing the cache has decreased dramatically, although it is still significant. The system time for the compilation decreased from 46.0 seconds to 29.9 seconds. The user time was not significantly different (301.9 seconds vs. 300.2 seconds). The overall speedup was 5.1%. (Note that kernel profile measurements are quite noisy, and the other differences between the columns in Table 1 are not necessarily significant.)

Table 2 shows an excerpt from the before-and-after LMBench results. (See `ftp://ftp.samba.org/pub/paulus/ols2003/lmb-argo-flush` for the full summary of results.) The optimization has produced worthwhile improvements in the fork, exec and shell process items. The first line shows the unoptimized results, and the second line shows the optimized results.

2.3 Further optimizations

The implementation in the previous section aimed at minimizing the number of flushes while still making sure that the instruction cache was consistent with memory for each page mapped into the process's address space. This includes anonymous pages and pages that are copied as a result of a write to a copy-on-write page, as well as page-cache pages. (A copy-on-write page is one which is mapped with a private writable mapping, including anonymous pages which are shared after a fork.)

Part of the reason that it is necessary to ensure consistency of the instruction cache is that the PowerPC architecture, as originally defined, doesn't provide any way to prevent a process from executing code from a readable page. That is, there is no execute permission bit in the page table entries (PTEs). If there was a way to trap attempts to execute from a page, it would be possible to defer the flush until a process first executed instructions from the page. That way, it would be possible to avoid the flush altogether on anonymous pages which are only used for data, not for code.

Embedded PowerPC implementations, such as the IBM PPC405, don't follow the original PowerPC memory management unit (MMU) architecture, but instead have a software-loaded TLB with a unique PTE format. The PTE format for the PPC405 includes an execute-permission bit. Also, the POWER4 processor uses one of the previously-unused bits in the PTEs as a no-execute bit.

We implemented an optimization on the PPC405 where the pages are not flushed in `update_mmu_cache`. Instead, if the `PG_arch_1` bit is clear, we clear the execute-permission bit in the PTE mapping the page. There is an added check in `do_page_fault` for an attempt to execute from a page with the execute-permission bit clear. In that case, we do the flush on the page and then set the execute-permission bit.

The kernel profiles shown in Table 3 show that the number of counts recorded in `flush_dcache_icache` while compiling the test kernel twice dropped from 1685 to 31. Thus the time spent doing cache flushes for instruction cache consistency has become negligible. The system time for a kernel compile was reduced by 147.7 seconds to 139.0 seconds, a de-

```
Processor, Processes - times in microseconds - smaller is better
----------------------------------------------------------------
Host                 OS  Mhz null null      open selct sig  sig  fork exec sh
                             call  I/O stat clos TCP   inst hndl proc proc proc
------ ------------- ---- ---- ---- ---- ---- ----- ---- ---- ---- ---- ----
argo    Linux 2.5.66  400 0.35 0.76 3.90 5.34  39.1 1.67 6.64 795. 5065 23.K
argo    Linux 2.5.66  400 0.35 0.76 3.88 5.33  40.3 1.67 6.13 659. 2254 11.K
```

Table 2: LMBench results before and after cache-flush optimization

Procedure	Orig.	Optim.
`ProgramCheckException`	4766	4685
`do_mathemu`	4475	4526
`ide_intr`	1745	1729
`flush_dcache_icache`	1685	31
`record_exception`	1369	1448
`copy_tofrom_user`	1357	1325
`do_page_fault`	1027	1035
`ret_from_except_full`	774	775
`fmul`	731	721
`fsub`	658	629

Table 3: Kernel profiles before and after execute-permission optimization

crease of 5.9%. The user time was not significantly different: 1460.6 seconds for the optimized kernel vs. 1457.9 seconds for the unoptimized kernel. The overall time for the kernel compilation was reduced by 0.37%. The reduction is less than might have been expected because a significant amount of the system time was spent in the kernel floating-point emulation routines (the PPC405 does not implement floating point instructions in hardware).

3 Memory copying

Copying memory is a fundamental operation in the kernel, used for:

- Copying data to or from a user process (e.g., for a `read` or `write` system call)
- Copying pages of memory, in particular for write faults on copy-on-write pages
- Copying other data structures within the kernel.

Separate procedures are used for these three operations: `copy_tofrom_user`, `copy_page` and `memcpy` respectively. The profiles in Table 1 show that `copy_tofrom_user` and `copy_page` are among the top ten most time-consuming operations in the kernel. These routines are thus a candidate for optimization.

However, these routines are already well optimized in the 32-bit PowerPC kernel for most 32-bit PowerPC implementations. In the 64-bit kernel, it becomes possible to use 64-byte loads and stores to move more data per instruction. This is easy in the `copy_page` case, since the source and destination addresses are page-aligned. However, in `copy_tofrom_user` and `memcpy`, where the source and destination are not necessarily 8-byte aligned, it becomes more complicated.

PowerPC processors generally handle most 2-byte and 4-byte unaligned loads and stores in hardware, without generating an exception. Older processors would generate an exception if the access crossed a page boundary, whereas most newer processors handle even that case in hardware. However, 64-bit PowerPC processors typically generate an exception on an 8-byte load or store if the address is not 4-byte aligned. The kernel has an alignment exception handler that emulates the load or store and allows the program to continue.

When copying memory and the source and/or destination addresses are misaligned, we generally copy a small number of bytes, one at a time, in order to get to an aligned destination address. If the source address is then misaligned (that is, the bottom 2 bits of the address are non-zero), there are two alternative strategies to handling the misalignment:

1. Use 32-bit or 64-bit loads and stores, ignoring the misalignment. In this case we will have misaligned load addresses and aligned store addresses.

2. Do loads with aligned addresses and use shift and OR instructions to shuffle the bytes into the correct positions to be stored to an aligned address.

For current PowerPC implementations, it turns out that while misaligned 32-bit loads are slower than aligned 32-bit loads, they are still faster than aligned 32-bit loads plus the extra instructions needed to shuffle the bytes into position. For this reason, `copy_tofrom_user` and `memcpy` in the 32-bit kernel use unaligned loads in a relatively simple loop. However, the situation is different for 64-bit loads. Since every misaligned 64-bit load will cause an exception, it is much faster to do the aligned loads and shuffle the bytes.

In fact, the behaviour of the processor on unaligned loads and stores is only one of many architectural and implementation characteristics that affect how an optimum memory copying routine should be written. Some of the others are:

- The number of levels in the storage hierarchy and the latency to each level;

- Presence or absence of automatic hardware prefetch mechanisms;

- Presence or absence of instructions to provide cache prefetch hints to the processor;

- Load-use penalty, that is, how many other instructions should be placed between a load and the store (or other operation) which uses the data from the load, so that the processor does not need to stall the store until the data from the load is available;

- Ability of the processor to issue instructions out of order, so that later instructions are not blocked by earlier instructions which do not have all their operands available;

- The penalty incurred for conditional branches (if this is large then there is an advantage to unrolling loops);

- Extended instruction sets such as Altivec on PowerPC, MMX/SSE on x86, or the VIS instructions on SPARC64, which provide the ability to operate on larger units of data (typically 128 bits).

Given how many factors can affect memory copying performance, it is not surprising that memory copy routines can become quite large and complicated, reaching tens of thousands of lines of assembly code on some architectures. Other factors that affect the performance of a memory copy routine include the size of the region to be copied, and whether the source and/or destination regions are already present in the processor's caches. Some optimizations, such as loop unrolling, might improve performance dramatically for larger copies (i.e., several cache lines or larger) but hurt performance for small copies by increasing the setup costs. Similarly, some optimizations, such as using extended instruction sets, might improve performance dramatically when all the source data is present in the level-1 data cache, but have no

effect or actually reduce performance when the data has to be brought in from main memory.

Thus it is interesting to know whether the kernel routinely does large copies, and whether they are misaligned or not. To test this, we added histogramming functions to `copy_tofrom_user` and `memcpy` in a 64-bit kernel. The results can be summarized as follows:

- 98% of calls to `memcpy` were for less than 128 bytes (one cacheline).
- 13% of calls to `memcpy` were not 8-byte aligned.
- 84% of calls to `copy_tofrom_user` were for less than 128 bytes, and 95% were for less than 512 bytes. Of the remainder, most were page-sized (4096 bytes) and page-aligned.
- 43% of calls to `copy_tofrom_user` were not 8-byte aligned.

These results indicate that it is important to optimize for the small-copy case, particularly for `memcpy` but also for `copy_tofrom_user`, and that performance on unaligned copies is important for `copy_tofrom_user`. The one large-copy case which is worth optimizing for is the case of copying a whole page, both in `copy_page` and also to a lesser extent in `copy_tofrom_user`.

3.1 Optimized POWER4 memory copy

On POWER4 the factors that need to be taken into account include the following:

- POWER4 aggressively executes instructions out of order and uses register renaming to avoid false dependencies between instructions.
- POWER4 includes automatic prefetch hardware which detects sequential memory accesses and prefetches cache lines which are likely to be needed in the near future.
- Correctly predicted conditional branches incur a one cycle penalty.
- The level-1 data cache on POWER4 is a write-through cache, thus all stores go through to the level-2 cache. The L2 cache is organized as three interleaved banks. Each bank has its own store queue.

The effect of the first three points is that while some degree of loop unrolling is beneficial, it is not necessary to aggressively unroll the main copy loop or to make sure that many instructions intervene between a load and the instruction that uses the result.

Because of the interleaved nature of the L2 cache, the optimum pattern of stores is one where stores go successively to each bank of the level 2 cache. In fact the best performance for large copies is obtained with a loop that works on six cachelines at a time, so that the loads and stores are interleaved across six cachelines.

Based on these considerations, the author developed optimized `copy_tofrom_user`, `copy_page` and `memcpy` implementations for POWER4, with the following characteristics:

- `copy_tofrom_user` detects page-sized page-aligned copies and calls a routine similar to `copy_page` for them. For other copies, it proceeds with an algorithm similar to `memcpy` below. (The main difference between `copy_tofrom_user` and `copy_page` or `memcpy` is that `copy_tofrom_user`

has to cope gracefully in the case where the source or destination address cannot be accessed, for example if a bad address is given to a system call.)

- The main loop of `copy_page` works on 6 cachelines at once, and contains 18 load and 18 store instructions, in three groups of 6 stores followed by 6 loads.
- `memcpy` has three main loops, each of which do two aligned 64-bit loads and two aligned 64-bit stores. One loop is for the case where the source and destination are 8-byte aligned with respect to each other, and the other two are for the misaligned case. These two loops are slightly different to handle an even or odd number of 64-bit doublewords to be transferred (excluding any bytes copied initially to get the destination 8-byte aligned).

These routines were tested on the 1.45GHz POWER4+ system using the same kernel compile test used in the previous section. The optimized copy routines were compared with the `copy_tofrom_user` and `memcpy` routines from the ppc32 kernel (thus using only 32-bit loads and stores) and a simple `copy_page` implementation which has two 64-bit loads and stores per iteration in its main loop.

Figure 1 shows selected LMBench results in graphical form. (See `ftp://ftp.samba.org/pub/paulus/ols2003/lmb-power4-copy` for the full summary of results.) From these it is evident that the optimized version is indeed noticeably faster than the unoptimized versions.

However, the results from the kernel compile test were more equivocal: the system time was reduced from 8.30 seconds to 8.19 seconds (a 1.3% improvement). When the user time (78.84 seconds in both cases) is added in, the overall improvement was only 0.13%.

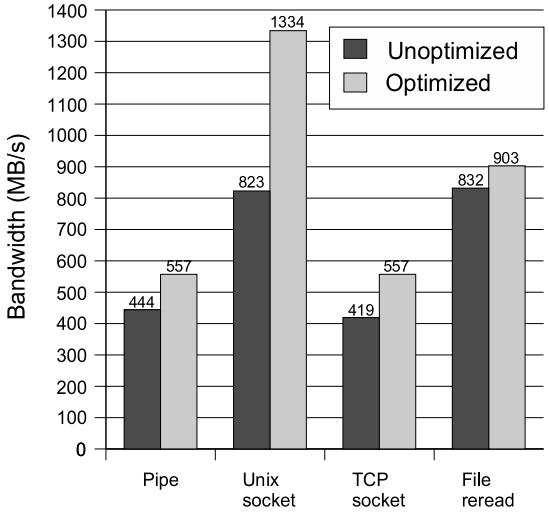


Figure 1: Results of memory copy optimization on 64-bit POWER4+.

4 PTE management

4.1 Introduction

In the PowerPC architecture, page table entries (PTEs) are stored in a hash-table structure which is accessed by the memory management unit (MMU) hardware. The hash table is divided into groups of 8 entries. Each group has an index between 0 and $N - 1$, where N is the number of groups in the table (N must be a power of 2). When the MMU needs to find the PTE for a given virtual address, it first computes a hash index using an XOR-based hash function on the virtual address. The hash index identifies one group, called the primary group for the virtual address, which is fetched and searched for a PTE which matches the virtual address. The PTE includes part of the virtual address so that a match can be verified. If no matching PTE is found, a secondary hash value is formed by subtracting the original hash value from $N-1$. This identifies the secondary group for the virtual address, which is also searched

for a matching PTE.

In contrast, the Linux virtual memory (VM) system uses a two-level or three-level tree structure for storing PTEs. It is the responsibility of the PowerPC-specific parts of the kernel to keep the MMU hash table accessed by the hardware synchronized with the Linux page table trees. Essentially, the MMU hash table is used as a large level-2 translation lookaside buffer (TLB).

To avoid confusion, we use the term HPTE (Hashtable PTE) to refer to a PTE in the MMU hash table, and LPTE (Linux PTE) to refer to a PTE in the Linux page table trees. The HPTE and LPTE formats are different, although related.

When a new LPTE is created, a corresponding HPTE must be created before the page can be accessed. This can be done in the `update_mmu_cache` hook or on demand when the page is first accessed.

Similarly, when an existing LPTE is invalidated, the corresponding HPTE (if any) must be found and invalidated. One reasonable approach is to do the HPTE invalidation in the TLB flushing routines. There are four TLB flushing routines called by the Linux VM code: `flush_tlb_page`, `flush_tlb_range`, `flush_tlb_mm` and `flush_tlb_kernel_range`. In addition, there are the `__tlb_remove_tlb_entry` and `tlb_flush` routines which are used in conjunction with the `mmu_gather` structure used when destroying page tables.

Of these, `flush_tlb_page` is relatively easy to implement efficiently since it only operates on a single page, whose address is given. The flushing routine just needs to search one primary HPTE group and possibly one secondary group, and invalidate the HPTE if it is found.

Implementing `flush_tlb_range` and `flush_tlb_mm` efficiently is more difficult, since the information about precisely which LPTEs have been changed or invalidated is not readily available. Searching the MMU hash table for every address in the range (for `flush_tlb_range`) or in the whole address space for a process (`flush_tlb_mm`) would be very time-consuming, and would be particularly inefficient if there were not actually HPTEs present for most of the addresses, as is typically the case for `flush_tlb_mm`.

Instead, we use a bit in the LPTE, called `_PAGE_HASHPTE`, to indicate whether a HPTE corresponding to this LPTE has been created. Then, `flush_tlb_range` and `flush_tlb_mm` can scan the Linux page tables looking for LPTEs with the `_PAGE_HASHPTE` bit set and only search the MMU hash table for the corresponding addresses. This scheme does however require some care in handling the `_PAGE_HASHPTE` bit:

- It must remain valid even if the LPTE is invalidated or used for a swap entry. Thus an atomic read-modify-write sequence must be used to invalidate or update an LPTE rather than a normal store instruction.
- The page holding the LPTE must still be allocated and present in the page table tree at the time that any of the TLB flush routines are called.

The 64-bit PowerPC kernel uses an additional 4 bits in the LPTE to indicate whether the HPTE is in the primary or secondary group, and which of the 8 slots in the group it is in. With this information, the TLB flush routines can invalidate the HPTE directly without having to search the primary and secondary groups. This approach isn't used in the 32-bit

PowerPC kernel since there are not 4 free bits in the LPTE.

4.2 Optimized implementation

Instead of having to scan the Linux page tables in `flush_tlb_range` and `flush_tlb_mm`, it would potentially be more efficient to invalidate the HPTE at the time that the LPTE is changed. Alternatively, it would be possible to make a list of virtual addresses when LPTEs are changed and then use that list in the TLB flush routines to avoid the search through the Linux page tables.

This approach was not possible in earlier kernels because there was not enough information supplied in the calls to the functions that update LPTEs (`set_pte`, `ptep_get_and_clear`, etc.) to determine the address space (represented by the `mm_struct` structure) and virtual address for the LPTE being modified. However, the infrastructure added for the reverse-mapping (rmap) support in the Linux VM system allows us to determine this information efficiently, since a pointer to the `mm_struct` for the address space and the base virtual address mapped by the LPTE page are now stored in the `page_struct` structure for each LPTE page.

The results for the 32-bit kernel are shown in Table 4. The times shown are the system and user times in seconds for the kernel compilation test on the 400MHz PPC750 (G3) system, and are averages of at least two repetitions. The first row is for the original unoptimized kernel, the second row ("Immediate update") is for a kernel which invalidates the HPTE at the time when the LPTE is modified, and the third row ("Batched update") is for a kernel that records the virtual addresses when LPTEs are modified and invalidates the HPTEs in the TLB flush routines.

Kernel version	System time	User time	Total time
Original	32.09	303.71	335.80
Immediate update	32.28	302.93	335.21
Batched update	32.40	303.22	335.62

Table 4: PTE optimizations, 32-bit kernel

Kernel version	System time	User time	Total time
Original	8.51	78.13	86.64
Batched update	8.44	78.04	86.48

Table 5: PTE optimizations, 64-bit kernel

Clearly, neither optimization gives a significant increase in performance. The differences in total time of less than 0.6s are less than the standard deviation of the total time measurement for the original kernel, which was 1.42s (from 7 measurements).

The results in Table 5 for the 64-bit kernel on the 1.45GHz POWER4+ system paint a similar picture. The table compares the original kernel with one that records the virtual addresses when LPTEs are modified and invalidates the HPTEs in the TLB flush routine (the "Batched update" row). The times are in seconds and are averages of 6 measurements. Previous experience has shown that it is important to batch up changes to the MMU hash table in the ppc64 kernel, particularly on SMP systems. Consequently we did not test the variant which invalidates the HPTE at the time that the LPTE is modified.

The difference in total time is 0.16 seconds, about 0.2% of the total time. Even if the difference were statistically significant, it hardly represents an important optimization. However, the optimized code is actually simpler and shorter (by 66 lines) than the original code, and may be worth adopting for that reason alone.

5 Conclusions

The previous sections demonstrated performance improvements of varying magnitudes from the optimizations considered. Some of the individual LMBench numbers were more than doubled by the first optimization, that of avoiding the instruction cache flush on pages which had already been flushed. The 64-bit memory copy optimizations improved the unix-domain socket bandwidth by over 60%, with smaller improvements on other bandwidth-related measurements.

The overall improvements for the kernel compile benchmark were more modest. This is to be expected since the time spent in the kernel is only about a tenth of the time spent in user processes for this benchmark, and the optimizations considered here only reduce the time spent in the kernel.

In sum, the optimizations presented here provide a substantial performance boost for the Linux kernel on PowerPC machines.

The results illustrate some general principles about optimization work:

- Kernel profiling is a useful tool for determining what a profitable target for optimization is, and whether a given optimization is effective. However, the kernel profile results are usually quite noisy, and care must be exercised in interpreting them.
- Some optimizations may produce dramatic improvements on benchmarks but have almost no effect on the speed of actual application programs.
- Measurement is key; some optimizations might seem like an extremely good idea but not produce any significant performance gains, either because of unforeseen side-effects or because the thing being optimized doesn't consume a significant amount of time.

Acknowledgements

The author would like to thank Anton Blanchard for assistance with implementing and benchmarking the cache-flushing optimizations presented here.

Legal statements

Sharing Page Tables in the Linux Kernel

Dave McCracken
IBM Linux Technology Center
Austin, TX
dmccr@us.ibm.com

Abstract

An ongoing barrier to scalability has been the amount of memory taken by page tables, especially when large numbers of tasks are mapping the same shared region. A solution for this problem is for those tasks to share a common set of page tables for those regions.

An additional benefit to implementing shared page tables is the ability to share all the page tables during *fork* in a copy-on-write fashion. This sharing speeds up *fork* immensely for large processes, especially given the increased overhead introduced by rmap.

This paper discusses my implementation of shared page tables. It covers the areas that are improved by sharing as well as its limitations. I will also cover the highlights of how shared page tables was implemented and discuss some of the remaining issues around it.

This implementation is based on an initial design and implementation done by Daniel Phillips last year and posted to the kernel mailing list. [DP]

1 Introduction

The Linux® memory management (MM) subsystem has excellent mechanisms for minimizing the duplication of identical data pages by sharing them between address spaces whenever possible. The MM subsystem currently does not, however, make any attempt to share the lowest layer of the page tables, even though these may also be identical between address spaces.

Detecting when these page tables may be shared, and setting up sharing for them is fairly straightforward. In the following sections the current MM data structures are described, followed by an explanation of how and when the page tables would be shared. Finally some issues in the implementation and some performance results are described.

2 Major Data Structures

To understand the issues addressed by shared page tables it is helpful to understand the structures that make up the Linux MM.

2.1 The *mm_struct* Structure

The anchor for the memory subsystem is the *mm_struct* structure. There is one *mm_struct* for each active user address space. All memory-related information for a task is found in the *mm_struct* or in structures it points to.

2.2 The *vm_area_struct* Structure

The address layout is contained in two parallel sets of structures. The logical layout is described by a set of structures called

*vm_area_struct*s, or *vma*s. There is one *vma* for each region of memory with the same file backing, permissions, and sharing characteristics. The *vma*s are split or merged as necessary in response to changes in the address space. There is a single chain of *vma*s attached to the *mm_struct* that describes the entire address space, sorted by address. This chain also can be collected into a tree for faster lookup.

Each *vma* contains information about a virtual address range with the same characteristics. This information includes the starting and ending virtual addresses of the range, and the file and offset into it if it is file-backed. The *vma* also includes a set of flags, which indicate whether the range is shared or private and whether it is writeable.

2.3 The Page Table

The address space page layout is described by the page table. The page table is a tree with three levels, the page directory (pgd), the page middle directory (pmd), and the page table entries (pte). Each level is an array of entries contained within a single physical page. The entry at each level contains the physical page number of the next level. The pte entries contain the physical page number of the data page.

For the architectures that use hardware page tables, the page table doubles as the hardware page table. This use of the page table puts constraints on the size and contents of the page table entries when they are in active use by the hardware.

2.4 The *address_space* Structure

In addition to the structures for each address space, there is a structure for each file-backed object called struct *address_space* (not to be confused with an address space, which is represented by an *mm_struct*). The *address_space* structure is the anchor for all mappings related to the file object. It contains links to every *vma* mapping this file, as well as links to every physical page containing data from the file.

All shared memory uses temporary files as backing store, so each shared memory *vma* is linked to an *address_space* structure. The only *vma*s not attached to an *address_space* are the ones describing the stack and the ones describing the bss space for the application and each shared library. These are called the 'anonymous' *vma*s and are backed directly by swap space.

2.5 The *page* Structure

All physical pages in the system have a structure that contains all the metadata associated with the page. This is called a "struct *page*." It contains flags that indicate the current usage of the page and pointers that are used to link the page into various lists. If the page contains data from a file, it also has a pointer to the *address_space* structure for that file and an index showing where in the file the data came from.

3 How Memory Gets Mapped

Memory areas are created, modified, and destroyed via one of the mapping calls. These calls can map new memory areas, change the protections on existing memory areas, and unmap memory areas.

3.1 Creating A Memory Mapping

New memory regions are created using the calls *shmget/shmmap* or *mmap*. Both calls create a mapped memory region, either backed by an open file passed as an argument or by a temporary file created for the purpose. The newly mapped memory region is represented by a new or modified *vma* that is attached to

the taskś *mm_struct* and to the *address_space* structure for the file. The page table is not touched during the mapping call.

Various flags are passed in when a page is mapped. These flags specify the characteristics of the mapped memory. Two of the key characteristics are whether the area is shared or private and whether the area is writeable or read-only. For read-only or shared areas, the file data is read from and written back to the file. Private writeable areas are treated specially in that while the data is read from the file, modified data is not written back to the file. The data is saved to swap space as an anonymous page if it needs to be paged out.

Pages are actually mapped when a task attempts to access a virtual address in the mapped area. The page fault code first finds the *vma* describing the area, then finds the *pte* entry that maps a data page for that address. A page is then found based on the information in the *vma*.

3.2 Locks and Locking

The MM subsystem primarily relies on three locks. Two locks are in the *mm_struct*. First is the *mmap_sem*, a read/write semaphore. This semaphore controls access to the *vma* chain within the *mm_struct*.

The second lock in the *mm_struct* is the *page_table_lock*. This spinlock controls access to the various levels of the page table.

The third lock is in the *address_space* structure. This lock controls the chain of *vma*s that map the file associated with that *address_space*. In 2.4, it is a spinlock and is called *i_shared_lock*. In 2.5, it was changed to a semaphore and is called *i_shared_sem*.

During a page fault the *mmap_sem* semaphore is taken for read at the beginning of the fault and is held until the fault is complete. The *page_table_lock* is taken early in the fault, but is released as necessary when the fault needs to block. Holding the *mmap_sem* semaphore for read allows other tasks with the same *mm_struct* to take page faults but not change their mappings.

4 Sharing the PTE Page

Normally the overhead taken up by page tables is small. On 32 bit architectures, there is typically a maximum of one pgd page, three pmd pages, and one pte page for every 512 or 1024 data pages. This ratio may be somewhat higher for sparsely populated address spaces, but virtual addresses are typically allocated in order so pte pages tend to be filled in fairly densely.

This ratio changes for shared memory areas. A shared memory region that covers an entire pte page may be shared among many address spaces. This sharing means there will be 1 pte page for each address space for every 512 or 1024 mapped data pages. Note that the pte pages, once allocated, are not freed even if the data pages have been paged out, so the pte page overhead is fixed even under memory pressure.

Shared memory is a common method for many applications to communicate among their processes. It is possible for a massively shared application to use over half of physical memory for its page tables.

Each address space in this scenario has an identical set of pte pages for its shared areas, with all its entries pointing to the same physical data pages. The premise behind shared page tables is to only allocate a single set of pte pages and set the pmd entry for each address space to point to it.

A beneficial side effect of sharing the pte page is once a data page has been faulted in by one

task, the page appears in the address space of all other tasks mapping that area. Without sharing, each task would have to take its own page fault to get access to that page.

There are clearly some constraints on when pte pages can be shared. The shared area must span the virtual memory mapped by the entire pte page. All address spaces must map the area at the same virtual address. While in theory the mapped areas only need to be aligned per pte page, the current implementation requires that the virtual addresses be the same.

4.1 Finding PTE Pages to Share

For sharing to work, it is necessary to find an existing pte page if one exists. Finding this page is accomplished at page fault time when there is not already a pte page for the faulting address.

First, the current *vma* is checked to see if it is eligible for sharing. It needs to be either shareable or read-only, and needs to span the entire address range for that pte page.

Next, the code searches for an existing pte page that can be shared. This search is done by going to the *address_space* for the current memory area, then walking its *vma* chain. Each *vma* is checked for compatibility. The *vma* needs to be at the same file offset and virtual address as the faulting *vma* and needs to also span the entire pte page. For each matching *vma*, its corresponding page table is checked for a pte page. If a pte page is found, it is installed in the pmd entry and its use count is incremented. While in theory it should be possible to share pte pages between any *vma*s whose mappings have the same pte page alignment, the current rmap implementation limits sharing to those that map to the same virtual address.

4.2 Copy On Write

On some architectures there is a second use for shared page tables. During *fork*, every pte entry is copied to the new page table. Data pages that can't be fully shared are marked as "copy on write." Marking a page as copy on write involves setting both the parent and the child pte entry to point to the same data page, but with write disabled. When either task tries to write to that page, the page is then duplicated and write access is enabled.

Some architectures (including x86) support disabling write access in the pmd entry, and interpret this to mean disabling write to all the data pages mapped by that entry. Disabling write allows the copy on write concept to be extended to shared page tables. Instead of copying each pte entry at *fork* time, each pmd entry is set to point to the same pte page and write access is disabled. When a write fault occurs, a new pte page is allocated and all the pte entries are copied in the same fashion as during *fork* in the non-shared version.

4.3 Locking Changes for Shared PTE Pages

When page tables have shared pte pages, the existing locking scheme becomes inadequate. The *page_table_lock* in the *mm_struct* can no longer protect the entire page table, since pte pages may be shared with other page tables.

There is a spinlock in the page struct that is normally used for *pte_chain*s in data pages. Since pte pages have no *pte_chain*s, the lock in the pte page's page struct can be used to control access to it. For pte pages this lock becomes the *pte_page_lock*. This change means the *page_table_lock* protects the pgd and pmd levels and the *pte_page_lock* protects the pte page.

Using this lock changes the locking protocol in

the page fault path. The *page_table_lock* must still be taken to until the pte page is found and selected. The *pte_page_lock* is taken for that page, at which point the *page_table_lock* can be released. The rest of fault resolution is done under the *pte_page_lock*.

5 Complications

While making pte pages shared seems like a simple task in theory, there are several things that complicated the task.

Part of *pte_chain*-based reverse mapping is a pointer in the pte page's *struct page* that points to the *mm_struct* the page belongs to. Sharing the pte page means it can belong to several *mm_struct*s. It was necessary to add an *mm_chain* structure which points to a chain of *mm_structs* that use the pte page.

There are various memory management-related system calls that can modify existing mappings. These include *mremap*, *mprotect*, *remap_file_pages*, and *mmap* itself. These calls can all change the mappings for an address space such that the pte page can no longer be shared. Each of those system calls was modified to identify shared pte pages and unshare them as necessary.

6 Performance

Shared page tables are primarily intended to reduce the space overhead of page tables, but there are some performance benefits, as well.

The primary performance gain is during *fork* because of copy-on-write sharing. Instead of duplicating a reference to each data page *fork* only needs to duplicate a reference to each pte page. This can improve *fork* performace by up to a factor of 10.

Fork speedup is balanced, however, by the cost of unsharing each pte page when one of its data pages is written to. Typically, three pte pages are unshared after every *fork* due to the user space layout. Since small programs generally only have three pte pages, only larger programs benefit from the improvement. In fact, the cost of sharing the pte, then unsharing it on page fault, has a small but measureable cost compared to copying. The simple solution to this is to copy the pte pages on *fork* if the address space only has 3 pte pages.

There is a corresponding performance improvement for *exit* and *exec* when they tear down the address space. Any pte pages that are shared can be detached simply by decrementing their reference count. For each pte page that is not shared, the code must examine each entry to determine what to do with its data page or swap page.

7 Conclusion

Shared page tables achieves its primary objective of dramatically improving the scalability of massively shared applications, as well as also improving the *fork* and *exit* performance of large tasks. While there is some cost in added complexity, the benefits far outweigh the cost.

Legal Statement

References

[DP] Daniel Phillips.
`http://nl.linux.org/phillips/page.table.sharing`

Kernel Janitors: State of the Project

Arnaldo Carvalho de Melo
Conectiva S.A.
acme@conectiva.com.br
http://advogato.org/person/acme

Abstract

The Kernel Janitors Project has been cleaning up the kernel for quite some time, in this paper I'll present what has been done, tasks that kernel hackers added to TODO list, tools used to help in the process, 2.5 changes that need to be propagated thru the tree and other aspects of kernel janitoring.

1 What is the Kernel Janitor Project?

The Kernel Janitors Project grew out of our search for things to help in the development of the Linux kernel, and learning from other patches submitted by more experienced people, we saw that some of these patches indicated error patterns that could exist in other parts of the kernel, we looked and... yes, we discovered that some parts of the kernel suffered from the same problems and in the process we found code bitrotting... I (acme) started maintaining a TODO list for things to fix or clean up and people started submitting suggestions for things to fix that I collected at `http://bazar.conectiva.com.br/~acme/TODO`.

2 A Way for Newbies to Start Hacking the Kernel

Looking at the httpd logs I discovered that lots of people accessed it, so this indeed was something useful as a starting point for people also wanting to help in cleaning up/fixing parts of the kernel.

Several people have the goodwill to help in kernel programming, but many don't have the time to help on areas that require more knowledge and effort, so the Kernel Janitor Project TODO list helps, as there are lots of simple tasks that needs work but are trivial enough to allow those people to help while not requiring much time and effort.

3 The Project Moves to SourceForge

Dave Jones suggested that we moved this to sourceforge, so that we could have it on CVS and allow more people to have admin rights to add more entries, add explanations about the tasks, etc.

I then registered the domain kerneljanitors.org and made it point to the sourceforge web page, that is very simple but has important pointers to resources for new janitors.

4 Project Committers

We're always accepting more people, preferably maintainers of parts of the kernel, so that we can improve the TODO list, and as there is always some boring and repetitive work that the maintainers postpone because there is always more important things to do.

In fact even Linus has posted at least once in the mailing list asking for help on simple repetitive tasks, such as the irqreturn_t conversion for all the interrupt handling routines.

These are the current commiters for this project, at this time:

- `Arnaldo Carvalho de Melo`
- `Dave Jones`
- `Jeff Garzik`
- `Matthew Wilcox`
- `Randy Dunlap`
- `Tariq Shureih`
- `William Lee Irwin III`

5 Mailing list

Using the SourceForge infrastructure we creates the kernel-janitor-discuss mailing list, where we discuss the aspects of the project and review janitor patches.

There has been a continuous arrival of people asking where to start helping, and a number of them became regular janitors.

6 Some Janitors

- `William Stinson`
  - *check_region removal*
- `Art Haas`
  - *C99 struct init style*

7 Tools

There are now several tools that help on pinpointing most of the entries in the TODO list and even some more involved problems that needs hands to fix, I'll briefly talk about several of such tools in the next sections.

7.1 Stanford Checker

Perhaps the first tool for source checking used in the Linux kernel, the Stanford Checker is based on a modified gcc that does several verifications for different types of common problems, and then the Stanford guys do some sanity checking and post the list on the linux-kernel mailing list, where people comment on it, checking if they are false positives and most frequently fixing the bugs.

It is unfortunately an unreleased tool, but it has been of great help over the last years.

7.2 kj.pl

This was a very simple perl script that Dave Jones wrote, that searched for some very trivial problems, but Dave has stopped working on it as now we have smatch.

7.3 smatch

Dan Carpenter's smatch is a released tool that allows developers to write perl scripts to search for problems, it is also based on a modified gcc.

Dan has created the kbugs.org web site where he has several smatch scripts and a list of the

results of those scripts, allowing janitors to pick real problems to work on.

The current scripts, for reference, are:

- `ReleaseRegion`
- `ReleaseIRQ`
- `DoubleSpinlock`
- `Dereference`
- `GFP_DMA`
- `UnreachedCode`
- `SpinlockUndefined`
- `FunctionStack`
- `UncheckedReturn`
- `SpinSleepLazy`
- `UnFree`

I'll quote Dan now just as an example of smatch results in kbugs.org:

"There were quite a few smatch related fixes in the 2.5.69 kernel. Someone fixed 4 SpinSleepLazy bugs, 3 UnreachedCode bugs and 6 unchecked calls of `copy_to_user()`. It's not entirely clear who did what from the changelog, but thank you anonymous heros."

7.4 cqual

Cqual is another tool that can help finding problems on codebases such as the kernel, here is the project description, from its authors:

"Cqual is a type-based analysis tool that provides a lightweight, practical mechanism for specifying and checking properties of C programs. Cqual extends the type system of C with extra user-defined type qualifiers. The programmer adds type qualifier annotations to their program in a few key places, and Cqual performs qualifier inference to check whether the annotations are correct. The analysis results are presented with a user interface that lets the programmer browse the inferred qualifiers and their flow paths."

It is part of a bigger project at the University of Berkeley called Open Source Quality, this seems to be a project that deserves investigation by janitors as it wasn't mentioned up to now on the kjp mainling list.

7.5 sparse

And now to something unreleased at this time: Linus Torvalds's sparse tool:

"Sparse is a semantic parser of source files: it's neither a compiler (although it could be used as a front-end for one), nor is it a preprocessor (although it contains as a part of it a preprocessing phase).

It is meant to be a small—and simple—library. Scanty and meager, and partly because of that easy to use. It has one mission in life: create a semantic parse tree for some arbitrary user for further analysis. It's not a tokenizer, nor is it some generic context-free parser. In fact, context (semantics) is what it's all about—figuring out not just what the grouping of tokens are, but what the *types* are that the grouping implies."

It is indeed very interesting that more and more people are working towards having tools that can help in improving the quality of Open Source projects such as Linux.

8 Documentation

- lwn.net Articles by Jonathan Corbet
- Tariq's "Drivers DOs and DONTs"

- Other Articles for janitors
- Arjan’s article about how not write a driver
- Greg KH’s article.

Linux Kernel Power Management

Patrick Mochel
Open Source Development Labs
mochel@osdl.org

Abstract

Power management is the process by which the overall consumption of power by a computer is limited based on user requirements and policy. Power management has become a hot topic in the computer world in recent years, as laptops have become more commonplace and users have become more conscious of the environmental and financial effects of limited power resources.

While there is no such thing as perfect power management, since all computers must use some amount of power to run, there have been many advances in system and software architectures to conserve the amount of power being used. Exploiting these features is key to providing good system- and device-level power management.

This paper discusses recent advances in the power management infrastructure of the Linux kernel that will allow Linux to fully exploit the power management capabilities of the various platforms that it runs on. These advances will allow the kernel to provide equally great power management, using a simple interface, regardless of the underlying archtitecture.

This paper covers the two broad areas of power management—System Power Management (SPM) and Device Power Management (DPM). It describes the major concepts behind both subjects and describes the new kernel infrastructure for implementing both. It also discusses the mechanism for implementing hibernation, otherwise known as suspend-to-disk, support for Linux.

1 Overview

Benefits of Power Management

A sane power management infrastructure provides many benefits to the kernel, and not only in the obvious areas.

Battery-powered devices, such as embedded devices, handhelds, and laptops reap most of the rewards of power management, since the more conservative the draw on the battery is, the longer it will last.

System power management decreases boot time of a system, by restoring previously saved state instead of reinitializing the entire system. This conserves battery life on mobile devices by reducing the annoying wait for the computer to boot into a useable state.

Recently, power management concepts have begun to filter into less obvious places, like the enterprise. In a rack of servers, some servers may power down during idle times, and power back up when needed again to fulfill network requests. While the power consumption of a single server is but a drop in the water, being able to conserve the power draw of dozens or hundreds of computers could save a company a significant amount of money.

Also, at the lower-level, power management may be used to provide emergency reaction to a critical system state, such as crossing a predefined thermal threshold or reaching a critically low battery state. The same concept can be applied when triggering a critical software state, like an Oops or a BUG() in the kernel.

System and Device Power Management

There are two types of power management that the OS must handle—System Power Management and Device Power Management.

Device Power Management deals with the process of placing individual devices into low-power states while the system is running. This allows a user to conserve power on devices that are not currently being used, such as the sound device in my laptop while I write this paper.

Individual device power management may be invoked explicitly on devices, or may happen automatically after a device has been idle for a set of amount of time. Not all devices support run-time power management, but those that do must export some mechanism for controlling it in order to execute the user's policy decisions.

System Power Management is the process by which the entire system is placed into a low-power state. There are several power states that a system may enter, depending on the platform it is running on. Many are similar across platforms, and will be discussed in detail later. The general concept is that the state of the running system is saved before the system is powered down, and restored once the system has regained power. This prevents the system from performing an entire shutdown and startup sequence.

System power management may be invoked for a number of reasons. It may automatically enter a low-power state after it has been idle for some amount of time, after a user closes a lid on a laptop, or when some critical state has been reached. These are also policy decisions that are up to the user to configure and require some global mechanism for controlling.

2 Device Power Management

Overview

Device power management in the kernel is made possible by the new driver model in the 2.5 kernel. In fact, the driver model was inspired by the requirement to implement decent power management in the kernel. The new driver model allows generic kernel to communicate with every device in the system, regardless of the bus the device resides on, or the class it belongs to.

The driver model also provides a hierarchical representation of the devices in the system. This is key to power management, since the kernel cannot power down a device that another device, that isn't powered down, relies on for power. For example, the system cannot power down a parent device whose children are still powered up and depend on their parent for power.

In its simplest form, device power management consists of a description of the state a device is in, and a mechanism for controlling those states. Device power states are described as 'D' states, and consist of states D0-D3, inclusive. This device state representation is inspired by the PCI device specification and the ACPI specification [ACPI]. Though not all device types define power states in this way, this representation can map on to all known device types.

Each D state represents a tradeoff between the amount of power a device is consuming and how functional a device is. In a lower power state (represented by a higher digit following

D), some amount of power to a device is lost. This means that some of the device's operating state is lost, and must be restored by its driver when returning to the D0 state.

D0 represents the state when the device is fully powered on and ready for, or in, use. This state is implicitly supported by every device, since every device may be powered on at some point while the system is running. In this state, all units of a device are powered on, and no device state is lost.

D3 represents the state when the device is off. This state is also implicitly supported by every device, since every device is implicitly powered off when the system is powered off. In this state, all device context is lost and must be restored before using the device again. This usually means the device must also be completely reinitialized.

The PCI Power Management spec goes on to define D3hot as a D3 state that is entered via driver control and D3cold that is entered when the entire system is powered down. In D3hot, the device may not lose all operating power, requiring less restoration that must take place. This is however, device-dependent. The kernel does not distinguish between the two, though a driver theoretically could take extra steps to do so.

D1 and D2 are intermediate power states that are optionally supported by a device. In each case, the device is not functional, but not entirely powered off. In order to bring the device back to an operating state, less work is required than reviving the device from D3. In D1, more power is consumed than in D2, but more device context is preserved.

A device's power management information is stored in `struct device_pm`:

```
struct device_pm {
#ifdef CONFIG_PM
    dev_power_t     power_state;
    u8              * saved_state;
    atomic_t        depend;
    atomic_t        disable;
    struct kobject  kobj;
#endif
};
```

`struct device` contains a statically allocated `device_pm` object. The PM configuration dependency guarantees the overhead for the structure is nil when power management support is not compiled in.

The kernel defines the following power states in `include/linux/pm.h`:

```
typedef enum {
    DEVICE_PM_ON,
    DEVICE_PM_INT1,
    DEVICE_PM_INT2,
    DEVICE_PM_OFF,
    DEVICE_PM_UNKNOWN,
} dev_power_t;
```

When a device is registered, its initial power state is set to `DEVICE_PM_UNKNOWN`. The device driver may query the device and initialize the known power state using

```
void device_pm_init_power_state(
  struct device * dev,
  dev_power_t state);
```

Controlling a Device's State

A device's power state may be controlled by the `suspend()` and `resume()` methods in `struct device_driver`:

```
int (*suspend)(struct device * dev,
               u32 state, u32 level);
int (*resume) (struct device * dev,
               u32 level);
```

These methods may be initialized by the low-level device driver, though they are typically initialized at registration time by the bus driver that the driver belongs to. The bus's functions should forward power management requests to the bus-specific driver, modifying the semantics where necessary.

This model is used to provide the easiest route when converting to the new driver model. However, a device driver's explicit initialization of these methods will be honored.

The same methods are called during individual device power management transitions and system power management transitions.

There are two steps to suspend a device and two steps to resume it. In order to suspend a device, two separate calls are made to the suspend() method—one to save state, and another to power the device down. Conversely, one call is made to the resume() method to power the device up, and another to restore device state.

These steps are encoded:

```
enum {
    SUSPEND_SAVE_STATE,
    SUSPEND_POWER_DOWN,
};

enum {
    RESUME_POWER_ON,
    RESUME_RESTORE_STATE,
};
```

and are passed as the 'level' parameter to each method.

During the `SUSPEND_SAVE_STATE` call, the driver is expected to stop all device requests and save all relevant device context based on the state the device is entering.

This call is made in process context, so the driver may sleep and allocate memory to save state. However during system suspend, backing swap devices may have already been powered down, so drivers should use `GFP_ATOMIC` when allocating memory.

`SUSPEND_POWER_DOWN` is used only to physically power the device down. This call has some caveats, and drivers must be aware of them. Interrupts will be disabled when this routine is called. However, during run-time device power management, interrupts will be reenabled once the call returns. Some devices are known to cause problems once they are powered down and interrupts reenabled—e.g. flooding the system with interrupts. Drivers should be careful not to service power management requests for devices known to be buggy.

During system power management, interrupts are disabled and remain disabled while powering down all devices in the system.

The resume sequence is identical, though reversed, from the suspened sequence. The `RESUME_POWER_ON` stage is performed first, with interrupts disabled. The driver is expected to power the device on. Interrupts are then enabled and the `RESUME_RESTORE_STATE` is performed, and the driver is expected to restore device state and free memory that was previously allocated.

A driver may use the `struct device_pm::saved_state` field to store a pointer to device state when the device is powered down.

Power Dependencies

Devices that are children of other devices (e.g. devices behind a PCI bridge) depend on their parent devices to be powered up to either provide power to them and/or provide I/O transactions.

The system must respect the power dependen-

cies of devices and must not attempt to power down a device which another device depends on being on. Put another way, all children devices must be powered down before their parent can be powered down. Conversely, the parent device must be powered up before any children devices may be accessed.

Expressing this type of dependency is simple, since it is easy to determine whether or not a device has any children or not. But, there are more interesting power dependencies that are more difficult to express.

On a PCI Hotplug system, the hotplug controller that controls power to a range of slots may reside on the primary PCI bus. However, the slots it controls may reside behind a PCI-PCI bridge that is a peer of the hotplug controller. The devices in the slots depend on the hotplug controller being on to operate, but it is not the devices' parent. There are similar transversal relationships on some embedded platforms in which some I/O controller resides near the system root that some PCI devices, several layers deep, may depend on to communicate properly.

Both types of power dependencies are represented using the `struct device_pm::depend` field. Implicit dependencies, like parent-child relationships, are handled by the depend count being incremented when a child is registered with the PM core. When that child device is powered down or removed, its parent's depend count is decremented. Only when a device's depend count is 0 may it be powered down.

Explicit power dependencies can be imposed on devices using

```
int device_pm_get(struct
    device *);
void device_pm_put(struct
     device *);
```

`device_pm_get()` will increment a device's dependency count, and `device_pm_put()` will decrement it. It is up to the driver to properly manage the dependency counts on device discovery, removal, and power management requests.

Disabling Power Management

There are circumstances in which a driver must refuse a power management request. This is usually because the driver author does not know the proper reinitialization sequence, or because the user is performing an uninterruptible operation like burning a CD.

It is valid for a driver to return an error from a `suspend()` method call. For example, a driver may know a priori that it can't handle the request. This works to the system's benefit, since the PM core can check if any devices have disabled power management before starting a suspend transition.

To disable or enable power management, a device may call

```
int device_pm_disable(struct
    device *);
void device_pm_enable(struct
     device *);
```

The former increments the `struct device_pm::disable` count, and the lattr decrements it. If the count is positive, system power management will be disabled completely, and device power management on that device.

These calls should be used judiciously, since they have a global impact on system power management.

3 System Power Management

System power management (SPM) is the process of placing the entire system into a low-power state. In a low-power state, the system is consuming a small, but minimal, amount of power, yet maintaining a relatively low response latency to the user. The exact amount of power and response latency depends on the state the system is in.

Power States

The states a system can enter are dependent on the underlying platform, and differ across architectures and even generations of the same architecture. There tend to be three states that are found on most archtitectures that support a form of SPM, though. The kernel explicitly supports these states—Standby, Suspend, and Hibernate, and provides a mechnanism for a platform driver (an architectural port of the kernel) to define new states.

```
typedef enum {
    POWER_ON         = 0,
    POWER_STANDBY    = 0x01,
    POWER_SUSPEND    = 0x02,
    POWER_HIBERNATE = 0x04,
} pm_system_state_t;
```

Standby is a low-latency power state that is sometimes referred to as "power-on suspend." In this state, the system conserves power by placing the CPU in a halt state and the devices in the D1 state. The power savings are not significant, but the response latency is minimal—typically less than 1 second.

Suspend is also commonly known as "suspend-to-RAM." In this state, all devices are placed in the D3 state and the entire system, except main memory, is expected to maintain power. Memory is placed in self-refresh mode, so its contents are not lost. Response latency is higher than Standby, yet still very low—between 3-5 seconds.

Hibernate conserves the most power by turning off the entire system, after saving state to a persistant medium, usually a disk. All devices are powered off unconditionally. The response latency is the highest—about 30 seconds—but still quicker than performing a full boot sequence.

Most platforms support these states, though some platforms may support other states or have requirements that don't match the assumptions above. For example, some PPC laptops support Suspend, but because of a lack of documentation, the video devices cannot be fully reinitialized and hence may not enter the D3 state. The hardware will supply enough power to devices for them to stay in the D2 state, which the drivers are capable of recovering from.

Instead of cluttering the code with a lot of conditional policy to determine the correct state for devices to enter, the PM subsystem abstracts system state information into dynamically registered objects.

```
    struct pm_state {
    struct pm_driver  * drv;
    pm_system_state_t   sys;
    pm_device_state_t   dev;
    struct kobject      kobj;
};
```

The drv field is a pointer to the platform-specific object configured to handle the power state. The sys field is the low-level power state that the system will enter. The dev field is the lowest power state that devices may enter. The kobj field is the generic object for managing an instance's lifetime.

The kernel defines default power state objects representing the assumptions above:

```
struct pm_state pm_state_standby;
struct pm_state pm_state_suspend;
struct pm_state pm_state_hibernate;
```

Platform drivers may also define and register additional power states that they support using:

```
int
pm_state_register(struct
                  pm_state *);
void
pm_state_unregister(struct
                    pm_state *);
```

The PM sysfs Interface

The PM infrastructure registers a top-level subsystem with the kobject core, which provides the `/sys/power/` directory in sysfs. By default, there is one file in the directory `/sys/power/state`.

Reading from this file displays the states that are currently registered with the system; e.g.:

```
# cat /sys/power/state
standby suspend hibernate
```

By writing the name of a state to this file, the system will perform a power state transition, which is described next.

Each power state that is registered receives a directory in `/sys/power/`, and three attribute files:

```
# tree /sys/power/suspend/
/sys/power/suspend/
|-- devices
|-- driver
‘-- system
```

The ‘devices’ file and the ‘system’ file describe which power state the devices in the computer and the state the computer itself are to enter, respectively. The ‘driver’ file displays which low-level platform PM driver is configured to handle the power transition. Writing to this file sets the driver internally.

Power Management Platform Drivers

The process of transitioning the OS into a low-power state is largely platform-agnostic. However, the low-level mechanism for actually transitioning the hardware to a low-power state is very platform specific, and even dependent on the generation of the hardware.

On some platforms, there may be multiple ways to enter a low-power state, presenting a policy decision for the user to make. Note this arises usually only in choosing whether to enter a minimal power state during a Hibernation transition, or turning the system completely off.

To cope with these variations, the PM core defines a simple driver model:

```
struct pm_driver {
    u32 states;
    int (*prepare)(u32 state);
    int (*save)   (u32 state);
    int (*sleep)  (u32 state);
    int (*restore)(u32 state);
    int (*cleanup)(u32 state);
    struct kobject  kobj;
};

int
pm_driver_register(struct
                   pm_driver *);

void
pm_driver_unregister(struct
                     pm_driver *);
```

The states field of `struct pm_driver` is a logical or of the states the driver supports. The methods are platform-specific calls that the PM

core executes during a power state transition. They are designed to perform the following:

prepare — Verify that the platform can enter the requested state and perform any necessary preparation for entering the state.

save — Save low-level state of the platform and the CPU(s).

sleep — Enter the requested state.

restore — Restore low-level register state of the platform and CPU(s).

cleanup — Perform any necessary actions to leave the sleep state.

A platform should intialize and register a driver on startup:

```
struct pm_driver acpi_pm_driver = {
  .states =  (POWER_STANDBY |
              POWER_SUSPEND  |
              POWER_HIBERNATE),
  .prepare = acpi_enter_sleep_state_prep,
  .sleep   = acpi_pm_sleep,
  .cleanup = acpi_leave_sleep_state,
  .kobj    = { .name = "acpi" },
};

static int __init acpi_sleep_init(void) {
    return pm_driver_register(
           &acpi_pm_driver);
}
```

Each registered PM driver receives a directory in sysfs in `/sys/power/`. Each driver receives one default attribute file named `states`, which displays the power states the driver supports. This file is not writable by userspace.

```
# tree /sys/power/acpi/
/sys/power/acpi/
'-- states
# cat /sys/power/acpi/states
standby suspend hibernate
```

Platform drivers may define and export their own attributes.

```
struct pm_attribute {
 struct attribute  attr;
 ssize_t (*show)(struct pm_driver *,
                 char *);
 ssize_t (*store)(struct pm_driver *,
                  const char *,
                  size_t);
};

int pm_attribute_create(
  struct pm_driver *,
  struct pm_attribute*);

void pm_attribute_remove(
  struct pm_driver *,
  struct pm_attribute *);
```

The semantics for `pm_driver` attributes follow the same semantics as other sysfs attributes. Please see the kernel sysfs documentation for more information.

Power State Transitions

Transitioning the system to a low-power state is, unfortunately, not as simple as telling the platform to enter the requested low power state. The file `drivers/power/suspend.c` contains the entire sequence, and should be used as reference material for the official process. A synopsis is provided here.

The first step is to verify that the system can enter the power state. The PM core must have a driver that supports the requested state, the driver must return success from its `prepare()` method, and the driver core must return success from `device_pm_check()`. Next, the PM core quiesces the running system by disabling preemption and 'freezing' all processes.

Next, system state is saved by calling `device_suspend()` to save device state,

and the driver's `save()` method to save low-level system state. If we're entering a variant of the Hibernation state, the contents of memory must be saved to a persistant medium. `pm_hibernate_save()` is called to perform this, which is described in the section Hibernation.

Once state is saved, the PM core disables interrupts and calls `device_power_down()` to place each device in the specified low power state. Finally, it calls the driver's `sleep()` method to transition the system to the low-power state.

The resume sequence has two variants, depending on whether the system is returning from a Hibernation state or not. If it is not, the platform is responsible for returning execution to the correct place (after the return from the driver's `sleep()` method). This may be a function of the processor, the firmware, or the low-level platform driver.

If we're returning from Hibernation, the system detects it during a boot process in the function `pm_resume()`. `pm_resume()` is a late_initcall, which means it is called after most subsystems and drivers have been registered and initialized, including all non-modular PM drivers. It calls `pm_hibernate_load()`, which is responsible for attempting to read, load, and restore a saved memory image. Doing this replaces the currently running system with a saved one, and execution returns to after the call to `pm_hibernate_store()`.

One way or another, the PM core proceeds to power on all devices and restore interrupts. The driver's `restore()` method is called to restore low-level system state, and `device_pm_resume()` is called to restore device context. Finally, the driver's `cleanup()` method is called, processes are 'thawed,' and preemption is reenabled.

A suspend transition is triggered by writing the requested state to the sysfs file `/sys/power/state`. Once the complete transition is complete, execution will return to the process that wrote the value.

4 Hibernation

This section describes the Hibernate power state; specifically the process the PM core uses to save memory to a persistant medium, and the model for implementing a low-level backend driver to read and write saved state from a specific medium.

As mentioned in the previous section, Hibernate is a low-power state in which system memory state is saved to a persistant medium before the system is powered off and restored during the system boot sequence.

Hibernate is the only low-power state that can be used in the absence of any platform support for power management. Instead of entering a low-power state, the configured PM driver may simply turn the system off. This mechanism provides perfect power savings (by not consuming any), and can be used to work around broken power management firmware or hardware. The PM core registers a default platform driver that supplies this mechanism. It is named 'shutdown' and supports the Hibernate state only.

Hibernation can also add value to situations which would otherwise ignore standard power management concepts. For example, system state can be saved and restored should a battery (either in a laptop or a UPS) become critcally low. Or, system state could be saved when the kernel Oops'd or hit a `BUG()`. The system could be rebooted and the state examined later.

Hibernation Backend Drivers

Hibernate is commonly referred to as "suspend-to-disk," implying that the medium that system state is saved to is a physical disk. This assumption does not offer the possibility that another type of media may be used to capture state, nor does it make the distinction of how the state is stored on disk, since it could theoretically be stored on a dedicated partition, in free swap space, or in a regular file on an arbitrary filesystem.

The PM subsystem offers the ability to configure a variable medium type to save state to.

```
struct pm_backend {
    int     (*open) (void);
    void    (*close)(void);
    int     (*read_image)(void);
    int     (*write_image)(void);
    struct kobject  kobj;
};

int pm_backend_register(struct p
    m_backend *);
void pm_backend_unregister(struct
     pm_backend *);
```

The PM core provides a default backend driver named `pmdisk` that uses a dedicated partition type to save state. The internals of pmdisk are discussed later.

Backend drivers are registered as children of the Hibernate `pm_state` object, and are represented by directories in sysfs.

They may also define and export attributes using the following interface:

```
struct pm_backend_attr {
 struct attribute attr;
 ssize_t (*show)(struct pm_backend *,
                 char *);

 ssize_t (*store)(struct pm_backend *,
                  const char *,
                  size_t);
};

int pm_backend_attr_create(
  struct pm_backend *,
  struct pm_backend_attr *);

void pm_backend_attr_remove(
  struct pm_backend *,
  struct pm_backend_attr *);
```

Snapshotting Memory

The Hibernate core 'snapshots' system memory by indexing and copying every active page in the system. Once a snapshot is complete, the saved image and index is passed to the backend driver to store persistantly.

The snapshot process has one critical requirement: that at least half of memory be free. This imposes a strict limitation on the use of the current Hibernate implementation during periods of high memory usage. However, this design decision simplifies the requirements of the implementaion itself.

The snapshot sequence is a three-step process. First, all of the active pages in the system are indexed, enough new pages are allocated to clone these pages, then each page is copied into its clone, or "shadow."

Active pages are detected by iterating over each page frame number (`'pfn'`) in the system and determining whether we should save it or not. A page's saveability is initially determined by whether or not the `PageNosave` bit is set, and then whether the page is free or not. Reserved pages may not be saveable, depending on whether they exist in the `'__nosave'` data section.

Pages marked `Nosave` or declared in the `__nosave` section (with the `'__nosavedata'` suffix) are volatile

data and variables internal to the Hibernate core. They are used and modified during the snapshot process, and are not saved.

Saveable pages are indexed in page-sized arrays called pm_chapters:

```
#define PG_PER_CHAPT \
    (PAGE_SIZE / sizeof(pgoff_t))

struct pm_chapter {
    pgoff_t c_pages[PG_PER_CHAPT];
};
```

pm_chapters are dynamically allocated based on the number of saveable pages in the system. The addresses of the allocated chapters are stored in another page-sized array, called a pm_volume:

```
#define CHAPT_PER_VOL \
    (PAGE_SIZE / \
    sizeof(struct pm_chapter *))

struct pm_volume {
    struct pm_chapter *
      v_chapters[CHAPT_PER_VOL];
};
```

There are two static pm_volumes in the Hibernate core—one for the memory index (pm_mem_index), and one for the snapshot (pm_mem_shadow). This imposes an upper limit on the amount of memory that can be snapshotted by the Hibernate core:

```
CHAPT_PER_VOL * PG_PER_CHAPT * PAGE_SIZE / 2
```

is the number of bytes that can be saved, assuming half of memory must be free to store the snapshot. On a 32-bit x86 machine with 4K-sized pages, this works out to be:

```
1024 * 1024 * 4096 / 2
= 2,147,483,648 bytes
= 2 GB
```

which is more than enough, since accessing memory above 1GB requires 4M-sized pages.

After memory has been indexed, but before it has been copied, the contents of pm_mem_index and pm_mem_shadow are copied to pm_mem_clone and pm_shadow_clone. The latter are also statically allocated objects, but are not declared "nosave." The purpose of the clones is to save the addresses of the dynamically allocated chapter pages so we can free them once the saved image has been restored.

At this stage, the Hibernate core calls a required architecture- specific function:

```
int pm_arch_hibernate(
     pm_system_state_t state);
```

The state parameter should be set to POWER_HIBERNATE. This call is responsible for saving low-level register state and calling pm_hibernate_save(), which copies each indexed page in pm_mem_index to its corresponding page in pm_mem_shadow.

Restoring Memory

During a resume sequence, the Hibernate core calls the backend's open() method, which is responsible for setting pm_num_pages, which the Hibernate core will use to pre-allocate pm_mem_index and pm_mem_shadow.

The backend's read_image() method is called, which populates pm_mem_index with the target location of each saved page, and pm_mem_shadow, which contains the saved pages.

The saved image will replace the memory on a different running system. The pages that have been allocated to store the saved image populated from the backend may conflict with pages

in the saved image that are to be restored. The Hibernate backend must guarantee that none of the pages currently pointed to by `pm_mem_shadow` conflict with the pages indexed by `pm_mem_index`. To do this, it loops through each page address in `pm_mem_shadow` and compares them with each page address in `pm_mem_index`. If any matches are found, a new page is allocated and the contents copied.

To replace memory, the Hibernate core calls

```
pm_arch_hibernate(POWER_ON);
```

The architecture is responsible for iterating over the pages in `pm_mem_shadow` and copying each one to its destination, as indexed in `pm_mem_index`. It is also responsible for restoring low-level register state once memory has been replaced.

This burden is placed on the architecture so it can implement a replacement algorithm without using the stack for variable storage. The saved memory image contains the saved stack, while the current stack pointer register will point to a location on the stack in the memory being replaced. These will likely not match and cause the system to crash very quickly.

Once the memory image is restored, the architecture must restore register context to get the stack pointer pointing to the right place. This is the reason that the same function is called to both save and restore the low-level registers.

Returning from `pm_arch_hibernate()` once memory has been replaced will restore execution to the point in `hibernate_write()` where `pm_arch_hibernate()` was called, in the saving sequence. To detect this, the Hibernate core declares:

```
static in_suspend __nosavedata = 0;
```

and sets to it one during the save path. Since it's not saved, it will be 0 during the restore path, allowing the Hibernate core to behave appropriately. The cloned volumes are copied back into `pm_mem_index` and `pm_mem_shadow`, and the dynamically allocated pages are freed.

Backend Driver Semantics

The Hibernate core calls the backend driver's `open()` method before any Hibernate operation. It is the backend's responsibility to verify the existence of the media and to open any necessary communication channels to it. The backend driver is responsible for reading image metadata from the medium and setting `pm_num_pages` to the number of saved pages if a saved image exists. The Hibernate core will use this value to pre-allocate storage for the saved pages.

It may also use this opportunity to verify there is enough free space on the device. The maximum requirement is the total amount of memory in the system, as indicated by:

```
num_physpages * PAGE_SIZE
```

This check is optional at this stage, since the size of the saved memory image may be much smaller than this, and may fit on a device with less free space than the total size of memory.

When the Hibernate core is done, it will call the backend's `close()` method. The backend is responsible for closing any communication channels to the storage medium and freeing any memory it had allocated.

After the Hibernate core has shadowed memory, it calls the backend's `write_image()` method. It does not pass any parameters. `pm_mem_index` and `pm_mem_shadow` must be used directly. The backend must saved each

page pointed to in each chapter of `pm_mem_shadow`. It must also save each chapter page of `pm_mem_index`. The exact format in which these are saved are up to the driver.

When restoring a memory image, after the Hibernate core has allocated storage for the saved memory, the backend's `read_image()` method is called. `pm_mem_index` contains enough allocated chapters to store the saved chapters and `pm_mem_shadow` contains enough allocated chapters and pages to store all of the saved pages. The backend must populate all of these.

pmdisk

`pmdisk` is a simple hibernate backend driver. It uses a dedicated partition with a custom format for storing system state. Internally, pmdisk uses the bio layer to read and write pages directly to/from the disk.

A pmdisk partition may be created using a utility called `pmdisk`. It simply writes a pmdisk header to a partition, which is defined as:

```
#define PM_HIBERNATE_SIG "PMHibernate"
#define PM_HIBERNATE_VER 1

#define PM_UNUSED_SPACE \
   (PAGE_SIZE - (4 * \
   sizeof(unsigned long) + 16))

struct pmdisk_header {
  char  h_unused[PM_UNUSED_SPACE];
  unsigned long h_version;
  unsigned long h_chksum;
  unsigned long h_pages;
  unsigned long h_chapters;
  char          h_sig[16];
} __attribute__((packed));
```

Internally, the pmdisk backend driver reads the header from the first page of the configured partition when its `open()` method is called. It verifies that it is a pmdisk partition, and sets `pm_num_pages` if there is an image stored on the disk.

On a `close()` call, pmdisk set the `h_pages`, `h_chapters`, and `h_chksum` fields of the header and writes it to the first page on the disk. Note that on a memory restore operation, `pm_num_pages` will be 0, signifying the memory image on the disk is no longer valid.

A saved memory image on a pmdisk partition is layed out like:

0:	pmdisk header
1 to `Nc`	Saved chapters of `pm_mem_index`
`Nc` to `Np`	Saved pages from `pm_mem_shadow`

On a `write_image()` call, pmdisk will first initialize an internal checksum variable.It will then write each chapter from `pm_mem_index` to disk, then each page from `pm_mem_shadow` to disk. As it writes each page, it will pass it to a checksum function. The checksum function is simple and definitely not cryptographically secure. But, it does provide an easy verification that an image on disk is valid.

On a `read_image()` call, pmdisk reads each chapter into `pm_mem_index` and each page into `pm_mem_shadow`. As it reads each page, it checksums them. Once all pages have been read, it compares the current checksum with the `h_chksum` field of the header. It returns success only if they match.

The internal pmdisk exports a sysfs attribute file named 'dev' which userspace must use to tell the kernel of the correct pmdisk partition to use. There is currently no way for pmdisk to automatically detect any valid partitions in the system.

The value that userspace must write to `dev` is a 16-bit `dev_t` value in hexadecimal format containing the major/minor number pair of the device to use. This format is not favored, but

is the only current method for obtaining a reference to a specific block device at the time of writing. This interface will change in the future.

5 Other Power Management Options

So far, a lot of talk has been dedicated to describing the internals of the new power management subsystem, but little has been given to describe how the new infrastructure interacts with current power management options. This section describes those relationships, and although it focuses on options specific to ia32 platforms, the relationships should be extendable to other platforms.

ACPI

In terms of system power management, fits nicely into the new PM infrastructure. It behaves as a PM driver, and provides platform-specific hooks to transition the system into a low-power state. At the basic SPM level, this is all that is required, though ACPI offers a potentially much more powerful solution, since it it exposes more intimate knowledge of the platform power requirements than has ever been available on ia32 platforms (e.g. response latencies, power consumption etc.). Exploiting this knowledge is up to the ACPI platform driver to expose these attributes via sysfs.

ACPI offers similar potential for device power management. Devices that appear in the firmware's DSDT (Differentiated System Description Table) may expose a very fine-grained level of detail about the devices' power requirements and capabilities.

ACPI also stress the capabilities of device Performance States. A performance state is a power state that describes a trade-off between the capabilities of a device against the power consumption of the device. In each performance state that a device supports, the device is fully running, but different functional hardware units may be powered off to conserve power. The driver model does not explicitly recognize performance states, though the new PM extensions to the driver model provide a framework that could easily be extended to recognize performance states.

APM

APM power management does not appear on very many new systems, but the current Linux installed base includes a large number of APM-capable computers. The new PM model was not developed with APM, or any firmware-driven PM model, in mind. However, care was taken to ensure that it conceptually made sense to use such mechanisms as low-level platform drivers for the PM model. No work has been done, however, to convert APM to act as a PM driver for the new model.

pm infratructure

The original PM infrastucture was developed by Andrew Henroid and was very groundbreaking, since nothing like it had been done for the Linux kernel before. It exists in its entirety in:

```
kernel/pm.c
include/linux/pm.h
```

The general idea is that drivers can declare and register an object with the pm infrastructure that is accessed during a power state transition. The idea is very similar to what we have now, though the registration now is implicit when a device is registered with the system. And, based on the implementation, we can guarantee

that each device is notified in proper ancestral order, which the old model cannot do.

Because the new model is far superior the old-style pm infrastructure, it is declared deprecated. All drivers that implement pm callbacks should be converted to use the hooks provided by the new driver model.

swsusp

swsusp is a mechanism for doing suspend-to-disk by saving kernel state to unused swap space. It was also a ground-breaking feature, as it was the first true suspend-to-disk implementation for Linux. There are some questionable characteristics of swsusp that many people have that the maintainers of swsusp counter are frivilous concerns, and it currently exists as an alternative to the new PM model. However, I've revoked any philosophical issues with swsusp. It can be, and should be ported to be, used as a backend driver for the generic Hibernate mechanism. The current code base could be reduced to a fraction of its current complexity.

6 Resources

The current power management kernel tree can be found in the BitKeeper repository:

```
bk://devloper.osdl.org/
linux-2.5-power
```

Information about the Linux power managment infrastructure, including GNU diffs, documentation and utilities like pmdisk can be found at

```
http://developer.osdl.org/
~mochel/power/
```

General OSDL developer resources can be found at:

```
http://developer.osdl.org/
```

7 Acknowledgements

Many people have contributed to this document, both explicitly and implicitly. First, Linus deserves a mention for encouraging me look at implementing ACPI suspend-to-ram as my first kernel project. Andy Grover and Paul Diefenbaugh of Intel for many things—contributing ACPI to the kernel, for talking with me, for always arguing with and motivating me internally to do things better, and for pushing me over the edge to write the finest OS driver model in existence. Andy Henroid for writing the first open-source power management model and providing a great base—despite its shortcomings—to learn and build from. Pavel Machek for constantly providing code and being energetic about the project. All the swsusp people for doing it in the first place and keeping it up, no matter how much I gripe about it.

Bringing PowerPC Book E to Linux

Challenges in porting Linux to the first PowerPC Book E processor implementation

Matthew D. Porter
MontaVista Software, Inc.
mporter@kernel.crashing.org | mporter@mvista.com

Abstract

The PowerPC *Book E* [1] architecture introduced the first major change to the PowerPC architecture since the original *Green Book* [2] PowerPC processors were introduced. Central to the *Book E* architectural changes is the MMU which is always in translation mode, even during exception processing. This presented some unique challenges for cleanly integrating the architecture into the Linux/PPC kernel.

In addition to the base PowerPC *Book E* architecture changes, the first IBM PPC440 core implementation included 36-bit physical addressing support. Since I/O devices are mapped above the native 32-bit address space, providing support for this feature illuminated several limitations within the kernel resource management and mapping system.

1 Overview of PowerPC Book E architecture

1.1 Book E MMU

It is important to note that *Book E* is a 64-bit processor specification that allows for a 32-bit implementation. Many of the register descriptions in the specification are written describing 64-bit registers where appropriate. In this paper, discussions of *Book E* architecture describe the 32-bit variants of all registers. Currently, all announced PowerPC *Book E* compliant processors are 32-bit implementations of the specification.

In order to understand *Book E* architecture, it is useful to follow the history of the original PowerPC architecture. The original PowerPC architecture was defined at a very detailed level in the *Green Book*. This architecture provides fine details on how the MMU, exceptions, and all possible instructions should operate. The familiar *G3* and *G4* processor families are recent examples of implementations of the *Classic PPC* [3] architecture.

Book E architecture is a result of a collaboration between IBM and Motorola to produce a PowerPC extension which lends itself to the needs of embedded systems. One of the driving forces behind the specification was the desire for each silicon manufacturer to be able differentiate their products. Due to this requirement, the specification falls short of providing enough detail to ensure that system software can be shared among Book E compliant processors.

[1] Full title of specification is *Book E: Enhanced PowerPC Architecture.*

[2] Full title is *PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors.*

[3] An affectionate name bestowed upon all processors that conform to the *Green Book* specification.

With the bad news out of the way, it can be said that all *Book E* processors share some common architecture features. In *Book E*, foremost is the requirement that MMU translation is always enabled. This is in sharp contrast to the *Classic PPC* architecture which uses the more traditional approach of powering up in real mode and disabling the MMU upon taking an exception.

Book E, on the other hand, powers up with a TLB entry active at the system reset vector. This insures that the Initial Program Load (IPL) code can execute to the point of loading additional TLB entries for system software start up. *Book E* architecture also defines several standard page sizes from 1KB through 1TB. In addition, *Book E* calls for the existence of two unique address spaces, AS0 and AS1. AS0 and AS1 are intended to facilitate the emulation of *Classic PPC* real mode on a *Book E* processor. This property can best be described by comparing the *Classic PPC* MMU translation mechanism to the manner in which *Book E* processors switch address spaces.

A *Classic PPC* processor has Instruction Translation (IR) and Data Translation (DR) bits in its Machine State Register (MSR). These bits are used to enable or disable MMU translation. A *Book E* processor has the same bits in the MSR but they are called Instruction Space (IS) and Data Space (DS). The IS and DS bits are used a little differently since they are used to control the current 4GB virtual address space that the processor is executing within. Both *Classic PPC* and *Book E* processors set these bits to zero when an exception is taken. On a *Classic PPC*, this disables the MMU for exception processing. On a *Book E* processor this switches to AS0. If the kernel and user space are run in the context of AS1, then TLB entries for AS0 can be used to emulate *Classic PPC* real mode operation.

1.2 Book E exception vectors

The *Book E* specification allows for a large number of exception vectors to be implemented. Sixteen standard exceptions are listed and space is reserved for an additional 48 implementation dependent exceptions.

The *Book E* exception model differs from the *Classic PPC* model in that the exception vectors are not at fixed memory offsets. *Classic PPC* exception vectors are each allocated 256 bytes. Using a bit in the MSR, the vectors can be located at the top or bottom of the PPC physical memory map.

Book E processors have an Interrupt Vector Prefix Register (IVPR) and Interrupt Vector Offset Registers (IVORs) to control the location of exception vectors in the system. The IVPR is used to set the base address of the exception vectors. Each IVORn register is used to set the respective offset from the IVPR at which the exception vector is located.

2 PPC440GP Book E processor

The first PowerPC *Book E* processor implementation was the IBM PPC440GP. This processor's PPC440 core was the next evolutionary step from the PPC405 cores that were a cross between a *Classic PPC* and a *Book E* PPC design.

The PPC440 core has a 64 entry unified Translation Lookaside Buffer (TLB) as its major implementation specific MMU feature. This TLB design relies on software to implement any desired TLB entry locking, determine appropriate entries for replacement, and to perform page table walk and load TLB entries. This approach is very flexible, but can be performance limiting when compared to processors that provide hardware table walk, Pseudo Least Recently Used (PLRU) replacement algorithms,

and TLB entry locking mechanisms.

The PPC440 core also implements a subset of the allowed *Book E* page sizes. Implemented sizes range from 1KB to 256MB, but exclude the 4MB and 64MB page sizes.

3 Existing Linux/PPC kernel ports

Linux/PPC already has a number of sub-architecture families which require their own `head.S` implementation. `head.S` is used by *Classic PPC* processors, `head_8xx.S` is used by the Motorola MPC8xx family, and `head_4xx.S` is used by the IBM PPC40x family of processors. In order to encapsulate the unique features of a *Book E* processor, it was necessary to create an additional `head_440.S`.

Traditionally, PowerPC has required fixed exception vector locations, so all ports have followed the basic *Classic PPC* `head.S` structure of a small amount of code at the beginning of the kernel image which branches over the exception vector code that is resident at fixed vector locations. This is true even on PPC405's `head_4xx.S` even though the PPC405 offers dynamic exception vectors in the same manner as a *Book E* compliant processor. With the standard Linux/PPC linker script, `head.S` is guaranteed to be at the start of the kernel image which must be loaded at the base of system memory.

4 Initial Book E kernel port

4.1 Overview

The first *Book E* processor kernel port in the community was done on the Linux/PPC 2.4 development tree [4]. The PPC440GP was the first publicly available *Book E* compliant processor available to run Linux.

4.2 MMU handling approaches

Several approaches were considered for implementing the basic handling of exception processing within the constraints of a *Book E* MMU. With the MMU always being enabled, it is not possible for the processor to access instructions and data using a physical address during exception processing. At a minimum, it is necessary to have a TLB entry covering the PPC exception vectors to ensure that the first instruction of a given exception implementation can be fetched.

One implementation path is to create TLB entries that cover all of kernel low memory within the exception processing address space (AS0). These entries would be locked so they could not be invalidated or replaced by kernel or user space TLB entries. This is the simplest approach for the 2.4 kernel where page tables are limited to kernel low memory. This allows task structs and page tables to be allocated via the normal `__get_free_page()` or `__get_free_pages()` calls using the `GFP_KERNEL` flag. Unfortunately, this approach has some drawbacks when applied to a PPC440 System on a Chip (SoC) implementation.

The PPC440 core provides large TLB sizes of 1MB, 16MB, and 256MB. A simple solution would be to cover all of kernel low memory with locked 256MB TLB entries. By default, Linux/PPC restricts maximum kernel low memory to 768MB. This would only require a maximum of 3 entries in the TLB to be consumed on a permanent basis. Unfortunately, this approach will not work since the behavior of the system is undefined in the event that system memory is not a multiple of 256MB. In

[4] Information on Linux/PPC kernel development trees can be found at `http://penguinppc.org/dev/kernel.shtml`

practice, this generates speculative cache line fetches past the end of system memory which result in a Machine Check exception.

The next logical solution would be to use a combination of locked large TLB entries to cover kernel low memory. In this approach, we quickly run into a situation where the locked TLB entries consume too much of the 64 entry TLB. Consider a system with 192MB of system RAM. In this system, it would be necessary to lock 12 16MB TLB entries permanently to cover all of kernel low memory. This approach would leave only 52 TLB entries available for dynamic replacement. Artificially limiting the already small TLB would put further pressure on the TLB and most likely adversely affect performance.

4.3 Linux 2.4 MMU Solution

A different approach is necessary because there does not seem to be a good method to lock all of kernel low memory into the PPC440 TLB. One possible approach is to limit the area in which kernel data structures are allocating by creating a special pool of memory. Implementing the memory pool approach involves the following steps:

1. Force all kernel construct allocation to occur within a given memory region.
2. Ensure that the given memory region is covered by a locked TLB within exception space.

The system is already required to maintain one locked TLB entry to ensure that instructions can be fetched from the exception vectors without resulting in a TLB miss. Therefore, the kernel construct memory region can simply be the pool of free memory that follows the kernel at the base of system memory. The locked TLB entry is then set to a size of 16MB to ensure that it covers both the kernel (including exception vectors) and some additional free memory. A TLB entry size of 16MB was chosen because it is the smallest amount of RAM one could conceivably find on a PPC440 system.

It was then necessary to create a facility to control allocation from a given memory region. The easiest way to force allocation of memory from a specific address range in Linux is to make use of the `GFP_DMA` flag to the zone allocator calls. The allocation of task structs, pgds, and ptes was modified to result in an allocation from the DMA zone. Figure 1 shows a code fragment demonstrating how this is implemented for PTE allocation.

The PPC memory management initialization was then modified to ensure that 16MB of memory is placed into `ZONE_DMA` and the remainder ends up in `ZONE_NORMAL` or `ZONE_HIGHMEM` as appropriate.

With this structure, all kernel stacks and page tables are allocated within `ZONE_DMA`. The single locked TLB entry for the first 16MB of system memory ensures that no nested exceptions can occur while processing an exception.

One complication that resulted from using the `ZONE_DMA` zone in this manner is that there can be many early consumers of low memory in `ZONE_DMA`. It was necessary to place an additional kludge in the early Linux/PPC memory management initialization to ensure that some amount of the 16MB of `ZONE_DMA` region would be free after the bootmem allocator was no longer in control of system memory. This was encountered when a run with 1GB of system RAM caused the `page structs` to nearly consume all of the `ZONE_DMA` region. This, of course, is a fatal condition due to the allocation of all task structs and page tables from `ZONE_DMA`.

```
static inline pte_t * pte_alloc_one(struct mm_struct *mm, unsigned long address)
{
        pte_t *pte;
        extern int mem_init_done;
        extern void *early_get_page(void);

        if (mem_init_done)
#ifndef CONFIG_440
                pte = (pte_t *) __get_free_page(GFP_KERNEL);
#else
        /* Allocate from GFP_DMA to get entry in pinned TLB region */
        pte = (pte_t *) __get_free_page(GFP_DMA);
#endif
        else
                pte = (pte_t *) early_get_page();
}
```

Figure 1: pte_alloc_one() implementation

4.4 Virtual exception processing

One minor feature of the PPC440 port is the use of dynamic exception vectors. As allowed by the *Book E* architecture, exception vectors are placed in head_440.S using the following macro:

```
#define START_EXCEPTION(label) \
        .align 5;              \
label:
```

This is used to align each exception vector entry to a 32 byte boundary as required by the PPC440 core. The following code from head_440.S shows how the macro is used at the beginning of an exception handler:

```
        /* Data TLB Error Interrupt */
        START_EXCEPTION(DataTLBError)
        mtspr        SPRG0, r20
```

This code fragment illustrates how each exception vector is configured based on its link location:

```
        SET_IVOR(12, WatchdogTimer);
        SET_IVOR(13, DataTLBError);
        SET_IVOR(14, InstructionTLBError);
```

The SET_IVOR macro moves the label address offset into a *Book E* IVOR. The first parameter specifies which IVOR is the target of the move. Once the offsets are configured and the IVPR is configured with the exception base prefix address, exceptions will then be routed to the link time specified vectors.

An interesting thing to note is that the Linux 2.4 *Book E* kernel actually performs exception processing at the kernel virtual addresses. I.e., the exception vectors are located at an offset from 0xc0000000.

5 New Book E kernel port

5.1 Overview

Working to get the *Book E* kernel support into the Linux/PPC 2.5 development tree resulted in some discussions regarding the long-term viability of the ZONE_DMA approach used in the 2.4 port. One of the major issues has been that the 2.5 kernel moved the allocation of task structs to generic slab-based kernel code. This move broke the current *Book E* kernel model since it is no longer possible to force allocation of task structs to occur within ZONE_DMA. Another important reason for considering a

change is that the current method is somewhat of a hack. That is, `ZONE_DMA` is used in a manner in which it was not intended.

5.2 In-exception TLB misses

The first method investigated to eliminate `ZONE_DMA` usage simply allows nested exceptions to be handled during exception processing. Exception processing code can be defined as the code path from when an exception vector is entered until the processor returns to kernel/user processing. On a lightweight TLB miss, this can happen immediately after a TLB entry is loaded. On heavyweight exceptions, this may occur when `transfer_to_handler` jumps to a heavyweight handler routine in kernel mode.

Upon examining the exception processing code, it becomes apparent that the only standard exception that can occur is the DataTLBError exception. This is because exception vector code must be contained within a locked TLB entry, so no InstructionTLBError conditions can occur. Further, early exception processing accesses a number of kernel data constructs. These include kernel stacks, pgds, and ptes. By writing a non-destructive DataTLBError handler it is possible to safely process data TLB misses within exception processing code.

In order to make the DataTLBError handler safe, it is necessary not to touch any of the PowerPC Special Purpose General Registers (SPRGs) when a DataTLBError exception is taken. Instead, a tiny stack is created within the memory region covered by the locked TLB entry. This stack is loaded with the context of any register that need to be used during DataTLBError processing. The following code fragment shows the conventional DataTLBError register save mechanism:

```
        /* Data TLB Error Interrupt */
        START_EXCEPTION(DataTLBError)
        mtspr   SPRG0, r10
        mtspr   SPRG1, r11
        mtspr   SPRG4W, r12
        mtspr   SPRG5W, r13
        mtspr   SPRG6W, r14
        mfcr    r11
        mtspr   SPRG7W, r11
        mfspr   r10, SPRN_DEAR
```

In the non-destructive version of the DataTLBError, the code looks like following:

```
        START_EXCEPTION(DataTLBError)
        stw     r10,tlb_r10@l(0);
        stw     r11,tlb_r11@l(0);
        stw     r12,tlb_r12@l(0);
        stw     r13,tlb_r13@l(0);
        stw     r14,tlb_r14@l(0);
        mfcr    r11
        stw     r11,tlb_cr@l(0);
        mfspr   r11, SPRN_MMUCR
        stw     r11,tlb_mmucr@l(0);
        mfspr   r10, SPRN_DEAR
```

Here, the `tlb_*` locations within the locked TLB region are used to save register state rather than the SPRGs.

If we were to continue to perform exception processing from native kernel virtual address, we would have a problem. The `tlb_*` locations allocated within `head_44x.S` would be at some offset from 0xc0000000. A store to any address with a non-zero most significant 16 bits would require that an intermediate register be used to load the most significant bits of the address.

This issue made it necessary to make the switch to emulation of *Classic PPC* real mode. This is accomplished by placing the dynamic exception vectors at a virtual address offset from address zero and providing a locked TLB entry covering this address space. By doing so it became possible to access exception stack locations using zero indexed loads and stores.

In the DataTLBError handler, each access to kernel data which may not have a TLB entry is

protected. A `tlbsx.` instruction is used to determine if there is already a TLB entry for the address that is to be accessed. If a TLB entry exists, the access is made. However, if the TLB entry does not exist, a TLB entry is created before accessing the resource. This method is illustrated in the following code fragment based on the `linuxppc-2.5 head_44x.S`:

```
        /* Stack TLB entry present? */
3:      mfspr   r12,SPRG3
        tlbsx.  r13,0,r12
        beq     4f
        /* Load stack TLB entry */
        TLB_LOAD;

        /* Get current thread's pgd */
4:      lwz     r12,PGDIR(r12)
```

Using this strategy, the DataTLBError handler gains the ability resolve any possible TLB miss exceptions before they can occur. Once it has performed the normal software page table walk and has loaded the faulting TLB entry, it can return to the point of the exception. Of course, that exception may now be either from a kernel/user context or from an exception processing context. A DataTLBError can now be easily handled from any context.

5.3 Keep It Simple Stupid

Sometimes one has to travel a long road to eventually come back to the simple solution. This project has been one of those cases. An implementation of the in-exception tlb miss method showed that the complexity of the TLB handling code had gone up by an order of magnitude. It is desirable (for maintenance and quality reasons) to keep the TLB handling code as simple as possible.

The KISS approach pins all of kernel low memory with 256MB TLB entries in AS0. The number of TLB entries is determined from the discovered kernel low memory size. A high water mark value is used to mark the highest TLB slot that may be used when creating TLB entries in AS1 for the kernel and user space. The remaining TLB slots are consumed by the pinned TLB entries.

This approach was previously thrown out due to the occurrence of speculative data cache fetches that would result in a fatal machine check exception. This situation occurs when the system memory is not aligned on a 256MB boundary. In these cases, the TLB entries cover unimplemented address space. The data cache controller will speculatively fetch past the end of system memory if any access is performed on the last cache line of the last system memory page frame.

The trick to make this approach stable is to simply reserve the last page frame of system memory so it may not be allocated by the kernel or user space. This could be done via the bootmem allocator, but in order to accomplish it during early MMU initialization it is necessary to utilize the PPC specific `mem_pieces` allocation API. Using this trick allows for a simple (and maintainable) implementation of PPC440 tlb handling.

5.4 Optimizations

One clear enhancement to the low-level TLB handling mechanism is to support large page sizes for kernel low memory. This support is already implemented[5] for the PPC405 family of processors that implement a subset of the *Book E* page sizes. Enabling the TLB miss handlers to load large TLB entries for kernel low memory guarantees a lighter volume of exceptions taken from accesses of kernel low memory.

Although this is a common TLB handling optimization in the kernel, a minor change to

[5] In the `linuxppc-2.5` development tree

the KISS approach could eliminate the need to provide large TLB replacement for kernel low memory. The change is to simply modify the KISS approach to run completely from AS0. AS1 would not longer be used for user/kernel operation since all code would run from the AS0 context. This yields the same performance gain by reducing TLB pressure as the large TLB replacement optimization. However, this variant leverages the TLB entries that are already pinned for exception processing.

6 36-bit I/O support

6.1 Overview of large physical address support

The PPC440GP processor implementation supports 36-bit physical addressing on its system bus. 36-bit physical addressing has already been supported on other processors with a native 32-bit MMU as found in IA32 PAE implementations. However, the PPC440GP implements a 36-bit memory map with I/O devices above the first 4GB of physical memory.

The basic infrastructure by which large physical addresses are supported is similar to other architectures. In the case of PPC440GP, we define a pte to be an `unsigned long long` type. In order to simplify the code, we define our page table structure as the usual two level layout, but with an 8KB pgd. Rather than allocating a single 4KB page for a pgd, we allocate two pages to meet this requirement.

In order to share some code between large physical address and normal physical address PPC systems, a new type is introduced:

```
#ifndef CONFIG_440
#include <asm-generic/mmu.h>
#else
typedef unsigned long long phys_addr_t;
extern phys_addr_t
fixup_bigphys_addr(phys_addr_t, phys_addr_t);
#endif
```

This `typedef` allows low-level PPC memory management routines to handle both large and normal physical addresses without creating a separate set of calls. On a PPC440-based core it is a 64-bit type, yet it remains a 32-bit type on all normal physical address systems.

6.2 Large physical address I/O kludge

The current solution for managing devices above 4GB is somewhat of a "necessary kludge." In a dumb bit of luck, the PPC440GP memory map was laid out in such a way that made it easy to perform a simple translation of a 32-bit physical address (or Linux resource) into a 36-bit physical address suitable for consumption by the PPC440 MMU.

All PPC440GP on-chip I/O devices and PCI address spaces were neatly laid out so that their least significant 32-bits of physical address did not overlap.

A PPC440 specific `ioremap()` call is created to allow a 32-bit resource to be mapped into virtual address space. Figure 2 illustrates the `ioremap()` implementation.

This `ioremap()` implementation works by calling a translation function to convert a 32-bit resource into a 64-bit physical address. `fixup_bigphys_addr()` compares the 32-bit resource value to several PPC440GP memory map ranges. When it matches one distinct range, it concatenates the most significant 32-bits of the intended address range. This results in a valid PPC440GP 64-bit physical address that can then be passed to the local `ioremap64()` routine to create the virtual mapping .

This method works fine for maintaining compatibility with a large amount of generic PCI device drivers. However, the approach quickly falls apart when a driver implements `mmap()`.

```
void *
ioremap(unsigned long addr, unsigned long size)
{
        phys_addr_t addr64 = fixup_bigphys_addr(addr, size);

        return ioremap64(addr64, size);
}
```

Figure 2: PPC440GP `ioremap()` implementation

The core of most driver `mmap()` implementations is a call to `remap_page_range()`. This routine is prototyped as follows:

```
int remap_page_range(unsigned long from,
                     unsigned long to,
                     unsigned long size,
                     pgprot_t prot);
```

The `to` parameter is the physical address of the memory region that is to be mapped. The current implementation assumes that a physical address size is always equal to the native word size of the processor. This is obviously now a bad assumption for large physical address systems because it is not possible to pass the required 64-bit physical address.

Figure 3 shows the implementation of `remap_page_range()` for the mmap compatibility kludge [6].

The physical address parameter is now passed as a `phys_addr_t`. On large physical address platforms, the `fixup_bigphys_addr()` call is implemented to convert a 32-bit value (normally obtained from a 32-bit resource) into a platform specific 64-bit physical address. The `remap_pmd_range()` and `remap_pte_range()` calls likewise have their physical address parameters passed using a `phys_addr_t`.

This implementation allows device drivers implementing `mmap()` using `remap_page_range()` to run unchanged on a large physical address system. Unfortunately, this is not a complete solution to the problem.

The `fixup_bigphys_addr()` routine cannot necessarily be implemented for all possible large physical address space memory maps. Some systems will require discrete access to the full 36-bit or larger physical address space. In these cases, there is a need to allow the `mmap()` system call to handle a 64-bit value on a 32-bit platform.

6.3 Proper large physical address I/O support

One approach to resolving this issue is to change the parameters of `remap_page_range()` and friends as was done in the mmap compatibility kludge. The physical address to be mapped would then be manipulated in a `phys_addr_t`. On systems with a native word size `phys_addr_t` there is no effect. The important piece of this approach is that all callers of `remap_page_range()` would need to do any manipulation of physical addresses using a `phys_addr_t` variable to ensure portability.

In the specific case of a `fops->mmap` implementation, a driver must now be aware that a `vma->pgoff` can contain an address that is greater than the native word size. In any case, the `vma->pgoff` value would be shifted by `PAGE_OFFSET` in order to yield a system specific physical address in a `phys_addr_t`.

[6] Patches for this support can be found at `ftp://source.mvista.com/pub/linuxppc/`

```
int
remap_page_range(unsigned long from, phys_addr_t phys_addr, unsigned long size, pgprot_t prot)
{
        .
        .
        .
        phys_addr = fixup_bigphys_addr(phys_addr, size);
        phys_addr -= from;
        .
        .
        .
}
```

Figure 3: PPC440GP `remap_page_range()` implementation

Once we allow for 64-bit physical address mapping on 32-bit systems, it becomes necessary to expand the resource subsystem to match. In order for standard PCI drivers to remain portable across standard and large physical address systems, it is necessary to ensure that a resource can represent a 64-bit physical address on a large physical address system. Building on the approach of using `phys_addr_t` to abstract the native system physical address size, this can now be the native storage type for resource fields. In doing so, it is also important to extend the concept to user space to ensure that common applications like XFree86 can parse 64-bit resources on a 32-bit platform and cleanly `mmap()` a memory region.

7 Conclusion

The Linux kernel is remarkably flexible in handling ports to new processors. Despite significant architectural changes in the PowerPC *Book E* specification, it was possible to enable the PPC440GP processor within the existing Linux abstraction layer in a reasonable amount of time. As with all Linux projects, this one is still very much a work-in-progress. Development in the Linux 2.5 tree offers an opportunity to explore some new routes for better PowerPC *Book E* kernel support.

During this project, I have had the opportunity to learn which features of a *Book E* processor would be most useful in supporting a Linux kernel port. Clearly, the most important feature in this respect is an abundance of TLB entries. The PPC440 core's 64 entry TLB is the single most limiting factor for producing a simple *Book E* port. If the PPC440 core had 128 or 256 TLB entries, the work on porting Linux to the processor would have been far easier.

Although these new processors are now running the Linux kernel, this support does not yet address 100% of this platform's new architectural features. As with many areas in the Linux kernel, support for large physical address mapping needs to evolve with emerging processor technologies. Without a doubt, the increased number of processors implementing large physical address I/O functionality help to make the Linux kernel community aware of the kernel requirements inherent in this technology.

8 Trademarks

Asynchronous I/O Support in Linux 2.5

Suparna Bhattacharya
Steven Pratt
Badari Pulavarty
Janet Morgan
IBM Linux Technology Center
suparna@in.ibm.com, slpratt@us.ibm.com,
pbadari@us.ibm.com, janetmor@us.ibm.com

Abstract

This paper describes the Asynchronous I/O (AIO) support in the Linux® 2.5 kernel, additional functionality available as patchsets, and plans for further improvements. More specifically, the following topics are treated in some depth:

- Asynchronous filesystem I/O
- Asynchronous direct I/O
- Asynchronous vector I/O

As of Linux 2.5, AIO falls into the common mainline path underlying all I/O operations, whether synchronous or asynchronous. The implications of this, and other significant ways in which the design for AIO in 2.5 differs from the patches that existed for 2.4, are explored as part of the discussion.

1 Introduction

All modern operating systems provide a variety of I/O capabilities, each characterized by their particular features and performance. One such capability is Asynchronous I/O, an important component of Enterprise Systems which allows applications to overlap processing with I/O operations for improved utilization of CPU and devices.

AIO can be used to improve application performance and connection management for web servers, proxy servers, databases, I/O intensive applications and various others.

Some of the capabilities and features provided by AIO are:

- The ability to submit multiple I/O requests with a single system call.
- The ability to submit an I/O request without waiting for its completion and to overlap the request with other processing.
- Optimization of disk activity by the kernel through combining or reordering the individual requests of a batched I/O.
- Better CPU utilization and system throughput by eliminating extra threads and reducing context switches.

2 Design principles

An AIO implementation can be characterized by the set of design principles on which it is based. This section examines AIO support in the Linux kernel in light of a few key aspects and design alternatives.

2.1 External interface design alternatives

There are at least two external interface design alternatives:

- A design that exposes essentially the same interfaces for synchronous and asynchronous operations with options to distinguish between mode of invocation [2].
- A design that defines a unique set of interfaces for asynchronous operations in support of AIO-specific requirements such as batch submission of different request types [4].

The AIO interface for Linux implements the second type of external interface design.

2.2 Internal design alternatives

There are several key features and possible approaches for the internal design of an AIO implementation:

- System design:
  - Implement the entire path of the operation as fully asynchronous from the top down. Any synchronous I/O is just a trivial wrapper for performing asynchronous I/O and waiting for its completion [2].
  - Synchronous and asynchronous paths can be separate, to an extent, and can be tuned for different performance characteristics (for example, minimal latency versus maximal throughput) [10].
- Approaches for providing asynchrony:
  - Offload the entire I/O to thread pools (these may be either user-level threads, as in glibc, or kernel worker threads).
  - Use a hybrid approach where initiation of I/O occurs asynchronously and notification of completion occurs synchronously using a pool of waiting threads ([3] and [13]).
  - Implement a true asynchronous state machine for every operation [10].
- Mechanisms for handling user-context dependencies:
  - Convert buffers or other such state into a context-independent form at I/O submission (e.g., by mapping down user-pages) [10].
  - Maintain dedicated per-address space service threads to execute context-dependent steps in the caller's context [3].

The internal design of the AIO support available for Linux 2.4 and the support integrated into Linux 2.5 differ on the above key features. Those differences will be discussed in some detail in later sections.

Other design aspects and issues that are relevant to AIO but which are outside the main focus of this paper include ([8] describes some of these issues in detail):

- The sequencing of operations and steps within an operation that supports

POSIX_SYNCHRONIZED_IO and **POSIX_PRIORITIZED_IO** requirements ([5]), as well as the extent of flexibility to order or parallelize requests to maximize throughput within reasonable latency bounds.

- AIO throttling: deciding on the queue depths and returning an error (**-EAGAIN**) when the depth is exceeded, or if resources are unavailable to complete the request (rather than forcing the process to sleep).
- Completion notification, queuing, and wakeup policies including the design of a flexible completion notification API and optimization considerations like cache-warmth, latency and batching. In the Linux AIO implementation, every AIO request is associated with a completion queue. One or more application threads explicitly wait on this completion queue for completion event(s), where flexible grouping is determined at the time of I/O submission. An exclusive LIFO wakeup policy is used among multiple such threads, and a wakeup is issued whenever I/O completes.
- Support for I/O cancellation.
- User/kernel interface compatibility (per POSIX).

2.3 2.4 Design

The key characteristics of the AIO implementation for the Linux 2.4 kernel are described in [8] and available as patches at [10]. The design:

- Implements asynchronous I/O paths and interfaces separately, leaving existing synchronous I/O paths unchanged. Reuse of existing, underlying asynchronous code is done where possible, for example, raw I/O.
- Implements an asynchronous state machine for all operations. Processing occurs in a series of non-blocking steps driven by asynchronous waitqueue callbacks. Each stage of processing completes with the queueing of deferred work using "work-to-do" primitives. Sufficient state is saved to proceed with the next step, which is run in the context of a kernel thread.
- Maps down user pages during I/O submission. Modifies the logic to transfer data to/from mapped user pages in order to remove the user-context dependency for the copy to/from userspace buffers.

The advantages of these choices are:

- Synchronous I/O performance is unaffected by asynchronous I/O logic, which allows AIO to be implemented and tuned in a way that is optimum for asynchronous I/O patterns.
- The work-to-do primitive permits state to be carried forward to enable continuation from exactly where processing left off when a blocking point was encountered.
- The need for additional threads to complete the I/O transfer in the caller's context is avoided.

There are, however, some disadvantages to the AIO implementation (patchset) for Linux 2.4:

- The duplication of logic between synchronous and asynchronous paths makes the code difficult to maintain.

- The asynchronous state machine is a complex model and therefore more prone to errors and races that can be hard to debug.
- The implementation can lead to inefficient utilization of TLB mappings, especially for small buffers. It also forces the pinning down of all pages involved in the entire I/O request.

These problems motivated a new approach for the implementation of AIO in the Linux 2.5 kernel.

2.4 2.5 Design

Although the AIO design for Linux 2.5 uses most of the core infrastructure from the 2.4 design, the 2.5 design is built on a very different model:

- Asynchronous I/O has been made a first-class citizen of the kernel. Now AIO paths underlie regular synchronous I/O interfaces instead of just being grafted from the outside.
- A retry-based model replaces the earlier work-to-do state-machine implementation. Retries are triggered through asynchronous notification as each step in the process completes. However in some cases, such as direct I/O, asynchronous completion notification occurs directly from interrupt context without requiring any retries.
- User-context dependencies are handled by making worker threads take on (i.e., temporarily switch to) the caller's address space when executing retries.

In a retry-based model, an operation executes by running through a series of iterations. Each iteration makes as much progress as possible in a non-blocking manner and returns. The model assumes that a restart of the operation from where it left off will occur at the next opportunity. To ensure that another opportunity indeed arises, each iteration initiates steps towards progress without waiting. The iteration then sets up to be notified when enough progress has been made and it is worth trying the next iteration. This cycle is repeated until the entire operation is finished.

The implications and issues associated with the retry-based model are:

- Tuning for the needs of both synchronous and asynchronous I/O can be difficult because of the issues of latency versus throughput. Performance studies are needed to understand whether AIO overhead causes a degradation in synchronous I/O performance. It is expected that the characteristics are better when the underlying operation is already inherently asynchronous or rewritten to an asynchronous form, rather than just modified in order to be retried.
- Retries pass through some initial processing steps each time. These processing steps involve overhead. Saving state across retries can help reduce some of the redundant regeneration, albeit with some loss of generality.
- Switching address spaces in the retry thread can be costly. The impact would probably be experienced to a greater extent when multiple AIO processes are running. Performance studies are needed to determine if this is a problem.

Note that I/O cancellation is easier to handle in a retry-based model; any future retries can simply be disabled if the I/O has been cancelled.

Retries are driven by AIO workqueues. If a retry does not complete in a very short time, it can delay other AIO operations that are underway in the system. Therefore, tuning the AIO workqueues and the degree of asynchrony of retry instances each have a bearing on overall system performance.

3 AIO support for filesystem I/O

The Linux VFS implementation, especially as of the 2.5 kernel, is well-structured for retry-based I/O. The VFS is already capable of processing and continuing some parts of an I/O operation outside the user's context (e.g., for readahead, deferred writebacks, syncing of file data and delayed block allocation). The implementation even maintains certain state in the `inode` or address space to enable deferred background processing of writeouts. This ability to maintain state makes the retry model a natural choice for implementing filesystem AIO.

Linux 2.5 is currently without real support for regular (buffered) filesystem AIO. While ext2, JFS and NFS define their `aio_read` and `aio_write` methods to default to `generic_file_aio_read/write`, these routines show fully synchronous behavior unless the file is opened with **O_DIRECT**. This means that an `io_submit` can block for regular AIO read/write operations while the application assumes it is doing asynchronous I/O.

Our implementation of the retry model for filesystem AIO, available as a patchset from [6], involved identifying and focusing on the most significant blocking points in an operation. This was followed by observations from initial experimentation and profiling results, and the conversion of those blocking points to retry exit points.

The implementation we chose starts retries at a very high level. Retries are driven directly by the AIO infrastructure and kicked off via asynchronous waitqueue functions. In synchronous I/O context, the default waitqueue entries are synchronous and therefore do not cause an exit at a retry point.

One of the goals of our implementation for filesystem AIO was to minimize changes to existing synchronous I/O paths. The intent was to achieve a reasonable level of asynchrony in a way that could then be further optimized and tuned for workloads of relevance.

3.1 Design decisions

- Level at which retries are triggered:

  The high-level AIO code retries filesystem read/write operations, passing in the remaining parts of the buffer to be read or written with each retry.

- How and when a retry is triggered:

  Asynchronous waitqueue functions are used instead of blocking waits to trigger a retry (to "kick" a dormant `iocb` into action) when the operation is ready to continue.

  Synchronous routines such as `lock_page`, `wait_on_page_bit`, and `wait_on_buffer` have been modified to asynchronous variations. Instead of blocking, these routines queue an asynchronous wait and return with a special return code, **-EIOCBRETRY**.

  The return value is propagated all the way up to the invoking AIO handler. For this process to work correctly, the calling routine at each level in the call chain needs to break out gracefully if a callee returns the **-EIOCBRETRY** exit code.

- Operation-specific state preserved across retries:

In our implementation [7], the high-level AIO code adjusts the parameters to read or write as retries progress. The parameters are adjusted by the retry routine based on the return value from the filesystem operation indicating the number of bytes transferred.

A recent patch by Benjamin LaHaise [11] proposes moving the filesystem API `read/write` parameter values to the `iocb` structure. This change would enable retries to be triggered at the API level rather than through a high-level AIO handler.

- Extent of asynchrony:

  Ideally, an AIO operation is completely non-blocking. If too few resources exist for an AIO operation to be completely non-blocking, the operation is expected to return **-EAGAIN** to the application rather than cause the process to sleep while waiting for resources to become available.

  However, converting all potential blocking points that could be encountered in existing file I/O paths to an asynchronous form involves trade-offs in terms of complexity and/or invasiveness. In some cases, this tradeoff produces only marginal gains in the degree of asynchrony.

  This issue motivated a focus on first identifying and tackling the major blocking points and less deeply nested cases to achieve maximum asynchrony benefits with reasonably limited changes. The solution can then be incrementally improved to attain greater asynchrony.

- Handling synchronous operations:

  No retries currently occur in the synchronous case. The low-level code distinguishes between synchronous and asynchronous waits, so a break-out and retry occurs only in the latter case while the process blocks as before in the event of a synchronous wait. Further investigation is required to determine if the retry model can be used uniformly, even for the synchronous case, without performance degradation or significant code changes.

- Compatibility with existing code:

  - Wrapper routines are needed for synchronous versions of asynchronous routines.
  - Callers that cannot handle asynchronous returns need special care e.g., making sure that a synchronous context is specified to potentially asynchronous callees.
  - Code that can be triggered in both synchronous and asynchronous mode may present some tricky issues.
  - Special cases like code that may be called via page faults in asynchronous context may need to be treated carefully.

3.2 Filesystem AIO read

A filesystem read operation results in a page cache lookup for each full or partial page of data requested to be read. If the page is already in the page cache, the read operation locks the page and copies the contents into the corresponding section of the user-space buffer. If the page isn't cached, then the read operation creates a new page-cache page and issues I/O to read it in. It may, in fact, read ahead several pages at the same time. The read operation then waits for the I/O to complete (by waiting for a lock on the page), and then performs the copy into user space.

Based on initial profiling results the crucial blocking points identified in this sequence were found to occur in:

- `lock_page`
- `cond_resched`
- `wait_on_page_bit`

Of these routines the following were converted to retry exit points by introducing corresponding versions of the routines that accept a wait-queue entry parameter:

```
lock_page --> lock_page_wq
wait_on_page_bit -->
   wait_on_page_bit_wq
```

When a blocking condition arises, these routines propagate a return value of **-EIOCBRETRY** from `generic_file_aio_read`. When unblocked, the waitqueue routine which was notified activates a retry of the entire sequence.

As an aside, the existing readahead logic helps reduce retries for AIO just as it helps reduce context switches for synchronous I/O. In practice, this logic does not actually cause a volley of retries for every page of a large sequential read.

The following routines are other potential blocking points that may occur in a filesystem read path that have not yet been converted to retry exits:

- `cond_resched`
- meta-data read (get block and read of block bitmap)
- request-queue congestion
- atime updates (corresponding journal updates)

Making the underlying readpages operation asynchronous by addressing the last three blocking points above might require more detailed work. Initial results indicate that significant gains have already been realized without doing so.

3.3 Filesystem AIO write

The degree of blocking involved in a synchronous write operation is expected to be less than in the read case. This is because (unless **O_SYNC** or **O_DSYNC** are specified) a write operation only needs to wait until file blocks are mapped to disk and data is copied into (written to) the page cache. The actual write out to disk typically happens in a deferred way in the context of background kernel threads or earlier in the process via an explicit sync operation. However, for throttling reasons, a wait for pending I/O may also occur in write context.

Some of the more prominent blocking points identified in the this sequence were found to occur in:

- `cond_resched`
- `wait_on_buffer` (during a get block operation)
- `find_lock_page`
- `blk_congestion_wait`

Of these, the following routines were converted to retry exit points by introducing corresponding versions of the routines that accept a wait-queue entry parameter:

```
down --> down_wq
wait_on_buffer --> wait_on_buffer_wq
sb_bread  --> sb_bread_wq
ext2_get_block --> ext2_get_block_wq
find_lock_page --> find_lock_page_wq
blk_congestion_wait -->
     blk_congestion_wait_wq
```

The asynchronous get block support has currently been implemented only for ext2, and only used by `ext2_prepare_write`. All other instances where a filesystem-specific get block routine is involved use the synchronous version. In view of the kind of I/O patterns expected for AIO writes (for example, database workloads), block allocation has not been a focus for conversion to asynchronous mode.

The following routines are other potential blocking points that could occur in a filesystem write path that have not yet been converted to retry exits:

- `cond_resched`
- other meta-data updates, journal writes

Also, the case where **O_SYNC** or **O_DSYNC** were specified at the time when the file was opened has not yet been converted to be asynchronous.

3.4 Preliminary observations

Preliminary testing to explore the viability of the above-described approach to filesystem AIO support reveals a significant reduction in the time spent in `io_submit` (especially for large reads) when the file is not already cached (for example, on first-time access). In the write case, asynchronous get block support had to be incorporated to obtain a measurable benefit. For the cached case, no observable differences were noted, as expected. The patch does not appear to have any effect on synchronous read/write performance.

A second experiment involved temporarily moving the retries into the `io_getevents` context rather than into worker threads. This move enabled a sanity check using `strace` to detect any gross impact on CPU utilization.

Thorough performance testing is underway to determine the effect on overall system performance and to identify opportunities for tuning.

3.5 Issues and todos

- Should the `cond_resched` calls in read/write be converted to retry points?
- Are asynchronous get block implementations needed for other filesystems (e.g., JFS)?
- Optional: should the retry model be used for direct I/O (DIO) or should synchronous DIO support be changed to wait for the completion of asynchronous DIO?
- Should relevant filesystem APIs be modified to add an explicit waitqueue parameter?
- Should the `iocb` state be updated directly by the filesystem APIs or by the high-level AIO handler after every retry?

4 AIO support for direct I/O

Direct I/O (raw and **O_DIRECT**) transfers data between a user buffer and a device without copying the data through the kernel's buffer cache. This mechanism can boost performance if the data is unlikely to be used again in the short term (during a disk backup, for example), or for applications such as large database management systems that perform their own caching.

Direct I/O (DIO) support was consolidated and redesigned in Linux 2.5. The old scalability problems caused by preallocating kiobufs and buffer heads were eliminated by virtue of the new BIO structure. Also, the 2.5 DIO code streams the entire I/O request (based on the underlying driver capability) rather than breaking the request into sector-sized chunks.

Any filesystem can make use of the DIO support in Linux 2.5 by defining a `direct_IO` method in the `address_space_operations` structure. The method must pass back to the DIO code a filesystem-specific get block function, but the DIO support takes care of everything else.

Asynchronous I/O support for DIO was added in Linux 2.5. The following caveats are worth noting:

- Waiting for I/O is done asynchronously but multiple points in the submission codepath can potentially cause the process to block (such as the pinning of user pages or processing in the filesystem get block routine).
- The DIO code calls `set_page_dirty` before performing I/O since the routine must be called in process context. Once the I/O completes, the DIO code—operating in interrupt context—checks whether the pages are still dirty. If so, nothing further is done; otherwise, the pages are made dirty again via a workqueue run in process context.

5 Vector AIO

5.1 2.5 readv/writev improvements

In Linux 2.5, direct I/O (raw and **O_DIRECT**) readv/writev was changed to submit all segments or iovecs of a request before waiting for I/O completion. Prior to this change, DIO readv/writev was processed in a loop by calling the filesystem read/write operations for each iovec in turn.

The change to submit the I/O for all iovecs before waiting was a critical performance fix for DIO. For example, tests performed on an aic-attached raw disk using 4Kx8 readv/writev showed the following improvement:

Random writev	8.7 times faster
Sequential writev	6.6 times faster
Random readv	sys time improves 5x
Sequential readv	sys time improves 5x
Random mixed I/O	5 times faster
Sequential mixed I/O	6.6 times faster

5.2 AIO readv/writev

With the DIO readv/writev changes integrated into Linux 2.5, we considered extending the functionality to AIO. One problem is that AIO readv/writev ops are not defined in the `file_operations` structure, nor are readv/writev part of the AIO API command set. Further, the interface to `io_submit` is already an array of `iocb` structures analogous to the vector of a readv/writev request, so a real question is whether AIO readv/writev support is even needed. To answer the question, we prototyped the following changes [12]:

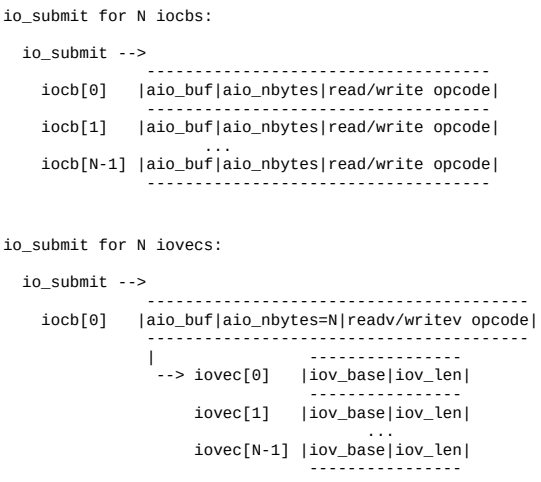
- added `aio_readv/writev` ops to the `file_operations` structure
- defined `aio_readv/writev` ops in the raw driver
- added 32- and 64-bit readv and writev command types to the AIO API
- added support for readv/writev command types to `fs/aio.c`:

```
fs/aio.c                |  156 ++++
include/linux/aio.h     |    1
include/linux/aio_abi.h |   14 ++++
include/linux/fs.h      |    2
4 files changed, 173 insertions(+)
```

5.3 Preliminary results

With the above-noted changes, we were able to test whether an `io_submit` for N iovecs is

more performant than an `io_submit` for N iocbs.



Based on preliminary data [1] using direct I/O, an `io_submit` for N iovecs outperforms an `io_submit` for N iocbs by as much as two-to-one. While there is a single `io_submit` in both cases, aio readv/writev shortens codepath (i.e., one instead of N calls to the underlying driver method) and normally results in fewer bios/callbacks.

5.4 Issues

The problem with the proposed support for AIO readv/writev is that it creates code redundancy in the custom and generic filesystem layers by adding two more methods to the `file_operations` structure. One solution is to first collapse the `read/write/readv/writev/aio_read/aio_write` methods into simply `aio_read` and `aio_write` and to convert those methods into vectored form [11].

6 Performance

6.1 System setup

All benchmarks for this paper were performed on an 8-way 700MHz Pentium™III machine with 4GB of main memory and a 2MB L2 cache. The disk subsystem used for the I/O tests consisted of 4 IBM® ServeRAID-4H™dual-channel SCSI controllers with 10 9GB disk drives per channel totalling 80 physical drives. The drives were configured in sets of 4 (2 drives from each channel) in a RAID-0 configuration to produce 20 36GB logical drives. The software on the system was SuSE Linux Enterprise Server 8.0. Where noted in the results, the kernel was changed to 2.5.68 plus required patches [7]. For AIO benchmarking, libaio-0.3.92 was installed on the system.

6.2 Microbenchmark

The benchmark program used to analyze AIO performance is a custom benchmark called rawiobench [9]. Rawiobench spawns multiple threads to perform I/O to raw or block devices. It can use a variety of APIs to perform I/O including `read/write`, `readv/writev`, `pread/pwrite` and `io_submit`. Support exists for both random and sequential I/O and the exact nature of the I/O request is dependent on the actual test being performed. Each thread runs until all threads have completed a minimum number of I/O operations at which time all of the threads are stopped and the total throughput for all threads is calculated. Statistics on CPU utilization are tracked during the run.

The rawiobench benchmark will be run a number of different ways to try to characterize the performance of AIO compared to synchronous I/O. The focus will be on the 2.5 kernel.

The first comparison is designed to measure the overhead of the AIO APIs versus using the normal read/write APIs (referred to as the "overhead" test). For this test rawiobench will be run using 160 threads each doing I/O to one of the 20 logical drives for both sequential and random cases. In the AIO case, an

`io_submit/io_get_events` pair is submitted in place of the normal read or write call. The baseline synchronous tests will be referred to in the charts simply as "seqread" "seqwrite" "ranread" "ranwrite" with an extension of either "ODIR" for a block device opened with the **O_DIRECT** flag or "RAW" for a raw device. For the AIO version of this test, "aio" is prepended to the test name (e.g., aioseqread).

The second comparison is an attempt to reduce the number of threads used by AIO and to take advantage of the ability to submit multiple I/Os in a single request. To accomplish this the number of threads for AIO was reduced from 8 per device to 8 total (down from 160). Each thread is now responsible for doing I/O to every device instead of just one. This is done by building an `io_submit` with 20 I/O requests (1 for each device). The process waits for all I/Os to complete before sending new I/Os. This AIO test variation is called "batch mode" and is referred to in the charts by adding a "b" to the front of the test name (e.g., bseqaioread).

The third comparison will improve upon the second by having each AIO process calling `io_getevents` with a minimum number equal to 1 so that as soon as any previously submitted I/O completes, a new I/O will be driven. This AIO test variation is called "minimum batch mode" and is referred to in the charts by adding a "minb" to the front to the test name (e.g., minbseqaioread).

In all sequential test variations, a global offset variable per device is used to make sure that each block is read only once. This offset variable is modified using the `lock xadd` assembly instruction to ensure correct SMP operation.

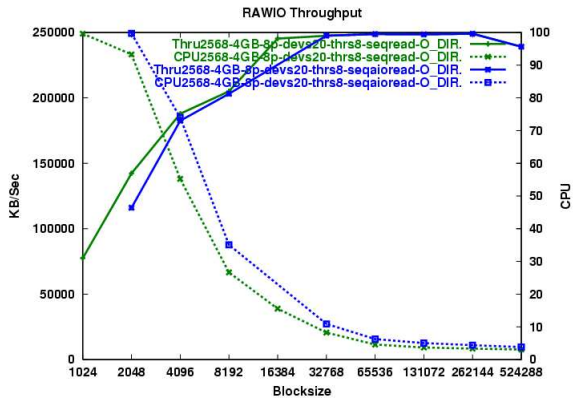


Figure 1: Sequential Read Overhead

6.3 Results, comparison and analysis

Raw and **O_DIRECT** performed nearly identically on all of the benchmarks tests. In order to reduce redundant data and analysis, only the data from **O_DIRECT** will be presented here.

For the first comparison of AIO overhead, the results show that there is significant overhead to the AIO model for sequential reads (Figure 1). For small block sizes where the benchmark is CPU bound, the AIO version has significantly lower throughput values. Once the block size reaches 8K and we are no longer CPU bound, AIO catches up in terms of throughput, but averages about 20% to 25% higher CPU utilization.

In Figure 2 we can see the performance of random reads using AIO is identical to synchronous I/O in terms of throughput, but averages approximately 20% higher CPU utilization. By comparing synchronous random read plus seeks with random pread calls (Figure 3) we see that there is minimal measurable overhead associated with having two system calls instead of one. From this we can infer that the overhead seen using AIO in this test is associated with the AIO internals, and not the cost of the additional API call. This overhead seems

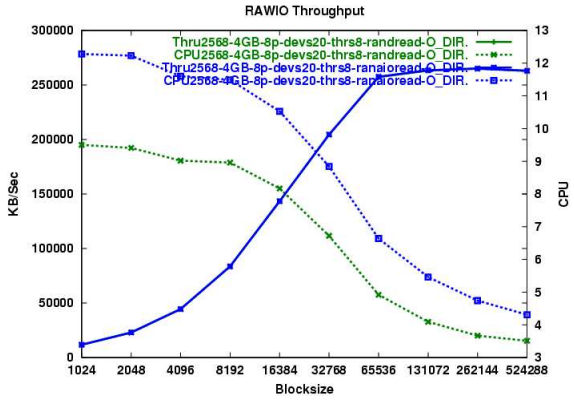


Figure 2: Random Read Overhead

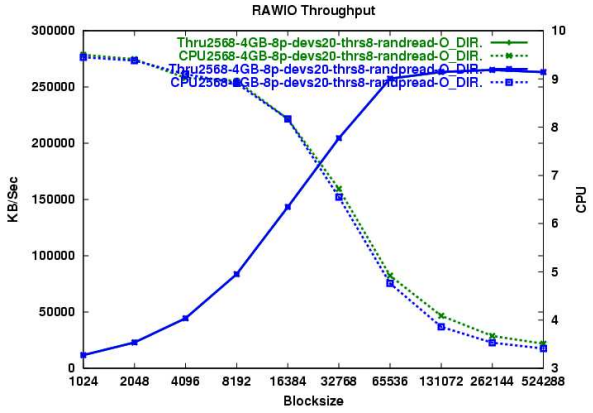


Figure 3: read vs. pread

excessive and probably indicates a problem in the AIO kernel code. More investigation is required to understand where this extra time is being spent in AIO.

For write performance we can see that AIO achieves approximately the same level of throughput as synchronous I/O, but at a significantly higher cost in terms of CPU utilization at smaller block sizes. For example, during the sequential write test at 2K block sizes, AIO uses 97% CPU while synchronous uses only 55%. This mirrors the behavior we see in the read tests and is another indication of problems within the AIO code.

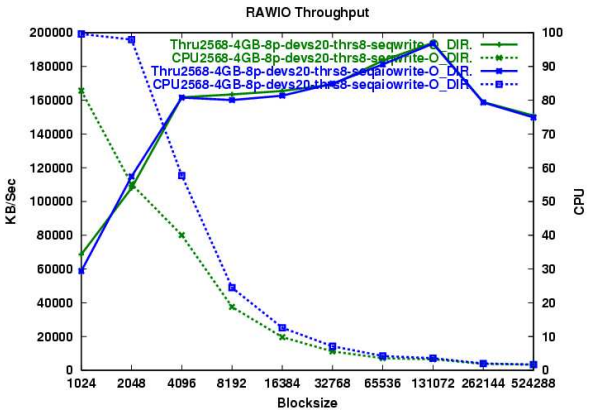


Figure 4: Sequential Write Overhead

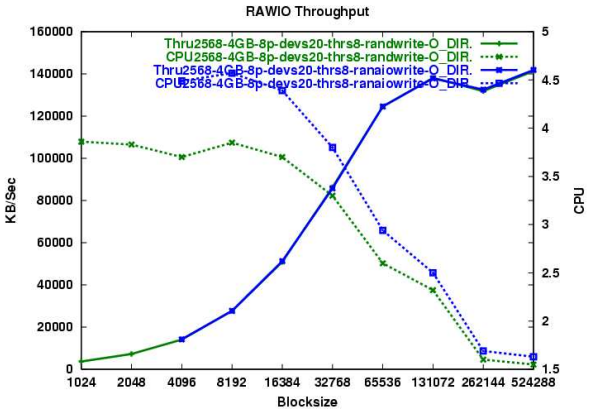


Figure 5: Random Write Overhead

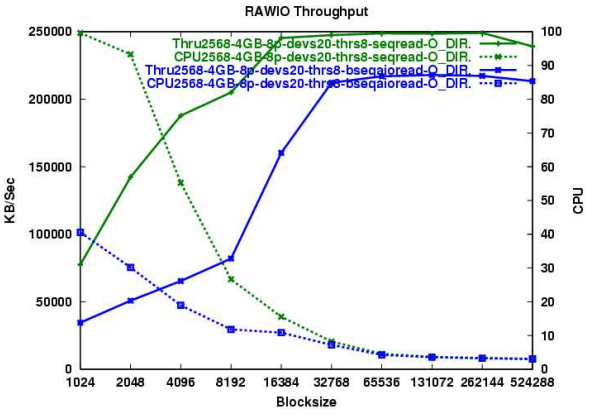


Figure 6: Sequential Read Batch

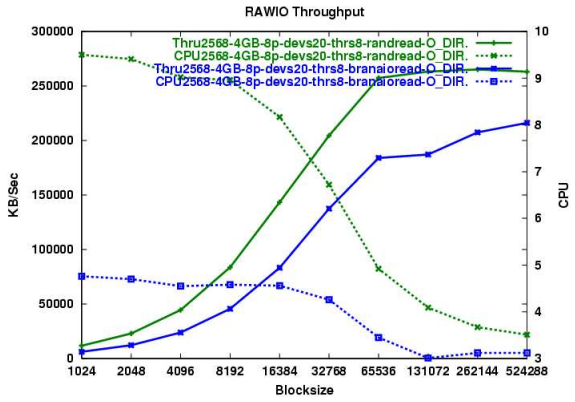


Figure 7: Random Read Batch

Batch mode AIO performs poorly on the sequential and random read tests. Throughput is significantly lower as is CPU utilization (Figures 6,7). This is probably due to the fact that we can drive more I/Os in a single request than the I/O subsystem can handle, but we must wait for all the I/Os to complete before continuing. This results in multiple drives being idle while waiting for the last I/O in a submission to complete.

AIO batch mode also under-performs on the write tests. While CPU utilization is lower, it never achieves the same throughput values as synchronous I/O. This can be seen in Figures 8 and 9.

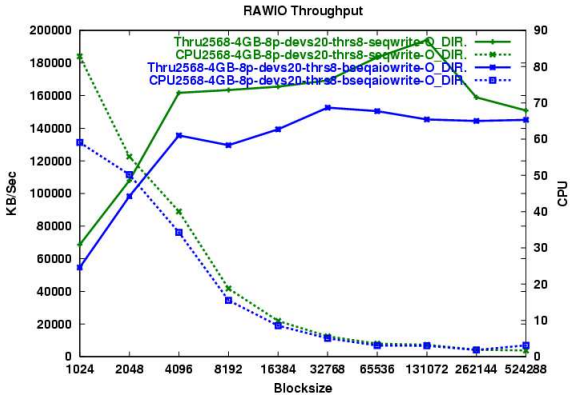


Figure 8: Sequential Write Batch

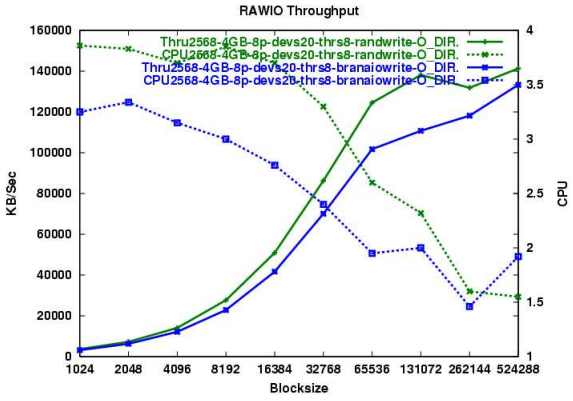


Figure 9: Random Write Batch

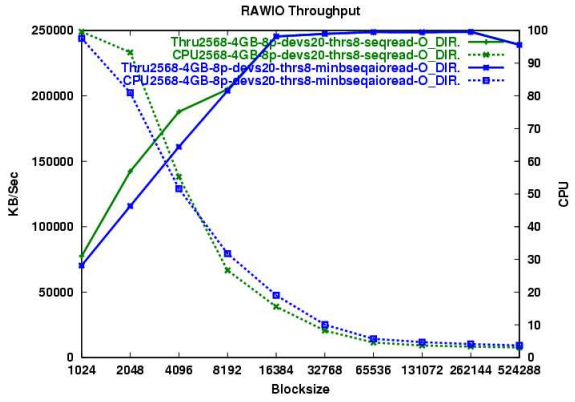


Figure 10: Sequential Read Minimum Batch

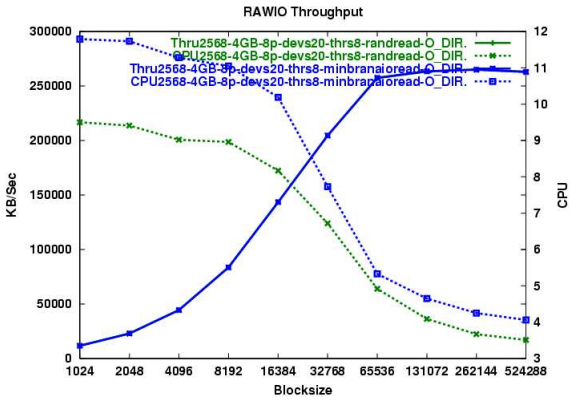


Figure 11: Random Read Minimum Batch

Minimum batch mode improves considerably on the overhead and batch mode tests; however, for **O_DIRECT** access AIO either lags in throughput or uses more CPU for all block sizes in the sequential read test (Figure 10). For the random read test minimum batch mode AIO has identical throughput to synchronous reads, but uses from 10% to 20% more CPU in all cases (Figure 11). Sequential minimum batch mode AIO comes close to the performance (both throughput and CPU utilization) of synchronous, but does not ever perform better.

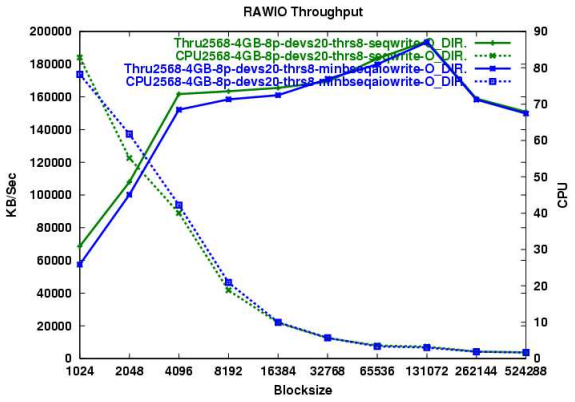


Figure 12: Sequential Write Minimum Batch

Minimum batch mode sequential writes, like reads, lag behind synchronous writes in terms of overall performance (Figure 12). This difference gets smaller and smaller as the block size increases. For random writes (Figure 9), the difference increases as the block size increases.

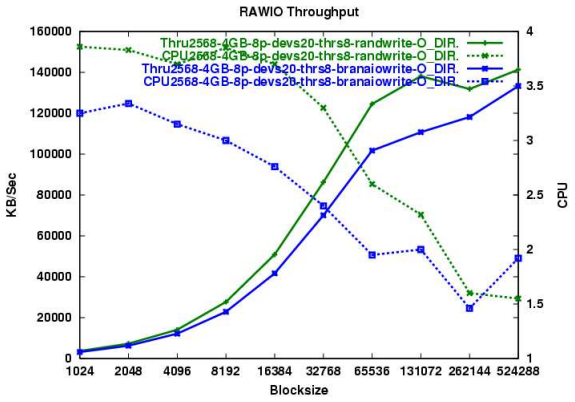


Figure 13: Random Write Minimum Batch

Since we are seeing lower CPU utilization for minimum batch mode AIO at smaller block sizes we tried increasing the number of threads in that mode to see if we could drive higher I/O throughput. The results seen in Figure 14 show that indeed for smaller block sizes that using 16 threads instead of 8 did increase the throughput, even beating synchronous I/O at the 8K block size. For block sizes larger than 8K the

increase in number of threads either made no difference, or causes a degradation in throughput.

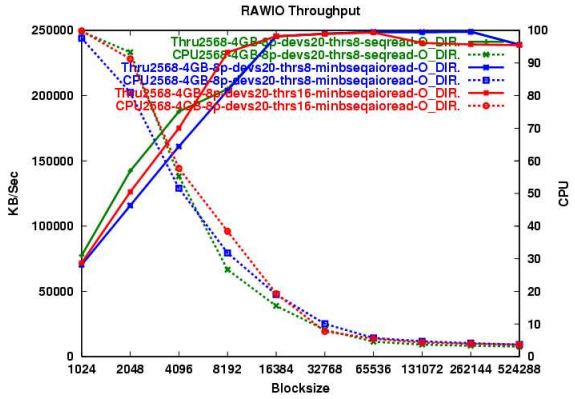


Figure 14: Sequential Read Minimum Batch with 16 threads

In conclusion, there appears to be no conditions for raw or **O_DIRECT** access under which AIO can show a noticeable benefit. There are however, cases where AIO will cause reductions in throughput and higher CPU utilization. Further investigation is required to determine if changes in the kernel code can be made to improve the performance of AIO to the level of synchronous I/O.

It should be noted that at the larger block sizes, CPU utilization is so low (less than 5%) for both synchronous I/O and AIO that the difference should not be an issue. Since using minimum batch mode achieves nearly the same throughput as synchronous I/O for these large block sizes, an application could choose to use AIO without any noticeable penalty. There may be cases where the semantics of the AIO calls make it easier for an application to coordinate I/O, thus improving the overall efficiency of the application.

6.3.1 Future work

The patches which are available to enable AIO for buffered filesystem access are not stable enough to collect performance data at present. Also, due to time constraints, no rawiobench testcases were developed to verify the effectiveness of the readv/writev enhancements for AIO [12]. Both items are left as follow-on work.

7 Acknowledgments

We would like to thank the many people on the `linux-aio@kvack.org` and `linux-kernel@vger.kernel.org` mailing lists who provided us with valuable comments and suggestions during the development of these patches. In particular, we would like to thank Benjamin LaHaise, author of the Linux kernel AIO subsystem. The retry based model for AIO, which we used in the filesystem AIO patches, was originally suggested by Ben.

This work was developed as part of the Linux Scalability Effort (LSE) on SourceForge (`sourceforge.net/projects/lse`). The patches developed by the authors and mentioned in this paper can be found in the "I/O Scalability" package at the LSE site.

8 Trademarks

References

[1] AIO readv/writev performance data. `http://osdn.dl.sourceforge.net/sourceforge/lse/vector-aio.data`.

[2] Asynchronous I/O on Windows® NT.

[3] Kernel Asynchronous I/O implementation for Linux from SGI. `http://oss.sgi.com/projects/kaio`.

[4] POSIX Asynchronous I/O.

[5] Open Group Base Specifications Issue 6 IEEE Std 1003.1. `http://www.opengroup.org/onlinepubs/007904975/toc.htm`, 2003.

[6] Suparna Bhattacharya. 2.5 Linux Kernel Asynchronous I/O patches. `http://sourceforge.net/projects/lse`.

[7] Suparna Bhattacharya. 2.5 Linux Kernel Asynchronous I/O rollup patch. `http://osdn.dl.sourceforge.net/sourceforge/lse/aiordwr-rollup.patch`.

[8] Suparna Bhattacharya. Design Notes on Asynchronous I/O for Linux. `http://lse.sourceforge.net/io/aionotes.txt`.

[9] Steven Pratt Bill Hartner. rawiobench microbenchmark. `http://www-124.ibm.com/developerworks/opensource/linuxperf/rawread/rawr%ead.html`.

[10] Benjamin LaHaise. 2.4 Linux Kernel Asynchronous I/O patches. `http://www.kernel.org/pub/linux/kernel/people/bcrl/aio/patches/`.

[11] Benjamin LaHaise. Collapsed read/write iocb argument-based filesystem interfaces. `http://marc.theaimsgroup.com/?l=linux-aio&m=104922878126300\&w=2`.

[12] Janet Morgan. 2.5 Linux Kernel Asynchronous I/O readv/writev patches. `http://marc.theaimsgroup.com/?l=linux-aio&m=103485397403768\&w=2`.

[13] Venkateshwaran Venkataramani Muthian Sivathanu and Remzi H. Arapaci-Dusseau. Block Asynchronous I/O: A Flexible Infrastructure for User Level Filesystems. `http://www.cs.wisc.edu/~muthian/baio-paper.pdf`.

Towards an O(1) VM:

Making Linux virtual memory management scale towards large amounts of physical memory

Rik van Riel
Red Hat, Inc.
riel@surriel.com

Abstract

Linux 2.4 and 2.5 already scale fairly well towards many CPUs, large numbers of files, large numbers of network connections and several "other kinds of big." However, the VM still has a few places with poor worst case (or even average case) behavior that needs to be improved in order to make Linux work well on machines with many gigabytes of RAM.

1 Introduction

In this paper I will explore the problem spaces and algorithmic complexities of the virtual memory subsystem. This paper will focus mostly on the page replacement code, which by definition has all of physical memory and parts of virtual memory as its search space. The following aspects of page replacement will be discussed:

- Page launder, the reclaiming of pages that are selected for pageout.
- Page aging, how to select which pages to evict.
- Balancing filesystem cache vs. anonymous memory.
- Reverse mapping, pte based vs. object based.

2 Page launder

Traditionally the virtual memory management subsystems in Unix and Linux systems have had either a clock algorithm or Mach-style active and inactive lists to do both LRU aging and eviction of pages. Linux 2.4 and 2.5 have what amounts to simple Mach-style active and inactive lists (Figure 1), at least when it comes to the writeout and reclaiming of pages that aren't mapped in processes. In this paper, the Mach VM pageout algorithm is used as an example because it is a decent approximation of what the different Linux VMs have done and the Mach VM is quite possibly the best documented virtual memory subsystem.

In the Mach VM, pages get recycled once they reach the end of the inactive list and are clean, meaning they do not need to be written to disk. If the page needs to be written to disk, a so-called dirty page, disk IO is started and the page is moved to the beginning of the inactive list. Presumably the disk IO will have finished and the page will be clean by the time it gets to the end of the inactive list again.

This organisation works reasonably well when dealing with filesystem cache pages, since

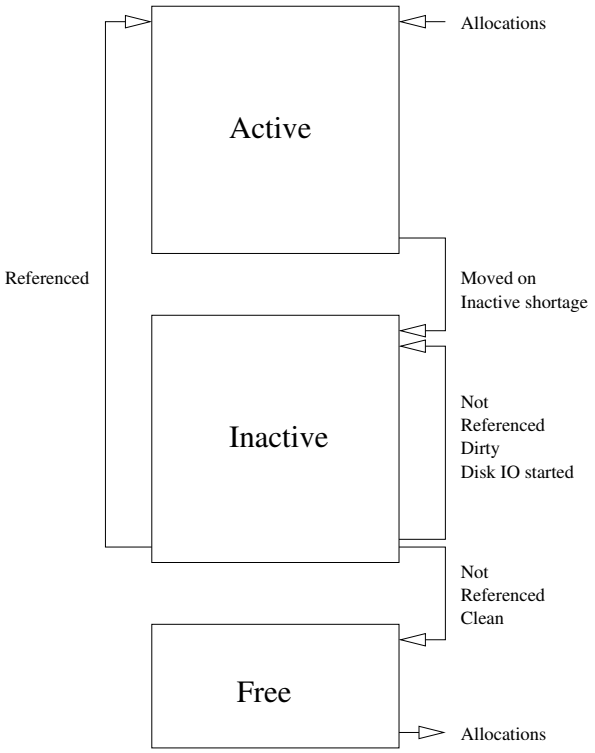


Figure 1: Mach pageout lists

those are usually clean pages which can be reclaimed the moment they reach the end of the inactive list. However, when the filesystem cache is small and the system is dealing mostly with dirty, swap or mmap backed pages from processes, this strategy has a big drawback on modern, large memory computers.

2.1 The problem with Mach-style page laundering

Memory used by processes is often dirty, meaning it needs to be written back to disk. The problem with this becomes obvious when we look at exactly what happens when all of the pages on the inactive list are dirty:

- The pageout code encounters a dirty page.
- Disk IO is started, the page is written to disk.
- The page is moved to the far end of the inactive list.
- The page reclaiming code encounters the next dirty page, starts writeout, etc...
- Since only a finite number of disk IO operations can be underway at any time, the page reclaiming code needs to wait for current IO operations to finish once it has started writeout on a certain number of pages.
- IO on the pages that were written out first finishes, meaning the pages are now clean and reclaimable.
- The page reclaiming code continues with the write out of the other dirty pages on the inactive list.

Of course, this has a number of serious drawbacks. The most obvious one is that on large memory systems the system will wait for most pageout IO to have finished before it can even start the last IO. Worse yet, it won't be able to free a page before all IO has been submitted.

In the early 1990s, when the Mach VM was popular, systems had up to a few megabytes of memory, with maybe a few hundred kilobytes of inactive pages, which could be written to disk in one or at most a few seconds. Modern systems, on the other hand, often have multiple gigabytes of memory. Since the speed of hard disks hasn't increased nearly as much as the size of memory, the time needed to write out all of the inactive list can be unacceptably high, up to dozens of seconds.

2.2 Solutions

One obvious solution is to only write out part of the pages on the inactive list. After all, if the system needs to free ten megabytes of memory, there is little reason to write out one gigabyte of

data. The implementation of this solution is a little less obvious, since there are various ways to approach this goal and there is a tradeoff to make between CPU usage and page freeing latency.

The first solution would be to simply write out a limited number of pages and skip the dirty pages on the list, scanning the list like usual and freeing all the clean pages encountered. In situations where the inactive list has both clean and dirty pages this tactic will allow you to always free the clean pages, reaching your free target and allowing allocations to go on with as little latency as possible. Of course, if the list only has dirty pages, then the system could end up spending a lot of CPU time scanning the list over and over again.

For the rmap VM a different, hopefully more predictable and CPU friendly solution (Figure 2) has been chosen. Instead of just one inactive list, there are various lists for the different stages of the pageout process a page can be in. Initially all rarely used pages are placed on the inactive_dirty list, regardless of whether or not they need to be written back to disk.

When a page reaches the end of the inactive_dirty list and wasn't referenced, the VM will move it to the inactive_laundry list, starting disk IO if the page was dirty. Referenced pages get moved back to the active list.

On the other end of the inactive_laundry list the VM removes clean pages, until the system has enough immediately freeable and free pages. Referenced pages are moved back to the active list; cleaned pages are moved on to the inactive_clean list, from where they can be immediately reused by the page allocation code.

The inactive_clean list is just an extension of the free page list. It contains clean pages that were not referenced and can be immediately reclaimed by the page allocation code. The reason for having an inactive_clean list is that the free page list in a VM is never the right size. The list should be as large as possible in order to be able to satisfy allocations with low latency, but at the same time the list should be as small as possible so almost all of memory can be used for processes and the cache. Having a list of immediately reclaimable pages with useful data in them avoids most of this dilemma.

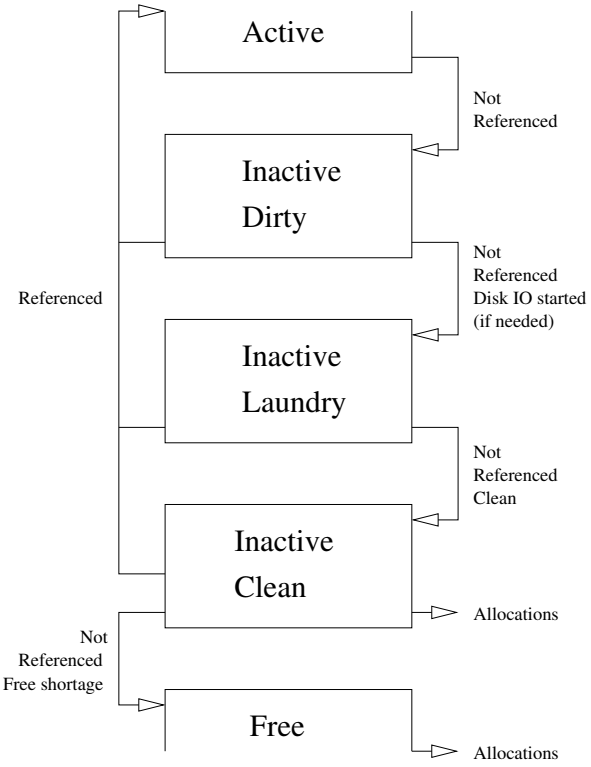


Figure 2: O(1) page launder

3 Page aging

Since the performance penalty of evicting the wrong page from memory is so high, due to the enormous speed differential between memory and disk, any virtual memory subsystem needs to take great care in selecting which pages to evict and which pages to keep in memory. On the other hand, on systems with more than a few megabytes of memory you do not want to

scan all the active pages every time the system is short on inactive memory.

While it is impossible to ensure this situation will never happen, because some applications just have access patterns you cannot tune a page replacement algorithm for, we can improve the situation a lot by pre-sorting the active pages in various lists (Figure 3), according to activity.

The pageout code will only look at the pages that most likely aren't very active, meaning it has a better chance of finding the proper pages for eviction without needing to resort to a full scan of memory. If the list with least used pages is empty, the pageout code simply shifts down all of the active lists and starts looking at the pages that came from the next list up.

The page aging (sorting) code scans the active lists periodically and moves the pages that were accessed to higher lists. It only needs to age pages upwards, because the downwards movement is done by the pageout code shifting down whole lists at a time. The period with which the page aging code scans the active lists is varied in reaction to the amount of pageout activity. Ideally the system would do a similar number of up aging scans as the number of times it shifts down active lists. The scan interval of the up aging code is reduced if the VM did too many down shifting of active pages and increased if the VM was quiet in-between two aging scans. The page aging interval has both a lower and an upper bound, to keep the overhead under control and to have some background aging in an otherwise idle system. The only time the page aging doesn't run is when there are more active pages on the higher lists than on the lower lists.

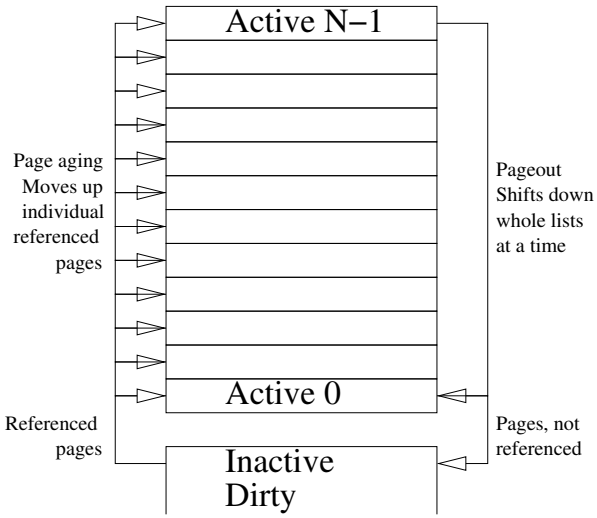


Figure 3: Multi list page aging

4 Balancing cache vs program memory

LRU style page replacement algorithms have well-documented, known problems. There are several replacement algorithms available that improve the replacement of pages within one set of data, e.g. EELRU, SEQ and LRFU; however none of these address the problem of balancing replacement between various sets of data. Since all currently implemented page replacement algorithms for Linux have this problem, the replacement algorithm needs some help balancing the file cache with memory used for programs.

The rmap VM borrows a common trick from other systems here. There are separate active lists for file cache memory and program memory, active_cache and active_anon, respectively. While the cache is larger than a certain percentage of active memory only the cache pages are a candidate for pageout, this value is the so-called borrow percentage and is 15 by default. Below the borrow percentage the VM will move both cache pages and pages belonging to processes to the inactive

list, reclaiming the pages that haven't been referenced again by the time they reach the far end of the inactive list. This gives cache and processes a chance to balance against each other by referencing pages. If the cache takes less than a predetermined minimum of the active list, one percent by default, the VM will only reclaim pages from processes.

The deeper reasons behind the need for these balancing hints are a little more complex than the reasons behind other design choices in the VM. One of the factors is that the amount of data on the filesystems tends to be several magnitudes larger than the amount of memory taken by the processes in the system. This means that the number of accesses to pages from the file cache could overwhelm the total number of accesses to the pages of the processes, even though the individual pages of the processes get accessed more frequently than most file cache pages. In other words, the system can end up evicting frequently accessed pages from memory in favor of a mass of recently but far less frequently accessed pages.

A replacement algorithm like LIRS, Low Interreference Recency Set, would probably do the right thing since it replaces pages with a higher interval between references before pages that have a lower interval between references. However, for LIRS to work properly the VM would need to keep track of pages that have already been evicted from memory. Since Linux does not have an infrastructure to keep track of those, the rmap VM uses an LRFU style page replacement algorithm with cache size hints.

Even if the direct value of LIRS over LRU/LFU for use as a primary cache wouldn't be big enough to offset the overhead of the needed infrastructure, the facts that LIRS would make the file cache vs process memory balancing automatic and that LIRS would also do the right thing as a second level cache (e.g. an NFS server, page cache on a web proxy where squid itself has the first level cache) make the implementation of LIRS for Linux a promising future experiment.

5 Reverse mapping

Reverse mappings provide an inverse to the page tables of the processes; that is, they keep track of which processes are using the physical pages, at which virtual addresses. Using reverse mappings, the pageout code can:

- Unmap a page from all processes using it, without needing to search the virtual memory of all processes.
- Unmap only those pages it really wants to evict, instead of scanning the virtual memory of all processes and unmapping more pages than it wants to evict in order to be on the safe side. This could reduce the number of minor page faults.
- Evict pages in a certain physical address range, which is useful since Linux divides physical memory into various zones.
- Scan only the virtual mappings of known inactive pages, which means the pageout code has a smaller search space in virtual memory. Combined with smarter page aging and page laundering, this results in a smaller overall search space for the pageout code.

5.1 Page based vs object based

There are pros and cons to doing reverse mapping on a per-page or a per-object basis. Reverse mapping on a per-page basis is more efficient for the pageout code, but the reverse

mapping code affects more than just the page-out code path. The page fault, fork, exit, and mmap paths all modify the reverse mappings, so doing reverse mappings on objects larger than a page (like a vma) would reduce the reverse mapping overhead in those code paths, at the cost of the pageout code needing to search more space.

The big question here is how much the overhead and algorithmic complexities would change, especially under larger workloads. A quadratic increase in complexity in the pageout path is almost certainly more expensive than what could be offset by a linear speedup in the other code paths, even though the pageout path is rarely run.

Large workloads, with many gigabytes of memory and hundreds or thousands of large, active processes are certainly able to bring out the worst of any VM; with the current implementations it doesn't even matter which style of reverse mapping is used. Bad behaviour can be triggered in either case.

It appears that for both object-based and page-based reverse mappings, Linux is in need of smarter data structures that aren't susceptible to quadratic algorithmic complexities anywhere. Once those are written we will be able to make a proper comparison between both methods of reverse mapping. It is conceivable that Linux would end up using a hybrid of object-based and page-based reverse mapping, with each type being used where it is most appropriate.

6 Conclusions

Linux memory management has come a long way in the last few years, but at the same time users have deployed Linux in more and more demanding environments. In fact, demand always seems to be one step ahead of whatever stage kernel development is at.

Users have shown beyond any doubt that there are legitimate workloads that bring out the worst case behaviour in any VM; because of this there is a constant need to bring the algorithmic complexity of any part of the virtual memory management subsystem closer to the holy grail of constant-time, or O(1) complexity. The author expects development of the Linux virtual management subsystem to remain challenging for years to come.

7 References

Draves, Richard P. *Page Replacement and Reference Bit Emulation in Mach.* In Proceedings of the USENIX Mach Symposium, Monterey, CA, November 1991.

Y. Smaragdakis, S. Kaplan, and P. Wilson, *EELRU: Simple and Effective Adaptive Page Replacement* in Proceeding of the 1999 ACM SIGMETRICS Conference, 1999.

Gideon Glass and Pei Cao. *Adaptive Page Replacement Based on Memory Reference Behavior.* In Proceedings of ACM SIGMETRICS 1997, June, 1997.

D. Lee, J. Choi, J.-H. Kim, S.H. Noh, S.L. Min, Y. Cho, and C.S. Kim, *LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies* IEEE Trans. Computers, vol. 50, no. 12, pp. 1352–1360, 2001.

S. Jiang and X. Zhuang. *LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance.* In Proc. of SIGMETRICS 2002.

Developing Mobile Devices based on Linux

Tim Riker
Texas Instruments
Tim@Rikers.org, http://Rikers.org/

Abstract

This presentation will cover available components of embedded solutions that leverage Linux. We discuss bootloaders, Linux kernel configuration, BusyBox, glibc, uClibc, GUI choices such as Qtopia, TinyX, GPE, Konqueror, Dillo, and similar packages that make up a commercial-grade consumer device. This presentation is aimed at those who are just getting into Linux on mobile or other embedded devices.

1 Why Linux?

Linux is a stable, tested platform for production use. The most often-used environment for Linux is as a server for key business services. Is it well suited for embedded use on mobile devices? It is. Linux has a low total cost of ownership as compared to other options. There is a large pool of developers that are familiar with the environment and this pool continues to grow rapidly. The use of an Open Source platform allows for flexible hardware and other product design choices that can save money. There is no dependency on a single vendor for support.

The largest advantage is quick time to market. The wealth of available projects that can be leveraged means that resources can be directed toward the specific value a product has to offer without spending undue resources duplicating what is done on devices that are otherwise similar. This advantage truly comes to light when a community is formed around the product. This community can focus on enhancing the product without direct development cost to the manufacturer. When planning the product lifetime, thought should be given to seeding hardware to key community members so that community support is available early in the product life cycle.

2 Special Needs of Mobile Devices

Mobile devices have features not often seen in desktop or server systems. Most use flash storage instead of traditional hard disk media. NOR flash is a common choice for average-sized storage, but becomes more expensive with larger systems. It is relatively easy to handle from a software perspective, and is commonly available as a direct memory mapped device. NAND flash is cheaper for larger (ie: >32MB) devices, but is not normally mapped to a specific memory location. In addition, flash devices track bad sectors and have error-correction code. Special device drivers and filesystems are required.

Removable storage is also an option. MMC (MultiMediaCard) is one common small-form-factor storage card. These use a 1-bit width serial interface to access the flash. SD (Secure Digital) storage cards can have an enhanced 4-bit interface that allows for faster data transfer and I/O devices, but the licensing from http://SDCard.org/ prohibits releasing

the source to any driver for these cards, so they are not currently recommended for Linux-based solutions.

Compact Flash (CF) storage cards are another popular option. Testing has shown that most, if not all, CF cards are unreliable when power-cycled during a write. Consider this strongly if power-cycles are likely to happen. Batteries often run down on mobile devices.

2.1 Power needs in hardware

Power consumption is critical in a mobile device. If the device has a display, the light is often the single largest power-consuming component. Software should be configured to be aggressive about turning off the lighting. This is commonly user-configurable, and often has a different setting when the device is connected to external power. Wireless interfaces are likely the next largest power consumer. It is wise to spend time during product development tuning the wireless setting for maximum power conservation. Choosing a different wireless chipset can make a large difference in the power needs of the device.

Linux has support for CPU scaling on a number of architectures. Research the devices that are affected when the CPU speeds up or down on different platforms. For example, on typical StrongARM platforms, the LCD display must be turned off during any CPU speed changes. This may mean your product cannot leverage CPU scaling in a useful manner. Other devices that may be affected by CPU speed changes include audio, USB, serial, network, and many other timing-sensitive devices.

CompactFlash and other removable devices may still consume significant power while in a suspended state. It is wise to add support for removing power in software to most system devices before entering a suspend state. It is also wise to avoid polling of any hardware device. As a general rule, interrupt-driven devices will have a lower load on the CPU and therefore consume less power. Physical keys on the device are one common area where this is overlooked. If the device has a power button, it should be on a separate hardware interrupt from the other keys on the device.

3 The bootloader

Choosing a bootloader has a big impact on the development environment. Most embedded systems do not have a traditional BIOS onboard. They do not have APM or ACPI interfaces. Some rely on the bootloader to take part in power-resume states; others just resume execution at the next address after where they entered suspend. Most bootloaders will initialize RAM configuration, and startup other devices in the system. Hardware designs may want to insure that the Linux kernel can be booted with a minimum of hardware setup so that there does not need to be driver code for the remaining hardware in the bootloader as well as in the kernel.

The most common interface to a bootloader is over a serial port. If the device has a keyboard and display, that may be another choice, but there is usually serial port support as well. The bootloader should support flashing new code on the device. New kernel images and filesystem images are loaded over this interface and stored to flash on the device. This requires that the bootloader is able to deal with whatever styles of flash are on the device. Xmodem, Ymodem, and even Zmodem code is available in existing open bootloaders.

If the device will have removable media such as CompactFlash storage cards, the bootloader can be configured to read images from there for flashing. This may be a good option for

upgrading devices in the field. Installing from MMC or other types of storage devices may require significant work implementing device support in the bootloader.

Some embedded systems have built-in ethernet interfaces and can use that for bootp or tftp updates. This is not very common for a mobile device.

For products with a USB client connection, USB serial support can be implemented in the bootloader. This will likely use a different USB device ID than presented by the product in its normal running state. If the product also has USB host connectivity, then it can be imaged off another working device. The kernel and base filesystem should be in a separate area for the device configuration in order to support this.

The bootloader will need to be aware of any partitioning in flash on the device. The kernel will need access to this same information. This information could then be shared with the kernel by dedicating an area of flash to store it. It could be compiled into both the bootloader and the kernel. The cleanest approach is for the bootloader to add it to the kernel command line when booting. There will likely be other kernel command-line options the bootloader will want to store and pass on to the kernel. All of these can be contained in one flash block if desired. Some bootloaders understand the filesystem, as well as the partitions on the device. This allows for one filesystem image that contains the base binaries and libraries as well as the kernel and its modules. This prevents the kernel and modules from ever being out of sync if they are all in the same image, which is a nice feature from a customer support perspective. If the bootloader can read files from the filesystem directly, then one can store the permanent kernel command-line options and other bootloader configuration inside the filesystem. This will likely not include flash partitioning information, as the partitioning would need to be known before the filesystem could be read.

Erasing the bootloader is a Bad Thing. Steps should be taken to insure that it it protected. This might mean putting it in ROM, or protecting the flash block(s) it lives in, or other methods. OS updates in the field should not replace the bootloader as part of the normal procedure.

4 Filesystems

Workstation and server filesystems like ext2 do not scale well for embedded devices. Smaller filesystems include `romfs` and `cramfs`, which are read-only. `cramfs` includes compression. Flash or ROM often is slow to access. The time it takes to read and decompress files is often faster than reading the same file if stored uncompressed. `cramfs` may be both the smallest and fastest choice for a read-only filesystem. In most cases it is also important to conserve memory. This implies that filesystems should be used directly from flash rather than using an initial ramdisk (initrd) whenever possible.

Some applications on the device will create temporary files. If the filesystem is read-only, ramfs support should be added to store these files. One way to handle this is to link /tmp to /var/tmp then mount ramfs on /var and unpack a tar archive into it, or just copy a tree from someplace else on the filesystem. You may want /dev linked in here as well if device files need to have permission or ownership changed at runtime.

An alternative to /dev is to include devfs support in the kernel. You may be able to avoid using devfsd by considering the needs of all the applications that will be included on the device and configuring default devices permissions in the kernel.

Many handheld portable devices use jffs2. This is a compressed journaled filesystem that understands NOR flash directly. It will need a few free blocks to use as workspace in each mounted partition. jffs2 has undergone a great deal of testing to insure that it is always in a readable state even when writes are interrupted by a powerfail or hard reboot. jffs2 support on NAND flash chips is in progress and may be complete by the time you read this. There are other flash filesystems like YAFFS designed just for NAND flash devices. This is a reasonable option with a lot of flash and where compression at the filesystem level is not needed. Removable storage media will still need to use vfat if it is going to be moved to other devices.

Some systems have dedicated partitions in flash for diagnostics or additional code that handles reflashing the device in the case where the normal filesystem is not usable. The impact of requiring this extra flash space should be considered carefully. Reflashing code can be added to the bootloader in much less space. User-space tools can be used for this task as long as the device can get to that point using the existing filesystem. If the device includes removable media access, diagnostics can be shipped on the removable media and the bootloader configured to load a kernel and filesystem from there.

5 Kernel device drivers

Embedded devices often do not include components like PCI, ISA, MCA busses. All devices that will not be present should be turned off in the kernel config to save space. These are not always obvious choices. For example, if the device includes a CompactFlash slot, all of the PCMCIA card drivers would be available as card options. There is no clear list of which drivers are exclusively for pcmcia cards and do not need to be available as they would not be inserted in a CompactFlash slot.

5.1 Connectivity

Many mobile Linux devices include some form of networking. This is probably not a normal ethernet interface but can include WiFi, Bluetooth, IrDA, USB, ppp over serial, cell phone, etc. Some devices will have multiple options available. Very good IPv6 support is available under Linux, but be aware that the IPv6 stack is larger than IPv4 or other options.

6 Libraries

Larger Linux systems are commonly based on the GNU C Library. This is not well suited to small devices. To quote the maintainer:

> . . . glibc is not the right thing for [an embedded OS]. It is designed as a native library (as opposed to embedded). Many functions (e.g., printf) contain functionality which is not wanted in embedded systems.
>
> —Ulrich Drepper
> <drepper@cygnus.com>
> 24 May 1999

Other alternatives will be smaller, but may require some modifications to source that is being ported. This should not be a large task when compared with the other aspects of the port. Alternatives include uClibc, dietlibc, newlib, and using pieces from things such as the Minix C library, libc5, *BSD libraries, etc. uClibc is the best choice today, as it is under the LGPL license to allow for commercial applications, includes shared library support, pthreads, c++, and even most locale features. It is configurable so you can remove things

you don't need. uClibc is tested for compliance with POSIX and ISO standards and passes more tests than glibc.

Other libraries included on the device should be reviewed closely to insure that they do not include components that are not needed. Many of these have compile-time options to exclude large portions of the functionality.

If no applications are going to be added to the system at a later date, library reduction tools can be used to remove the unneeded portions. Some examples of these tools include mklibs, lipo, and libraryopt. The python script `mklibs.py` is used in the Debian project on multiple architectures and is the best place to start.

7 Applications

There are many basic tools that run on top of Linux and make up a basic distribution. Many of the GNU tools could be here. GNU fileutils, textutils, grep, modutils, and many others are found in most every Linux distribution, hence the GNU/Linux naming convention. While this is the norm for a desktop distribution, it is likely that none of the binaries or libraries on an embedded Linux system will be the GNU flavor. Applications like BusyBox and TinyLogin are not GNU packages, nor is uClibc.

BusyBox includes close to 200 different applications in one multicall binary that is under 600k. This allows you to hardlink or symlink to the single binary, and depending on how it is called, it acts as each of those different applications. Each applet is normally less feature-rich than the GNU equivalent, but much smaller in size. If you were to include the normal binaries for all of these from a GNU/Linux distribution, you could have over 5 MB of binaries. Each applet in BusyBox can be enabled or disabled so it's likely that your version will be much smaller when you only include the programs you need.

Your system will likely want to have some services and perhaps a way to get shell access remotely. A small inetd (24k) with a minimal telnetd (32k) could do the trick. The latest version of BusyBox includes both of these. Expansion capabilities might warrant including pcmcia-cs and hotplug code. Use caution in this area as well. Linux supports a lot of PCMCIA hardware but the size of included drivers can add up quickly.

8 Crypto

For secure access OpenSSH is the common choice on Linux desktops and servers. OpenSSH client (215k) and server (254k) normally uses OpenSSL (181k). When OpenSSL is compiled for a smaller set of supported hashes and algorithms, it can be smaller, but even then it is large as compared to most embedded applications. The author is still searching for a small ssh/sshd solution.

FreeSWAN is another secure access method. It is normally even larger, close to 2M in size. FreeSWAN supports Opportunistic Encryption which can be very useful in an enterprise environment. Normal IPSec support is also valuable for many solutions. Adding hardware crypto support as provided by the Texas Instruments OMAP161x chips and converting all crypto systems in the kernel and userland over to use this can save memory as well as boost performance.

9 Package Management

For devices that are expandable, some form of package management is needed. Many desktop systems use RPM or dpkg for this purpose. Both of these are large binaries and more im-

portantly they normally have a large amount of package information stored on the filesystem. BusyBox includes a stripped-down version of both rpm and dpkg which might be a good starting place. The Handhelds.org project has a package manager called ipkg that is small and better suited for embedded systems. The Debian installer uses .udebs for about the same purpose. ipkg packages and udeb's do not normally include man pages or other supporting files, but have just the minimum files included. These packages are both the same format as a .deb file.

Note that Linux Standards Base requires the ability to install rpm files. It also requires many things like libgcc_s and glibc that the embedded device may not provide.

10 Graphical User Interface

There are many choices of GUIs for a mobile device. The Sharp Zaurus models use Qtopia with Qt/Embedded. These are available under the GPL, but using these versions requires that all applications available for the platform are also under the GPL. These components are also available from Trolltech under a different license as royalty-bearing components. This allows applications to use licenses other than the GPL. These components might be the only ones on the device that are not free. The base files needed for a common Qtopia and Qt/E configuration would be about 3 MB in size. This does not include the applications.

The Handhelds.org distribution can use X11 and GTK as the basis for an environment called GPE (GPE Palmtop Environment). This uses TinyX from the XFree86 version of the X Window System. The X server itself will be around 700k and the X11 and GTK libraries add up to about 3 MB. Note that this system retains all the remote display options of a normal X Window System desktop machine.

Smaller systems are available such as Pixel or NanoGUI. These might not include advanced web browsers or other large applications, but they are appropriate for smaller devices. A custom interface could be built on top of systems like DirectFB, GTKfb, Qt/E, SDL, or by writing directly to the Linux framebuffer.

The Qtopia interface has gotten more attention lately and is currently ahead of the GPE work. The OPIE project has enhanced Qtopia and is starting work on a next-generation library that would replace the Qtopia library and clean up the interface in the process. This new project should be available under LGPL, and if it was configured to run on top of TinyX using Qt/X11 instead of Qt/E, could avoid a per-unit royalty. Removing duplicate code between X and Qt/X11 could drop the storage size down very close to that of Qt/E and Qtopia. This retains the remote display options that the X Window System provides, while leaving a path to use much of the available Qt widgets and applications.

11 Web Browser

As time progresses, an embedded web browser will become more and more important. Browsers like Netscape and Mozilla are large. Other browsers might use the same engine but be much more space-constrained. Browsers like Dillo and ViewML are less feature-rich, but even smaller. Text-based browsers like lynx or links could be wrapped with a graphical front end, but this would be time-consuming and not in line with the goals of those projects. There are commercial options like Opera to consider as well. Konqueror Embedded, which uses Qt, is a good mix of size and features. Some minor interface tweaks could make it even better.

12 A Small Example

The TuxScreen project is very tightly storage-constrained. There is no web browser in the Linux-based base filesystem image. This device is a Desktop phone with a StrongARM 1100, a 640x480x8 color touch screen, and only 4 MB of flash storage. In this space we have a 128k bootloader partition (with only 32k used) and a 4MB minus 128k jffs2 root partition. In the root partition is the kernel and modules, uClibc, BusyBox, TinyLogin, pcmcia-cs, lrzsz, inetd, telnetd, XFree86 TinyX, rxvt, matchbox (a window manager), and more, with over 500k left in writable space. For devices that are only remote displays, this is a reasonable target size.

13 Where to Go from Here?

There are many Linux consulting companies that can help with a new product release. Monta Vista and Metroworks are two of the larger players in that space. Red Hat also has a group focused on embedded work. Some of these solutions might include other royalty-bearing components or have the option to spread out the engineering cost over the product lifecycle. There are plenty of individual consultants active in embedded systems work based on Linux. Some are looking for permanent placement and some work on a consulting basis. You should determine the finances of your project and pick a solution that matches with that plan.

The big benefits of going with Linux as the solution is this large available pool of resources to draw from and the ability to have all the source so you retain the option of doing the work internally or through another vendor. Do not ignore the benefit of growing a community around your product. The work that active, qualified developers in the community do to improve your software solution is a valuable benefit. Much of it will be available to you for merely the time it costs to monitor those activities. You can effectively kill off this effort by making it difficult for others to get involved. This will likely have a detrimental effect on your product sales.

When working out the details of contract work, you will likely want to add a requirement that all work be pushed upstream. This implies that it will be peer-reviewed by other Linux developers and may need to be fixed to comply with existing Linux kernel code. This is a very important part of new work done in the kernel. Without it, you and your product will be stuck with an old version of the kernel which the community and other vendors will not be eager to support. When the changes are pushed upstream, they are much more likely to get fixed as the kernel develops. A few free units of fun hardware in the right hands can go a long ways as far as overall software development expenses go.

Conclusion

We have touched on many projects that can be leveraged to offer rapid time to market when developing mobile and other embedded devices based on Linux. I hope this has been useful. There are many useful web sites that offer more information. The author plans to add links to all of the projects mentioned here and others as well in the wiki found on `http://eLinux.org/wiki/` if you would like more details. Good luck with your projects, and welcome to the community.

Lustre: Building a File System for 1,000-node Clusters

Philip Schwan
Cluster File Systems, Inc.
phil@clusterfs.com, http://www.clusterfs.com/

Abstract

Lustre is a GPLed cluster file system for Linux that is currently being tested on three of the world's largest Linux supercomputers, each with more than 1,000 nodes. In the past 18 months we've tried many tactics to scale to these limits, and the first half of this paper will discuss some of our successes and failures. The second half will explore some of the changes that we plan to make over the next year, as we scale towards tens of thousands of clients and petabytes of data.

1 Introduction

The Lustre cluster file system has been designed and implemented with the goal of removing the bottlenecks traditionally found in such systems. Lustre runs on commodity hardware and provides a cluster storage layout that is efficient, scalable, and redundant. Metadata Servers (MDSs) contain the file system's directory layout, permissions, and extended attributes for each object. Object Storage Targets (OSTs) are responsible for the storage and transfer of actual file data, and already scale to many dozens of OSTs and hundreds of terabytes of data. Both types of service node can operate in pairs which automatically take over for each other in the event of failure. Each also runs an instance of the Lustre distributed lock manager, access to which forms the core of the Lustre protocols.

Although Lustre's design dates from 1999, development began in earnest in early 2002. In the time since, surprisingly few of the major points have changed from the original plan, and the implementation has undergone fewer false starts as a result. Our distributed lock manager has weathered the storm and remains largely as it was a year ago. The choice of a system designed around object protocols has proven to be correct, and Lustre has so far scaled to the limits of available hardware. Lustre's internal networking has shown itself to be relatively flexible and high-performance, and network abstraction layers exist for TCP/IP, Quadrics Elan, Myrinet, and SCI.

Not all of our original decisions were ideal, however. One source of bugs continues to be the interaction between Lustre and the Linux VFS layer, which is not very well suited to network file systems that want a great deal of control. This interaction had a significant impact on one of Lustre's major metadata architecture choices, the concept of "intent-based" locking operations, described in more detail later. Ultimately, we had to make significant changes to our intent-based metadata implementation.

The last year of working with government and industry has suggested which activities are most important to pursue in the next year. First, and most significantly, two major caching im-

provements will be made beginning this summer: a write-back metadata cache, and a persistent data/metadata cache. The write-back cache will be enabled in times of low concurrency, and allows for metadata updates which can be made in local memory and later replayed on the server. Making this cache persistent for both metadata and file data will enable features such as disconnected operation and server replication. Finally, our collaborative read cache will reduce the load on primary OSTs for the most frequently accessed files, removing a very common bottleneck in distributed systems.

2 Distributed Lock Manager

All of Lustre's consistency guarantees are enforced, in one way or another, by the Lustre distributed lock manager (DLM). Core operational decisions, such as when to switch between writeback caching and synchronous metadata updates, will be delegated to the DLM.

The design of the Lustre DLM borrows heavily from the VAX Clusters DLM, plus extensions that are not found in others. Although we have received some reasonable criticism for not using an existing package (such as IBM's DLM[1]), experience thus far has seemed to indicate that we've made the correct choice: it's smaller, simpler and, at least for our needs, more extensible.

The Lustre DLM, at just over 4,000 lines of code, has proven to be an overseeable maintenance task, despite its somewhat daunting complexity. The IBM DLM, by comparison, is nearly the size of all of Lustre combined. This is not necessarily a criticism of the IBM DLM, however; to its credit, it is a complete DLM which implements many features which we do not require in Lustre.

In particular, Lustre's DLM is not really *distributed*, at least not when compared to other such systems. Locks in the Lustre DLM are always managed by the service node, and do not change masters as other systems allow. Omitting features of this type has allowed us to rapidly develop and stabilize the core functionality required by the file system.

Next, we feel that our extensions to the basic DLM API and protocol have been quite successful. File range locking is managed internally as part of the regular lock matching and compatiblity functions. Through the use of a small policy function, the lock manager is able to grant larger locks than originally requested. In this way we avoid the bottleneck found in some other file systems, for which a client must lock each page or block individually. For the very common case of a file being accessed by only one user, Lustre's DLM will grant exactly one lock for the entire file.

Finally, intent locking is designed around the concept of allowing the lock manager to choose between different modes depending on its view of resource contention. In a directory with very little contention—a user's home directory, for example—the DLM can grant a *write-back* lock, allowing the client to cache a large number of metadata updates in memory. In this way it will avoid an interaction with the server for each request and batch them at some later time. In a directory with very high concurrency—such as `/tmp`—the DLM will refuse to grant any lock at all. Instead, it will perform the operation on the client's behalf, notify it of the result, and avoid bouncing the directory lock between hundreds or thousands of simultaneous users.

3 Object Protocols

Of all of the concepts that went into Lustre's architecture, the use of *object protocols* is by far the most pervasive. Although Lustre is certainly not unique in its use of storage objects, we have also designed many of the internal APIs to allow for additional layering (RAID 0 as one example) or short-circuiting (a client running on an OST with no networking layer between them). This symmetry between internal APIs and network protocols has served us well.

During the initial design it became quite clear that a shared block file system would absolutely not scale to the required limits for many reasons. First, shared disk arrays on anything but the smallest clusters quickly become cost ineffective for even the largest customers; this would certainly violate our goal of running on inexpensive commidity hardware. Second, high-level object protocols remove a key bottleneck for scaling beyond a dozen or two nodes: locking and allocation of metadata.

In a traditional shared-block file system, those blocks which store inode and block allocation information are subject to incredible contention. By organizing the protocol around objects instead of blocks, the OSTs remain responsible for the internal metadata allocation. Parallel file I/O to a single file has been shown to scale to more than 1,100 nodes, the limit of available hardware.

For those customers who have already invested in a large storage area network (SAN) based around shared disk, Lustre is still an option. In the SAN mode, OSTs are still responsible for managing the object locking and shared storage metadata, but clients can read and write individual data pages directly from the SAN.

4 Networking

Lustre's networking layer has not changed significantly from its original form more than a year ago. It uses a simple message-passing package called Portals[2], which has from an API standpoint served us fairly well. Importantly, it provides the right abstractions for enhancements such as remote DMA as supported by the networking hardware. We've made a few relatively minor API changes to accomodate the different needs of the filesystem, as opposed to the scientific community from which Portals emerged.

The original implementation of Portals, however, caused many serious problems. The port to run in the Linux kernel and userspace was fairly straightforward, but Portals had never been run in a multi-threaded environment and had absolutely no internal locking. Given that we had to rewrite more than 80% of the code and put up with serious race conditions for many months, it would likely have been a better choice to keep the API and start the implementation from scratch.

The API and the internal abstraction layers, however, have been both simple enough to understand and modify, and flexible enough to cope with the needs of many networking drivers. Lustre (and therefore Portals) needs to support a variety of interconnects, including kernel TCP/IP, TCP/IP offload cards, Quadrics Elan 3, Myrinet, and SCI.

For each network type we have a Portals Network Abstraction Layer (NAL), approximately one to two thousand lines of code each. Although they are small, they are generally quite complicated, and may depend on a fair bit of wizardry to get the most out of a particular interconnect. Nevertheless, the Lustre network regression test running on our Elan NAL between two nodes is bottlenecked by the PCI bus

at more than 300 MB/s.

5 Intent Operations Explained

Most distributed file systems perform metadata operations in the same way all the time, regardless of contention. Some systems choose to give locks on objects to clients, and some choose to perform all operations synchronously on the server.

In the first mode, a client wanting to perform metadata operations will first take a lock on the parent directory, download the applicable directory information, and make many changes locally. This is extremely efficient in times of low contention; it can perform as many operations as it wants locally without contacting the server, as long as no other nodes try to acquire the lock. In times of high contention, however, it is a disaster: imagine users on 1,000 nodes all running `touch /tmp/foo`. The lock on `/tmp` will have to be given to each node in turn, and the cluster will grind to a halt.

In the second mode, the client sends a message to the server for each operation, and the server performs the operation without giving any locks out. Not only is this much simpler to code properly, it also avoids the problem with lock ping-pong. When this mode is used, however, even directories with no contention have this behaviour, and you suffer the effects of a server round-trip for each operation.

Lustre currently executes all operations as if there were high concurrency, with exactly one RPC per metadata operation. With the completion of the writeback metadata cache later this year, the DLM will be able to make the choice between giving the client a writeback lock on a subtree or performing one RPC per op.

6 Intents Gone Wrong

Our first attempt at writing the client-side VFS code to support the intent mechanism was roughly as follows. Consider the case of a mkdir operation: normal filesystems will lock the parent directory, lookup the new directory to see if it already exists, create it, then release the lock. Lustre added a *lookup intent* structure to each lookup call, to tell the lock manager on the server why we asked for the lock (in this case, to mkdir). If the server decided not to give out the lock, it would perform the operation on the client's behalf and return a success or error code.

When the Lustre client received this reply, it would do complicated things to cooperate with the VFS. If the mkdir succeeded, for example, it needed to create a *negative* directory entry (dentry) before returning from lookup (if we returned a new positive dentry, the VFS would return `-EEXISTS`). Later, the VFS would call us back to do the "actual" mkdir, at which time we would instantiate the dentry based on the reply stored in the lookup intent.

This turned out to be a disaster of race conditions, both on the server and on the client. On the server, the lock manager would perform these operations before the lock was granted, so that it could give the client a lock on the new file. By the time the lock was actually granted, however, anything could have changed. On the client, our ability to control the dcache, particularly in the window between lookup and final creation, proved insufficient.

Our final solution was to make two fairly major changes to both sides. Instead of the lock manager performing an operation before locks are granted, the metadata server is able to specify an already-granted lock to give to the client. This allows the MDS to perform the operation and then return the still-granted lock on

the new file without races. The client has been simplified to call directly into the filesystem and return the result immediately: no dcache, no VFS code, no races.

7 Client Metadata Caching

When a node asks to lock a directory for reading or writing, the Lustre DLM will soon be able to grant a *subtree lock*, if the directory has not recently seen conflicting activity. This allows the client to keep a cache which can be filled and selectively revalidated as necessary.

As a client with a subtree lock fills its cache from the MDS, the MDS may revoke locks on other objects. If during this process the MDS encounters an opened file or a file with hard links, it flags this file for special attention by the client. Specifically, the client also flags these as shared objects which cannot be cached locally and must use the intent path for updates.

Once a client has a subtree lock, it can begin to keep a local journal of updates. Each update is a short record which describes one logical filesystem operation on an object, for example "create directory, mode 0755, parent inode 12, new directory name foo". Because all update operations are now reduced to the creation of a single record in client memory, they are incredibly fast.

When another client attempts to perform a conflicting operation beneath a subtree lock, that lock must be found and revoked. The MDS server code can easily walk the dentry tree, looking at each path component of the affected object, and revoke subtree locks as necessary. It is now easy to see why we flag hard linked files for special handling, as they have more than one path by which they can be reached.

When a subtree lock is revoked, any accumulated updates must be flushed to the MDS and replayed on stable storage. This is exactly the same mechanism which already exists for intent locks, except that they are grouped into pages of operations which are all guaranteed to succeed by virtue of the subtree lock. Now a single network exchange can contain hundreds of records.

8 Persistent Caching

To this design there is one particularly attractive extension, which is a persistent cache in the style of AFS[3], Coda[4], and InterMezzo[5]. Two new pieces are required for such an extension: the local cache itself, and a way to revalidate its contents after locks are lost and re-acquired.

Lustre's stackable object protocols allow a very symmetric design for the persistent cache by adding a metadata server to the client. Today the file system code interacts with a metadata client (MDC) via function calls, which executes network commands to an MDS. In this new model, the MDC can be replaced with a *caching MDC* which can talk to both a local and remote MDS. The local MDS is responsible for maintaining the local cache, either in memory or on disk; the caching MDC resolves cache misses and replays updates to the real MDS as before.

At some point, after a client has lost and reacquired a lock, we need a way to validate the data that already exists in the cache. Unix filesystems already provide the concept of a *change time* (ctime), which is updated whenever the inode changes. For Lustre directories we will add a new *subtree change time* (stctime) which will be updated whenever any inode in the subtree is changed. These stctimes have a nanosecond granularity and will allow a client to very quickly establish whether large portions of a cache are up to date.

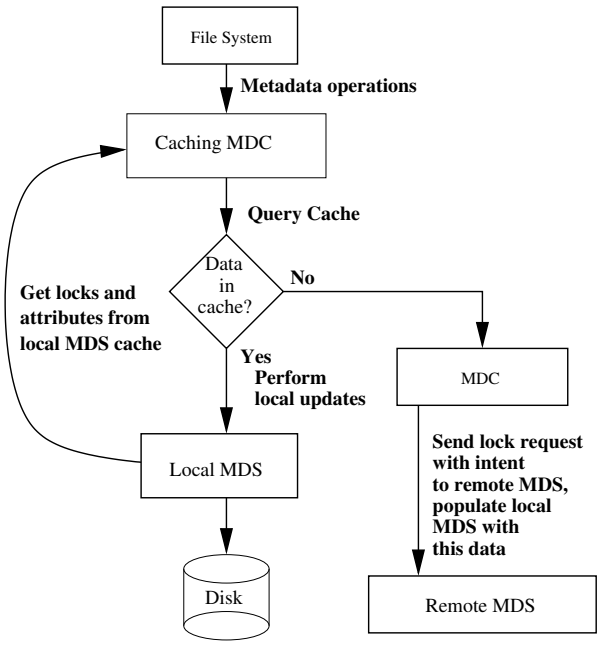


Figure 1: Persistent Caching

Updates to the sctime will of course dirty more (possibly many more) inodes during each update. For customers pushing their metadata server to the limits, they have the option of disabling the sctime and revalidating each object individually, or going without a persistent cache.

9 Collaborative Caching

A very common load pattern found in industry filesystem installations is one where read traffic vastly outnumbers write traffic. One such example is a cluster of web servers serving mostly static content.

Unless some effort is made to distribute the load in these situations, the servers will be completely overwhelmed, based purely on the raw bandwidth that a single server can provide. Consider the load placed on central servers if workstations have remote root filesystems and are all booted simultaneously following a power outage; or a lab of students all loading large applications at the beginning of a class.

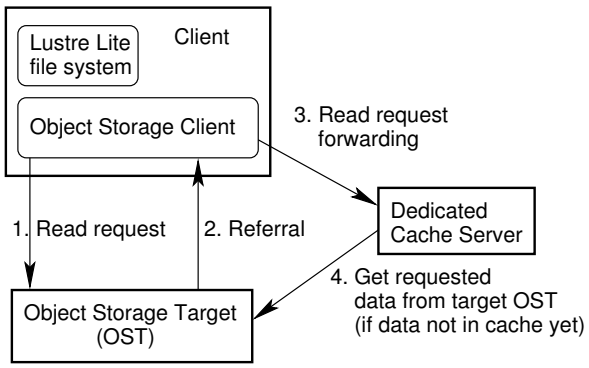


Figure 2: Collaborative Caching

Lustre is once again leveraging the object protocols into a symmetric collaborative caching device. Instead of working directly with an OST, a client communicates with one or more caching devices which can take locks on the client's behalf, locally service cache misses, and provide an alternative path to the high-demand data. These caching devices can also run on the clients themselves, which is the *collaborative* part of the collaborative cache.

With the addition of caching devices, recovery becomes somewhat more complicated. Normally the decision to give up on a particular OST following a network timeout is a very simple one. However with a cache in the middle it's important to distinguish between a failure of the cache node and a failure of the OST itself.

10 Conclusion

After more than a year of development, the Lustre framework has deviated surprisingly little from the original architecture. The lock manager and object protocols have served us well and will continue to form the centre of the design. Despite the fairly serious setbacks with

the VFS and client caching, our intent locking strategy has been shown to be successful on very large clusters and will be the primary mechanism for dealing with high-contention directories.

Today Lustre scales comfortably to more than 1,000 nodes and is running on 3 of the 8 largest clusters in the world[6] (at Lawrence Livermore and Pacific Northwest National Laboratories). We have every reason to believe that today's Lustre code will scale to 2,000 nodes without serious difficulties; the next two years of development are planned to address scalability and performance issues on a completely new scale of clusters which are only just beginning to be designed.

11 Acknowledgments

Lustre has benefitted significantly from the experience, guidance, and funding of several US Government national laboratories, notably Lawrence Livermore National Laboratory and Pacific Northwest National Laboratory. We have also received the support of the National Nuclear Security Administration ASCI Path-Forward Program, which provides funding for many of the advanced features in Lustre's future. Amongst our partners we are grateful for the support of and opportunities with Hewlett-Packard and Dell. The views and conclusions contained in this paper are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of our partners or the US Government.

12 Availability

Lustre is released under the terms and conditions of the GNU General Public License, and can be downloaded from our FTP site or checked out of our public CVS repository. More information can be found at `http://www.lustre.org/`

Founded in 2001 by Dr. Peter Braam, Cluster File Systems, Inc. is a privately held company headquartered on the internet, with developers in five countries. CFS is focused on high-end storage solutions, including the development of advanced file systems, novel architectures, and the storage industry as a whole. More information about how we can improve your storage offering, business, or laboratory can be found at `http://www.clusterfs.com/` or by writing to `info@clusterfs.com`

References

[1] IBM, *Programming Locking Applications*, Version 4.3.1, Second Edition, 1999.

[2] Brightwell et al, *The Portals 3.1 Message Passing Interface*, Revision 1.0, 1999.

[3] J. Howard, *An Overview of the Andrew File System*, In Proceedings of the USENIX Winter Technical Conference, 1988.

[4] Satyanarayanan, M, Kistler, J. J., Kumar, et. al., *Coda: a Highly available File System for a Distributed Workstation Environment*, IEEE Trans. on Computers, 39(4): 447-459, 1990.

[5] Braam, Callahan, and Schwan, *The InterMezzo Filesystem*, In Proceedings of the Ottawa Linux Symposium, 1999.

[6] `http://www.top500.org/list/2003/06/`

OSCAR Clusters

John Mugler, Thomas Naughton[*] and Stephen L. Scott[†]
Oak Ridge National Laboratory, Oak Ridge, TN

Brian Barrett[‡], Andrew Lumsdaine and Jeffrey M. Squyres[§]
Indiana University, Bloomington, IN

Benoît des Ligneris, Francis Giraldeau[¶]
Université de Sherbrooke, Québec, Canada

Chokchai Leangsuksun[‖]
Louisiana Tech University, Ruston, LA

Abstract

The Open Source Cluster Application Resources (OSCAR) is a cluster software stack providing a complete infrastructure for cluster computing. The OSCAR project started in April 2000 with its first public release a year later as a self-installing compilation of "best practices" for high-performance classic Beowulf cluster computing. Since its inception approximately three years ago, OSCAR has matured to include cluster installation, maintenance, and operation capabilities and as a result has become one of the most popular cluster computing packages worldwide. In the past year, OSCAR has begun to expand into other cluster paradigms including Thin OSCAR, a diskless cluster solution, and High-Availability OSCAR, embracing fault tolerant capabilities. This paper will cover the current status of OSCAR including its two latest invocations—Thin OSCAR and High-Availability OSCAR—as well as the details of the individual component technology used in the creation of OSCAR.

1 Introduction

The growth of cluster computing in recent years has primarily been fueled by the price performance quotient. The hardware investment to obtain a supercomputer caliber system extends the capabilities to a much broader audience. This advancement enables individuals to have exclusive access to their own super computer.

The capability of individual computing clusters involves the well known exchange of hardware costs for software costs. This software is necessary to build, configure and maintain the ever growing number of distributed heterogeneous machines that make up these clusters. The Open Cluster Group (OCG) was formed to address cluster management needs. The

[*]Contact author: Thomas Naughton, <naughtont@ornl.gov>

[†]This work was supported by the U.S. Department of Energy, under Contract DE-AC05-00OR22725.

[‡]Supported by a Department of Energy High Performance Computer Science Fellowship.

[§]Supported by a grant from the Lilly Endowment.

[¶]Supported by Centre de Calcul Scientifique

[‖]Supported by Center for Entrepreneurship and Information Technology (CEnIT), Louisiana Tech University

first working group formed by OCG was the Open Source Cluster Application Resources (OSCAR) project. The OSCAR group began by pooling "best practice" techniques for High Performance Computing (HPC) clusters into an easy to use toolkit. This included the integration of HPC components with a basic wizard to drive the installation and configuration.

The initial HPC oriented OSCAR has evolved with the base cluster toolkit being distilled into a more general cluster framework. The separation of the HPC specific aspects enables the framework to be used for other cluster installation and management schemes. This has led to the creation of two additional OCG working groups that leverage the basic OSCAR framework. The "Thin-OSCAR" working group uses the framework for diskless clusters. The "HA-OSCAR" group is focusing on high availability clusters. The relationship of OCG working groups and the framework is depicted in Figure 1.

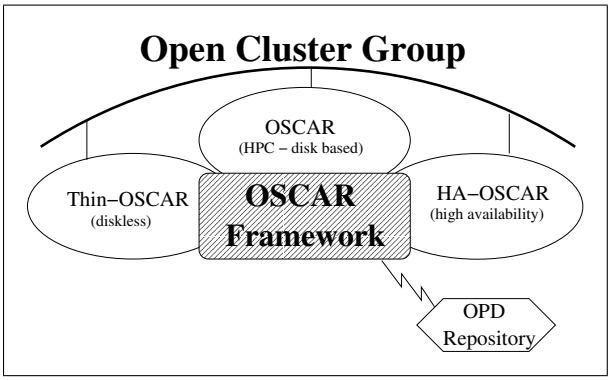


Figure 1: The Open Cluster Group (OCG) working groups share the OSCAR framework for cluster installation and management.

The following sections will briefly discuss the OCG umbrella organization and the current working groups. The basic OSCAR framework will be discussed followed by a brief summary of the HPC specific components. Then the diskless (Thin-OSCAR) and high availability (HA-OSCAR) working groups will be covered followed by concluding remarks.

2 Background

The OSCAR working group was the first working group that was formed under the umbrella Open Cluster Group (OCG) organization. The OCG was formed after a handful of individuals met in April 2000 to discuss their forays into cluster construction and management [1]. The consensus was that much effort was being duplicated and a toolkit to assist with the integration and configuration of current "best practices" would be beneficial to the participants and the HPC community in general.

This initial meeting led to subsequent discussions and ultimately a public release of the cluster installation, configuration, and management toolkit OSCAR v1.0 in April 2001. This initial release addressed OCG's mission statement to make cluster computing simpler while making use of the commonly used open source software solutions. In the years that have followed, the OSCAR working group has enhanced the toolkit with features such as modular OSCAR packages and improved cluster management facilities.

The OSCAR project is comprised of a mixture of industry and academic/research members. The overall project is directed by a steering committee that is elected every two years from the current "core organizations." This "core" list is composed of those actively contributing to project development. The 2003 core organizations include: Bald Guy Software (BGS), Dell, IBM, Intel, MSC.Software, Indiana University, the National Center for Supercomputing Applications (NCSA), Oak Ridge National Laboratory (ORNL), and Université de Sherbrooke.

There have also been new OCG working groups created to address other cluster envi-

ronments. These new working groups are named "Thin-OSCAR" and "HA-OSCAR." The "Thin-OSCAR" project provides support for diskless clusters. The "HA-OSCAR" group is focused on high availability clusters. The different groups focus on different cluster environments but leverage much of the core facilities offered by the base OSCAR framework.

3 OSCAR Framework

The standard OSCAR release targets usage in a High Performance Computing (HPC) environment. The base OSCAR framework is not necessarily tied to HPC. Currently the framework and toolkit are simply referred to as OSCAR and are used to build HPC clusters. To avoid confusion throughout this paper, a distinction will be made by using OSCAR to refer to the entire toolkit and OSCAR framework for the base facilities.

The framework includes the set of base or "core" packages needed to build and maintain a cluster. There are two other package classifications: "included" and "third-party." The *included* class of packages includes commonly used HPC applications for OSCAR. These packages are closely maintained and tested for compatibility with each OSCAR release. The *third-party* distinction is provided for all other OSCAR packages (see Section 3.5).

The core components enable a user to construct a virtual *image* of the target machine using System Installation Suite (SIS). There is also an OSCAR database (ODA) that stores cluster information. The final two components include a parallel distributed "shell" tool set called C3 and an environment management facility called Env-Switcher.

The fundamental function of OSCAR is to build and maintain clusters. This is greatly comprised of software package management. The guiding principle behind OSCAR and OCG is to use "best practices" when available. Thus, the Red Hat Package Manager (RPM) [2] is leveraged by OSCAR.[1] RPM files are pre-compiled binary versions of the software with meta data that is used to manage the addition, deletion, and upgrade of the package. RPM handles the conflict and dependence analysis. This add/delete/upgrade capability is a key strength of RPM. This is made use of by the framework in "OSCAR Packages."

3.1 System Installation Suite

The System Installation Suite (SIS) is based on the well known SystemImager tool [3, 4]. SystemImager is used to build an image—a directory tree that comprises an entire filesystem for a machine—that is used to install cluster nodes. The suite has two additional components: System Installer and System Configurator. These two components extend the standard SystemImager to allow for a description of the target to be used to build an image on the head node. This image has certain aspects generalized for on-the-fly customization via System Configurator. This dynamic configuration phase enables the image to be more general so items such as the network interface card are not in the SIS image. This capability allows for heterogeneity within the cluster nodes, while leveraging the established SystemImager management model.

SIS is used to "bootstrap" the node installs—kernel boot, disk partitioning, filesystem formatting, and base OS installation. The image used during the installation can also be used to maintain the cluster nodes. Modifying the image is as straight-forward as modifying a local filesystem. Once the image is updated,

[1] The underlying framework is designed to be as distribution agnostic as possible. The RPM name is slightly misleading but the system is available on distributions other than Red Hat.

`rsync`[2] is used to update the local filesystem on the cluster nodes. This method can be used to install and manage an entire cluster, if desired. This image based cluster management is especially useful for maintaining diskless clusters and is used by the Thin-OSCAR working group, (see Section 5).

3.2 C3 Power Tools

The distributed nature of clusters introduces a need to execute commands and exchange files throughout the cluster. The Cluster, Command and Control (C3) tool set offers a comprehensive set of commands to perform parallel command execution across cluster(s) as well as file scatter and gather operations [6, 7]. The tools are useful at both administrative and user levels. The tool set is the product of scalable systems research being performed at ORNL [8].

C3 includes commands to execute (`cexec`) across the entire cluster—or a subset of nodes—in parallel. File scatter and gather (`cpush`/`cget`) operations are also available. The C3 power tools have been developed to span multiple clusters. This multi-cluster capability is not fully harnessed by OSCAR currently but is available for administrators or standard users.

C3 is used internally throughout the OSCAR toolkit to distribute files and perform parallel operations on the cluster. For example, the user management commands, e.g., `useradd`, are made cluster aware using the C3 tools. Since C3 enables standard Linux commands to be run in parallel, administrators can use the tools to maintain clusters. A common example is the installation of a RPM on all nodes of the cluster, e.g.,

```
shell$ cexec rpm -ivh \
    foo-1.0.i386.rpm
```

[2] `rsync` is a tool to transfer files similar to `rcp`/`scp` [5].

3.3 Environment Switcher

Managing the shell environment—both at the system-wide level as well as on a per-user basis—has historically been a daunting task. For cluster-wide applications, system administrations typically need to provide custom, shell-defendant startup scripts that, create and/or augment `PATH`, `LD_LIBRARY_PATH`, and `MANPAGE` environment variables. Alternatively, users could hand-edit their "dot" files (e.g., `$HOME/.profile`, `$HOME/.bashrc`, and/or `$HOME/.cshrc`) to create/augment the environment as necessary. Both approaches, while functional and workable, typically lead to human error—sometimes with disastrous results, such as users being unable to login due to errors in their "dot" files.

Instead of these models, OSCAR provides the `env-switcher` OSCAR package. `env-switcher` forms the basis for simplified environment management in OSCAR clusters by providing a thin layer on top of the Environment Modules package [9, 10]. Environment Modules provide an efficient, shell-agnostic method of manipulating the environment. Basic primitives are provided for actions such as: add a directory to a `PATH`-like environment variables, displaying basic information about a package, and setting arbitrary environment variables. For example, a module file for setting up a given application may include directives such as:

```
setenv FOO_OUTPUT $HOME/results
append-path PATH /opt/foo-1.2.3/bin
append-path \
   MANPATH /opt/foo-1.2.3/man
```

The `env-switcher` package installs and configures the base Modules package and creates two types of modules: those that are unconditionally loaded, and those that are subject to system- and user-level defaults.

Many OSCAR packages use the unconditional modules to append the `PATH`, set arbitrary environment variables, etc. Hence, all users automatically have these settings applied to their environment (regardless of their shell) and guarantee to have them executed even when executing on remote nodes via `rsh`/`ssh`.

Other modules are optional, or a provide one-of-many selection methodology between multiple equivalent packages. This allows the system to provide a default set of applications, that optionally can be overridden by the user (*without* hand-editing "dot" files). A common example in HPC clusters is having multiple Message Passing Interface (MPI) [11, 12] implementations installed. OSCAR installs both the LAM/MPI [13] and MPICH [14] implementations of MPI. Some users prefer one over the other, or have requirements only met by one of them. Other users wish to use both, switching between them frequently (perhaps for performance comparisons).

The `env-switcher` package provides trivial syntax for a user to select which MPI package to use. The `switcher` command is used to select which modules are loaded at shell initialization time. For example, the following command shows a user selecting to use LAM/MPI:

```
shell$ switcher mpi = lam-6.5.9
```

3.4 OSCAR Database

The OSCAR database, or ODA, is used to describe the software in an OSCAR cluster. As of OSCAR version 2.2.1, ODA has seven tables in a MySQL database named *oscar* which runs on the head node. Every OSCAR package has an XML meta file named `config.xml` that is used to populate the package database. This XML file contains a description of the package, and the RPM names associated with the package. Thus the database enables a user to find information on packages quickly and easily.

ODA provides a command line interface into the database via the `oda` command. This offers easier access into the database without having to use a MySQL client. ODA also provides an abstraction layer between the user and the actual backing store method. The internal organization of data is masked through the use of a uniform interface irrespective of this underlying database engine. The retrieval and update of cluster information is aided by the use of ODA "shortcuts" to assist with common tasks.

ODA is intended to provide OSCAR package developers with a central repository for information about the installed cluster. This alleviates the problem of every package developer having to keep an independent data store, and makes the addition and removal of packages easier.

3.5 OSCAR Packages

An OSCAR package is a simple way to wrap software and a given configuration for a cluster. The most basic OSCAR package is an RPM in the appropriate location in the package directory structure. The modularity of this facility allows for easy addition of new software to the framework. OSCAR packages are most useful, however, when they also provide supplemental documentation and a meta file describing the package. The packaging API provides authors the ability to make use of scripts to configure the cluster software outside of the RPM itself. The scripts fire at different stages of the installation process and test scripts can be added to verify the process. Additionally, an OSCAR Package Downloader (OPD) (see Section 3.6) is provided to simplify acquisition of new packages.

With this modular design, packages can be updated independently of the core utilities and core packages. The OSCAR toolkit is freely redistributable and therefore requires all "se-

lected" packages to adhere to this constraint. However, any software which does not fulfill this requirement can be made available via an OSCAR repository with access via OPD.

The contents and directory structure of a typical OSCAR package are listed below. For further details on the creation of packages for the toolkit, see the OSCAR Architecture document in the development repository [15].

`config.xml` – meta file with description, version, etc.

`RPMS/` – directory containing binary RPM(s) for the package

`SRPMS/` – directory containing source RPM(s) used to build the package

`scripts/` – set of scripts that run at particular times during the installation/configuration of the cluster

`testing/` – unit test scripts for the package

`doc/` – documentation and/or license information

3.6 OSCAR Package Downloader

The OSCAR Package Downloader (OPD) provides the capability to download and install OSCAR software from remote package repositories. A package repository is simply an FTP or web site. Given the ubiquitous access to FTP and web servers, any organization can host their own OSCAR package repository and publish their packages on it. There is no central repository; the OPD network was designed to be distributed such that no central authority is required to publish OSCAR packages. Although the OPD client program downloads an initial list of repositories from the OSCAR Working Group web site,[3] arbitrary repository sites can be listed on the OPD command line.

Since package repositories are FTP or web sites, any traditional FTP client or web browser can also be used to obtain OSCAR packages. Most users prefer to use the OPD client itself, however, because it provides additional functionality over that provided by traditional clients. OPD offers two interfaces: a simple menu-based mechanism suitable for interactive use and a command-line interface suitable for use by higher-level tools (or automated scripts).

Partially inspired by the Comprehensive Perl Archive Network (CPAN), OPD provides the following high-level capabilities:

- Automating access to a central list of repositories
- Browsing packages available at each repository
- Providing detailed information about packages
- Downloading, verifying, and extracting packages

While the job that OPD performs is actually fairly simple and could be performed manually, having an automated tool for these functions provides ease of use for the end-user, performs multiple checks to ensure that downloaded and extracted properly, and lays the groundwork for higher-level OSCAR package/retrieval tools.

[3] The centralized repository list is maintained by the OSCAR working group. Upon request, the list maintainers will add most repository sites.

4 OSCAR Toolkit

The integration of common HPC packages is a key feature of the OSCAR toolkit. The focus of this paper is to talk about the framework, however, and its use by the various OCG working groups. A brief highlight of the packages is provided with further details available in other articles [16, 17].

The "selected" packages that are included enable a user to install and configure the head node and cluster nodes to run HPC applications. These HPC packages include parallel libraries like LAM/MPI, MPICH and PVM. A batch queue system and scheduler—OpenPBS and MAUI—is included and setup with a reasonable set of defaults. The toolkit sets up common cluster services such as Network File System (NFS) and Network Time Protocol (NTP). Security packages like Pfilter [18] are setup as well as OpenSSH with non-interactive access to all nodes in the cluster from the head node. The testing scripts provided with the packages are executed by the OSCAR framework to validate the cluster installation.

5 Thin-OSCAR

5.1 Goals

Thin-OSCAR[4] is a workgroup dedicated to integrate diskless clustering techniques into OSCAR so that the OSCAR infrastructure can use diskless nodes. There are three class of nodes in the thin-OSCAR perspective: diskless, systemless (a disk is present in the node but there is no operating system on the disk) and diskfull (regular OSCAR node with disk). At the time of this writing, only diskless and diskfull nodes are supported out of the box. Systemless nodes are supported to some extent but you will have to fiddle with some config files manually. Moreover, a generic abstraction of a node is necessary to build a generic but comprehensive interface.

[4]Workgroup web site: `http://thin-oscar.ccs.usherbrooke.ca/`

5.2 Principle of operation

The thin-OSCAR model is the following: the thin-OSCAR package is a collection of Perl scripts and libraries that are used to transform a regular SIS image (as used by OSCAR) into the two ram disks necessary for diskless and systemless nodes. The first ram disk to be transferred is called the "BOOT image" and is used in order to ensure that the node has network connectivity, NFS client capabilities and to create a raid0 array of ram disk [19]. Once this minimal image (less than 4Mb) has booted, the RUN image from the second ram disk can be transferred. The RUN image is build directly from the SIS image and contains the complete system that will run on the node. Some directories are copied from the SIS image while others are NFS exports (read-only) directly exported from the SIS image directory [20].

In order to build the BOOT image, you will need the SIS image name, modules name to integrate into this ram disk as well as the modules used by your NIC adapter and the kernel version you want to use. The use of the root raid in ram technique is necessary because one of the thin-OSCAR goals is to support the regular kernel (only 4 Mb or ramdisk) without recompilation.

5.3 mini howto

In order to download thin-OSCAR, please use Oscar Package Downloader and select the Université de Sherbrooke repository. Then download thin-OSCAR. Before actually using thin-OSCAR, you have to complete a regular installation from stage 1 (selection of packages) to stage 6 (setup networking). Assign nodes IP to

MAC addresses and don't forget to click on the "Setup Network Boot" button.

Once this is done, you are ready to use the thin-OSCAR package. Go to `/usr/lib/oscar/packages/thin-oscar/` and run the `./oscar2thin.pl` script. You will go into an interface where you will have to define your diskless model, link the model to an OSCAR node and then generate all the necessary ram disk (Configure All). Once this is done, you are finished and can reboot all the diskless nodes. You must know which kernel modules are necessary for your NIC before starting this process in order to setup network connectivity.

5.4 thin-OSCAR future

We will certainly move to a more modern RAM filesystem (tmpfs) as it is now available on regular kernels and toward an automatic detection of NIC modules so that BOOT ramdisk creation can be fully automated. Multicast transfer of the ramdisk is under study as it will shorten the boot time and the network usage during boot (especially for big clusters!).

The proof of concept of the thin-OSCAR has been done and, at the time of this writing, thin-OSCAR is used on a 180 node production diskless cluster [21]. An improved integration into the OSCAR framework is under development and will be available as soon as the OSCAR GUI enables it.

6 HA-OSCAR

High-Availability (HA) computing, once thought as only important to industry applications such as telecommunications, has become critically important to the fundamental mission of high-performance computing. This is because very large and complex application codes are being run on increasingly larger scale distributed computing environments. Since COTS (common off the shelf) hardware is typically employed to construct these environments (clusters), quite often the application code's runtime exceeds the hardware's aggregated mean-time-between-failure rate for the entire cluster. Thus, in order to efficiently run these very large and complex applications, high-availability computing techniques must be employed in the high-performance computing environment (HPC).

The current HPC release of OSCAR is fully suitable for mission critical systems as it contains several individual system elements that exhibit a single-point-of-failure trait. In order to support HA requirements, clustered systems must provide ways to eliminate single-point-of-failures. HA-OSCAR is a focus group with goals to add HA features to the original HPC OSCAR distribution. While HA-OSCAR is still a work in progress, the scope of this effort has been defined into three incremental steps; the creation of the HA-OSCAR whitepaper [22], Active-Hot-Standby, and n + 1 Active-Active distributions.

Hardware duplication and network redundancy are common techniques utilized for improving the reliability and availability of computer systems. To achieve the HA-OSCAR cluster system, we must first provide a duplication of the cluster head node. There are different ways for implementing such an architecture, which includes Active-Active, Active-Hot Standby and Active-Cold Standby [23]. Currently, the Active-Hot Standby configuration is the initial model of choice. Figure 2 shows the HA-OSCAR cluster system architecture. We have experimented with and planned to incorporated Linux Virtual Server and Heartbeat mechanisms to our initial Active- Hot Standby HA-OSCAR distribution. However, we will extend the initial architecture to support the

Active-Active HA after release of the Hot-Standby distribution. The Active-Active architecture will provide a better resource utilization since both head nodes will be simultaneously active and providing services. The dual master nodes will run redundant OpenPBS, MAUI, DHCP, NTP, TFTP, NFS, rsync and SNMP servers. In the event of a head node outage, all functions provided by that node will failover to the second redundant head node and all service requests will continue to be served, although at a reduced performance rate (i.e. in theory, 50% at the peak or busy hours).

An additional HA functionality to support in HA-OSCAR is that of providing a high- availability network via redundant Ethernet ports on every machine in addition to duplicate switching fabrics (network switches, cables, etc.) for the entire network configuration. This will enable every node in the cluster to be present on two or more data paths within its networks. Backed with this Ethernet redundancy, the cluster will achieve higher network availability. Furthermore, when both networks are up, an improved communication performance may be achieved by using techniques such as channel bonding of messages across the redundant communication paths.

7 Conclusion

The Open Cluster Group (OCG) has formed several working groups focused on improving cluster computing management. The first working group, OSCAR, has evolved over time as has the toolkit by the same name. The OSCAR toolkit has also been distilled in order to allow for more general usage. This more general OSCAR framework is being leveraged by subsequent working groups seeking to extend support for new cluster environments. These new environments include the areas of diskless and high availability clusters. These are

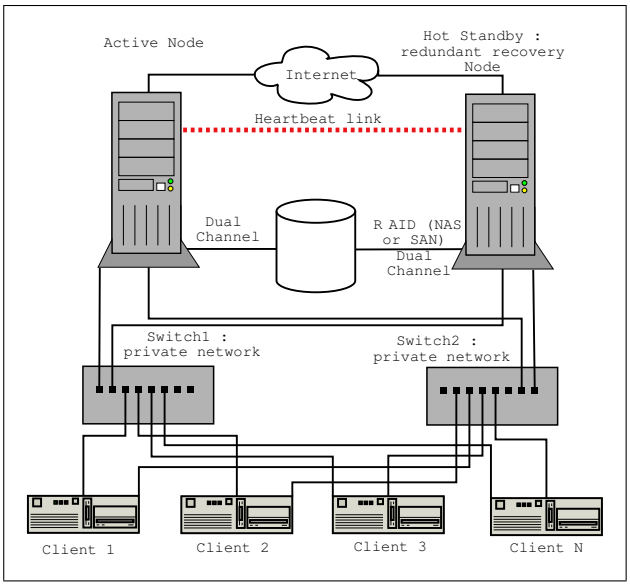


Figure 2: Diagram of HA-OSCAR architecture.

being pursued by the Thin-OSCAR and HA-OSCAR working groups respectively. These OCG working groups are providing the cluster community with sound tools to simplify and speed cluster installation and management.

References

[1] Richard Ferri. The OSCAR revolution. *Linux Journal*, (98), June 2002. `http://www.linuxjournal.com/article.php?sid=5559`.

[2] Edward C. Bailey. *Maximum RPM: Taking the Red Hat Package Manager to the Limit*. Red Hat Software, Inc., 1998.

[3] Sean Dague. System Installation Suite Massive Installation for Linux. In *The 4th Annual Ottawa Linux Symposium (OLS'02)*, Ottawa, Canada, June 26-29, 2002.

[4] System Installation Suite (SIS), `http://www.sisuite.org/`.

[5] A. Tridgell and P. Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, Australian National University, Department

of Computer Science, June 1996. (see also: `http://rsync.samba.org/`).

[6] M. Brim, R. Flanery, A. Geist, B. Luethke, and S. Scott. Cluster Command & Control (C3) tools suite. In *To be published in, Parallel and Distributed Computing Practices, DAPSYS Special Edition*, 2002.

[7] Cluster Command & Control (C3) Power Tools, `http://www.csm.ornl.gov/torc/C3`.

[8] Al Geist et al. Scalable Systems Software Enabling Technology Center, March 7, 2001. `http://www.csm.ornl.gov/scidac/ScalableSystems/`.

[9] John L. Furlani. Modules: Providing a flexible user environment. In *Proceedings of the Fifth Large Installation Systems Administration Conference (LISA V)*, pages 141–152, San Diego, CA, September 1991. `http://modules.sourceforge.net/`.

[10] John L. Furlani and Peter W. Osel. Abstract yourself with modules. In *Proceedings of the Tenth Large Installation Systems Administration Conference (LISA '96)*, pages 193–204, Chicago, IL, September 1996. `http://modules.sourceforge.net/`.

[11] Al Geist, William Gropp, Steve Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, William Saphir, Tony Skjellum, and Marc Snir. MPI-2: Extending the Message-Passing Interface. In Luc Bouge, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par '96 Parallel Processing*, number 1123 in Lecture Notes in Computer Science, pages 128–135. Springer Verlag, 1996.

[12] Message Passing Interface Forum. MPI: A Message Passing Interface. In *Proc. of Supercomputing '93*, pages 878–883. IEEE Computer Society Press, November 1993.

[13] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In John W. Ross, editor, *Proceedings of Supercomputing Symposium '94*, pages 379–386. University of Toronto, 1994.

[14] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[15] Open Cluster Group: OSCAR Working Group. OSCAR: A packaged cluster software for High Performance Computing. `http://www.OpenClusterGroup.org/OSCAR`.

[16] Thomas Naughton, Stephen L. Scott, Brian Barrett, Jeff Squyres, Andrew Lumsdaine, and Yung-Chin Fang. The Penguin in the Pail – OSCAR Cluster Installation Tool. In *Proceedings of SCI'02: Invited Session – Commodity, High Performance Cluster Computing Technologies and Applications*, Orlando, FL, USA, 2002.

[17] Benoît des Ligneris, Stephen L. Scott, Thomas Naughton, and Neil Gorsuch. Open Source Cluster Application Resources (OSCAR) : design, implementation and interest for the [computer] scientific community. In *Proceeding of 17th Annual International Symposium on High Performance Computing Systems and Applications (HPCS 2003)*, pages 241–246, Sherbrooke, Canada, May 11-14, 2003.

[18] Neil Gorsuch. PFILTER in OSCAR - Industrial Strength Cluster Firewalls in an Open Source Environment. In *Proceeding of 1st Annual OSCAR Symposium (OSCAR 2003)*, Sherbrooke, Canada, May 11-14, 2003.

[19] Mehdi Bozzo-Rey, Michel Barrette, Benoît des Ligneris, and Francis Giraldeau. Root raid in ram how to. In *Proceeding of 17th Annual International Symposium on High Performance Computing Systems and*

Applications (HPCS 2003), pages 241–246, Sherbrooke, Canada, May 11-14, 2003.

[20] Benoît des Ligneris, Michel Barrette, Francis Giraldeau, and Michel Dagenais. Thin-OSCAR : Design and future implementation. In *Proceeding of 1st Annual OSCAR Symposium (OSCAR 2003)*, pages 261–265, May 11-14, 2003.

[21] M. Barrette, X. Barnabé-Thériault, M. Bozzo-Rey, C. Gauthier, F. Giraldeau, B. des Ligneris, J.-P. Turcotte, P. Vachon, and A. Veilleux. Development, installation and maintenance of Elix-II, a 180 nodes diskless cluster running thin-oscar. In *Proceeding of 1st Annual OSCAR Symposium (OSCAR 2003)*, pages 267–271, May 11-14, 2003.

[22] I. Haddad, F. Rossi, C. Leangsuksun, and S. L. Scott. Telecom/High Availability OSCAR Suggestions for the 2nd Generation OSCAR. Technical Report TR-LTU-12-2002-01, Louisiana Tech University, Computer Science Program, December 2002.

[23] P. S. Weygant. *Cluster for high availability: A Primer of HP solutions*. Hewlett-Packard Company, Prentice-Hall, Inc., second edition, 2001.

Porting Linux to the M32R processor

Hirokazu Takata
Renesas Technology Corp., System Core Technology Div.
4-1, Mizuhara, Itami, Hyogo, 664-0005, Japan
takata.hirokazu@renesas.com

Naoto Sugai, Hitoshi Yamamoto
Mitsubishi Electric Corp., Information Technology R&D Center
5-1-1, Ofuna Kamakura, Kanagawa 247-8501, Japan
{sugai,hitoshiy}@isl.melco.co.jp

Abstract

We have ported a Linux system to the Renesas[1] M32R processor, which is a 32-bit RISC microprocessor designed for embedded systems, and with an on-chip-multiprocessor feature.

So far, both of UP (Uni-Processor) and SMP (Symmetrical Multi-Processor) kernels (based on 2.4.19) have been ported and they are operating on the M32R processor. A Debian GNU/Linux based system has been also developed on a diskless NFS-root environment, and more than 300 unofficial `.deb` packages have already been prepared for the M32R target.

In this paper, we describe this new architecture port in detail and explain the current status of the Linux/M32R project.

[1]Renesas Technology Corp. is a new joint semiconductor company established by Hitachi Ltd. and Mitsubishi Electric Corp. on April 1, 2003. It would be the industry's largest microcontroller (MCU) supplier in the world. The M32R family microcontroller and its successor will be continuously supplied by Renesas.

1 Introduction

A Linux platform for Renesas M32R processor has been newly developed. The Renesas M32R processor is a 32-bit RISC microprocessor, which is designed for embedded systems. It is suitable for a System-on-a-Chip (SoC) LSI due to its compactness, high performance, and low power dissipation. So far, the M32R family microcomputers have widely used for the products in a variety of fields—for example, automobiles, digital still cameras, digital video camcorders, cellular phones, and so on.

Recently, the Linux system has begun to be used widely and employed even in the embedded systems. The embedded systems would be more software-oriented systems hereafter. The more complex and larger the embedded system is, the more complicated the software becomes and harder to develop. In order to build these kinds of embedded systems efficiently, it will be more important to utilize generic OSes such as Linux to develop software.

This is the first Linux architecture port to the M32R processor. This porting project, called a "Linux/M32R" project, has been active since

2000. Its goal is to prepare a Linux platform for the M32R processor. At first, this Linux porting was just a feasibility study for the new M32R processor development, and it was started by only a few members of the M32R development team. Then, this project has grown to a lateral project among Renesas Technology Corp., Renesas Solutions Corp., and Mitsubishi Electric Corp.

In this feasibility study, we have ported not only Linux kernel, but also whole GNU/Linux system including GNU tools, libraries, and other software packages, so called "userland." We also enhanced the M32R processor to add MMU (Memory Management Unit) facility in order to port Linux system. And we have also developed an SMP kernel to investigate multiprocessing by M32R's on-chip-multiprocessor feature[1]. At present, the Linux/M32R system can operate on the dual M32R cores in SMP mode.

In this paper, we describe this new architecture port in detail and explain about the current status of the Linux/M32R project.

2 Linux/M32R Platform for Embedded Systems

Recently, due to the continuous evolution of semiconductor technologies, it is possible to integrate a whole system into one LSI chip, so called "System-on-a-Chip (SoC)."

In an SoC, microprocessor core(s), peripheral I/O functions, internal memories, and user logics can be integrated into a single chip.

By making use of wide internal buses, an LSI can achieve high performance which can not be realized by combination of several general-purpose LSIs. In other words, we can optimize system performance and cost by using SoC, because we can employ optimum hardware architecture and circuit configuration.

In such an SoC, a microprocessor core is a key part; therefore, the more compact and higher performance microprocessor is significantly required. To make such a high performance embedded processor core, not only in circuit and process technology but also architectural breakthrough is necessary. Especially, multiprocessor technology is important even for the embedded processor, because it can improve processor performance and lower system power dissipation by increasing processor number scalably and without increasing operating clock frequency.

For an SoC with embedded microprocessor, software is also a key point. The more system is highly functional, the more software will be complex and there is an increasing demand for shortening development time of SoC. Under such circumstance, recently the Linux OS becomes to be adopted for embedded systems. Linux platform makes it easy to port application programs developed on a PC/EWS to the target system. We believe that a full-featured Linux system will come into wide use in embedded systems, because embedded systems will become more functional and higher performance system will be required.

In the development of embedded systems, it is important to tune system performance from both hardware and software points of view. Therefore, we used M32R softmacro and FPGA (Field Programmable Gate Array) devices to implement an evaluation board for rapid system prototyping. FPGA devices are slow, but make it possible to develop a system in short turn-around time.

In this feasibility study, to construct a Linux platform, we ported Linux to the M32R architecture, and validated the hardware system architecture through the porting, and developed the software development environment.

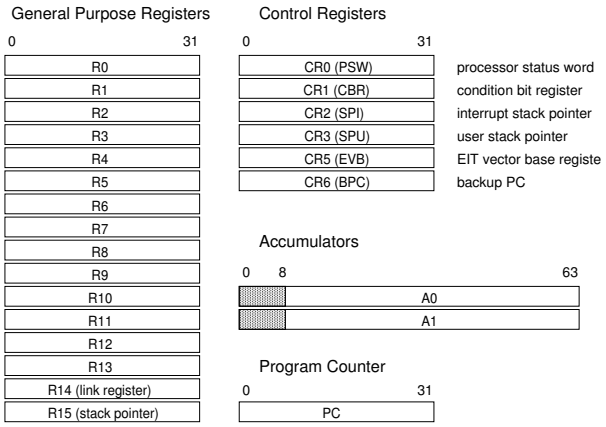


Figure 1: M32R register architecture

2.1 M32R architecture

The M32R is a 32-bit RISC microprocessor, and employs load-store architecture like other RISC processors. Therefore, memory access is executed by only load and store instructions and logical and arithmetic operation is executed among registers. Except for multiply and divide instructions, most of instructions can be executed in one clock, and instruction completion does not depend on the the instruction issuing order (out-of-order completion). The M32R supports DSP operation instructions such as multiply and accumulate instructions.

Figure 1 shows the M32R register architecture. The M32R has sixteen 32-bit general purpose registers (R0 $\sim$ R15) and 56-bit accumulators for the multiply and accumulate operations. R14 is also used as a link register (LR) which keeps return address for a subroutine call. There are two stack pointer registers, SPI (interrupt stack pointer) and SPU (user stack pointer). The CPU core selects one of them as a current stack pointer (R15; SP) by the SM (stack mode) bit of the PSW (processor status word).

2.2 M32R softmacro

The M32R softmacro is a compact microprocessor, developed to integrate into a SoC. It is a full synthesizable Verilog-HDL model and it has an excellent feature that the core does not depend on a specific process technology. Due to a synchronous edge-triggered design, it has good affinity to EDA tools.

This M32R softmacro is so compact that it can be mapped into one FPGA. Utilizing such an M32R softmacro, we developed software on a prototype hardware and co-designed hardware and software simultaneously.

To port Linux to the M32R, some enhancement of the M32R softmacro core was needed; processor mode (user mode and supervisor mode) was introduced and an MMU module was newly supported.

- TLB (Translation Lookaside Buffer): instruction/data TLBs are full-associative, 32-entries each, respectively.
- page size : 4kB/16kB/64kB (user page), 4MB (large page)

2.3 Integrated debugging function and SDI

The integrated debugging function is a significant characteristic of the M32R family microcomputer. The M32R common debugging interface, called as SDI (Scalable Debug Interface), is utilized via five JTAG pins; the internal debug functions are controlled through these debug pins.

Using the JTAG interface defined as IEEE 1149.1, internal debug function can be used. No on-chip or on-board memory to store monitor programs is necessary, because such monitor programs can be provided and executed via JTAG pins.

3 Porting Linux to the M32R

The Linux system consists of not only the Linux kernel, but also the GNU toolchain and libraries. Of course, a target hardware environment is also necessary to execute Linux.

Therefore we had to accomplish the following tasks:

- Porting the Linux kernel
- Development of Linux/M32R platforms (M32R FPGA board, etc.)
- Enhancement of the GNU toolchain
- Porting libraries (GNU C library, etc.)
- Userland; preparing software packages

Actually, in the Linux/M32R development, these tasks have developed concurrently.

3.1 Porting Kernel to the M32R

In the Linux kernel, the architecture-dependent portion is clearly distinguished. Therefore, in case of a new architecture port, all you need to do is prepare only architecture dependent code, which is far less than whole Linux code. In the M32R case, `include/asm-m32r/` and `arch/m32r/` are needed.

To be more precise, we can prepare the architecture-dependent portion in reference to the other architecture implementation. However, it has some difficulties in rewriting these portions:

asm function : It was very difficult to port some headers in which asm statement is extensively used, because an insufficient and inadequate rewriting easily cause bugs which are very hard to debug.

function inlining : In the Linux source code, function inlining is heavily used. We can not disable the compiler's inline optimization function to compile the kernel source, but a buggy toolchain sometimes causes a severe problem in optimization.

In this porting, we started with a minimum configuration kernel. We prepared stub routines, built and checked the kernel operation, and gradually added files of `kernel/`, `mm/`, `fs/`, and other directories. When we started kernel porting, there was no evaluation board. So, we made use of GNU M32R simulator to port a kernel at first.

The GNU simulator was very useful at the initial stage of kernel porting, though it does not support MMU. It had also good characteristics that downloading was quite fast comparing with evaluation board and C source level debug was possible.

Employing the simulator and initrd image of romfs root filesystem, it is possible to develop and debug kernel's basic operation, such as scheduling, initialization and memory management. Indeed, the demand loading is performed after `/sbin/init` is executed by a `execve()` system call at the end of kernel boot sequence of `init()`.

At first, we started to port the kernel 2.2.16. The current stable version of the M32R kernel is 2.4.19 and now we are developing 2.5 series kernel for the M32R.

3.1.1 System Call Interface

In the M32R kernel, like other processors, a system call is handled by a system call trap interface. In the system call interface (syscall I/F), all general purpose registers and accumulators are pushed onto the kernel stack to save

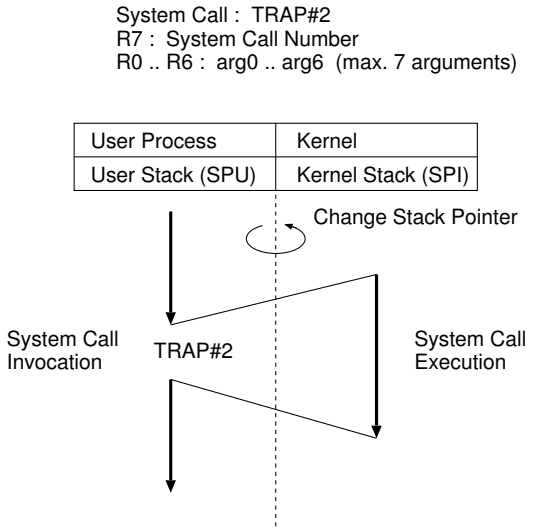


Figure 2: System call interface

the context.

In Fig. 2, the syscall ABI (Application Binary Interface) for the M32R is shown. Two stack pointers, kernel stack (SPI) and user stack (SPU), are switched over by software at the entry point of syscall I/F routine, because stack pointers do not change automatically by `TRAP` instruction. In order to switch stack pointers without working register and avoid multiple TLB miss exception, `CLRPSW` instruction is newly introduced.

The stack frame formed by the syscall I/F routine is shown in Fig. 3. It should be noted that there is a special system call in Linux, like sys_clone(), that has particular interface passing a stack top address as a first argument. Therefore, we employ a parameter passing method: The stack top address (∗pt_regs) is always put onto the stack as a implicit stack parameter like the SuperH implementation.

According to the ABI of the M32R gcc compiler, the first 4 arguments are passed by registers and the following arguments are passed by stack. Therefore, the ∗pt_regs parameter can be accessed as the eighth parameter on the kernel stack.

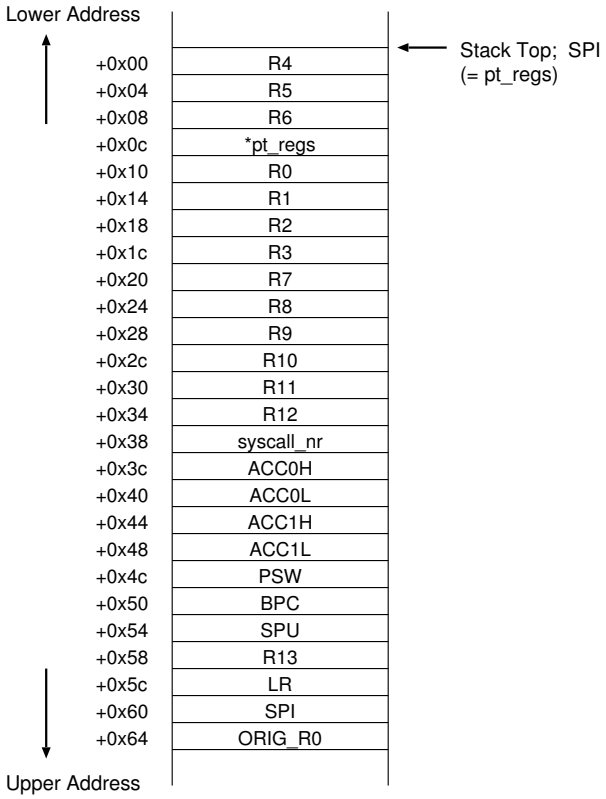


Figure 3: Stack frame formed by a system call

The `syscall_nr` and `ORIG_R0` field are used for the signal operations. When a system call is issued, its system call number is stored into `R7` and trap instruction is executed. `syscall_nr` also holds the system call number in order to determine if a signal handler is called from a system call routine or not. Because the `R0` field might be changed to the return value of a system call, `ORIG_R0` keeps the original value of `R0` in preparation to restart the system call.

3.1.2 Memory Management

Linux manages the system memory by *paging*. In the M32R kernel, the page size is 4kB like the other architecture.

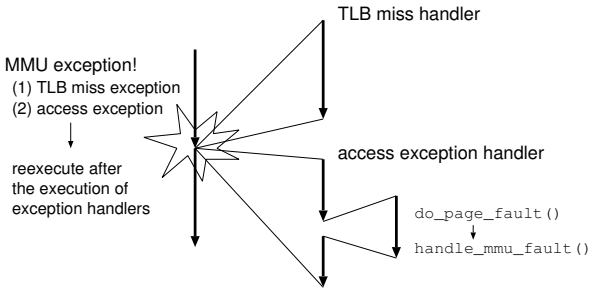


Figure 4: Exception handling for the demand-loading operation

In demand loading and copy-on-write operations, a physical memory page can be newly mapped when a *page fault* happens. Such a page fault is handled by both *TLB miss handler* and *access exception handler* (Fig. 4).

demand loading : If an instruction fetch or operand access to the address which is not registered in page table, an MMU exception happens. In case of a TLB miss exception, TLB miss handler is called. To lighten the TLB miss handling operation, TLB handler only sets TLB entries. Page mapping and page-table setting operations are to be handled by the access exception handler; For accessing a page which does not exist in the page table, the TLB miss handler sets the TLB entry's attribute to not-accessible at first. After that, since the memory access causes an access exception due to not-accessible, access exception handler deal with the page table operations.

copy-on-write : In Linux, copying a process by `fork()` and reading a page in read-only mode are handled as a copy-on-write operation to reduce vain copy operations. For such a copy-on-write operation, TLB miss handler and access exception handler are used like a demand loading operation.

The M32R's data cache (D-cache) is indexed and tagged physically. So, it does not have to take care the cache aliasing. Therefore the D-cache is flushed only for a signal handler generation and a trampoline code generation.

To simplify and speed up the cache flushing operations for trampoline code, a special cache flush trap handler (trap#12) is established in the M32R kernel.

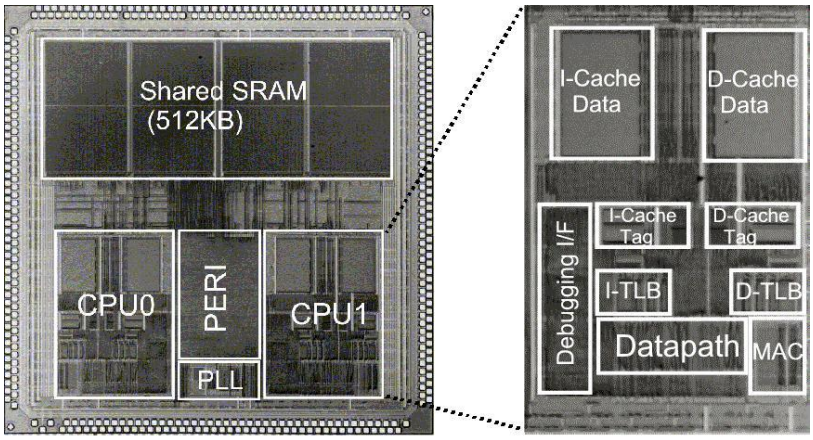
3.1.3 SMP support

In Linux 2.4, multiprocessing performance is significantly improved compared with Linux 2.2 or before, because the kernel locking for accessing resources is finer.

To implement such a kernel locking on SMP kernel, *spinlock* is generally used for mutual exclusion control. But the M32R has no atomic test-and-set instruction, the spinlock operations can be implemented with `LOCK` and `UNLOCK` instructions in the M32R kernel.

The `LOCK` and `UNLOCK` instructions are a load and store instructions for mutual exclusion operation, respectively. `LOCK` acquires and keeps a privilege to access the CPU bus until `UNLOCK` instruction is executed. Accessing a lock variable by `LOCK/UNLOCK` instruction pair under a disabled interruption condition, we implemented an atomic access.

Figure 5 shows an M32R on-chip-multiprocessor prototype chip. Linux SMP kernel can be executed on the on-chip multiprocessor system. On-chip multiprocessor might be a mainstream in near future even in embedded systems, because multiprocessor system can enhance the CPU performance without increasing operating clock frequency and power dissipation. In this chip, two M32R cores are integrated and each has its own cache for good scalability.



(a) Chip (b) CPU

Figure 5: A micrograph of an on-chip-multiprocessor M32R prototype chip

3.2 Development of Linux/M32R Platform

To execute full-featured Linux OS, an MMU is necessary; therefore, we developed a new M32R softmacro core with an MMU and made an evaluation board "Mappi," which used FPGAs to map the M32R softmacro core, as a Linux/M32R platform.

As shown in Fig. 6, the Mappi evaluation board consists of two stacked boards. The upper board is a CPU board and the lower board is an extension board. The CPU board has no real CPU chip, but it has two large FPGAs on it. We employ the M32R softmacro core and map it onto the FPGAs.

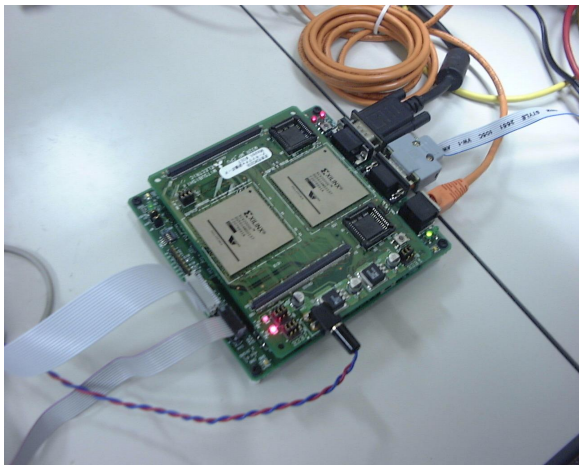
The Mappi board is a very flexible system for prototyping. If we have to modify a CPU or other integrated peripherals, we can immediately change and fix them by modifying their Verilog-HDL model.

At first, we could only use `initrd` and `busybox` on it, because the Mappi system had only a CPU board and it had only 4MB SRAMs. After the extension board was developed, more memory (SDRAM), Ethernet, and PC-card I/F became available. So, we introduced NFS and improved the porting environment. It was Dec. 2001 that we succeeded in booting via a network using the extension board.

Utilizing the M32R's SDI function and JTAG-ICE, mentioned before, we can download and debug a target program via JTAG port. It is much faster than a serial connection because the Debug DMA function is used for downloading and refering internal resources. Of cource, it is also possible to set hardware breakpoints for the PC break and the access break via SDI.

Generally speaking, it is too difficult to develop and debug software programs on an unsteady hardware which is under development. But, we could debug and continued to develop the system by using the SDI debugger, because the SDI debugger made it possible to access the hardware resources directly and it was very useful for the kernel debugging.

Finally, we constructed an SMP environment to execute the SMP kernel, mapping the M32R softmacro cores to two FPGAs on the Mappi



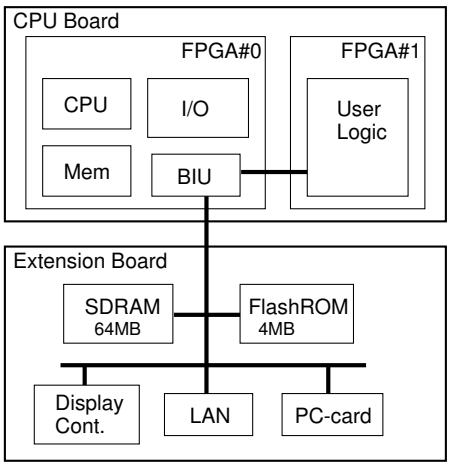


Figure 6: Mappi: the M32R FPGA evaluation board; it has the M32R softmacro on FPGA (CPU, MMU, Cache, SDI, SDRAMC, UART, Timer), FPGA Xilinx XCV2000E ×2, SDRAM(64MB), FlashROM, 10BaseT Ethernet, Serial 2ch, PC-card slot ×2, and Display I/F(VGA)

CPU board; concretely, we replaced the user logic portion in FPGA#1 shown in Fig. 6 with an another M32R core with a bus arbitor, and modified the ICU (Interrupt Control Unit) to support inter-processor interruption for the multiprocessor.

After the M32R prototype on-chip-multiprocessor chip was developed, the Linux/M32R system including userland applications has been mainly developed by using the real chip, because the operating clock frequency of the M32R FPGA is 25MHz but the M32R chip can run more than 10 times faster.

3.3 M32R GNU toolchain enhancement

The GNU toolchain is necessary to develop the Linux kernel and a variety of Linux application programs. When we started the Linux porting, we had only Cygnus's (now it's Red Hat, Inc.) GNUPro™m32r-elf toolchain. It was sufficient for the kernel development; however, it could not be applicable to user application development on Linux, because in a modern UNIX system a dynamic linking method is strongly required to build a compact system and achieve higher runtime performance. (Although a static linked program is much faster than a dynamic linked program if the program size is small. The bigger a program becomes, the larger cache miss penalty would be.)

We enhanced the M32R GNU toolchain to support shared libraries:

- Change BFD library to support dynamic linking; some relocations were added for dynamic linking.
- Change GCC and Binutils to support PIC (Position Independent Code).

Because the version of the GNUpro m32r-elf gcc was 2.8 and too old, we had to upgrade and develop a new m32r-linux toolchain. We applied GNUpro patch to the gcc of the FSF version and developed GCC (v2.95.4, v3.0, v3.2.2) and Binutils (v2.11.92, v2.13.90).

In a prologue portion of a C function, the following code is generated when the `-fPIC` option is specified.

```
; PROLOGUE
push r12
push lr
bl .+4  ; get the next instruction's
        ; PC address to lr
ld24 r12,#_GLOBAL_OFFSET_TABLE_
add r12,lr
```

We also modified BFD libraries to support dynamic linking. We referenced the i386 implementation and supported the ELF dynamic linking. In the ELF object format [3], GOT (Global Offset Table) and PLT (Procedure Linkage Table) are used for the dynamic linking. In the M32R implementation, the GOT is refered by R12 relative addressing and the RELA type relocation is emoployed. Like a IA-32 implementation, the code fragment of PLT refers the GOT to determine the symbol address, because it is suitable and efficient for the M32R's cache which can be simply flushed whole caching data.

As for GDB, we enhanced it to support a new remote target `m32rsdi` to use the SDI remote connection. By using the gdb, we can do a remote debugging of the kernel in C source-level. In the latest version of `m32r-linux-gdb/ m32r-linux-insight` (v5.3), we have employed a `SDI server` that engages in accessing the JTAG port of the ICE/emmulator connected with the parallel port of the host PC. This gdb makes it possible to debug using SDI, communicating with the SDI server in background. Though the SDI server requires privileged access to use parallel port, we can use gdb in user mode.

3.4 Porting GNU C library

The GNU C library is the most fundamental library, which is necessary to execute a variety kind of application programs. So we decided to port it to implement full-featured Linux system for a study, though its footprint is too large for a tiny embedded system.

We started to port glibc-2.2.3 (v2.2.3) in eary stage of the Linux/M32R porting, it was about the same time that the kernel's scheduler began to work.

Then, the glibc for the M32R have been developed step by step;

- Check by a statically linked program, for example, `hello.c` (newlib version → glibc version).
- Build a shared library version of glibc and check by dynamically linked programs, `hello.c, busybox, etc.`
- Port the LinuxThreads library to support Pthreads (POSIX thread).

The latest version of the glibc for the M32R is glibc-2.2.5 (v2.2.5). It also supports a LinuxThreads library, that implements POSIX 1003.1c kernel threads for Linux. In this LinuxThreads library, we implemented fast user-level mutual exclusion using the Lamport's algorithm [2], because the system call implementation was quite slow due to context switching.

After the glibc porting was finished, we started to build various kind of software. But it has taken several months to implement and debug the following:

- Fixup operations of the user_copy routines in the kernel
- Resolve the relocation by a dynamic linker `ld-linux.so`
- Signal handling

Especially, the dynamic linking operation was the one of the most difficult portions in this

GNU/Linux system porting, because the dynamic linker/loader resolved global symbols and subroutine function addresses in runtime. Furthermore, the dynamic linker itself is also a shared library, so we can not debug it in C source-level. However, we debugged the linker, making use of a simulator, a SDI debugger, and all kinds of things.

3.5 Userland

For the sake of preparing software packages and making the Linux/M32R distributable, we built major software packages.

We chose the Debian GNU/Linux as a base distribution, because it is well-managed and all of the package sources are open and published. In Debian, using command programs such as `dpkg` and `apt`, it is possible to manage abundant software packages easily.

To build a binary package for the M32R, we did as the following:

1. Expand the source tree from the Debian source package (*.dsc and *.orig.tar.gz)
2. Rebuild a binary package by using a `dpkg- buildpackage` command, specifying the target architecture to `m32r` (`dpkg-buildpackage -a m32r -t m32r-linux`).

So far, more than 300 unofficial `.deb` packages have been prepared for the M32R target, including the basic commands, such as self-tools and shells, utilities, package management tools (`dpkg`, `apt`), and application programs as follows:

> adduser, anacron, apt, base-files, bash, bc, binutils, bison, boa, bsdgames, bsdutils, busybox, coreutils, cpp-3.2, cvs, debianutils, devfsd, diff, dpkg, e2fsprogs, elvis-tiny, expect, file, fileutils, findutils, flex, ftp, g++-3.2, gcc-3,2, grep, gzip, hostname, klogd, less, libc6, locales, login, lynx, m4, make, mawk, modutils, mount, nbd-client, net-tools, netbase, netkit-inetd, netkit-ping, passwd, perlbase, perl-modules, portmap, procps, rsh-client, rsh-server, samba, sash, sed, strace, sysklogd, tar, tcl8.3, tcpd, tcsh, telnet, textutils, util-linux, wu-ftpd, ...

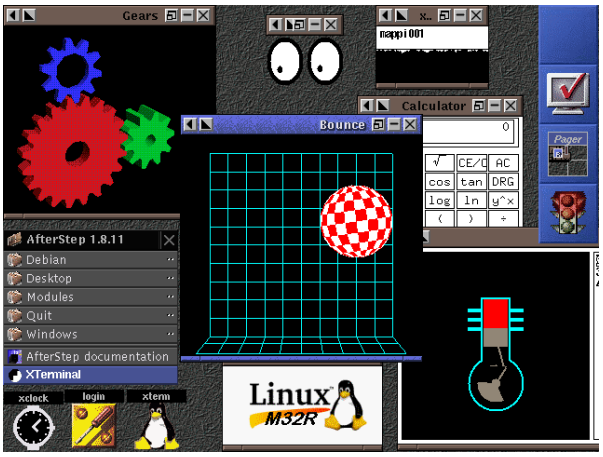


Figure 7: A snapshot of the desktop image of X; the window manager is `AfterStep`

Most of these packages were developed under the cross environment, except some software packages, such as Perl, Xserver, gcc, etc. Because they had to be configured in the target environment. Therefore, self tools were necessary in order to build packages under the self environment. Fortunately, by using `.deb` packages and `dpkg-cross` commands, the same package files can be completely shared between the self and the cross environment.

Regarding the GUI environment, we have ported some window systems (X, Microwindows, Qt-Embedded). We ported them easily using a framebuffer device. Figure 7 is a sample screen snapshot of X desktop image. `gears` and `bounce` are demonstration pro-

grams of the Mesa-3.2 3D-graphics library.

4 Evaluation

The Linux/M32R system's conformance have been checked and validated by using the LSB (Linux Standard Base) testsuites, which are open testsuites and based on the LSB Specification 1.2 [5]. In this validation, we compared the following two environments.

Linux/M32R
based on the Debian GNU/Linux (sid)
linux-2.4.19 (m32r_2_4_19_20030109)
glibc-2.2.5 (libc6_2.2.5-6.4_m32r.deb)
gcc-3.0 (self gcc; m32r-20021112)

RedHat7.3 [2]
linux-2.4.18-10 (kernel-2.4.18-10.i686.rpm)
glibc-2.2.5 (glibc-2.2.5-42.i686.rpm)
gcc-2.96

The result of validation is shown in Table 1. Judging from the result, the LSB conformance of the Linux/M32R is no less good than the RedHat Linux 7.3, because the original Debian distribution has basically good LSB conformance and quality.

5 Future work

To apply the Linux/M32R to embedded systems, it is indispensable to tune and shrink the whole system more and more. As for the kernel, particulary, tuning and improving realtime performance will be strongly required.

At present, we are porting the Linux 2.5 series kernel for the M32R in order to support the state of the art kernel features, such as O(1) scheduler, the preemptible kernel, the no-MMU support, the NUMA support, and so on.

We are also planning to utilize DMA function and internal SRAM to increase Linux system performance. And for the high-end embedded systems, we intend to continuously focus on the SMP kernel for the on-chip-multiprocessor.

6 Summary

To build a Linux platform, we have ported a GNU/Linux system to the M32R processor. In this work, a hardware/software co-design methodology was employed, using a full synthesizable M32R softmacro core to accelerate Linux/M32R development. To develop SoC in a short time, such a hardware and software co-design and co-debugging methodology will become more important hereafter.

Linux will play a great role in the field of not only PC servers but also embedded systems in the near future. Through the feasibility study, we believe that the Open Source will provide a quite large impact on developing embedded system design and development. If we have opportunity, we hope to publish the Linux/M32R and M32R GNU toolchain.

7 Acknowledgements

The authors greatly acknowledge the collaboration and valuable discussion with the M32R development team [1] and thank Takeo Takahashi, Kazuhiro Inaoka, and Takeshi Aoki for their special contributions, and we would also like to thank Dr. Toru Shimizu and Hiroyuki Kondo for their promotion of the M32R processor development project.

[2] The kernel and glibc are upgraded and different from the original Red Hat 7.3 distribution.

Section		ANSI.hdr	ANSI.os		POSIX.hdr	POSIX.os		LSB.os		Total	RedHat7.3
			F	M		F	M	F	M		Total
Total	Expect	386	1244	1244	394	1600	1600	908	908	8284	8284
	Actual	386	1244	1244	394	1600	1600	908	908	8284	8284
Succeeded		176	1112	86	207	1333	0	695	0	3609	3583
Failed		4	0	0	5	2	0	49	0	60	45
Warnings		0	12	0	0	5	0	2	0	19	18
FIP		2	0	0	2	2	0	1	0	7	7
Unresolved		0	0	0	0	0	0	5	0	5	4
Uninitiated		0	0	0	0	0	0	0	0	0	0
Unsupported		203	0	0	179	72	0	59	0	513	513
Untested		0	4	0	0	7	0	39	0	50	43
NotInUse		1	116	1158	1	179	1600	58	908	4021	4021

Key: F:function, M:macro; FIP: Further Information Provided

Table 1: LSB 1.2 testsuites result

References

[1] Satoshi Kaneko, Katsunori Sawai, Norio Masui, Koichi Ishimi, Teruyuki Itou, Masayuki Satou, Hiroyuki Kondo, Naoto Okumura, Yukari Takata, Hirokazu Takata, Mamoru Sakugawa, Takashi Higuchi, Sugako Ohtani, Kei Sakamoto, Naoshi Ishikawa, Masami Nakajima, Shunichi Iwata, Kiyoshi Hayase, Satoshi Nakano, Sachiko Nakazawa, Osamu Tomisawa, Toru Shimizu, *A 600MHz Single-Chip Multiprocessor with 4.8GB/s Internal Shared Pipelined Bus and 512kB Internal Memory*, Proceedings of 2003 International Solid-State Circuits Conference, 14.5.

[2] Laslie Lamport, *A Fast Mutual Exclusion Algorithm*, ACM Trans. on Computer System, Vol. 5, No. 1, Feb. 1987, pp. 1-11.

[3] *Executable and Linkable Format (ELF)*, `http://www.cs.northwestern.edu/~pdinda/ics-f01/doc/elf.pdf`

[4] *Debian GNU/Linux*, `http://www.debian.org/`

[5] *LSB testsuites*, `http://www.linuxbase.org/test/`, `ftp://ftp.freestandards.org/pub/lsb/test_suites/released-1.2.0/runtime/`

Linux* in a Brave New Firmware Environment

Matthew Tolentino
Intel Corporation
Enterprise Products & Services Division
matthew.e.tolentino@intel.com

Abstract

Initially included exclusively on Intel®[1] Itanium®[2] platforms, the Extensible Firmware Interface Architecture (EFI) will soon be supported on IA-32 server, workstation, and desktop systems. This paper provides insight into the design and composition of an EFI enabled IA-32 Linux kernel capable of booting on legacy free platforms. An overview of the EFI development environment is provided, including the specifications, development tools, and software development kits available for development today.

The design and prototype implementation of the kernel initialization sequence from the instantiation of the EFI enabled Linux boot loader to the login prompt is detailed with an emphasis on maintaining backward compatibility with existing legacy platforms. The legacy free VGA replacement, the Universal Graphics Adapter (UGA), is introduced in the context of Linux, including the requirements for use within the kernel. Additionally, details of a prototype implementation of the Universal Graphics Adapter Driver stack, including an EFI Byte Code Interpreter and Virtual Machine (EBC VM), are presented and analyzed. This paper concludes with a call for kernel developers to review and provide feedback on the design and implementation presented.

1 Introduction

This paper begins with an overview of the Extensible Firmware Interface and the Universal Graphics Adapter. The architecture of an EFI enabled Linux kernel is presented as well as the design and implementation details of the EFI Linux Boot Loader, kernel initialization changes, and support for the Universal Graphics Adapter. Future enabling work is outlined and conclusions presented.

2 EFI Overview

The EFI Specification defines a consistent, architecturally neutral interface between platform firmware and operating systems. Designed to address the limitations and issues inherent in legacy BIOS support for PC-AT systems, EFI provides a core set of services and protocol interfaces used to initialize platform hardware as well as common interfaces to access platform capabilities. Additionally, EFI aggregates platform configuration information that operating systems require for initialization – information that has been traditionally obtained through BIOS calls. This includes details ranging from the system memory map and the number of processors in the system to the

*Linux is a trademark of Linus Torvalds

[1]Intel is a registered trademark of the Intel Corporation

[2]Itanium is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries

parameters of the current video console. Further, the structure of EFI is modular such that as incarnations of new technologies are incorporated into systems, the interfaces to the operating system for these technologies will not require significant modifications. The system level view of the conceptual design of EFI is depicted in figure 1.

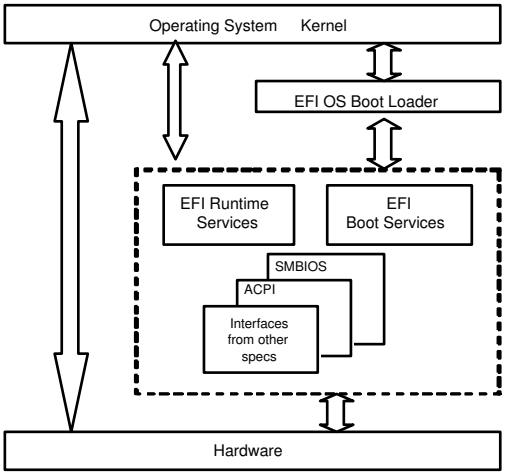


Figure 1: EFI System View

Once the system is powered on, the system firmware initializes and owns all hardware resources. Before an operating system is loaded, EFI provides a pre-boot environment and manages platform resources through the Boot Services and Runtime Services tables. These tables, encapsulated in the EFI System Table provide access to system resources. Unlike legacy BIOS which executes in 16bit real mode, EFI provides a 32bit protected mode operating environment.

2.1 EFI Boot Services

Accessed through the EFI System Table, EFI Boot Services provide platform independent functionality that is only available before an operating system is loaded. This includes services such as memory allocation routines, device access protocols, time services, and others detailed in [1]. Once control of the system is transferred to the operating system, EFI Boot Services are terminated. The OS loader is required to call ExitBootServices() as part of the transition of control to the operating system.

2.2 EFI Runtime Services and Drivers

The EFI Runtime Services provide OS neutral platform specific functionality that persists into OS runtime. One example of an EFI Runtime driver is the floating point software assist driver (fpswa.efi) on Itanium platforms discussed in [4]. Another example supported by both IA-32 and Itanium platforms is the Universal Graphics Adapter driver. EFI implementations may provide any number of EFI drivers for use during OS runtime in order to take advantage of the generic functionality. Details of these drivers are passed to the operating system via the EFI Configuration Table and the memory locations occupied are specified as Runtime Memory.

2.3 EFI Configuration Table

Leveraging existing standards and technologies, the EFI configuration table provides an interface to commonly used tables that describe platform resources. Each entry in the Configuration Table is comprised of 2 elements - a Globally Unique Identifier as defined in [9] and a pointer to the respective table. Examples of configuration table entries include ACPI tables, UGAs discovered by the firmware, and SMBIOS tables.

2.4 EFI Boot Loader

Generally, loading and initializing an operating system involves several steps. The first step is the determination of the kernel image to load into memory as well as any additional required components, such as a RAM disk, that are re-

quired. The second is the act of loading the chosen images into memory. The third and final step involves transferring control to the loaded kernel, thus enabling the kernel initialization sequence to commence.

Several boot loaders have been developed to load Linux on IA-32 platforms, including the popular lilo and grub loaders described in [7] and [8] respectively. Each of these provides the capability to boot Linux kernels from a legacy BIOS and offer varying degrees of functionality as discussed in [3]. However, these loaders do not provide the capability to boot from the EFI environment. In order to boot a legacy-free Linux kernel from EFI, an EFI native application is necessary to set up and affect the transfer of control from the firmware to the kernel.

3 UGA Overview

The UGA is a software abstraction that provides an interface for simple graphical output that does not require specific implementation knowledge of the video hardware. UGA serves as a replacement for VGA hardware and video BIOS providing graphical display capabilities in an image that can be used by both system firmware and operating systems. Unlike VGA, UGA enables several significant features including support for controlling multiple output devices and higher default screen resolutions. UGA ROMs may reside anywhere in memory, alleviating the need for static reservation of specific video RAM and ROM memory regions. Additionally, multiple UGA ROMs may be used in a single system without conflict.

A key design consideration of UGA is the processor and platform independent nature of the driver image. Compilation of driver images into EFI Byte Code (EBC) enables a single image to execute on multiple architectures. The resultant byte code image must be interpreted and executed in the context of an EBC Virtual Machine capable of translating EBC instructions to native instructions.

Because the UGA is an EBC image and operating systems are expected to use the same image as the pre-boot firmware environment, an OS resident EBC Virtual Machine is required. This Virtual Machine provides an EFI execution environment within the OS, interfaces with the console layer and the PCI subsystem of the operating system, and provides the means to interpret and execute the EBC instruction stream of the UGA. Figure 2 pictorially describes this generic design.

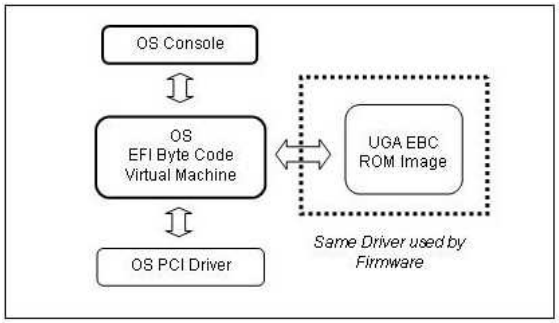


Figure 2: UGA Conceptual Design

Although UGA does provide graphics display capabilities, it is not intended to replace high-performance, operating system specific graphics drivers. Rather, the UGA is specified for use in the absence of a performance oriented graphics driver. For example, a back-end server may not require extensive graphics-oriented functionality during normal operation. There may also be cases where an installation kernel may require the user to provide a graphics driver. In these cases, a UGA driver may be used as a default console driver. Further advantages of UGA can be found in section 10 of [1].

3.1 OS Console Driver

Support for UGA at OS runtime requires an interface to the OS specific console subsystem. This driver is responsible for affecting change to the video controller via the UGA protocols outline in section 10 of [1].

3.2 OS EFI Byte Code Virtual Machine

The OS EBC Virtual Machine, as depicted in figure 2, serves two functions. The first is to provide an EFI emulation environment functionally equivalent to the EFI pre-boot environment. This enables the use of the same UGA driver as firmware through emulation of the EFI Boot Services.

The second function is to provide the facility to decode and execute EBC instructions. UGA drivers compiled into EBC can not be directly executed. Therefore, an OS present mechanism is needed translate the EBC instruction stream of the UGA image into instructions of the native processor.

3.3 OS PCI Driver Interface

In pre-boot space, the UGA driver uses the PCI I/O Protocol, a Boot Services structure, to access the video controller. Because this mechanism is no longer available when the operating system takes control, PCI access must be provided via an OS level implementation of the PCI I/O Protocol interface structure. This enables the UGA to use the standard operating system interface to access PCI space.

3.4 UGA EBC ROM

The OS support for UGA is capable of executing any UGA EBC ROM discovered by firmware that is compliant with the EFI Driver Model. In order to facilitate the transfer of control from EFI to the operating system, UGA drivers used in pre-boot space will be halted upon termination of Boot Services. The image must be re-initialization to be used during OS runtime.

4 Architecture of an EFI enabled Linux Kernel

The advent of EFI on legacy free, IA-32 systems necessitates additional support in several key areas of the Linux kernel. This section presents an overview of the architectural differences in booting the kernel from EFI versus legacy BIOS as well as the impact of the changes. The details of the changes specific to these areas are discussed in the remainder of this paper.

4.1 Key Differences

One of the key differences in booting from EFI versus legacy BIOS on IA-32 systems is the capability to launch the kernel in 32bit, protected mode. The firmware no longer invokes the boot loader in 16bit real mode. As a result, the kernel no longer needs to affect the transition to 32bit, protected mode.

Loading an EFI enabled kernel requires an EFI Linux boot loader. The loader is a native EFI application, which is responsible for obtaining platform configuration information EFI and loading the kernel into memory. Included as boot parameters, all platform configuration information is collected by the loader and passed to the kernel. This permits the loader to transfer control directly to the architecturally specific entry point of the kekernel. This difference alleviates the need for the kernel code that collects sytem information via BIOS calls.

On EFI platforms, EFI defined services are employed to invoke firmware functionality as op-

posed to legacy BIOS calls. The EFI Runtime Services provide a standard interface for accessing firmware and hardware in a platform independent manner. For example, access to the real time clock is accessed via a firmware function in the Runtime Service table as opposed to directly reading CMOS. The practice of scanning memory to find the signatures of hardware description tables is no longer necessary because tables such as ACPI are included as part of the EFI interface.

The advent of the Universal Graphics Adapter no longer requires static reservation of fixed memory regions and presents the opportunity for kernel level multi-head console support.

The design of an EFI enabled kernel requires key modifications to the following three crucial areas:

- Boot loader
- Kernel initialization sequence
- Console subsystem

4.2 Impact of EFI Kernel Changes

The changes to the kernel to support booting from EFI are relatively straightforward. The modifications to the boot loader involve extending the functionality of the IA-32 aspects of the Elilo loader to pass EFI data structures to the kernel.

The changes to the IA-32 kernel initialization sequence provide the capability to initialize kernel data structures, such as the memory manager, using EFI tables and data structures versus those obtained via BIOS calls. Primarily isolated in the architecturally specific directory of the kernel source tree, EFI support provides additional initialization options without affecting existing functionality. In other words, the changes to the kernel initialization sequence do not radically alter the architecture of the kernel.

Inclusion of UGA support necessitates several new Linux kernel drivers; however, this merely provides an additional console display option. Collectively, these drivers serve as an alternative to the legacy VGA console driver, but may also operationally coexist.

4.3 Architectural Influence

Itanium Linux kernels have included support for EFI for several years. Accordingly, much of the design discussed in this paper for IA-32 kernels has been leveraged from the Itanium port of Linux. Additionally, the prototype implementation has reused EFI related code to also support IA-32.

5 EFI Linux Boot Loader

Launching an operating system from EFI requires an EFI aware boot loader. This section provides background on the Elilo Linux Boot Loader, design considerations for improved IA-32 support, and details of the new boot parameters structure used to convey platform configuration information to the kernel.

5.1 Elilo Background

Elilo is the predominant loader used to launch Linux on Itanium platforms (on which EFI is the only pre-boot firmware solution) and has been included in numerous Itanium specific Linux distributions. Despite the pointed focus on supporting the Itanium architecture, a framework for booting self-extracting compressed IA-32 kernels has been incorporated into Elilo. This support permits booting IA-32 kernels without passing EFI information to the kernel. Instead, there is an implicit assumption that legacy mechanisms, such as BIOS calls,

still exist for traditional platform configuration information retrieval. Essentially, Elilo manually fabricates the legacy boot parameters data structure without any semblance of EFI awareness.

5.2 Elilo Design Considerations

The primary consideration for modifying the Elilo loader is to provide adequate EFI information to the kernel to ensure proper initialization and functionality in a legacy free environment. The information required by the kernel consists of:

- Kernel Location and Size
- RAM disk (initrd) Location and Size
- Kernel Command Line
- EFI Memory Map
- Console Information
- EFI System Table
- ACPI Tables
- Other EFI Configuration Table Entries (HCDP, SMBIOS, etc.)

In addition to collecting and passing salient platform configuration information to the kernel, Elilo is also responsible for setting up the environment for passing control to the kernel. Further information regarding the features and capabilities of Elilo can be found in [6].

5.3 EFI Aware Boot Parameters

The following boot parameter structure is introduced that encapsulates the information the kernel requires. The form of the structures is as follows:

```
struct ia32_boot_params {
  UINTN command_line;
  UINTN efi_systab;
  UINTN efi_memmap;
  UINTN efi_memmap_size;
  UINTN efi_memdesc_size;
  UINTN efi_memdesc_version;
  UINTN initrd_start;
  UINTN initrd_size;
  UINTN loader_addr;
  UINTN loader_size;
  UINTN kernel_start;
  UINTN kernel_size;
  struct {
    UINT16 num_cols;
    UINT16 num_rows;
    UINT16 orig_x;
    UINT16 orig_y;
  } console_info;
} boot_parameters;
```

This data structure provides all necessary information to enable kernel initialization. Note that inclusion of the EFI system table permits access to the EFI Runtime Services and Configuration Tables that contain further platform configuration information and provide access to runtime firmware functionality. For example, the location of the ACPI tables is presented to the kernel as an entry in the EFI Configuration Table.

6 EFI Kernel Initialization

The Linux kernel initialization sequence requires modification to utilize the EFI data structures that describe the platform hardware configuration. This section presents the details of the kernel modifications and contrasts these with the existing kernel initialization. The methodology for dynamic, runtime determination of the appropriate structures to use is presented as are details on the EFI support routines necessary for proper kernel initialization.

6.1 Existing EFI Kernel Initialization

The Linux kernel initialization sequence was developed to boot in 16bit real mode and obtain platform configuration information via BIOS calls. Figure 3 outlines the current initialization sequence of the Linux kernel.

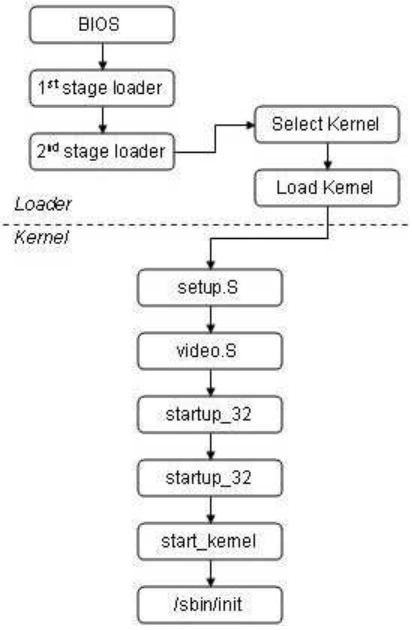


Figure 3: Existing Kernel Initialization

6.2 Kernel Initialization on EFI Platforms

Booting from EFI simplifies the kernel initialization sequence, but requires modifications to parse EFI data structures. Figure 4 depicts the proposed modified kernel initialization sequence.

In this model, the processor is already in 32 bit, protected mode when the boot loader is invoked, hence the real mode to protected mode transition code in the kernel is not necessary. Also, all platform configuration information traditionally obtained through BIOS calls is collected by the EFI Elilo loader and passed via the boot parameter structure. This alleviates the need for the code resident in setup.S,

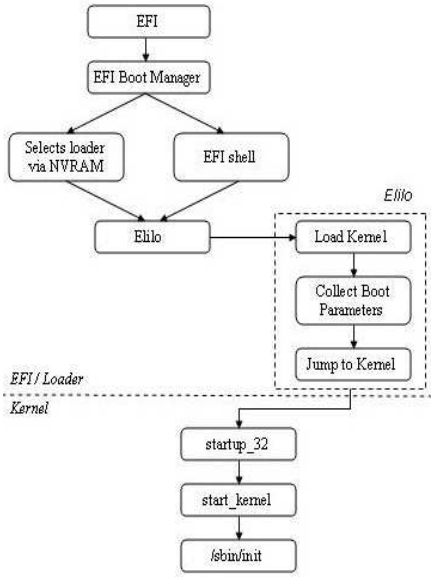


Figure 4: EFI Kernel Initialization

video.S, and bootsect.S. Consequently, control is transferred directly from the loader to the architecturally specific startup_32() routine in 32 bit, protected mode.

6.3 Dynamic Configuration Detection

Because the structure of platform configuration information on EFI platforms differs from the structure of BIOS provided information, the kernel must determine which to use to initialize kernel data structures. This determination is based on the location of the boot parameters in memory.

The boot parameters of legacy kernels are placed in a designated page in memory. Leveraging the re-use of this memory area, the boot parameters of EFI aware kernels are placed within the same page, but at a different offset. The correct kernel initialization code path to follow is determined by verifying which boot parameters have been passed to the kernel. A new global flag `efi_enabled` is set if the kernel was found to be loaded from EFI. This flag is used during the kernel initialization se-

quence to determine the appropriate data structures to use during initialization. This capability enables a single kernel image to load on platform with either legacy BIOS or EFI.

6.4 EFI Data Structure Mapping

Initially, only a limited kernel virtual address space is available through the temporary, statically initialized page global directory `swapper_pg_dir`. This maps the first eight megabytes of physical memory to kernel virtual address space starting at 3GB.

This limited mapping is not sufficient because EFI drivers and related structures may be located anywhere in memory (below 4GB), thus the kernel requires the capability to dynamically map EFI pages into kernel virtual address space for use during kernel initialization. Examples of these structures include the EFI memory map, RAM disk details, etc. Because these structures are only used during early initialization, the memory is only required temporarily. Once the kernel memory manager is initialized these memory regions will be available as for normal kernel memory allocations.

6.5 EFI Memory Map

The EFI memory map provides a snapshot of system memory usage before control is passed to the kernel. Consisting of memory descriptor entries that describe contiguous ranges of physical memory by type, attribute, and size, the EFI memory map serves as a replacement for the e820h memory map obtained via the legacy INT 15h (ax=0xe820) BIOS call. Each EFI memory map descriptor consists of the following structure:

```
struct efi_memory_desc {
  u32 type;
  u64 phys_start;
  u64 virt_start;
  u64 num_pages;
  u64 attribute;
};
```

In order to properly initialize the kernel memory manager as well as the EFI Runtime Drivers and Services, the EFI memory map is required. Several routines from the Itanium kernel have been massaged into the IA-32 kernel to support EFI memory map traversal and parsing.

6.5.1 EFI Memory Map Walking Routine

The prototype for the EFI memory map descriptor traversal routine is:

```
void efi_memmap_walk(
       efi_freemem_callback_t
       callback, void *arg);
```

This routine employs a callback mechanism used to discern further information about memory regions during memory map traversal. For example, used in concert with the find_max_pfn() callback routine, the kernel is capable of discovering the maximum page frame number.

6.5.2 Other Memory Map Related Routines

Several additional routines have been added to accommodate initialization using the EFI memory map. The following routine is used to determine the memory type of the region described by a given memory descriptor.

```
static int is_available_memory(struct
                efi_memory_desc *md);
```

Differentiation between memory descriptor types is necessary to determine usable regions

by the kernel. The following types constitute memory regions available for kernel use.

- EFI_LOADER_CODE
- EFI_LOADER_DATA
- EFI_BOOT_SERVICES_CODE
- EFI_BOOT_SERVICES_DATA
- EFI_CONVENTIONAL_MEMORY

The following two functions are convenience functions used to determine the type and attributes of memory regions in the EFI memory map given an address.

```
u32 efi_mem_type(unsigned long
                 phys_addr);
u64 efi_mem_attributes(unsigned long
                       phys_addr);
```

6.6 Persistent EFI Drivers and Services

Unlike EFI Boot Services, which are terminated when control is transitioned to the kernel, several EFI drivers and services are available for use during OS runtime. The following sections detail these services and describe the support framework for maintaining access to these drivers and services as well as managing the mappings into kernel virtual address space.

6.6.1 EFI Runtime Drivers and Services

The IA-32 Linux kernel requires the capability to call the EFI Runtime Drivers and Services to take advantage of platform specific functionality, such as access to the real time clock (RTC), persistent EFI NVRAM environment variables, and the capability to reset the system. The following structure, included in the include/linux/efi.h header constitutes the kernel data structure through which calls to the EFI Runtime Services and Drivers are managed.

```
struct efi_runtime_services {
  struct efi_table_hdr hdr;
  unsigned long get_time;
  unsigned long set_time;
  unsigned long get_wakeup_time;
  unsigned long set_wakeup_time;
  unsigned long
    set_virtual_address_map;
  unsigned long convert_pointer;
  unsigned long get_variable;
  unsigned long get_next_variable;
  unsigned long set_variable;
  unsigned long
    get_next_high_mono_count;
  unsigned long reset_system;
};
```

Additional EFI Runtime Drivers may be employed to exploit OS independent functionality. For example, the floating point software assist driver (fpswa.efi) on Itanium platforms discussed in [4] and the Universal Graphics Adapter are EFI compliant runtime drivers used on both IA-32 and Itanium platforms. Details of runtime drivers are passed to the operating system via the EFI Configuration Table.

The memory occupied by runtime drivers is reserved by the kernel to prevent the memory manager from viewing the area as free memory. Additionally, these memory regions are mapped into kernel virtual address space to avoid the overhead of invoking these services in flat, physical addressing mode. The mapping of runtime services and drivers into kernel virtual address space is provided by the following routine:

```
void efi_enter_virtual_mode(void);
```

This routine walks the EFI memory map and maps all regions described by memory descriptors of the following type:

- RunTimeServicesCode
- RunTimeServicesData

Once mapped, the VirtualStart field of the memory descriptor is updated with the virtual addressed returned by the mapping function, `ioremap()`. After all memory descriptors have been updated with virtual addresses, the EFI Runtime routine SetVirtualAddressMap is invoked and passed the updated EFI memory map. SetVirtualAddressMap must be called in physical mode requiring the capability to transition to physical mode before invocation and return. This function updates all EFI runtime images with virtual addresses and completes all necessary fix-ups to enable EFI Runtime Services to be called in virtual mode. Should the call to SetVirtualAddressMap fail to complete or return an error status code, the kernel will panic.

During the mapping process, each memory descriptor is also checked to determine if the address for the EFI system table is included in the range. Once SetVirtualAddressMap returns, the EFI System Table pointer is updated with the newly assigned kernel virtual address as are as the kernel's EFI data structures.

Because the ioremap capability is not available until the kernel memory manager is initialized the efi_enter_virtual_mode() function must be called after the mem_init() and kmem_cache_sizes_init() functions in start_kernel().

6.7 ACPI Initialization

In order to support device discovery and power management, kernel support for ACPI is required. The ACPI tables contain vital platform configuration information necessary for proper kernel initialization, such as the number of processors in a system, PCI interrupt routing, etc. Inclusion of ACPI support in kernel builds requires the kernel configuration flag CONFIG_ACPI_EFI to be defined in order to enable inspection of the kernel's EFI data structure for the ACPI tables. An additional requirement for proper ACPI table discovery is to update the following ACPI initialization function:

```
unsigned long __init
         acpi_find_rsdp(void);
```

On legacy systems this function scans memory looking for the Root System Description Pointer (RSDP). On EFI based systems, the address of the ACPI tables is included in the EFI Configuration Table. This function has been updated to examine the EFI Configuration Table for address of the RSDP.

7 Kernel UGA Architecture

The Linux kernel requires UGA support in order to provide console display functionality on legacy free platforms. Because the UGA is compiled into EFI Byte Code and is programmatically designed for execution in the EFI environment, kernel level support for UGA requires the addition of new driver functionality. These drivers, also pictorially described in figure 5, include:

- EFI Boot Service Emulation Driver
- EBC VM & Interpreter Driver
- UGA Console Driver

Each of these components may be built as kernel driver modules, although all are required for proper console display. Because these drivers have minimal architectural dependencies, all EBC drivers are included in a new `drivers/ebc` directory of the kernel tree.

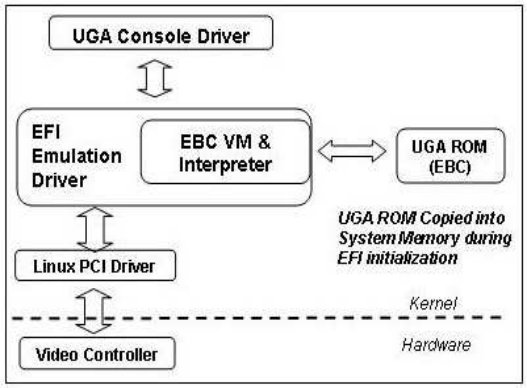


Figure 5: Kernel UGA Architecture

7.1 EFI Boot Services Emulation Driver

As is the case with all EFI drivers, the UGA requires the EFI system table, as well as a pointer to itself, as initialization parameters. All requests from the UGA for services to allocate memory, bind to the respective controller, and other routines are based on EFI Boot Services functionality. Because EFI Boot Services are terminated when ExitBootServices() is invoked, a minimal framework must be fabricated within the kernel in order to simulate EFI Boot Services. The Boot Services Emulation Driver fulfills this requirement by providing kernel implementations of the Boot Service routines. For example, the EFI memory allocation routine AllocatePages() is implemented with the kernel's kmalloc() service.

7.2 EBC VM & Interpreter Driver

A kernel implementation of an EBC Virtual Machine and interpreter is also required. This driver provides the framework and execution engine to:

- Interpret and execute all EBC instructions
- Provide an interface to handle calls between a native environment and the VM
- Provide an interface to fix-up calls to EBC driver images.

Details on the full EBC instruction set can be found in Chapter 19 of [1]. However, details on two instructions that require particular attention are included in the following sections.

7.2.1 BREAK 5 Instruction

The BREAK 5 instruction enables the transition of the instruction stream from native to EBC instructions. This technique is referred to as thunking in [1]. During compilation of the UGA ROM image, the EBC compiler inserts BREAK 5 instructions in the initialization instruction sequence to handle the transition for each of the image's protocol entry points. Functionally, the BREAK 5 instruction introduces a level of indirection, as the VM/Interpreter must replace the address of every entry point in an image with the address of a thunk. The thunk is an area in memory that includes the hexadecimal encoding of native instructions to transition control to the interpreter with an entry point of the image. This enables the seamless interpretation and execution of the EBC instruction stream via the UGA protocol interfaces. Figure 6 depicts the logical conceptual calling mechanism and an example implementation used to invoke the VM and interpreter at the appropriate UGA protocol entry point.

7.2.2 CALLEX Instruction

The CALLEX instruction provides the mechanism to facilitate calls outside the context of the EBC instruction stream or VM. For example during compilation, when the compiler observes a call to a Boot Service function, it inserts a CALLEX instruction, such that the tran-

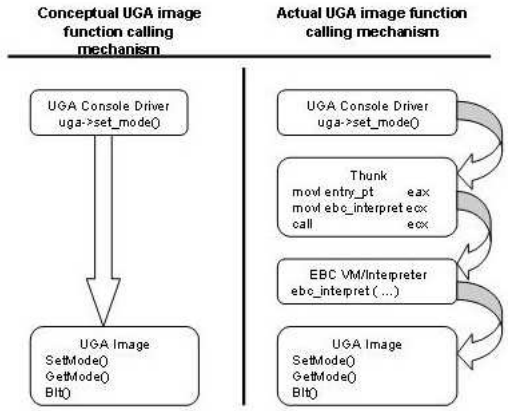


Figure 6: Thunking Mechanism

sition of the stack, IP, return value, etc. are handled gracefully.

7.2.3 Architectural Dependencies

This driver supports both the IA-64 and IA-32 architectures. Architecturally specific, required features are included through inclusion of appropriate header files. For example, the inline assembly routines used to manipulate the stack pointer and obtain the entry point from a processor register are included in architecturally specific headers.

7.3 UGA Console Driver

The UGA console driver provides an interface between the Linux kernel console subsystem and the UGA ROM. Because the UGA is conceptually encapsulated (i.e. registered) with the EFI Boot Services Emulation driver, the console driver will affect changes to the display through the use of the UGA_IO_PROTOCOL and UGA_DRAW_PROTOCOL protocols. As a result, the console driver must obtain the UGA protocol interface structures from the EFI Boot Services Emulation Driver.

Unlike VGA, multiple UGAs may be used in a single system. Therefore, the UGA console driver must maintain data structures to handle multiple UGA simultaneously. For early boot message display, the console driver will initially employ the UGA used by the firmware, the details of which are included in the boot parameters.

Similar to the VGA console driver, the UGA console driver supports the generic console routines through the `consw` structure. Once the console driver obtains the protocol interfaces from the EFI Emulation Driver and the driver is initialized the UGA may be used for console display.

7.3.1 Firmware to OS Handoff Structure Parsing

Details of the UGAs discovered during firmware initialization are passed via the EFI Configuration Table. Each entry in the Configuration Table consists of a GUID and pointer pair. During kernel initialization, the pointer in the Configuration Table is stored in the kernel's efi structure. In the case of UGA, this points to the firmware-to-OS handoff header, which is of the following form:

```
struct efi_os_handoff_hdr {
  u32 version;
  u32 hdr_size;
  u32 entry_size;
  u32 num_entries;
};
```

This header is immediately followed by the driver handoff entries for all UGAs discovered by the firmware. Each entry driver handoff structure is of the following form:

```
struct efi_driver_handoff {
  int type;
  struct efi_dev_path *dev_path;
```

```
  void *pci_rom;
  u64 pci_rom_size;
};
```

The UGA Console driver parses each of these driver handoff structures to obtain device information as well as the address of the PCI Expansion ROM that has been copied into memory by the firmware. The PCI Expansion ROM is then parsed to locate the EBC UGA image. Once the UGA image is located, the console driver updates its internal data structures with the image address and device information. Figure 7 shows the organization of the information passed from the firmware to the kernel.

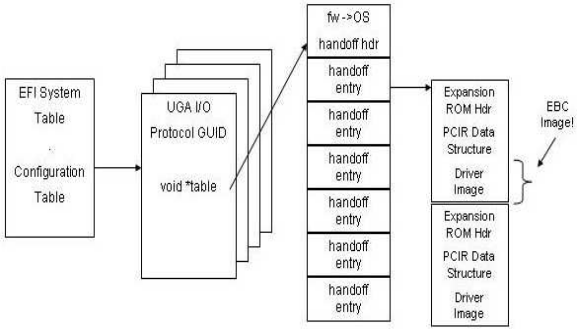


Figure 7: UGA Configuration Table Parsing

Details on the header information and layout of the ROM image may be found in [2].

7.3.2 Executable Image Parsing

After the EBC image is located, the console driver initializes a list of UGAs in the system with device information as well as pointers to the actual PE32+ UGA EBC image in memory. In order to use the driver, the image must be effectively loaded. Although the image is already in memory, the image must be parsed to correctly identify the entry point and image specific information. Once the entry point is obtained, the EBC VM and Interpreter is invoked through the EFI Boot Services Emulation Driver and the UGA is initialized. The effective loading and parsing of PE32+ EFI images requires PE header structure information be included in the kernel.

8 Future Challenges

The possibility exists to offload kernel decompression to the Elilo loader. This loader extension would provide the capability to boot both compressed and uncompressed IA-32 kernel images, similar to existing functionality on Itanium platforms. This is an area that is still under investigation.

Additional work to enable the use of the more advanced features of UGA is ongoing. A prototype implementation has been developed that supports the UGA functionality discussed. However, this support is currently limited to a single console and does not account for possible UGA related changes to XFree86. Further, the current implementation does not address issues involved in supporting multi-head console within the kernel through the use of UGA.

9 Conclusions

EFI provides a standard interface to platform firmware from which to launch operating systems on legacy free platforms. The IA-32 Linux kernel requires several key changes to initialize properly on EFI supported platforms. The first change involves the inclusion of additional IA-32 support in the Elilo Linux boot loader. Modification of the kernel initialization sequence to enable the use of EFI constructs such as the EFI memory map and EFI Runtime Drivers and Services are required. The kernel also requires several additional driver modules in order to use the legacy-free UGA for console display.

As platform hardware evolves and legacy hard-

ware and legacy BIOS support is phased out, operating systems must adapt. Inclusion of the capabilities discussed in this paper constitutes a significant step towards enabling Linux to boot on EFI based, legacy free platforms.

10 Acknowledgements

Special thanks to Mark Doran, Andrew Fish, Harry Hsiung, Mike Kinney, and Greg McGrath of the Intel EFI team for their contributions to this paper and always helpful advice. Thanks to Steve Carbonari, Mark Gross, Tony Luck, Asit Mallick, Mark Gross, Mohan Kumar, Sunil Saxena, and Rohit Seth for reviewing this paper.

Special credit is due to Mark Gross for his efforts spent working on the design and implementation of the prototype EFI enabled kernel.

References

[1] *EFI Specification Version 1.1, Intel Corporation, 2003* `http://developer.intel.com/technology/efi`

[2] *PCI Local Bus Specification,* Revision 2.3, PCI Special Interest Group, Hillsboro, OR `http://www.pcisig.org/specifications`

[3] Werner Almesberger, *Booting Linux: The History and the Future* Proceedings of the Ottawa Linux Symposium 2000, July 2000

[4] David Mosberger and Stephane Eranian, *ia-64 Linux Kernel, Design and Implementation.* Prentice Hall, NJ 2002.

[5] *ACPI Specification* Version 2.0b `http://www.acpi.info/spec.htm`

[6] David Mosberger and Stephane Eranian, *elilo-3.3a Documentation.* 2000. `ftp://ftp.hpl.hp.com/pub/linux-ia64/elilo-3.3a.tar.gz`

[7] Werner Almesberger, *Lilo Technical Overview*, `ftp://metalab.unc.edu/pub/Linux/system/boot/lilo`

[8] Eric Boleyn, et al. *GNU Grub* `http://www.gnu.org/software/grub/grub.html`

[9] *Wired for Management Baseline,* Intel Corporation, 1998. `http://developer.intel.com/ial/WfM/wfmspecs.htm`

Implementing the SMIL Specification

Malcolm Tredinnick
CommSecure
malcolm@commsecure.com.au

Abstract

Synchronized Multimedia Integration Language (SMIL) is a W3C recommendation for encoding multimedia presentations. It provides presentational control over not just the spatial layout of the document, but also the relationship between elements over time. At the present time there does not appear to be a high quality Open Source implementation of the SMIL 2.0 specification available. This paper describes one attempt at an implementation. Some ideas about where future software development could take this implementation to fulfill the requirements of other projects are also mentioned.

1 A Potted History Of SMIL

1.1 SMIL 1.0 — June 1998

In mid-1998, the W3C promoted the SMIL 1.0 specification [4] to Recommendation status. The goals of this recommendation were (from its abstract)

1. to describe the temporal behavior of a presentation,
2. describe the layout of the presentation on a screen, and
3. associate hyperlinks with media objects.

Here, media objects are either continuous media, such as video or audio presentations, or discrete media such as text blocks and images.

The SMIL 1.0 specification was not overly complex. It was not short (about 30 pages when printed out), but that was because it introduced a number of elements with specific semantics that needed to be explained clearly. Implementations appeared, but SMIL did not set the world on fire. This was possibly because although it fulfilled the design requirements mentioned above, that was *all* it did. A presentation needed to be separated out into individual media elements in practice and leveraging existing content was difficult[1].

1.2 SMIL 2.0 — August 2001

Roughly three years later the SMIL 2.0 specification reached Recommendation status [5]. Superficially, this specification looked vastly different from the SMIL 1.0 version. It was also much larger (the 1.0 document was around 150 KB in size, 2.0 was 3.3 MB). However, upon closer inspection, it became apparent that the changes fell broadly into a only a few categories.

1. Additional markup for transitions between media components, extra layout

[1] This is just the present author's observation. The real reason may be that SMIL 1.0 was ahead of its time; a solution in search of a problem.

possibilities, content control (so that presentations on PDAs and large monitors could be driven from the same source) and extra events for controlling the presentation.

2. Improved accessibility features (for example, offering a choice between closed captioning and audio descriptions).

3. A modular design so that SMIL modules could be reused in conjunction with other XML modules. This also provides a way to allow small-footprint implementations to implement the commonly used portions of the language and omit the more complex parts whilst still being able to process documents intended for more featureful implementations.

SMIL 2.0 provided backwards-compatibility with SMIL 1.0 documents and precisely laid out how to process older documents with respect to the newer features. Content written for the older specification did not become obsolete. In view of this, an implementation of "SMIL" need only focus on the 2.0 specification since 1.0 compatibility comes for free[2].

Today, SMIL is *still* not setting the world on fire. It is slowly gaining visibility, though, and is becoming incorporated into a number of auxiliary specifications which is forcing it into peoples' consciousness. We consider a few of these uses later in this paper.

2 Building A SMIL Library

Around June of 2002, the present author stumbled across SMIL whilst looking for something entirely different and started to read the specification, as one does in those circumstances. In November, a rough design for a library to parse SMIL documents and somehow present them was sketched out. Over the following weekends and the odd evening or two, code started to come together to the point where at the present time[3] a complete implementation of the specification is within sight. The implementation has been done without reference to third-party products, since the whole idea was that it was meant to be a fun project and a test of how difficult it might be to implement from the specification alone.

Existing SMIL implementations all appear to be designed along the lines of building a complete presentation application for SMIL documents. The library being considered in this paper, `libsmil`, has different goals.

The `libsmil` library parses a SMIL document and extracts the data into a format that can easily be used by a client application to make the presentation. It is then up to the client application to start and stop the presentation of the various media objects in windows as directed by the library (see Figure 1).

In essence, `libsmil` manages a bunch of data structures containing information about the presentation for the client. From time to time, the library will poke the client and tell it to start or stop presenting on of the media objects in a particular location. Similarly, the client will call the library whenever the user (or other external stimulus) triggers an event that may influence the progress of the presentation. This encompasses actions like clicking on a "stop" button or following a hyperlink to some other location in the document. The library itself is agnostic about the means used to present the information by the client. This should provide the opportunity to reuse this library in a variety of different situations, since it has very few dependencies on its own and

[2] Well, it comes after a lot of work, in practice. But you still only need to implement a single specification.

[3] As this paper is being written in April 2003

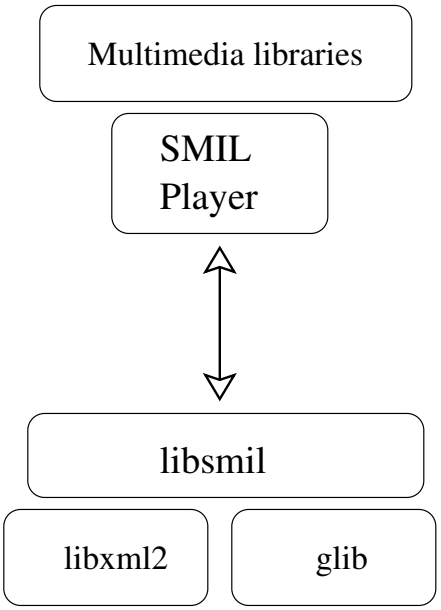


Figure 1: Basic `libsmil` architecture

places few restrictions on the client's behavior and operation.

3 An Interaction With A Presentation Client

In order to illustrate the features provided by `libsmil`, we consider a typical series of calls between the library and a client application. For our purposes here, it suffices to understand that a standalone SMIL document consists of a `head` and a `body` portion, much as XHTML does. The head of the document contains data about the presentation and the body contains the presentation itself. The explanation in this section necessarily skips over some of the minor steps, but we will endeavor to touch on all the highlights.

3.1 Pre-Processing

After initializing `libsmil`, the client reads in the document and creates a DOM representation using the GNOME XML library `libxml2`. This responsibility is given to the client since an internal representation of the data to be presented may already exist or is being created on the fly. This would be typical, for example, in the scenario where the client is using SMIL as part of a larger language, such as XHTML+SMIL (see [8]). For more information on this type of usage, see section 5.

The DOM is then passed to `libsmil` for initial, pre-presentation processing. Initially, this involves parsing all of the contents of the `head` element

The library extracts from the document the regions that are going to be used in the presentation and what their dimensions are. A SMIL document can have a number of top-level regions (essentially, separate windows for a GUI client) and each top-level region can contain any number of named regions within it. These internal regions can be used to display videos, captions, text overlays on images or videos or whatever the content author desire. The regions may overlap and can have a Z-ordering applied to them so that one region always appears on top of another region. Each region can also have attributes specified to indicate how media within it is treated in terms of clipping, scrolling or scaling.

A SMIL document can specify any number of customized tests which are used in the body of the document. These are a sequence of named data elements which may contain arbitrary values. The values can be set by the user—through some interface on the client—or via a URI. Some of the tests are marked as intended for user setting, whereas others are nominated as "hidden," meaning that although an advanced client may permit them to be set, they will typically take their value either from a default setting or by retrieval from some remote location. So `libsmil` extracts out all the custom tests, sets up their default values

and types (visible or hidden) and puts them into a data structure for the client to present to the user upon request.

In the final phase of the pre-presentation scan, the library makes a pass over the `body` element of the document and tries to extract two more pieces of data.

1. A list of all the media types that are required to make the presentation, and
2. a graph of all the timing and synchronization dependencies between the elements in the body of the document.

The first item here is straightforward: every media object is marked with one of a limited number of tags and includes a compulsory attribute indicating the type of the media. Building up a list of required media types and linking the types to the URIs for the contents is just a matter of parsing these elements. This list is given back to the client, who can use it to pre-load the appropriate presentation libraries or prepare itself (and the user) for the fact that some items will not be presentable due to the lack of an appropriate output method.

The timing and synchronization data is what makes implementing the SMIL specification interesting and challenging. It is non-trivial! Rather than interrupt the flow of this section, we will just take it as given that some magic happens and a timing graph is constructed which `libsmil` will use during the presentation to control events. In section 4 we will come back to this problem, since it lies at the heart of the benefits SMIL can provide to external applications. The timing graph that is constructed here is only of use to the library and will remain hidden from the client. That is a fair division of knowledge: most of the complexity of a presentation lies in getting the timing issues correct and that is ultimately what the client is relying on the library for.

After collecting all this data, control is returned back to the client. The client can now query the library to extract a list of top-level regions that it will need to provide, or to find out which media formats are going to be required. The client then initializes any extra libraries it requires to make the presentation and registers a single callback with `libsmil`.

3.2 Running The Presentation

After all the data collection in the previous section, running a presentation is a reasonably simple process, with a couple of exceptions mentioned below.

The client program calls `smil_run_presentation()` to start the action[4]. If the user does nothing, the majority of the presentation will consist of `libsmil` calling the function that the client had registered at the end of the last section. This function contains instructions to start or stop the presentation of a particular media element in a particular region. For continuous media, a "start" instruction may also contain an offset at which to begin playing the fragment, a repeat count and possibly a time multiplier. This last option is an advanced feature in SMIL that permits authors to alter the natural duration of a media fragment by, in effect, accelerating or decelerating time as seen by that item only. Not all players will be able to support this feature and presentation authors need to allow for that possibility.

From time to time, the user may trigger an event at the user-interface. This could be something like clicking on a hyperlink, hitting a hotkey, or closing a window on the desktop. The client will pass this event, along with any asso-

[4]This must be done in a thread or separate process, since at the same time as `libsmil` is creating events, the client must be responding to user interaction *and* presenting the media to the user.

ciated contextual information such as the name of the link or region that was clicked, back to the library. These events will be used by the library to control the future of the presentation, but from the client's point of view they simply result in further calls to the registered callback for starting and stopping presentations.

The only slightly complex feature when running a presentation is how to present features from the Animation modules. These SMIL modules provide a means to change the value of attributes on elements over time. The values can change linearly and non-linearly over time. They can jump to discrete values at different moments. The value against time graph can even be specified using Bezier splines.

The Animation modules are most commonly used when SMIL is incorporated into another language profile. So the attributes being changed might be things like the display style property on an XHTML element, or the length of a line in SVG. Displaying the host language elements correctly is clearly the job of the client. However, managing the complex value against time relationships is something that the SMIL library should be doing, since it possesses complete knowledge of the required algorithms.

Currently, when an `animation` element is begun, `libsmil` calls the client as normal with a start instruction and supplies the appropriate initial values for the attribute(s) in question. Subsequently, whenever the client is ready to redraw the element being animated, it calls `smil_update_animation(...)` with the given element / attribute pair identifier and retrieves the new value for the attribute that is being animated. This is a rare case of the client pulling the information it needs, rather than waiting to be notified of an update. Due to the small number of users of `libsmil` at the time of writing, it is unclear if this method of running an animation is the best. As more experienced is gathered, this interface may change.

The other exception to the normal run algorithm outlined above is precipitated by the Transition Effects module. This module provides a number of transitions that can be used when moving between media objects within the same region. They include various fades and screen wipes. A SMIL implementation (a player) supporting the Transition Effects module is required to support at least four transitions. However, the specification outlines over 100 transitions, with a fallback algorithm for when a transition is specified that the player cannot handle.

In the present implementation, a client is required to know the required behavior for each of the transitions it supports. For example, the library might call the client with an instruction to begin the horizontal windshieldWipe with a duration of three seconds. How the client implements this wipe is left up to the client. It is possible (but not compulsory) for the client to tell `libsmil` which wipes it can handle so that the library can implement the fallback algorithm on behalf of the client. This way of using the library is consistent with keeping as many SMIL-related decisions in the library's domain as possible.

A client can take this to extremes and tell the library that it can do no transitions at all—that is, it does not support the Transition Effects module—in which case `libsmil` will simply optimize away the calls to begin and end transitions. This would be appropriate, for example, in a client that is designed to present audio and braille information for blind users; implementing wipes makes no sense in that situation.

This implementation is not perfect—it is one case where the client is required to have knowledge of the SMIL specification in some de-

tail. The alternative, however, is for the library to pass back a bitmap of how a region should look as the wipe progresses. The problem with this is that it would involve putting knowledge about presentational techniques into a library that is otherwise devoid of such information. It would also remove some possibilities for the client to optimize or improve the wipe according to circumstances. For example, the same wipe performed on a region that is 300 by 200 pixels may look different on a PDA than on a screen capable of much higher resolutions and with more CPU power available.

In practice (limited as it may be), this implementation appears to work. The minimal four transitions required by the module are trivial to implement in a client. Some multimedia libraries, such as the `gstreamer` library from the GNOME project, also provide ways of doing standard wipes (the SMIL specification uses a number of wipes already specified in standards for the television and motion picture industries). Therefore, requiring clients to implement their own versions of each wipe may not be particularly onerous. Once again, continued use of `libsmil` should provide better indications on this front.

4 The Timing And Synchronization Module

As mentioned in section 3.1, the heart of `libsmil` is the timing and synchronization code. This is the longest and possibly the most complex portion of the specification. The `libsmil` implementation has been rewritten three times so far and although it is approaching completeness, testing continues in an effort to try and gain some confidence that the behavior is correct in all circumstances. A fourth (or further) rewrite is not out of the question as this code is required to be both easy (or even just *possible*) to maintain and fairly fast, since it is where most of the library's work is done while a presentation is running.

Without going into too many details, SMIL contains three container elements for holding content that is temporally related. The `seq` ("sequential") element contains items that are presented one after the other in the order they are given in the document. The `par` ("parallel") element contains items that are to be presented in parallel, subject to time constraints on their begin and end times and lengths. Finally, the `excl` ("exclusive") element wraps content of which only one item can be playing at any given time, although the order is not important. One might use `excl` to present a number of video clips from amongst which the user can select arbitrarily with each new clip replacing the one that is currently playing.

Within each of these containers, each element can have a begin time (absolutely specified or relative to the start of the container), an end time, a duration and a repeat count[5]. Further to this, the containing element (the `seq`, `par`, or `excl` element) can also have begin, end, duration, and repeat counts attached to it. After all, this container may reside inside another temporal container and so on. In fact, this last possibility is universally true. All elements are assigned a behavior that is equivalent to one of the containers. By default, all elements, including and, in particular, `body`, act as `seq` elements. So everything in the document is played in order from beginning to end with well-defined semantics as it concerns scheduling.

By and large, scheduling the beginning and ends of elements inside a `seq` or `excl` element is fairly straightforward. At least, these cases certainly fall out easily after the logic for

[5] There are some restrictions on these values depending on the type of containing container, but we will treat them as all roughly the same here; no real confusion should result from doing so.

```
<par>
   <video id="vid"
          begin="-5s"
          dur="10s"
          src="movie.mpg" />
   <audio begin="vid.begin+2s"
          dur="8s"
          src="sound.au" />
</par>
```

Figure 2: A sample `par` container holding video and audio elements.

```
<par>
  <img id="foo"
       begin="0; bar.begin+2s"
       dur="3s" .../>
  <img id="bar"
       begin="foo.begin+2s"
       dur="3s" .../>
</par>
```

Figure 3: A ping-pong effect.

handling the contents of a `par` is implemented.

The difficulty for a `par` lies in the fact that elements may have multiple begin times (and multiple end time, but we shall omit a detailed discussion of those here). These times may also be relative to the begin or end times of other elements, even those within the same `par` container. Further, an element's starting time may be before the starting time of its containing element. The contained element will not begin before its parent, so the net effect is that when it does begin, it will appear to have already been playing for some time.

An example may make this clearer (see Figure 2). In this example, when the containing `par` element begins, the video will begin playing five seconds into its length. Thus it will appear as though it started from the beginning five seconds *before* the `par` element started. The video will then play for five seconds, since its total duration is ten seconds and it effectively started at minus five seconds. The audio element in this container begins two seconds after the video begins. This is effectively at minus three seconds, from the point of view of the `par` container. Thus the audio will really start playing three seconds in from its beginning and will run for a further five seconds, ending at the same time as the video element. This is a very typical example of how audio and video sequences can be synchronized despite having possibly come from different sources.

In this example, we saw a case of an element that has a definite starting time (the `video` element) coupled with one that has an indefinite starting time (the `audio` element whose start time depends upon the `video` element). It is those elements with indefinite starting (and ending) times that can make life difficult for the implementation. In some cases, such as the above example, the effective start time of the element can be easily determined, even as early as the pre-presentation pass, since its only dependency is on something with a definite starting time. However, the start time could be dependent upon a something such as a button click event being sent, which is unresolvable until presentation time.

One significant test of any synchronization implementation is something like the code fragment in Figure 3.

The effect here is that the image labeled *foo* is displayed at time zero and lasts for three seconds, image *bar* is displayed initially two seconds into the element and lasts for three seconds. This triggers *foo* to be displayed again at four seconds into the element's period (the start time of *bar* plus two more seconds) and so on, back and forth between the two image. The duration of three seconds here is relatively meaningless: it could be anything longer than

two seconds and the same effect would occur.

Implementing the code to process this fragment requires a little planning. It appears that all of the indefinite time periods (the `bar.begin+2s` parts) can be resolved, since everything can be traced back to a dependency on the instance of *foo* that starts at time zero. However, there is no definite end to this element (although there may be one hidden in the omitted portion of the above example). It would obviously be a mistake to try and resolve all of the image start times out to infinity. Instead, `libsmil` detects that there is a loop in the dependency chain and stops resolving at that point. It then becomes a matter of resolving portions of the infinite dependency chain as required when the container element is being presented.

The examples that we have considered in this section are indicative of the problems to be solved by the timing and synchronization code. A glance at [5] shows that there are many more cases to consider, but the logic is fairly well explained in the specification. The problem is that there is just a lot of it and the interactions between cases is complex.

In theory, getting the timing information correct is just a problem in directed graph theory. In practice, it is a maze of twisty passages, all alike, and somewhat difficult to navigate correctly.

5 Using `libsmil` To Extend Other Languages

In section 1.2 it was explained that one of the changes between versions 1.0 and 2.0 of the SMIL specification was that modularity was introduced. This was done along the same lines as the XHTML modular design and for the same reasons—it enables the language to be extended or for portions to be lifted and dropped into another language profile in order to extend the latter. It is natural, therefore, to try and design `libsmil` in such a fashion that it can assist with presenting these extension languages.

On the whole, this has not been too difficult. The languages that one would choose to extend with SMIL are things like XHTML, SVG and MathML—languages which normally are static presentations once rendered. SMIL adds the ability to change attribute values over time, particularly via the Timing and Synchronization modules and the Animation modules.

Using `libsmil` to render a document in, say, SVG+SMIL is very similar to rendering a pure SMIL document. The library does a prepresentation pass over the document to build up information about the nodes it will influence and to create a time graph, just as in the standalone case. Once this pass is finished, the client renders the document using the initial values for all attributes. It then calls `smil_run_presentation()` and waits for the registered callback to be triggered with the usual instructions about starting or stopping some action. In this case, these actions will typically be things like changing the value of an attribute, rather than playing a media item.

The main difference from the standalone case that will arise is when the document being displayed is changed by some event outside the control of `libsmil`. In the standalone case, all document navigation is controlled by `libsmil`; in the extension case, the SMIL library does not have the knowledge of how navigation works in the external (hosting) language, so that is up to the client to manage. Therefore, the client may from time to time call `smil_new_document()` to load a completely different document or `smil_jump_to_xpath()` to move to a location

within the current document.

For client applications that already have decent access to their documents' parsed object model, adding support for SMIL's temporal activity appears not to be too difficult.

6 Applications Of SMIL

In case the reader is still wondering about the practical benefits of SMIL, which have probably not been made clear in the previous sections, here are a small number of typical use cases.

Recorded presentations It is possible to coordinate the automatic presentation of a conference speaker's slides with the audio recording of the their talk. The slides will progress at the right moment. Extra navigation possibilities for both the audio and visual portions of the talk can be presented as well.

Digital Talking Books SMIL is already part of the DAISY 2 [2] and ANSI/NISO Z39.86 [1] talking book standards—the latter standard being also known as DAISY 3. DAISY 2 requires SMIL 1.0 support, while Z39.86 requires a minimal SMIL 2.0 implementation. These two standards provide visually impaired people and people with reading difficulties a means to access literature that would otherwise be closed to them.

Captioning for video formats Many digital movie and video formats do not contain subtitles as sideband information. Sometimes, subtitles are provided, but not for the required language. The ability to synchronize a video presentation with arbitrary textual captions will provide a benefit both to people with hearing difficulties and to those watching a presentation given in a foreign language.

Educational presentations As authoring tools become available, pulling together disparate sources into a coherent presentation should become relatively straightforward. This will permit educators to begin to build up a library of coordinated presentations using information that currently might be scattered all over the Web. It was not mentioned earlier in this paper, but the media objects displayed by SMIL can be retrieved from remote URLs as well as local files. Also, SMIL provides a mechanism for pre-fetching any content that may take time to download so as not to hold up later portions of the presentation.

Kiosk and conference display front-ends SMIL provides a simple way to create a menu-based presentation. It can also revert to a standard looping presentation once the requested video or audio has completed. This makes it ideal for writing control documents for video kiosks or product displays at conventions.

7 Future Work

Development on `libsmil` is focused on creating a complete implementation of the specification. Simultaneously, some demonstration applications and a small presentation program are being written to show off the library's features.

Following on from that work, a number of obvious "next steps" present themselves. The following list is in no particular order, but all are achievable tasklets.

1. Implement the Timed Text specification that is currently being developed by the

W3C [7]. This will allow for scrolling captions and easier synchronization of captions with audio and video.

2. Implement a digital talking book player. Currently, no Open Source implementation of the DTB standards is available. With proprietary software for presenting these books already available, it is important to have a source code available implementation around to prevent inadvertent commercialization of the standard.

3. Write plugins for various browsers. Initially plugins that act like an embedded PDF reader and display only SMIL documents would the goal. Then integration with the main rendering engine for displaying XHTML+SMIL and SVG+SMIL documents (which is a much harder job).

4. Implement any missing pieces of the SMIL Animation Recommendation and the SMIL DOM interface. These two documents from the W3C provide extensions to the initial SMIL 2.0 specification. Extending `libsmil` to cover these features should not be too much of a stretch.

8 Playing With `libsmil`

The `libsmil` implementation discussed in this paper can be downloaded from the GNOME CVS repository (see [3] for instructions if you are unfamiliar with accessing that repository). The code is in the `smil` module, which contains the library as well as a few small applications and extensive documentation for library hackers and client developers alike.

Once the library has stabilized a little more, tarball releases will be made and the download site posted in a few popular locations.

References

[1] ANSI/NISO Z39.86-2002 Digital Talking Book standard `http://www.niso.org/standards/resources/Z39-86-2002.html`

[2] DAISY 2 Digital Talking Book standard `http://www.diasy.org`

[3] The GNOME CVS repository `http://developer.gnome.org/tools/cvs.html`

[4] SMIL 1.0 specification. `http://www.w3.org/TR/REC-smil/`

[5] SMIL 2.0 specification. `http://www.w3.org/TR/smil20`

[6] The Synchronized Multimedia group at the W3C. `http://www.w3.org/AudioVideo/`

[7] The Timed Text group at the W3C. `http://www.w3.org/AudioVideo/TT/`

[8] XHTML+SMIL—a W3C Note. `http://www.w3.org/AudioVideo/TT/`

Benchmarks that Model Enterprise Workloads

Using macrobenchmarks to measure and improve Linux scalability for real-world applications

Vaijayanthimala Anand, Hubertus Franke, Hanna Linder, Shailabh Nagar,
Partha Narayanan, Rajan Ravindran, Theodore Ts'o
IBM Corp.
{manand,frankeh,hannal,nagar,partha,rajancr,tytso}@us.ibm.com

Abstract

In this paper we demonstrate the use of macrobenchmarks in Linux® kernel development. We describe two macrobenchmarks, SPECjAppServer2002™ benchmark application and IBM®'s Trade, which are based on the Java™ platform and modeling enterprise applications typically found in large data centers. This paper shows how these macrobenchmarks were used to analyse potential improvements in the load balancing and yield behaviour of the 2.5 kernel's O(1) CPU scheduler. We also demonstrate how the macrobenchmarks helped debug the 2.5 kernels and compare their performance improvements over the 2.4 series.

1 Introduction

1.1 Microbenchmarks vs. Macrobenchmarks

Performance is a key driver for Linux kernel development. Several patches have been developed explicitly to improve Linux performance on various architectures. Most patches which seek to add a new kernel feature are expected to show that they minimize, if not eliminate, any negative performance impact on the system.

Over the years, various benchmarks have become popular in the kernel development community to assess the performance of patches. Most of these microbenchmarks measure specific aspects of system performance, such as tiobench for filesystem performance[Tiobench] and pipetest[Pipetest] for event delivery. Microbenchmarks have two advantages. First, they are typically both easy to set up and run and are free, making them accessible to all developers. This is particularly important for the widely dispersed open source kernel community. Second, microbenchmarks can be pivotal in determining the impact of a patch on a specific kernel subsystem.

The specificity of a microbenchmark limits its suitability for predicting overall system impact. Hence, developers often use microbenchmark *suites* such as lmbench[lmbench] and Contest[Contest]. By running a collection of microbenchmarks, each stressing a different aspect of the kernel, a more accurate picture of the overall system impact can be obtained.

Microbenchmarks (singly or in suites) suffer from two major disadvantages. First, they do not adequately capture the dynamic interactions between different kernel control paths which may be impacted by the same patch. Even if these control paths are tested individually, their interactions will not be appar-

ent. Even if the microbenchmarks could be made to run together, the interactions being tested would be ad-hoc. Second, microbenchmark suites are less representative of real world workloads. As such, while they can be used to gain a better understanding of the impact of a patch on a single subsystem, they are not ideal.

Macrobenchmarks such as Trade[Trade] and SPECjAppServer2002[SJAS] help fill this void. They exercise different parts of the kernel during runtime in ways that are representative of real world workloads that run on Linux. Such benchmarks are designed to compare hardware and software differences based on performance and cost-performance criteria. However, they can also be used to guide software development because they permit an orderly isolation and elimination of system-wide performance bottlenecks. Macrobenchmarks also allow non-kernel bottlenecks to be identified, further encouraging an evolutionary approach to kernel development.

Macrobenchmarks have their disadvantages as well. They are often expensive to purchase and are not open source. They are not easy to set up and often require multiple machines with above average physical resource requirements especially memory and disks. They may also need proprietary middleware, such as databases and Web Application Servers (hereafter referred to as AS), if freely available open-source alternatives are not performant enough or do not have the right feature set yet to allow the benchmark to be run.

One notable effort to provide free macrobenchmarks is being done by the Open Source Development Lab(OSDL). The OSDL's Database Test (DBT) benchmark suite[DBT] development effort is a welcome step in reducing the need to purchase specialized middleware in order to run macrobenchmarks. The Scalable Test Platform, also from OSDL, helps make enterprise class hardware available to all developers, further easing the hurdles in using macrobenchmarks.

1.2 J2EE-based Macrobenchmarks

The Java 2 Platform, Enterprise Edition, J2EE [J2EE] framework is a mechanism for creating distributed and Java-based enterprise class applications for various business domains such as manufacturing, supply-chain management, and on-line financial applications. Compared to the traditional transaction processing benchmarks such as TPC-H, TPC-C and TPC-W, the J2EE framework has not received much attention in the Linux benchmarking efforts. For this paper, two J2EE based macrobenchmarks, Trade and SPECjAppServer2002, are used to investigate Linux kernel performance.

J2EE applications consist of multiple layers. Performance analysis of such applications are involved and demanding as they depend on many factors. A typical J2EE stack is illustrated in Figure 1.

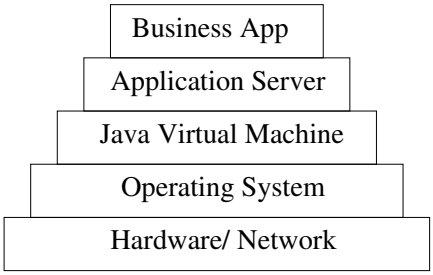


Figure 1: J2EE stack

The component which implements the actual application depends on the AS services. The AS in turn takes advantage of the underlying Java Virtual Machine (JVM) implementation. The Java applications call methods from the Java API libraries that provide access to the system resources through appropriate system calls.

The AS performance depends on many fac-

tors: caching support, transaction execution efficiency, JVM implementation, Enterprise Java Beans component pooling mechanisms, efficiency of persistent storage mechanisms, Java Database Connectivity, optimized driver support, etc. More information on J2EE best practices can be found in the literature such as Oracle9i and Java Performance [Oracle9i, EnterpriseJava, JavaPerf]. These studies focus on J2EE and its associated components rather than the operating system. By contrast, the focus in this paper is on the operating system; specifically, the Linux kernel. Two complex enterprise workloads are used to identify kernel performance issues and suggest possible kernel improvements.

1.3 Description of Trade

Trade [Trade] is a freely available benchmark developed by IBM. It is designed to measure AS performance. Trade is an end to end benchmark that models an online financial application. Specifically, an electronic stock brokerage providing web-based online securities trading.

Two versions of the Trade benchmark, Trade 2.7 and Trade 3.0, are used in this study. Trade 2.7 is a collection of Java classes, Java servlets, Java Server Pages (JSP), and Enterprise Java Beans integrated into a single application.

While Trade 2.7 is written based on J2EE 1.2, Trade 3.0 is the third generation of this benchmark making use of many features of J2EE 1.3 [J2EE] including local-interfaces, message driven beans, Container-Managed Relationship (CMR), etc. It also incorporates Web Services as one of its major enhancements. Many Application Servers in the industry implement these features.

This benchmark is used for performance research on a wide range of software components including the Linux operating system, AS, Java and more.

The Trade benchmark can be run in either a two tier or in a three tier configuration. In the two tier model, the client driver (which simulates clients of the online brokerage application) runs on one system, while the AS and the backend database runs on another. The AS executes J2EE applications which consist of two parts: the server side presentation logic and the server side business logic. In a three tier model, the AS and the backend databases run on separate systems, interconnected by a high speed network. We were more interested in the performance and scaling of the AS, so we chose to do our testing using a three tier configuration. Figure 2 shows such a configuration.

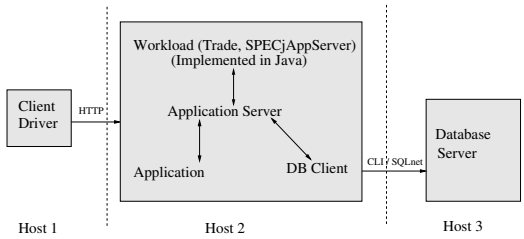


Figure 2: Three tier configuration for Trade 3.0 and SPECjAppServer2002

The client driver simulates requests of an online stock brokerage application, which makes a predefined mix of login, register, buy, sell, and quote requests of online securities. These requests come in as HTTP requests to the AS. Trade 3.0 has been configured to use the Enterprise Java Beans (EJB) mode meaning that all accesses to the back-end pass through the EJB container of the AS as opposed to the use of direct Java database connectivity (JDBC). All the orders are executed in a synchronous mode by the session and entity beans rather than being queued for asynchronus execution. The communication between the servlets and EJBs are done using the Remote Method Invocation (RMI) protocol. The backend database

stores 5000 users and 1000 securities applications. Database records are inserted, then modified as the benchmark progresses. To maintain reproducibility of the benchmark results, a database is initialized once and backed up. The database is restored before each test run.

The Trade application generates a large number of threads, of the order of 160, during its operation. The metric of interest in this benchmark is the number of web pages that are served by the AS.

1.4 Description of the SPECjAppServer2002 Benchmark

SPECjAppServer2002 [SJAS] (hereafter referred to as SJAS) is a benchmark for measuring the performance and scalability of J2EE (Java 2 Enterprise Edition) application servers and containers, by emulating the heavyweight manufacturing, supply chain management, and order/inventory system representative of a Fortune 500 company. It is a derivative of the ECperf 1.1 benchmark [ECperf]. SJAS supports multiple configurations such as single, dual, multiple, and distributed nodes. We chose dual mode (3-tier configuration) for our setup: (i) a client driver emulator, (ii) an AS tier and (iii) a database backend tier. This paper always refers to the 2002 version of SPECjAppServer.

SJAS models four logical business entities (domains): customer, manufacturing, supplier and service provider, and corporate. In the customer domain, large and small orders are distinguished in that they trigger different transactions (e.g., credit checks, order change). The manufacturing domain processes the different orders and schedules parts with suppliers. The supplier domain decides which supplier to use and handles the transaction (e.g., order size, due date) with the supplier. The corporate domain handles the list of all customers, parts, suppliers, and credit information. SJAS can be implemented either in a centralized or distributed mode. In this paper we chose the centralized mode, which allows us to put all four business entities on a single AS.

The throughput of SJAS is driven by the number of order entries in the customer domain and the manufacturing domain and is measured in TOPS, which is the average number of successful total operations per second completed during the measurement interval. TOPS is linearly related to the injection rate (IR). The IR refers to the rate at which business transaction requests from the order entry application in the customer domain are injected into the AS. The goal is to drive the injection rate as high as possible. An injection rate is sustainable if at least 90% of each type of business transactions completes within a required response time.

Though a full SJAS benchmark run requires more with respect to reporting [SJAS], we are using the sustainable injection rate as a means to evaluate scalability and changes to the kernel.

Note: SPECjAppServer2002 is a trademark of the Standard Performance Evaluation Corp. (SPEC). The SPECjAppServer2002 results or findings in this publication have not been reviewed or approved by SPEC, therefore no comparison nor performance inference can be made against any published SPEC result. The official web site for SPECjAppServer2002 is located at `http://www.spec.org/osg/jAppServer2002`.

1.5 Hardware Configuration

The Trade and SJAS macrobenchmarks are complex and require a fair amount of tuning for getting useful results. Combined with the multiplicity of issues being investigated, it was difficult to ensure that all results presented in

this paper came from the same hardware setup. Four different environments were used to collect the experimental data shown in later sections. These environments will be denoted hereafter as Configurations A, B, C, and D.

Configuration A consists of a 4-way 700 Mhz Pentium(tm) III, 1MB L2 Cache, and 4GB RAM for the AS and a 4 way 700 Mhz Pentium III, 1MB L2 Cache and 4GB RAM for the backend. A 2-way Pentium III 1GHz system was used to drive the workload.

Configuration B consists of a 4-way Pentium III 900 Mhz, 2 MB L2 Cache, 2.5GB RAM for the AS and a 4-way Pentium III 500 Mhz, 512 KB L2 Cache, 3.2 GB RAM for the backend. The client was a 2-way Pentium III 850 Mhz, 256KB L2 Cache, 2GB RAM system.

Configuration C differed from Configuration B only in doubling the number of processors in the AS tier. Thus, it has an 8-way Pentium III 900 Mhz, 2 MB L2 Cache, 2.5GB RAM for the AS, and a 4-way Pentium III 500 Mhz, 512 KB L2 Cache, 3.2 GB RAM for the backend. The client remained a 2-way Pentium III 850 Mhz, 256KB L2 Cache, 2GB RAM system.

Configuration D includes a 8-way Pentium III 900 Mhz, 2MB L2 Cache, 24GB RAM for the AS, and a 8-way Pentium III 700 Mhz, 1MB L2 cache, 8 GB RAM for the backend.

2 Kernel Bug Detection Using Macrobenchmarks

One benefit of complex macrobenchmarks is their ability to find bugs in the kernel that otherwise might not be found until the kernel is run on a large real-world system. During initial experiments with the SJAS benchmark, one such bug was found, fixed, and included in the 2.5.63 kernel.

The sole symptom was a complete system hang of the middle tier, with no oops or diagnostic of any kind produced. The hang could be reproduced by stopping and restarting the application server between five and ten times. The problem was traced using the NMI (Non maskable interrupt) watchdog timer and taking stack traces of all CPUs in the system.

The problem turned out to be threads deadlocking in the kernel. On any multiprocessor system, one task (say A) acquired a spinlock with interrupts disabled. Thereafter A performed an operation requiring all other processors to flush their Translation Lookaside Buffers (TLBs). To flush remote TLBs, task A would send an inter-processor interrupt (IPI) and go into a busy wait for an acknowledgement. However, if the tasks on the other CPUs were busy waiting on the same spinlock and also had their interrupts disabled, they would never receive the IPI, thus leading to a deadlock.

This issue was resolved by modifying the code to ensure that the spinlock was not held with interrupts disabled. The fix was included in the 2.5.63 kernel. Although the problem was easy to fix once the cause was determined, it took the right set of dynamic interactions, provided in this case by SJAS, to trigger the bug.

3 Comparing 2.4 and 2.5 Kernels

The project was initiated by using Trade 2.7 to test 2.4-based distribution kernels as well as then-current stock 2.5 kernels. Presented in Table 1 are the results of running Trade 2.7 in a three-tiered mode using configuration A.

The results obtained were unexpected. It was found that the 2.4 based distribution kernel (2.4-distro) performed better than the 2.5.59 stock kernel. To recheck the results, we ran the SJAS benchmark on a stock 2.4.20 kernel (2.4.20-stock) and compared results with

Kernel Version	#CPUs	TOPS	CPU Utilization (%)		
			user	system	idle
2.4.20-stock	4	Base1	68	14	17
2.5.59	4	Base1-3.8%	60	10	28
2.5.66	4	Base1+11.6%	70	12	16
2.4.20-stock	8	Base2	47	10	41
2.5.59	8	Base2+0%	36	6	56
2.5.66	8	Base2+24.0%	49	9	41

Table 2: Performance and middle tier CPU utilization of SJAS on 2.5.59 and 2.5.66 kernels using 2.4.20 as a baseline for 4-way (Base1) and 8-way (Base2) servers

Kernel	TOPS	%CPU Usage on AS
2.4-distro	Base1	87
2.5.59	Base1-14.4%	66
2.5.59+D7	Base1+2%	86

Table 1: Trade 2.7 results

processes runnable	context switches	CPU Utilization (%)		
		user	sys	idle
8	15689	74	18	8
12	18844	76	18	6
10	15778	71	21	8
11	16114	74	20	6
11	17629	74	17	8

Table 3: Output from vmstat for AS on a 2.4-x distro kernel using a 4-way server and Trade 2.7. Number of runnable processes are 2-3 times the number of processes.

the 2.5.59 kernel using Configurations C and D. The results, shown in Table 2, confirm that the 2.4.20-stock kernel exhibits better performance than 2.5.59 with the latter's TOPS decreasing by 3.8% on Configuration C and remaining unchanged in Configuration D.

At a later date, we also compared the performance on a 2.5.66 kernel and found that it performed significantly better than 2.4.20-stock with an *increase* in TOPS of 11.6% and 24.0% on Configurations C and D respectively. Table 2 shows that system time remained approximately the same for these two kernels though overall utilization was higher for 2.5.66. Isolating the performance changes between 2.5.59 and 2.5.66 is part of our future work. We felt our first task was to determine why the 2.5.59 kernel performed worse than the 2.4.20 and 2.4.20-distro, despite several scalability enhancements in 2.5.59.

Since distribution kernels have patches added on top of a 2.4 stock kernel, the profile data was analyzed in order to understand the observed differences. Comparing the vmstat outputs for a 2.4-x distro kernel (Table 3) to a 2.5.59 kernel (Table 4) we clearly see that the latter has fewer runnable processes in general and often has fewer runnable tasks than processors. Consequently, 2.5.59 shows higher idle times. The data initiated further investigation of the CPU scheduler behaviour.

In the next step, readprofile data was collected at a 60 second granularity during the steady run of Trade 2.7 on the same configuration as above. Comparing the data for the 2.4-distro kernel (Table 5) and 2.5.59 (Table 6), we see that `schedule()` is the costliest kernel function in both kernels.

The calls to `schedule()` drew our attention because they were still high on both lists even though 2.4-x uses the old scheduler and 2.5.59

runnable tasks	context switches	CPU Utilization (%)		
		user	sys	idle
3	12195	41	10	49
5	12079	41	9	50
7	17508	49	10	42
2	12087	41	9	50
3	11898	44	9	47

Table 4: Output from vmstat for AS on a 2.5.59 kernel running Trade 2.7 on a 4-way server. Number of runnable processes often dip below the number of processors and are low compared to the 2.4-x data shown earlier.

Ticks	Kernel function	Normalized Ticks
23969	**Total**	0.02
7071	default_idle	110.48
2388	schedule	1.52
822	csum_partial_copy...	3.31
799	send_sig_info	4.54
744	save_i387	1.29
511	tcp_v4_rcv	0.31

Table 5: Readprofile data for AS on a 2.4-distro kernel running Trade 2.7 on a 4-way server. Normalized ticks gives the number of ticks divided by the size of the function. `schedule()` figures are high though idle times are low compared to 2.5.59.

Ticks	Kernel function	Normalized Ticks
60332	**Total**	0.05
54048	default_idle	844.50
397	schedule	0.41
365	csum_partial_copy...	1.47
191	tcp_sendmsg	0.04
181	__kfree_skb	0.60
177	load_balance	0.19

Table 6: Readprofile data for AS on 2.5.59 running Trade 2.7 on a 4-way server. Normalized ticks gives the number of ticks divided by the size of the function. `schedule()` is costly despite the usage of the O(1) scheduler; also idle time is higher than in the 2.4-distro kernel.

uses the O(1) scheduler. To examine our hypothesis that the O(1) scheduler was causing the high idle times, we tested a 2.4.20 kernel with and without the O(1) scheduler using the same configuration as above. The results, not shown in this paper, were similar to the data shown earlier and confirmed the hypothesis. The 2.4.20 stock kernel produced 20% better throughput than the 2.4.20+O(1) scheduler. Further, 2.4.20+O(1) had fewer tasks in the run queue than the number of CPUs in the system and 40% idle time, similar to the results found in the 2.5.59 kernel.

Using snapshots of runqueue lengths in all CPUs at each timer tick, it was found that CPUs were going idle while there were runnable tasks on other runqueues. The imbalance in runqueue lengths across various CPU's while using O(1) led us to a careful examination of the load balancing logic of the O(1) scheduler. The analysis is discussed in the next section.

4 Load Balancing

Before discussing the results of various experiments, we revisit the load balancing differences between the old 2.4 scheduler and O(1). The 2.4 scheduler uses a single runqueue for all CPUs which leads to high lock contention and lock hold times when the number of tasks and CPUs start increasing. O(1) replaces the single runqueue with per-CPU runqueues. While choosing the next task to run on a CPU, it does not look at remote runqueues, maintaining the O(1) behaviour and preserving cache affinity. Consequently, it needs to explicitly balance the load on each runqueue by calling a `load_balance()` function. Workloads which are sensitive to load imbalance, such as Trade and SJAS, get affected by the effectiveness of `load_balance()`. In 2.5.59, `load_balance()` is called periodically on each CPU, with the frequency of invocation determined by the idleness of the CPU.

To improve the load balancing behaviour of the O(1) scheduler, we tried a series of patches from Ingo Molnar's D7 patch [D7-PATCH] to Andrea Arcangeli's try_to_wake_up patches (included within his O(1) patch [AA-O1-PATCH]) to a *find_busiest_queue* patch, created in-house [FBQ].

4.1 D7 patch

Ingo Molnar's D7 patch unconditionally migrates a task from a remote to the current runqueue if the current CPU is about to go idle. Table 1 shows that this patch helps 2.5.59 perform 2% better than 2.4-distro for Trade 2.7, more than making up for the 14.4% performance loss seen by stock 2.5.59. For an SJAS workload, the same patch helps 2.5.59 draw on a par with the 2.4.20 stock kernel, overcoming the 3.8% degradation seen by 2.5.59 versus 2.4.20 (Table 7). The 10% degradation of 2.4.20+O(1) compared to 2.4.20 in the same table reconfirm the earlier hypothesis that the O(1) scheduler is at least partially responsible for the performance loss of 2.5.59 compared to 2.4.20.

Kernel level	CPU Usage	% TOPS improved
2.4.20-stock	82%	baseline
2.4.20+O(1)	66%	-10.6%
2.5.59-stock	70%	-3.8%
2.5.59+D7	64%	no change

Table 7: SPECjAppServer2002 - v1.14, 4-way results on 2.5.59

Kernel level	CPU Usage	%TOPS improved
2.5.66-stock	82%	baseline
2.5.66+trytowakeup1	83%	+4.3%
2.5.66+trytowakeup2	89%	+0%
2.5.66+busiestqueue	82%	-4.3%

Table 8: SPECjAppServer2002 - v1.14 4-way results on 2.5.66

4.2 Load Balancing on Task Wakeup

The O(1) scheduler used in Andrea Arcangeli's 2.4 kernel tree contains two changes to do load balancing on task wakeup events in addition to the periodic invocations of `load_balance()` in the stock kernel's O(1). We implemented these changes as two separate patches for the 2.5.66 kernel.

The first patch, henceforth called try-

Kernel level	CPU Usage	%TOPS improved
2.5.66-stock	56%	baseline
2.5.66+trytowakeup1	60%	+4.4%
2.5.66+trytowakeup2	72%	+3.0%
2.5.66+busiestqueue	65%	+5.2%

Table 9: SPECjAppServer2002 - v1.14 8-way results on 2.5.66

towakeup1, modifies the `try_to_wake_up()` function to invoke load balancing whenever a task is being woken up. Using the task wakeup event as a load balancing trigger was also motivated by the high count for calls to `tcp_data_wait()`; the high count causes task wakeups in the profiling data similar to the one shown in Table 6 for 2.5.59. The trytowakeup1 patch improved SJAS throughput performance by 4.7% on Configuration C compared to the 2.5.66 stock kernel, as shown by Table 8. Configuration D showed a similar 4.3% improvement as seen in Table 9.

The second patch, henceforth called trytowakeup2, tries to explicitly address the problem of CPUs going idle by trying to place the task being woken up onto an idle CPU if possible. This is in contrast to trytowakeup1 which is only concerned with pulling tasks to the runqueue of the waker. While trytowakeup2 increases SJAS performance by 4.5% in Configuration D (Table 9), it has no effect in Configuration C(Table 8). The behaviour can be explained by the relative lack of idle CPUs in Configuration C (4-way AS) compared to Configuration D.

The final patch called busiestqueue [FBQ], aimed at improving the aggresiveness of the existing load balance function itself rather than changing the frequency or location of its invocation. In the normal O(1) scheduler, the `find_busiest_queue()` function is used by `load_balance()` to find the remote runqueue with the maximum number of runnable tasks from which tasks can be pulled to the current runqueue. The `load_balance()` code checks whether tasks on the remote runqueue are suitable for migration but if none are found suitable, it does not try to find another runqueue and try again. The busiestqueue patch remedies this behaviour by modifying `find_busiest_queue` and its invocation by `load_balance()` to ensure that all remote runqueues are examined for tasks to migrate. The results from using the patch are mixed. Configuration C shows a performance degradation of 5.1%(Table 8) whereas Configuration D shows a 2.4% improvement(Table 9). Evidently, the patch is too aggressive and the extra cycles spent in trying to find another remote runqueue prove too costly. We are in the process of finetuning the patch by limiting the number of iterations in the search for a busy queue.

The trytowakeup1 and busiestqueue patches increased performance by around 5% on the 8-way Configuration D when applied individually and in combination (data not shown). This suggests that one or the other approach is sufficient in achieving better load balancing and leads to the question of which one should be used. The answer will lie in the effect of the patches on other workloads and is part of our future work.

5 Yielding to Other Tasks

The system call `sys_sched_yield()` causes the calling task to yield execution to another task that is ready to run on the current processor. Multi-threaded applications often use `sys_sched_yield()` to improve interactive response or to improve the performance of the system by letting the scheduler use the processor resources more effectively. This is particularly true if the applications use traditional userspace locks (not based on futexes).

However, the benefits realized from the use of `sys_sched_yield()` are heavily dependent on the implementation of the system call. The CPU scheduler selects the next task to run and determines how long the yielding task will remain on the runqueue before getting a chance to run again. The following implementations

of `sys_sched_yield()` are feasible:

PA The yielding process is queued right after the next task on the same priority queue. Effectively, it yields only to the next task at the same priority level.

PB The yielding process is queued at the tail of its priority queue making it yield to all runnable tasks at the same priority level.

PC The yielding process is moved to the priority queue on the expired list effectively making it yield to all runnable tasks in the system (as the expired list becomes the active list only after all runnable tasks have exhausted their timeslices).

The yielding task rarely knows how long it needs to yield before it can attempt to acquire a shared resource again as the availability of the shared resource depends on external events and progress made by competing tasks. For instance, an interactive application might see reduced response time if policy PA were used. But a task polling for a shared resource such as a userspace spinlock, might benefit from PB or PC which allows the task holding the resource to run and potentially release it for use by the yielding task. As the CPU scheduler is unaware of the task's rationale for using `sys_sched_yield()`, it cannot decide the best yielding interval either. Hence different Linux distributions have tried all three policies.

To understand the impact of these policies on macrobenchmarks such as Trade and SPECjAppServer2002, we collected profile data to see the number of `sys_sched_yield()` calls issued. Table 10 shows that `sys_sched_yield()` accounts for almost one third of all calls to `schedule()` when Trade 2.7 is run on Configuration A.

Table 11 shows the data collected by instrumenting the `sys_sched_yield()` for a 1 minute run of Trade 2.7 on 2.4.20 using Configuration A. In the table, *higher, lower* and *same* refer to the number of `sys_sched_yield()` calls in which there was at least one task on a higher, lower and same priority level as the yielding task. The row labelled *only* counts the number of `sys_sched_yield()` calls in which the yielding task was the only one on its runqueue. We see that most tasks in the system are on the same priority queue as the yielding task. Hence, the policy adopted by `sys_sched_yield()` is likely to have a significant impact on performance.

Ticks	Kernel Functions
6826403	**Total**
2523245	sys_sched_yield+11d
2236660	cpu_idle+3e
1312369	schedule_timeout+9d
327747	schedule_timeout+184

Table 10: Functions calling `schedule()` for a 2 minute run of Trade 2.7 on Configuration A

The 2.5.65 stock kernel uses the PC policy. We implemented the other two policies, PA and PB and compared their performance with PA using Trade 3 in Configuration D. PB and PC turned out to have the same results for the benchmark which follows from Table 11. As there are very few tasks on lower priority levels when `sys_sched_yield()` is called, PB and PC are effectively the same policy. Hence only PA and PC results are shown in Table 14. We see that the pages per second (pg/s) drops by 32.6% if PA is used instead of the default PC policy. CPU usage (usg) and efficiency (effncy) also see a corresponding drop. Similar results are seen for SJAS (not shown). Using PA instead of PC decreases TOPS by 10% on a 4-way.

The reasons become clear from the vmstat outputs of PA and PC shown in Tables 12 and 13 respectively. The number of context switches

Relative Priority	CPU 0	CPU 1	CPU 2	CPU 3	CPU 4	CPU 5	CPU 6	CPU 7
Same	145103	157055	163064	156379	162783	161733	167366	177876
Only	117653	112196	112387	105653	101420	96053	108830	92293
Higher	26	34	28	29	31	25	33	36
Lower	929	937	1000	1073	1036	1016	1156	1132
Total	263711	270222	276479	263134	265270	258827	277385	271337

Table 11: sys_sched_yield call count and the distribution of tasks on priority queues relative to the yielding task, using Trade 2.7 on 2.4.20 in Configuration A. The data indicates that most tasks in the system were on the same priority queue as the yielding task.

increases almost fourfold (from approximately 6700 to 27000) when PA is used. As PA causes the yielding process to get scheduled much sooner, the shared resource it waits on is generally not available, thus leading to frequent context switches as it yields again and again. For such an application, the default policy of letting all other runnable tasks run once is a good choice.

The kernel development community has been discussing alternative policies for `sys_sched_yield()` in order to improve response time for interactive applications. The results shown here indicate that such changes might adversely affect macrobenchmarks like Trade. However, this is only true until application servers start using the new fast user-level mutex (futex) feature provided by the 2.5 kernels.

6 Conclusions and Future Work

In this paper, we have examined two macrobenchmarks, Trade and SPECjAppServer2002. Both benchmarks model complex workloads utilizing the J2EE framework, which are popular in many enterprise data centers. We have shown a case study of a kernel bug that was triggered by these benchmarks and which would have been hard to find otherwise.

procs		system		cpu	
r	in	**cs**	us	sy	id
14	6067	27204	63	10	27
9	5868	29230	60	9	31
12	5337	24765	61	8	30
10	6021	27753	61	9	30
5	5947	27496	64	10	25

Table 12: vmstat output collected while using Policy A showing high context switches and high idle times. *r*, *in*, and *cs* refer to the number of runnable tasks, interrupts, context switches respectively, while *us*, *sy*, and *id* refer to the percentage of time spent by CPUs in user mode, system mode, and idling respectively.

procs		system		cpu	
r	in	**cs**	us	sy	id
18	7788	6903	85	14	0
26	7168	6686	84	11	6
24	8010	6798	87	12	1
23	8083	6727	87	13	0
22	7934	6212	87	13	1

Table 13: vmstat output collected for Trade 3 running on Configuration D, while policy PC is used to implement `sys_sched_yield()`. The other labels are explained in the caption for Table 12. Context switches and idle time are significantly lower for PC compared to PA.

Kernel	Policy	Pg/s	Usg	Effncy
2.5.65	PC	Baseline	95%	100%
2.5.65	PA	-32.6%	75%	85%

Table 14: Comparison of `sys_sched_yield()` implementations using Trade 3 on Configuration D.

The macrobenchmarks were also used to reveal deficiencies in the load balancing logic of the 2.5 kernel's O(1) CPU scheduler. Various patches were used to increase the aggressiveness of load balancing and reduce the probability of CPUs going idle despite the presence of runnable tasks in the system. Based on our observations, we suggest the following four load balancing policies might be of help for workloads sensitive to load imbalance such as Trade and SJAS:

- Load balance during initial placement of tasks by choosing the idle processor
- Load balance during wakeup by choosing the idle processor
- Load balance the queues aggressively (similar to patches described above) when a processor goes idle
- Consider providing aggressive load balancing through a configuration option

More patches will be produced to implement the above catagory of improvement and the investigation will continue to find a fair load balancer to improve these workloads for SMP (Symmetrical Multi Processor) and NUMA (Non Uniform Memory Access) systems. Any load balancing patches proposed will need to be tested using different workloads to make sure that they do not degrade performance by unnecessary balancing.

The final part of this paper examined different implementations of the `sys_sched_yield()` call and concluded that the existing 2.5.65 implementation performed well for macrobenchmarks such as Trade and SJAS.

There is still much work to be done in exploring how the kernel can more efficiently support J2EE-based workloads. As we have seen, these workloads tend to be very sensitive to scheduler issues, and changes which benefit one workload may actually cause harm to other workloads.

Further tuning of the application and improvements in the Linux kernel has improved the CPU utilization of these benchmarks. Hence, while initial attempts to use spinlock metering to find lock contention was not fruitful, we anticipate that future work in improving the benchmark score of these workloads will include finding and fixing lock contention problems.

We have used, and are continuing to use, macrobenchmarks as a method for finding potential areas for improvement in the Linux 2.5 kernel, especially as it relates to the Linux scheduler. We hope we have demonstrated to the reader that more complex benchmarks are a useful tool for the kernel developer interested in improving the performance and scalability of the Linux kernel.

7 Acknowledgments

The authors would like to thank the following people for their help: Jianwen Alex Feng, Bill Hartner, Wilfred Jamison, Sandra Johnson, and Rick Lindsley.

8 Legal Statement

This work represents the view of the authors and does not necessarily represent the view of IBM.

References

[AA-O1-PATCH] Andrea Arcangeli, `http://www.kernel.org/pub/linux/kernel/people/andrea/kernels/v2.4/2.4.20rcaa1`

[FBQ] Rick Lindsley, `http://www.ibm.com/linux/ltc/patches/?patch_id=865`

[Contest] Con Kolivas, `http://members.optusnet.com.au/ckolivas/contest/`

[D7-PATCH] Ingo Molnar, `http://people.redhat.com/mingo/O(1)-scheduler/sched-2.5.59-D7`

[DBT] Open Source Development Lab, `http://sourceforge.net/projects/osdldbt`

[J2EE] Java 2 Platform, Enterprise Edition `http://java.sun.com/j2ee/overview.html`

[ECperf] `http://java.sun.com/j2ee/ecperf/`

[EnterpriseJava] Kingsum Chow, Ricardo Morin, and Kumar Shiv, *Enterprise Java Performance: Best Practices*, Intel Technology Journal, Vol 07 (Feb 2003) p. 32–48

[JavaPerf] Java Performance Tuning, `http://www.javaperformancetuning.com`

[lmbench] Carl Staelin and Larry McVoy, `http://sourceforge.net/projects/lmbench`

[Oracle9i] Oracle9i Application Server, *Architecting for J2EE performance*, `http://otn.oracle.com/products/ias/pdf/ArchitectingForJ2EEPerformance.pdf`

[Pipetest] Ben LaHaise, `http://www.kvack.org/~blah/ols2002.tar.gz`

[PMQS] Hubertus Franke, Shailabh Nagar, Mike Kravetz, Rajan Ravindran, *PMQS:Scalable Linux Scheduling for High End Servers, ALS'01*, Annual Linux Symposium, Oakland, 11/2001

[SJAS] Standard Performance Evaluation Corp., `http://www.specbench.org/jAppServer2002/`

[Tiobench] Mika Kuoppala, `http://sourceforge.net/projects/tiobench`

[Trade] Application Server Benchmark, `http://www-3.ibm.com/software/webservers/appserv/benchmark3.html`

Large Free Software Projects and Bugzilla

Lessons from GNOME Project QA

Luis Villa
Ximian, Inc.
luis@ximian.com, http://tieguy.org/

Abstract

The GNOME project transitioned, during the GNOME 2.0 development and release cycle, from a fairly typical, mostly anarchic free software development model to a more disciplined, release driven development model. The educational portion of this paper will focus on two key components that made the GNOME QA model successful: developer/QA interaction and QA process. Falling into the first category, it will discuss why GNOME developers bought in to the process, how Bugzilla was made easier for them and for GNOME as a whole, and why they still believe in the process despite having been under Bugzilla's lash for more than a year. Falling into the second, some nuts and bolts: how the bugmasters and bug volunteers fit into the development process, how we coordinate, and how we triage and organize. Finally, the paper will discuss how these lessons might apply to other large projects such as the kernel and Xfree86.

1 Introduction

During the GNOME 2.0 development and release cycle in 2002, the GNOME project grew from a fairly typical, fairly anarchic free software development model to a more disciplined, release driven development model. A key component of this transition was the move towards organized use of Bugzilla as the central repository for quality assurance and patch tracking. This was not a process without problems-hackers resisted, QA volunteers did too little, or too much, we learned things we need to know too late, or over-engineered the process too early. Despite the problems, though, GNOME learned a great deal and as a result, GNOME's latest releases have been more stable and reliable than ever before (even if far from perfect yet :)

The purpose of this paper isn't to teach someone to do QA, or to impress upon the reader the need for good QA—other tomes have been written on each of those subjects. Instead, it will focus on QA in a Free Software context—how it works in GNOME, what needed to be done to make it work both for hackers and for QA volunteers, and what lessons can be learned from that experience. To explain these things, I'll focus on three main sections. The first will be a very brief history of GNOME's transition to a more Bugzilla-centric development style, in order to provide some background for the rest of the paper. The second part will focus on the lessons learned from this transition. If a reader needs to learn how to manage QA and Bugzilla for a large Free Software project (either as bugmaster or as a developer), this section should serve as a fairly concise guide to GNOME's free software QA best practices—explaining how developers and QA should work together, what processes make for good free software QA, and how a free soft-

ware project can build a QA community that works. Finally, in the third section, the paper will attempt to discuss how GNOME's lessons might be applied to the usage of Bugzilla in other projects.

2 Background

Before going further, I'll offer a brief bit of background on the GNOME project and how it came to be a project where QA was an integrated part of the Free Software development process.

2.1 The GNOME Code Base

GNOME is not the kernel or X, but it is on roughly the same order of magnitude in terms of code and complexity. And it is continuing to grow as we integrate the functionality expected by modern desktop users. Desktop is, of course, a very vague term, but at the current time, GNOME includes a file manager, a menu system, utilities, games, some media tools, and other things you'd expect from the modern proprietary OSes. Of course, there is also a development infrastructure as well, including accessibility for the disabled, internationalization, advanced multimedia, documentation, and support for many standards.

To provide all of this, GNOME is a whole lot of code. A very basic build is more than 60 modules, each with its own build and dependencies. Wheeler's sloccount in a 'basic' build (no web browser and no multimedia infrastructure) shows 8,000 .c or .h files and roughly 2.0 million lines of code.[Wheeler]. When counting word processing, web browsing, spreadsheets, media handling, and e-mail and calendaring (all of which are provided by fairly robust, complex GTK/GNOME apps) the total grows to roughly 4.2 million lines of code.

2.1.1 The GNOME User Base

To a free software hacker, of course, users are not just users—they are potential volunteers. The 'modern desktop users' GNOME developers like and/or hate to talk about are actually quite numerous, which means a large base of people who generate bug reports (especially stack traces) and who also can be possibly persuaded to do QA work. For context, Ximian GNOME 1.4 had a million and a half installations. By late in the 2.0 cycle daily rpm builds of CVS HEAD drew thousands of downloads a day, and each tarball pre-release of GNOME 2.0 was downloaded and built by tens of thousands of testers. So, even during the relatively unstable runup to 2.0, thousands of users were pounding gnome's pre-releases on a daily basis, and many of those became willing helpers in the QA process- over 1,500 people submitted bugs during the 2.0 release cycle, and several thousand more crashes were submitted anonymously. This type of QA is difficult for anything but the largest proprietary software companies to match, and has been invaluable to GNOME.

2.1.2 QA in GNOME 2.0

As can be imagined, this much code, with this many users, trying to excersize a lot of functionality, while developers seek to make things more functional, usable and accessible, generates a lot of crash data and bug reports. Between January 2002 and the release of GNOME 2.0 on June 26th, 2002, slightly over 10,000 bugs were tagged as 2.0 bugs and an additional 13,000 non-2.0 bugs were filed. The QA team triaged or closed over 17,000 of those. Over 5,000 more were eventually marked as fixed, including over 1,000 deemed high priority by the QA team. For a project with a fairly small active core development

team, these were huge numbers. It's only because of volunteer help in identifying and triaging bugs that dealing with this at all was possible. During the 2.0 cycle, regular 'bug days'—12 hour long meetings of new volunteers—and a mailing list helped coordinate and recruit volunteers.

3 So what did we learn?

None of this was a pretty or clean process; lots of mistakes were made and not quite as many lessons learned. But we did learn a number of general rules for volunteer driven, high-volume bug handling. The gist of these lessons can be summarized simply—QA volunteers must work with their community to find, identify, and track the most important bugs. But the details are more complex and will, I hope, make this paper worth reading.

3.1 QA and Hackers

While ESR may have his critics, he was undoubtedly right in observing that we are all in this to scratch our itches, whatever those itches may be. Free software QA is a slightly odd bird in this light—QA volunteers are in it to scratch the itch of higher quality software, but they can't do it themselves. That means paying a lot more attention to the needs of others than may be typical for free software. Following are some of the GNOME team's lessons learned.

3.1.1 Rule 1: free software QA must support the needs and desires of developers in order to succeed

This seems fairly obvious, but it is also fairly easy to forget or ignore. Free software begins and ends with developers who are having fun. There may be others involved for reasons other than 'fun', but if QA's sole purpose is to whine about what QA thinks are the important flaws, volunteers will leave, and leave quickly. QA must think first and foremost not about their own goals, but about the goals of developers.

To put it another way: developers think they can do their thing without QA (which, in free software, they mostly can) and QA absolutely cannot do its job (which is getting bugs fixed, not just finding bugs) without developers. If QA forgets this in proprietary software, developers have to suck it up. If QA forgets this in free software, developers will ignore them, or worse, walk away from the project.

This is not to say that QA must be silent servants of hackers, never giving feedback or their own input. QA volunteers can be and should be trusted individuals whose advice is valued. But that will happen more quickly and more easily if the goal of supporting and aiding hackers is always first and foremost.

3.1.2 Rule 2: QA needs guidance from maintainers

In order for QA volunteers to serve the needs of maintainers and developers in general, maintainers and developers must clearly communicate their priorities. This falls out pretty cleanly from rule 0: if a project doesn't know where the project is going, or what the project's developers want, then it is going to be very hard for QA to help reach those goals. This also means that when a project is conflicted, QA teams may not be of as much utility as a project expects.

'What a project wants', from a QA perspective, is usually pretty obvious in GNOME—stability, stability, stability. If a program can be crashed, or a button doesn't work, everyone typically agrees that this should raise a red flag.

But past that, things often get murkier—some maintainers may, for example, care deeply about code quality, while others may be deeply involved with fixing usability issues. And how does one weigh a difficult to reproduce crasher against, say, a build problem on Solaris? That's not typically an easy or fun call to make; it's nearly an impossible one to make correctly unless a QA volunteer has guidance from the developer.

In proprietary QA, these answers are usually pretty easy—compare against a design doc and go. In free software QA, where design docs are often lacking in details if they exist at all, developers must do the best they can to communicate to QA what exactly the priorities are so that when the QA team finds a problem they can classify it correctly.

3.1.3 Rule 3: QA must persuade hackers they are useful and intelligent

When I came on board the Evolution team, the universal response was 'oh, someone is going to mark duplicates for us, that's nice.' When I left, I was very flattered to know that the team was worried about a lot more than duplicates. The difference between coming and going was not just that I was effective, but that the very first thing I did was work very hard to learn the lay of the land in Evolution. Instead of reading one bug and deciding 'this is bad', I read nearly two thousand bugs before doing anything more than rudimentary marking of duplicates in the bug database. This is an extreme example, of course, but it is the direction Free Software QA volunteers should lean if they can.

In contrast, some first-time GNOME volunteers have dropped in, read one or two bugs, and decided 'oh, this bug is hugely important', and tried to mark it as such. Worse, some will try to guess at the source of problems in code they've never looked at, or (this is inevitably the most irritating to developers) they'll declare that something 'must be easy to fix'. Invariably, this leads to irritation from developers who have seen a lot more issues and have, unsurprisingly, a much better grasp of their own code. The best way to avoid this is to work hard to always make the right call, especially when first working on a project. There aren't the obvious checks of functionality and code review that typically establish trust between hackers—so very sound and conservative judgement has to substitute at first.

3.1.4 Rule 4: Bugzilla cannot be the end-all and be-all of communication between hackers and QA

Bugzilla is a wonderful tool, that allows for great communication and incredibly flexible ways to sort, parse, and otherwise mangle bugs. But it doesn't speak to mailing lists, and it can't selectively poke hackers about important issues. QA volunteers must actively seek out other, non-Bugzilla forms of communication—mailing lists and IRC, primarily, but also web pages and other forums. Use these channels to draw attention to QA and to QA processes—most important new or outstanding bugs, important recent fixes, new features or reports in Bugzilla, or even simple 'this many bugs were opened and this many closed last week.' By doing this, a QA team can establish an identity as a 'regular' part of the development process even amongst developers who aren't familiar or comfortable with Bugzilla.

This was a lesson learned the hard way in GNOME. During the 2.0 cycle, the bug squad assembled and emailed regular Friday status reports to GNOME's main development list. It was well recieved by hackers, but like many things in free software, it wasn't completely appreciated until (during the 2.2 cycle) it was

gone, done away with by lack of attention on the part of the bug squad and the mistaken belief that it wasn't very useful to developers. Developers let us know, and as a result we'll try to bring

3.2 Triage

Triage is a word that has been thrown around in this paper a fair amount—before going further it may be useful to define it. In medicine, battlefield triage is the process of separating the very badly wounded from those who are lightly wounded and those who are so wounded that they will die regardless of treatment. In a free software context, it's the process of separating and identifying bugs that are most severe and/or most useful to developers out from the inevitable mountain of bugs that will come in for any popular project. Specifically, in GNOME, we triage by setting 'priority', 'severity', and 'milestone' fields in Bugzilla. Like communication, GNOME has learned a fair amount about this in the past year.

3.2.1 Rule 5: bugs need to be triaged, not just tracked

When I came into GNOME and Evolution, both projects knew that having a Bugzilla was a good thing. So, they dutifully entered bugs in their bug tracking systems—they tracked bugs. But neither project had useful definitions of severity and priority—they couldn't or didn't triage their bugs. So what they had, when they needed to know what came next, was a large list of bugs in basically random order. Not surprisingly, that wasn't very helpful and so bugs ended up getting entered in to Bugzilla and never read again. Developers ended up maintaining lists outside of Bugzilla to help them figure out what to do next—a silly duplication inefficiency in projects that can't really afford much inefficiency.

If this kind of thing is happening, it indicates that bugzilla is not being used properly. The solution is to carefully define priorities, severities, and milestones, and use them religiously, by looking at every bug and making at least an attempt to judge how bad it is and when it should be fixed by. When it comes down to release time, having consistently marked bugs with these priorities means that it will be much easier to say 'these things must be fixed, these we fix if we have time, these we pretend just don't exist.' And that will leave you with much better software.

3.3 Rule 6: triage must be tied to release goals

This is a whole lot like Rules 0 and 1, so I'll be brief. It's worth repeating, though—triage is basically the art of determining what is important, and if QA and hackers frequently disagree on what is important, QA will get ignored. This greatly reduces the space for personal freedom in QA—several volunteers have come into GNOME, picked up on a pet theme and marked those bugs up, and I've spent a great deal of time apologizing for them. Bugzilla cannot be allowed to become a soapbox, for anything except the goals maintainers have already agreed to. If there is dissent on those goals, take it to the lists—Bugzilla is not a good forum for setting or arguing goals.

(In proprietary software, this is easy—'project goals' are in a tightly defined project spec that must be followed. Bug volunteers, especially those coming from a proprietary background, must remember that this just isn't possible in Free Software.)

3.3.1 Rule 7: triage new bugs agressively, or Bugzilla will quickly terrify maintainers

The initial temptation of almost all the QA volunteers I've dealt with is to assume that a bug they've just read is extremely important, and should be prioritized to reflect that. In some ways, this is true—we do see a lot of very ugly bugs, that in an ideal world given infinite resources and time would be high priority. But we live in a world of volunteers and spare time, so marking bugs as more important than others should be done only carefully and conservatively.

Most free software hackers work on their projects in blocks of very short periods of time. That means that if Bugzilla is their TODO list, the smaller and the more sorted it is, the more beautiful. In practice, we've found that it means that once maintainers trust their QA, they tend to only look at high priority bugs. This can be scary- it puts a lot of power in the hands of QA, and messing up by deciding that a bug is not important enough for a maintainer to look is seemingly very bad. This is utterly true in proprietary QA—if a QA guy screws up and punts something that he shouldn't, there may not be much of a system of checks and balances to catch the error. Free Software QA saves us from such a fate—punt a bug or mark it low priority, and if it is important, ten other people will file it or add comments. The massive volume of bugs we get is a constant check.

For example, in GNOME, we regularly see crashes that a maintainer or QA volunteer (or often the original reporter) decides is completely impossible to reproduce. We knock them down in importance or close them in that case. Often, they actually are impossible to reproduce- build problem, transient issue that got fixed the next day, or other such. But in some cases, after everyone has thrown up their hands, you'll continue to see reports of the crash. The 'mistake' of triaging or punting the original crash can then be revisited—thanks to the volume of bugs we recieve, we've gotten ample confirmation that maybe it wasn't such a bogon after all.

This isn't perfect, of course—in Evolution, for example, we get relatively few bugs on first-time installation, so a single punt on an installation issue may obscure much deeper and more important issues that won't be filed again for some time. But, unfortunately, it's something that frequently must be done—the alternative is often for maintainers to query Bugzilla and face massive lists that are quickly overwhelming. QA can and should serve as a buffer for that if necessary.

3.3.2 Rule 7: closing old bugs, even completely unread, is unpleasant but OK

GNOME's QA was publicly flamed several months ago by someone (we'll call them 'james w. z.') for mass-closing old GNOME bugs without substantially reviewing them. This was an unfortunate thing that we hate to do, but it was justified. In the typical free software cycle, a project starts off too unstable and with too few users to get many bug reports. After the project builds and grows, you still have all the old bugs from the early period, and an increasing number of users and bug reporters, many of whom are filing bugs you can't possibly have time to fix or even sometimes look at before your next rewrite and release.

Faced with an escalating number of bugs, a volunteer-driven project that can't easily bring in more resources has two options: mass close old bugs with an 'if you still see this in our latest release, please reopen the bug', or let the DB grow so large that it is unusable for hackers and QA volunteers alike. From these two,

the choice is obvious if unpleasant. Furthermore, as 'james' reminded us, this isn't something that is easy for bug filers to understand. But when doing it, remind yourself: if it is still a bug, someone will file it again.

3.3.3 Rule 8: triage rules can't be just in one person's head

As already mentioned, the first step I took when moving in to GNOME was to revise and rewrite GNOME's definitions for priorities. Previously, they'd been fairly broad and inspecific. The new priorities gave specific examples, and tried to group problems into specific classes as well. This was an important first step for sane triage across Bugzilla. But it was not enough—nearly all judgment calls by volunteers ended up coming back to me for validation, since the definitions did not include a lot of my experience and judgement—just examples and definitions. So, I've started (and the QA volunteers have rewritten and completed) a GNOME triage guide [`http://developer.gnome.org/projects/bugsquad/triage/`]. This document attempts to put a lot of collective wisdom down onto paper, and makes it easier for new volunteers to come in and get started, and for old volunteers and developers to understand more precisely what should be going on.

This will hold true for any project without strong guidelines, I believe—either a large group of volunteers will inconsistently apply their own judgments (confusing developers) or the project will become overly dependent on one person, which will eventually again lead to inconsistency as the mass of bugs becomes too much for that one person to handle. Again, this was a lesson learned by GNOME only after 2.0- during the 2.0 cycle, much of the triage wisdom stayed in my head and when I had less time (during the 2.2 cycle) the process grew a bit creaky, because triage often blocked on my availability to answer judgment calls.

3.4 Some Miscellaneous (But Important) Observations on Free Software QA

There are a few other important lessons GNOME has learned that aren't rules, per se, but which everyone trying to do Free Software QA should always keep in mind.

3.4.1 Observation 1: volunteers and hackers are expensive, and bugs are cheap

You could also think of this as 'volunteers are scarce, bug filers are like locusts.' This has a number of implications for Free Software QA- many of the rules I've previously cited are almost the direct results of this observation. Many others I haven't cited also fall out of it. If you keep this simple observation (almost more a law than a rule) in mind, you'll find the others with time.

3.4.2 Observation 2: triage is an imperfect art

Despite the immediately previous suggestions about how to make triage consisten, it must be understood that triage is an imperfect art, where a certain amount of inconsistency is inevitable.

As already mentioned, the best way to triage is to read a lot of bugs first, to gain an appreciation for what types of bugs a project is seeing and how severe they are. But even after having read 20,000 or so bugs in the past year, over four projects, drawing the line even between seemingly simple things like 'is this an enhancement or a bug' is a frequent borderline judgement call for me.

Everyone involved in the QA process—bug reporters, bug fixers, and bug triagers (both casual and regular) must learn to accept this and work with it. The important lesson here is that volunteers should not be held to an impossible standard—both volunteers and developers must understand that differences of opinion will happen and aren't the end of the world. There will be thousands more bugs to work with if one gets screwed up. :)

3.4.3 Observation 3: QA is winning when people are interested in the process, not just the results

So how can one know when QA is starting to win? At what point can a QA volunteer sit back and say 'the hard part is done, now all I have to do is read bugs?' I'd suggest that one important metric is noting the point when the standard response by developers to bug reports is 'put it in Bugzilla.' GNOME moved very slowly in this direction, but that's now pretty much the standard response on mailing lists when a bug is reported to a list—'take it to Bugzilla.' There are other parts of the process as well—bug days, noting bug numbers in CVS commits or code comments, and an overall commitment by developers to working with QA volunteers.

4 And these rules apply to other large projects how?

4.1 XFree

A few months back, XFree had a discussion on their mailing list about use of Bugzilla to track XFree86 bugs. The response was... underwhelming. Why? The main fears were pretty straightforward: 'will we get lots of useless bug email?', 'will people try to control what we do?', and of course 'what benefit do we get?'

The answers to these questions aren't always obvious to a project just embarking on doing serious Free Software QA for the first time. Being more public can definitely open maintainers up to more mail. Obviously, this can happen—as I discussed, it's even possible that less buggy software will get more bug reports. That's not truly a requirement—even if Bugzilla is used only to triage and track bugs that come in through other forums (say, open only to developers, and used to track issues reported to a mailing list) it can still be of great use to a project, assuming that other rules I laid out about supporting developer goals and defining the triage process well are followed. GNOME actually allows anonymous bug submission, the opposite end of the spectrum, and while this is far from perfect, it has helped us make huge leaps and bounds in terms of stability by encouraging stack-trace submission.

Concerns about 'control' were equally unfounded—even borderline paranoiac. Xfree, like all other Free Software projects, is controlled by hackers and hackers alone. If a hacker decides that QA volunteers aren't to be trusted, or simply disagrees with triage decisions, they can ignore them and move on. The burden is on those running the QA process to prove that their bugs are valid and useful. I've given some suggestions on how this can happen already.

Finally, the most obvious and at the same time most difficult question—"what benefit do we get?" I got into Linux because I'd heard about, heard it didn't crash, and one night Outlook crashed 10 times, while I was trying to write a single email to a professor. So for me, more stable software is an obvious benefit of working in QA. That is, admittedly, not for everyone. Answering this question really requires some introspection on the part of hackers and maintainers—if you want to make software that is good for your users (virtually no matter

how you define good), then your project wants a QA process and wants Bugzilla. If you are in free software purely because you want to write cool hacks, or because in free software, no one can tell you what to do, Bugzilla may not be for your project. But that's an answer only you can answer. Frankly, on reading the XFree lists, it was not altogether clear that many of the XFree hackers were particularly concerned about their users. If that is the case (and that is most definitely their prerogative as authors of the code) then perhaps Bugzilla is not for them. No matter how hard they try, it would be hard for QA volunteers to support the pursuit of power or cool hacks.

4.2 Kernel

Reading kernel traffic, I was very pleased to see that Andrew Morton had proposed not just a list of bugs, but actually defined what he felt should be the standard for "when should we go to 2.6.0?" I'm a long time k-t reader, and this was the first time I'd seen something of the sort. Defining and agreeing on that is part of my Rule 0—QA has to support development, and developers have to tell QA what they want. The list had even been split into rudimentary "can't ship without fixing" and "it would be nice" lists—a big first step towards solid triage.

It was sort of disappointing, though, to read through the details—the vast majority of issues had no bugs associated with them, and squirelled down at the bottom was "and there are several hundred open Bugzilla bugs." This was the kind of opportunity kernel bug people should have seized on (and possibly have since this paper has been written, of course.) Bugzilla is perfectly designed to track these kinds of issues and their progress. Some intrepid volunteer could easily have volunteered to enter every issue into Bugzilla and assign it a high priority and assign it to the owner of the issue. From there, patches could have been attached and tracked, punting it from one list to another would have been as simple as changing a single field, outsiders could easily discover what bugs had and hadn't been fixed already, and a host of other things. Instead of ongoing IRC status meetings where many things were reported fixed or irrelevant, a simple query could have reported a list before the meeting that could be updated by all participants in parallel if need be. (And of course, no more diffs to show what had changed—again, simple, dynamic query to show what has changed over any period of time.)

Similarly, the "several hundred open Bugzilla bugs" was a great invitation for someone to work with Bugzilla, trawl them (it only takes a weekend, at worst, to read a couple hundred bugs, once you've got the knack) and start making preliminary suggestions to maintainers aobut important bugs that were in Bugzilla but not on the list. Remember rule two—persuade the hackers you are useful. Filling in the blanks on information they knew must be there but didn't have time to find themselves is a great step towards that, and reading the (currently small) open bugs to get perspective would have been a great start for those looking to help out and do effective triage.

Pre-release is the best time for QA and Bugzilla—priorities are typically clear cut, hackers are most pressed for time and so most appreciative of the help, and hackers are the most motivated to work on bug-fixing instead of pie-in-the-sky features. Hopefully, someone involved in the kernel community will find this general advice useful and can take advantage of this relatively rare time in the kernel cycle.

5 Conclusion

Free Software QA is a slightly different beast, playing with different sets of data inputs and

different sets of motivations than a typical QA process. As a result, making QA central to the release process is not easy for any Free Software project, and it's even harder to stay with it once it is successful, since success breeds difficulty. But it can be done if communication, motivation and technique are all brought in line with each other. GNOME did, and benefited immensely from it. It is my hope that other large projects will be able to learn from our lead.

6 Acknowledgments

Thanks to Ximian and Sun, for allowing me to work so extensively with the GNOME community.

Thanks to the Bugsquad and all the volunteers who preceded it, first for doing so much work for their own communities, and second for keeping me sane while I worked on Evolution and GNOME. And thirdly for suggesting the title of my talk.

Thanks to all those who proceeded me, at Mozilla and GNOME, for giving me something to work with—tools, skills, and data.

Thanks to ed on gimpnet, for helping me fight through the structure of this paper.

7 Availability

This paper and slides from the associated presentation will be available from

`http://tieguy.org/talks/`

References

[Wheeler] From David Wheeler's SLOCCount—`http://www.dwheeler.com/sloccount/`

Performance Testing the Linux Kernel

The re-aim workload

Cliff White
Open Source Development Labs
cliffw@osdl.org, http://www.osdl.org/archive/cliffw

Abstract

Good performance testing requires good tests and good procedures. This paper discusses experiences creating and using an automated test environment.

The paper also describes work done at Open Source Development Labs (OSDL™) in rewriting and modernizing the AIM7 and AIM9 benchmarks. The intent is to make the benchmarks relevant for modern hardware by making it flexible and extensible.

This paper talks about how to create a testing environment, how to automate it, and how to select and evaluate potential tests. The paper talks about the differences between low-level (micro) workloads and application-modeling (macro) workloads, using OSDL Scalable Test Platform tests as examples, and talk about the difference between tests that focus on specific areas and tests that exercise broad areas.

1 Introduction

Performance testing in kernel context is necessary to show that a projected improvement is in fact an improvement. Performance tests are used to measure large-scale application performance and small-scale system routine and system call performance.

There are two areas not specifically addressed by performance testing. One area is compliance which the Linux Standard Base and Linux Test Project test suites both address. The other area is reliability—demonstrating the ability to sustain proper operation over long time spans.

A goal of the OSDL's Scalable Test Platform is to measure performance, over and over again. To do this, we run publicly available workloads, and we create a few of our own. This paper describes work being done to revise an old workload suite, the AIM tests.

2 Creating a Proper Test Environment

A good test must be repeatable. It is very important that multiple runs of a test on the same hardware with the same kernel produce the same results. OSDL's STP creates this repeatable environment by re-loading the test machines with a new OS before every run. Thus, every test starts from an identical state. For non-STP testing, the system is set up for repeated runs by using a Makefile. Whenever a new test is created, the first thing created is a setup/tear-down Makefile. In the Makefile, careful track is kept of everything added to the system for the test.

When running the test, care should be taken to understand and control the test environment. There are a few areas to consider:

- Networking – This should be obvious, but any test of networking should be run on a private network, where no other traffic impacts the test measurement. This is especially important when a test is controlled or monitored via a network.

- Other shared resources – Might include shared storage arrays, or other devices. Again, it is best to use dedicated hardware or stop other users before testing.

- State of the system prior to test startup – This is especially important for repeating test results. Rebooting the system prior to every test run is one way to assure a known state. However, many tests are very influenced by cache effects and this must be considered. When testing database workloads, it is common to warm the database cache prior to taking any performance measurement.

- Repeat testing for repeatability – A single test result might be useful. A repeatable test result is much more usable. Statistically, three runs are about the minimum for good data, five or more runs are better.

2.1 Experiences from the STP

Here is some advice, culled from experiences adding tests to the OSDL's Scalable Test Platform.

- When scripting for automation, error recovery is everything. Error reporting is more important. Error discovery is most important. You will find that making things happen in a script is easy—knowing when things have not happened and doing the right thing thereafter is hard.

- Be very paranoid. Review and sanity check test results frequently. Hardware failures can be very sneaky; repeating known tests can be a good way to spot flaky hardware.

- When running a large or even a medium number of systems, administration tools are very important, especially tools that allow you to look at health over time.

- When testing kernels, sometimes the most interesting tests are the ones that do not run at all.

- Likewise, be aware of timeout conditions—the tests that never complete can also be interesting. You should have timeout conditions for each phase of an automated process.

- Build the tools to parse and present results when you build the test. If possible, build the tool so you can compare multiple runs.

- Likewise, instrument the test when you build the test. Add readprofile or oprofile if possible. However, be aware of the impact of your instrumentation; touching `/proc` too frequently can impact your results.

- Results presentation is very important. design the report so that the most important data is the easiest to see.

- Establish a baseline run you can use for comparison purposes.

- Compare frequently to that baseline. Test results in isolation are less interesting than comparisons to known conditions.

- Establish your hardware baselines in as much detail as you can. In a perfect world, what is the maximum rate your disk subsystem can deliver? Knowing these rates

can help you determine when a test is using real hardware and when a test is running from disk cache.

2.2 Macro and Micro Workloads

STP uses two very different types of tests when testing kernel performance. These tests are divided into macro and micro workloads. Micro workloads are tests that exercise a very small piece of the system, such as a single system call. These tests focus on the low-level performance details.

A macro workload is a simulation of a real-world task. Macro workloads are sometimes created from real customer workloads, or may be designed to a specification, such as the Transaction Processing Performance Council's TPC-N specification. These large-scale workloads might include OLTP applications, decision support systems or reservation and inventory systems. (OSDL's macro workloads include the Database test suite—the subject of another paper at this conference.)

It is important that we do not confuse the results of macro and micro workloads, or attempt to extrapolate too much real-world behavior from micro measurements. Micro benchmarks are usually developer-focused and not very useful for understanding customer needs.

When looking at macro benchmarks, avoid confusing simulation with reality, and extrapolating results beyond the specific configuration and problem tested. Many macro benchmarks are grounded in real customer needs and situations, but some are designed more for marketing price/performance comparisons.

2.2.1 The AIM Suite

The AIM suite was created by AIM Technology in the 1980's. The AIM company sponsored a yearly 'Hot Iron' [DEC] award for hardware manufactures, with prizes awarded in various price/performance categories. To quote from a press release[HP]:

> Since 1981, AIM Technology has provided vendors and end users comprehensive, unbiased performance testing to help users determine the best fit between their application needs and available systems and configurations.. . . AIM is an independent organization, as opposed to a vendor consortium, which allows AIM to bring an expert eye to performance measurement, not restrained by objectives of consortium members.

The company no longer exists and the awards are no longer being given out. SCO acquired rights to the AIM technology in 2000, and placed suites VII and IX of the test under the GPL. From the SCO web page[SCO]:

> The AIM benchmark technology has proved useful for more than a decade in measuring performance of hardware and versions of the Unix operating system. The benchmarks were licensed by nearly all of the vendors of Unix system hardware. More than 70 companies used these benchmarks to compare and tune products. In addition, because of the stressful multidimensional nature of the AIM workload many OS and hardware vendors have used the benchmarks as part of their quality assurance process.

The AIM suite combines features of both macro and micro tests. The suite consists of a list of sub-tests, also known as "jobs." Each job exercises a specific area of system functionality, such as file I/O, shared memory, process creation, and compute-intensive math tasks. Each job consists of a C function which is linked to the driver code. A job may loop repeatedly internal to the C function. (For example, each addition test does 1.5 million internal loops.) Lists of jobs are grouped into a "workload," contained in a workload file.

The test runs in two modes. In the single-user mode (AIM suite IX), each job in the workload file is executed serially by the test driver. An alarm is set, and the job is executed repeatedly until the alarm expires. The alarm time is referred to as the "test period." Performance is calculated by multiplying the number of job executions by the internal job loop count, and dividing by the test period. Results reported are iterations per test period and operations per second for each job.

In the multiuser mode, the driver forks a number of subprocesses, giving each an identical list of jobs. The length of the job list is variable, the default is 100 jobs per child. The jobs are identical to the single user case. Each job is given a weight (the 'hit' count). This weight is used to calculate the fraction of the total work performed by each subtest, the total work is the sum of all job weights.

A typical test executions consists of a series of passes wherein the number of subprocesses is increased on each pass. Each subprocess runs a randomly-ordered set of jobs until its list is exhausted. The driver waits for all the child processes to exit, and records the time between child start and child exit. From this data two numbers are calculated—the jobs per minute (JPM) and jobs per child process per minute (JPM_C)[SGI].

As the system load increases the jobs per minute increases until it reaches a peak. If the number of child processes continues to increase, the work per child per minute begins to decrease. Depending on the command line options, the test run terminates when child work decreases below a threshold or the number of child processes reaches the maximum desired. Results reported are parent time, total child time, jobs per minute, and jobs per minute per child.

The AIM suite provides a set of building blocks (the sub-tests) that can be combined to create a simulation of a real-world workload. The old test has several examples of these workloads, including simulations of databases, file servers, and compute servers. The workload can be adjusted by altering test weight or changing the test mix.

3 Re-aim – AIM rework

3.1 The driver

The AIM code has remained untouched since 1991. I re-wrote the driver portion of the code so that I could understand it, maintain it and enhance the list of sub-tests. After studying the old code for a time, I choose to write a new driver, preserving as much of the functionality of the old driver as was useful. No doubt a different coder could have continued to maintain the existing structure, I choose not to.

The old driver used global data structures and static defines to control the size of the test list, the number of test arguments, and other details. The static definitions were replaced with dynamicly linked lists for flexibility. The AIM7 and AIM9 tests use almost the same list of tests, so a common driver was desirable.

For convenience, the GNU autoconf tools were used for the build and install system. The fol-

lowing parts of the old AIM framework are essentially unchanged:

- The method of statically linking test modules to the driver engine code, and calling those modules through a function pointer.
- The method for calculating workload task distribution and weighting.
- The method for calculating disk file size and distribution.
- The majority of the test module code (not changed at this time).
- The method of calculating the results is unchanged, however the timing method and location of timestamps relative to driver sleep() has.
- The adaptive timer remains the default.

The current driver has the following options, shown in Table 1 which may help explain usage.

Most of the parameters can be specified in a configuration file, options in this file are ignored if the command line option is present. Disk directories and disk file sizes must be specified in a configuration file.

Several things were noted while re-doing the driver.

- The old multi-user test ran until the jobs/child/minute was less than 1.0. This is quite a load on modern systems, resulting runs greater than eight hours to attain convergence. This length of a run is generally not useful for such a performance test so the default crossover is jobs/child/minute less than 10.0, with a second switch to set this to a quick test

Options	Description
-d(x), –debug(x)	Turns on debugging output, 1 is default
-v, –verbose	Produces more output
-s(x), –startusers	Number of users at start
-e(x), –endusers	Number of users at end
-i(x), –increment	Number to increment by
-f(s), –file(s)	Workfile name, (default 'workfile')
-l(s), –list(s)	Config file name, (default 'reaim.config')
-c, –crossover	Run to crossover, (JPM/user less than 10.0)
-q, –quick	Run to quick crossover, (JPM/user less than 60.0)
-x	Runs until max JPM detected
-j(x),–jobs(x)	Number of jobs in tasklist, (minimum is workfile size)
-m, –multiuser	AIM7 style, default
-t, –timeroff	No adaptive timer
-o, –oneuser	Runs AIM9 style single thread
-p, –period	Length for single thread
-r, –repeat	Iterate entire test
-h, –help	This message

Table 1: Re-aim Options

value of 60.0. On a 2-CPU 800MHZ Pentium III system, the quick test converges at 15→50 users, depending on workload. The default crossover point is 50→200 users depending on the workload.

- The jobs per child is now adjustable, with a default value of 100. This can be used to cause a set number of child processes to do more or less work without changing the workload.

- In a perfect world, all children (doing equal work) should receive equal favor from the scheduler. In reality, as the number of children exceeds the number of CPUs, unfairness occurs and the child exit is serialized. In addition, the child exit timing is collected serially by the parent using `wait()`. The maximum and minimum child exit times are recorded to reflect this. This variance also appears in the standard deviation calculated by keeping a running total across all child exits.

- Timestamps are collected with the `times()` function. The parent time figure is effectively wall clock time for the test. This function also allows us to extract the system and user time as seen by the child process. This information is reported as a running total. The child time thus exceeds the parent time in the report.

- Filesize and poolsize (see below) are now set in the configuration file. If either is specified in the workfile, that setting overrides the configuration file, maintaining old behavior.

- A method for detecting the maximum jobs per minute was added. When the `-x` option is used, the jobs per minute rate is tested by taking the standard deviation across the last five test iterations. If the standard deviation is less than 1.0 percent of the average, the test exits. In addition, if the the JPM rate drops more than 1.0 below the average, the test exits. Maximum jobs per minute are always reported.

3.2 Math tests

Time changes everything. Years ago, when computing was frequently referred to as "number-crunching," math performance was an exciting topic. Today, in the kernel context, when run single threaded, these math tests tell us very little. Fluctuations in the single-user (AIM9) integer math test times are undoubtedly due to non-math causes, and do not typically reflect a change in the kernel. The multiuser is a bit different—when we examine the multiuser case we see that all these test run entirely in user space. If we think of each subtest as a part of a larger workload, these user space functions are quite useful.

Table 2 shows typical parent times and child system and user times when running these tests on a 2-CPU system (Linux-2.4.18).

Test_Name	Parent	Child Sys	Child usr
add_short	5.96	0.00	1.18
add_double	15.71	0.00	3.13
add_float	10.58	0.00	2.09
add_int	16.90	0.00	3.37
add_long	16.93	0.00	3.38
mul_short	0.50	0.00	0.09
mul_long	0.42	0.00	0.07
mul_int	0.40	0.00	0.07
mul_float	17.43	0.00	3.48
mul_double	17.55	0.00	3.48
div_double	15.40	0.00	3.07
div_float	15.83	0.00	3.07
div_int	18.94	0.00	3.76
div_long	18.83	0.00	3.76
div_short	18.94	0.00	3.76

Table 2: Re-aim Math Tests

Number Forked	Parent Time	Jobs per Minute
Without math tests		
10	75.23	797.55
With math tests		
10	58.07	1064.23

Table 3: Database Load Comparisons

Number Forked	Parent Time	Jobs per Minute
Equal Weight		
10	39.17	1531.78
20	66.41	1806.96
Disk:math - 4:1		
10	57.38	1045.66
20	91.33	1313.92
Disk:math - 1:4		
10	26.53	2261.59
20	49.07	2445.49

Table 4: Effects of Test Weight

Adding these user space workloads to the multiuser test produces these results, shown in Table 3:

This appears a bit counter-intuitive—we have a longer test list, but it runs faster! Remember that the number of tests per child is constant (100 in this case). Adding the short user-space math tests to the workload actually decreases the amount of work per child. Here are some further examples of how changing a simple mix can change the run time. We'll start with four tests, equally weighted, then we will set the disk test weight to four times the weight for the math tests, then do the reverse. Results shown in Table 4:

There are fifteen of these math tests, all are tight loops. No changes in these tests are planned.

Num Forked	Parent Time	Child SysTime	Child UTime
10	23.70	7.64	4.10
20	26.29	15.30	8.27
30	29.02	23.02	12.30
40	31.55	31.48	16.38
50	35.91	39.54	20.33

Table 5: High System Time Load

3.3 Other Tests

The math tests are notable for consuming mostly user time. There is another list of tests that consume mostly system time. These tests include the various memory tests (brk, shared memory) and the various system call tests. (create/close, link, fork, exec.) Combining these tests into into a single workload does consume more system time, as seen in Table 5:

The current list of system-call focused test is a bit short. Repeated runs of various workloads have not yielded memory consumption at reasonable user levels.

Another current question involves the shell_rtn tests, which currently use the shortest possible shell script. In addition, the three functions calling the shell are identical. The reason for this duplication is unknown.

The intent is to examine other open sources of test routines for incorporation into this run framework.

3.4 Disk Tests

The disk tests in the old AIM test consist of three groups: basic block I/O tests, the same tests with an added sync, and the sync I/O tests. Each test determines file size from a global variable, `disk_iteration_count`. There are two configuration variables that control this, FILESIZE and POOLSIZE (speci-

fied in kilobytes or megabytes). If POOLSIZE is zero, each child will write or read a total of FILESIZE bytes. If POOLSIZE is non-zero, child file size is equal to FILESIZE + (POOLSIZE/number_of_children). Thus when POOLSIZE is non-zero, I/O per child will be reduced on each increase in child count.

For example, specifying a FILESIZE of 10K and a POOLSIZE of 100K will result in a single child creating a 110K byte file on each disk device listed. Two children will create a 60K file, etc. 24 children will create a 14K file, consuming 328KB per disk device.

The old AIM tests follow this sequence:

- `creat()` file
- write file
- `close()` file descriptor
- `open()` file descriptor
- do test

This results in the disk test running entirely from cache. I added a second set of disk tests using this method:

- `creat()` file
- write file
- `close()` file descriptor
- `sync()`
- `open()` file descriptor
- do test

This simple change noticeably impacted performance:

		Random Disk Writes
disk_rw	without sync()	21922 (1K) per second
disk_rw	with sync()	1218.78 (1K) per second

The first number is more indicative of real-world hardware performance, but the cache-only version of the tests may be of greater interest to kernel developers.

The third category of disk tests performs the same operations, but descriptors are opened with the O_SYNC flag. (The read-only test is not performed, of course.) This test is of lesser interest, due to the relative slowness of O_SYNC.

The current disk tests do all IO at 1K block sizes. Future improvements to the disk test suite include:

- Tests that use O_DIRECT and raw IO.
- Tests that use a common file created during the test setup or prior to the test run, requiring noticeable non-cached IO.
- Tests that produce measurable read activity, period. This is a weakness of the cache-intensive design of the current tests. Many test runs show little or no real read IO—files are created, read and destroyed too quickly.
- Tests that attempt to consume a noticeable percentage of the cache.
- Temporary file creation is currently serialized, multiple devices should work in parallel.

The final test is disk_src, which does a series of directory searches. This test is of interest due to its use of dcache. Future enhancements include creating a script which will allow other trees to be searched by disk_src, in place of the current fakeh.tar.

Run Time	Change
2 seconds	2.39%
4 seconds	2.17%
8 seconds	2.02%
15 seconds	1.52%
30 seconds	1.46%
45 seconds	1.62%

Table 6: Single user variation—3 runs each

3.5 Comparison of AIM9 Duration

This comparison attempts to show the useful duration for the single user (AIM9) test run. A proper duration should produce stable results from run to run. To test this, a single user test was run three times using a list of fifty-four tests. Average change between tests was compared across the three runs, as shown in Table 6. (Note: Each test must complete one full loop.) While the run-to-run performance does stabilize slightly when the test duration exceeds fifteen seconds, run-to-run stability does not improve noticeably beyond that point. This has been reflected in the choice of default settings for the single user run duration (10 seconds).

4 Run results

4.1 List of the workloads

Appendix A has a list of the various workloads with run times on several sample configurations.

4.2 Comparisons – 2.5

Table 7 is a quick comparison of a 2.5 patch set, which is a subset of one of Martin Bligh's trees. We can see by this quick comparison that the patch does improve performance. The test

Forks	JPM-mjb	JPM	delta
10	1167.50	1074.15	8.3%
20	1240.24	1219.13	1.7%
22	1252.72	1219.14	2.7%
100	1247.36	1203.68	3.6%

Table 7: Comparison of 2.5.68 and 2.5.68-mjb0.5

was run on a small 2-CPU system, with 1GB of physical memory and IDE disks.

5 Conclusions

I have described the work that has been done to change from AIM to Re-aim. I intend to spend a great deal more time adding to the list of test cases and otherwise improving the usefulness of the tests.

6 Availability

The Re-aim code is available from Sourceforge:

```
http://sourceforge.net/
   projects/re-aim-7
```

Or via BitKeeper:

```
bk://bk.osdl.org/aimrework
```

7 Trademarks

8 Acknowledgements

Thanks to Ruth Forester and John Hawkes for advice, and OSDL for support.

References

[SCO] Web page announcing AIM suite release, `http://www.caldera.com/developers/community/contrib/aim.html`, 2000.

[HP] Press Release with AIM description, `http://www.compaq.com.hk/press/release/99press/990623.html`, 1995.

[DEC] Press Release with AIM description, `http://wint.decsy.ru/alphaservers/digital/v0000022.htm`, 1995.

[SGI] Ruth Forester, et al. "Filesystem Performance and Scalability in Linux 2.4.17," *Proceedings of the 2002 USENIX Annual Technical Conference*, Berkeley, CA 2002 `http://oss.sgi.com/projects/xfs/papers/filesystem-perf-tm.pdf`.

A Re-aim Results

A.1 Example runs

This appendix shows how various workloads perform on some sample systems. Workloads were run until max sustainable jobs were reached. The results shown below are the maximum users obtained by each workload. Several iterations are shown in some cases to demonstrate typical run termination—the adaptive timer was used for these runs. See the source package for a listing of each workfile. Some of the workload have arbitrary names reflecting time. This is not intended as a hardware comparison.

We notice that for several of the workloads, scaling is roughly linear across the three configurations. For other workloads, most noticeable the fserver and Dbase, performance on the quad system jumps markedly. However, the adaptive timer skews the increment such that comparisons may not be relied upon—any true comparison should be made without the adaptive timer. (The adaptive timer was used in this case to reduce total run time.) The additional disks on the Quad system appear to impact run times. The other systems under test use disks which are shared by the system. (`/tmp` or `/usr/tmp`) The quad system has 5 spindles of disk devoted to the tests. The actual test report includes data on standard deviation and confidence levels. These columns have been removed, due to text formatting requirements.

The systems:

1. Single CPU
   PIII - 600MHZ
   384KB RAM
   single IDE disk
   Linux-2.5.68 -stock
   FILESIZE 10k
   POOLSIZE 100k

2. Dual CPU
   PIII - 868MHZ
   1GB RAM
   Dual IDE disk
   Linux-2.5.68 - stock
   FILESIZE 10k
   POOLSIZE 1m

3. Quad CPU
   PIII - 700MHZ

4GB RAM
5 SCSI disks
Linux-2.4.20 - stock
FILESIZE 10k
POOLSIZE 1m

A.2 The Workloads

workfile.all_utime Table 8. All these tests run entirely in user space.

workfile.alltests Table 9. The full test list.

workfile.compute Table 10. From the old test. Simulation of a compute-intensive server. 31.7% of this workload are tests from the all_utime list.

workfile.dbase Table 11 Simulation of a database load. 21.8% percent of this workload are tests from the all_utime list.

workfile.disk Table 12. The disk tests with no other work. All tests in this list are weighted equally. Notice the difference between this workload and the fserver workload, which includes other subtests.

workfile.fivesec Table 13 A completely artificial grouping of tests, based on their run duration when tested on a UP system.

workfile.fserver Table 14 Simulation of a file server. 21.8% of this mix is 100% user time tests, which matches the dbase workfile.

workfile.fivesec Table 15 A completely artificial grouping of tests, based on their run duration when tested on a UP system.

workfile.shared Table 16. Simulation of a multi-user shared server, assumed to be supporting telnet clients. 39.7% of the work mix are 100% user time tests.

workfile.short Table 17 A completely artificial

Max Jobs per minute				
Single - 1044.37 (1 user)				
Dual - 2938.27 (7 users)				
Quad - 4896.00 (12 users)				
Num Forked	Parent Time	Child SysTime	Child UTime	Jobs per Minute
Single				
5	29.32	0.00	29.32	1043.66
Dual				
14	29.27	0.01	56.72	2927.23
Quad				
20	25.04	0.03	100.04	4888.18

Table 8: All User Time Workload

Max Jobs per minute				
Single - 1839.22 (118 users)				
Dual - 4233.31 (345 users)				
Quad - 7207.78 (281 users)				
Num Forked	Parent Time	Child SysTime	Child UTime	Jobs per Minute
Single				
222	674.92	79.09	583.36	1835.42
Dual				
545	727.78	288.98	973.43	4223.53
Quad				
281	217.54	219.01	611.41	7207.78
343	317.68	494.41	750.89	6024.74

Table 9: All Tests Workload

grouping of tests, based on their run duration when tested on a UP system.

Max Jobs per minute				
Single - 803.29 (5 users)				
Dual - 1429.25 (7 users)				
Quad - 4708.68 (753 users)				
Num Forked	Parent Time	Child SysTime	Child UTime	Jobs per Minute
Single				
6	45.57	1.90	43.63	797.89
Dual				
10	43.35	3.73	78.56	1397.92
Quad				
753	969.10	246.37	3620.94	4708.68
1007	1300.27	348.33	4838.78	4693.19

Table 10: Compute Workload

Max Jobs per minute				
Single - 806.63				
Dual - 1186.52				
Quad - 1124.23				
Num Forked	Parent Time	Child SysTime	Child UTime	Jobs per Minute
Single				
10	73.64	3.92	68.51	806.63
Dual				
53	265.33	38.30	457.17	1186.52
Quad				
383	2626.73	7089.40	2706.01	866.10

Table 11: Dbase Workload

Max Jobs per minute				
Single - 1059.20				
Dual - 2753.83				
Quad - 9723.69				
Num Forked	Parent Time	Child SysTime	Child UTime	Jobs per Minute
Single				
52	309.29	19.06	11.38	1059.20
65	396.09	24.01	14.32	1033.86
Dual				
259	592.52	272.62	45.07	2753.83
349	816.90	370.08	61.38	2691.52
386	927.06	423.03	67.48	2623.13
464	1131.55	518.55	81.22	2583.36
Quad				
352	374.70	766.36	53.39	5918.33
510	330.43	899.31	76.65	9723.69
807	867.55	3087.03	119.11	5860.30

Table 12: Disk Workload

Max Jobs per minute				
Single - 2014.13				
Dual - 3872.86				
Quad - 10995.93				
Num Forked	Parent Time	Child SysTime	Child UTime	Jobs per Minute
Single				
24	70.78	16.60	12.21	2014.13
32	101.06	22.16	16.34	1880.86
35	110.06	24.29	17.73	1888.97
41	130.73	28.48	20.89	1862.92
Dual				
136	208.59	158.24	51.78	3872.86
180	287.05	210.50	68.99	3724.79
198	319.77	231.98	75.75	3678.02
Quad				
432	265.98	698.35	170.78	9647.64
550	297.11	875.30	214.95	10995.93
799	1023.44	3577.29	314.54	4637.36

Table 13: FiveSec Workload

Max Jobs per minute				
Single - 1617.78				
Dual - 4267.88				
Quad - 149.06				
Num Forked	Parent Time	Child SysTime	Child UTime	Jobs per Minute
Single				
17	63.68	9.74	37.24	1617.78
19	72.01	11.17	41.57	1598.94
23	89.66	13.41	50.25	1554.54
Dual				
328	465.73	241.80	483.01	4267.88
367	525.04	272.13	540.58	4235.91
449	639.48	339.93	661.62	4254.93
531	756.40	402.28	782.39	4254.18
Quad				
141	5732.32	4832.17	283.59	149.06
145	5968.47	4803.54	290.69	147.22
146	6112.96	5288.60	292.91	144.74

Table 14: Fserver Workload

Max Jobs per minute				
Single - 952.59				
Dual - 2945.31				
Quad - 5007.89				
Num Forked	Parent Time	Child SysTime	Child UTime	Jobs per Minute
Single				
12	81.63	3.42	69.14	952.59
Dual				
187	411.42	57.24	730.43	2945.31
Quad				
48	62.11	12.72	231.40	5007.89

Table 15: Long Workload

Max Jobs per minute				
Single - 1177.14				
Dual - 2232.94				
Quad - 2153.06				
Num Forked	Parent Time	Child SysTime	Child UTime	Jobs per Minute
Single				
12	59.33	5.09	51.69	1177.14
16	80.37	6.84	68.84	1158.64
Dual				
28	72.98	23.81	95.95	2232.94
34	103.61	26.71	116.53	1909.85
Quad				
132	520.91	386.16	436.88	1474.80
182	624.65	585.43	628.17	1695.73
291	786.61	1135.61	1002.55	2153.06
400	1409.01	3649.24	1380.71	1652.22

Table 16: Shared Workload

Max Jobs per minute				
Single - 45333.33				
Dual - 166909.09				
Quad - 222545.45				
Num Forked	Parent Time	Child SysTime	Child UTime	Jobs per Minute
Single				
6	0.82	0.38	0.42	44780.49
Dual				
9	0.33	0.27	0.39	166909.09
Quad				
4	0.11	0.24	0.23	222545.45
8	0.26	0.47	0.45	188307.69

Table 17: Short Workload

Stressing Linux with Real-world Workloads

Mark Wong
Open Source Development Labs
markw@osdl.org

Abstract

Open Source Development Labs (OSDL™) have developed three freely available real-world database workloads to characterize the performance of the Linux kernel. These customizable workloads are modeled after the Transaction Processing Performance Council's industry standard W, C, and H benchmarks, and are intended for kernel developers with or without database management experience. The Scalable Test Platform can be used by those who do not feel they have the resources to run a large scale database workload to collect system statistics and kernel profile data. This paper describes the kind of real world activities simulated by each workload and how they stress the Linux kernel.

1 Introduction

OSDL[1] is dedicated to providing developers with resources to build enterprise enhancements into the Linux® kernel and its Open Source solution stacks. This paper focuses on real-world database workloads, based on industry standard benchmarks, used to stress Linux.

OSDL currently provides three workloads, derived from specifications published by the Transaction Processing Performance Council[2] (TPC). TPC is a non-profit corporation created to define database benchmarks with which performance data can be disseminated to the industry.

TPC benchmarks are intended to be used as a competitive tool. All published TPC benchmark results must comply with strict publication rules and auditing to ensure fair comparisons between competitors. Furthermore, it is required that all the hardware and software used in a benchmark must be commercially available with a full disclosure report of the pricing of all the products used as well as support and maintenance costs.

As a basis for our three workloads, we used the TPC Benchmark* W (TPC-W*), TPC Benchmark C (TPC-C*), and TPC Benchmark H (TPC-H*). Each benchmark is briefly described here. TPC-W is a Web commerce benchmark, simulating the activities of Web browsers accessing an on-line book reseller for browsing, searching or ordering. TPC-C is an on-line transaction processing (OLTP) benchmark, simulating the activities of a supplier managing warehouse orders and stock. TPC-H is an ad hoc decision support benchmark, simulating an application performing complex business analyses that supports the making of sound business decisions for a wholesale supplier.

It is impractical for most Linux kernel developers to adhere to TPC rules, simply for cost alone. The executive summaries of the pub-

[1] http://www.osdl.org/
[2] http://www.tpc.org/

lished results on the TPC Results Listing Web page [3] show system costs in the millions of dollars.

To illustrate, the highest performing TPC-W result at the 10,000 item scale factor[4] [TPCW10K] uses forty-eight dual-processor Web servers, twelve dual-processor and two single-processor Web caches, and one system with eight processors and approximately 150 hard drives for the database management system. This does not include the systems emulating Web browsers that drive the benchmark.

The highest performing TPC-C [TPCCRESULT] result uses thirty-two eight-processor system with 108 hard drives each, four four-processor systems with fourteen hard drives each for the database management system, and sixty-four dual-processor clients. This does not include the systems emulating the terminals required to drive the benchmark.

The highest performing TPC-H result at the 10,000 GB scale factor [TPCH10000GB] uses sixty-four dual-processor systems with four gigabytes of memory each and a total of 896 hard drives.

2 Database Test Suite

The Database Test Suite[5] is a collection of simplified derivatives of the TPC benchmarks that simulate real-world workloads in smaller scale environments than that of a full blown TPC benchmark. These workloads are not very well suited to compare databases or systems because the variations allowed in running the workload would result in an apples-to-oranges comparison. However, these workloads can be used to compare the performance between different Linux kernels on the same system.

The amount of database administration knowledge and resources needed to run one of these workloads may still be intimidating to some, but the OSDL Scalable Test Platform[6] (STP) offsets these concerns. How the STP can be used is discussed towards the end of this paper.

These tests were initially developed on Linux with SAP DB but each test kit is designed to allow them to be usable with any database, such as MySQL[7] or PostgreSQL[8], with some porting work. Members of the PostgreSQL community are currently contributing to the Database Test Suite so that it may be used with PostgreSQL and also run on FreeBSD[9].

Each test provides scripts to collect system statistics using sar, iostat, and vmstat. Database statistics for SAP DB are also collected by using the tools that are provided with the database. The data presented for each workload in this paper are primarily profile data to show what parts of the Linux kernel is exercised. Keep in mind that the profile data presented here characterizes the workload for a specific set of parameters and system configurations. For example, as we review the profile data we will see that Database Test 2 appears to be stressing a SCSI disk controller driver, yet the workload can be customized so that the working set of data can fit completely into memory to put the focus of the workload on the system memory and processors as opposed to the storage subsystem.

[3] Current results can be viewed on the Web at: http://www.tpc.org/information/results.asp

[4] The *scale factor* determines the initial size of the database.

[5] http://www.osdl.org/projects/performance/

[6] STP can be accessed through the Web at http://www.osdl.org/stp/

[7] http://www.mysql.com/

[8] http://www.postgresql.org/

[9] http://www.freebsd.org/

2.1 Database Test 1 (DBT-1)

Database Test 1 (DBT-1) is derived from the TPC-W Specification [TPCW], which typically consists of an array of Web servers, hosting the on-line reseller's store front, that interfaces with a database management system, shown in Figure 1. An array of systems is also required to support the benchmark by simulating Web browsers accessing the Web site.

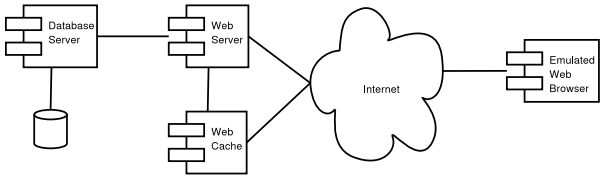


Figure 1: TPC-W Component Diagram

DBT-1, on the other hand, focuses on the activities of the database. There are no Web servers used, and a simple database caching application can be used to simulate some of the effects that a Web cache would have on the database. Since no Web servers are used, emulated Web browsers are not fully implemented. Instead, a driver is implemented to simulate users requesting the same interactions against the database that a Web browser would make against a Web server.

Figure 2 is a component diagram of the programs used in DBT-1. The Driver is a multi-threaded program that simulates users accessing a store front. The Application Server is another multi-threaded application that manages a pool of database connections and handles interaction requests from the Driver. The Database Cache is yet another multi-threaded program that extracts data from the database before the test starts running. If the caching component is not used, the Application Server queries the database directly. Each component of DBT-1 can run on separate or shared systems, in other words, in a three-tier or one-tier environment.

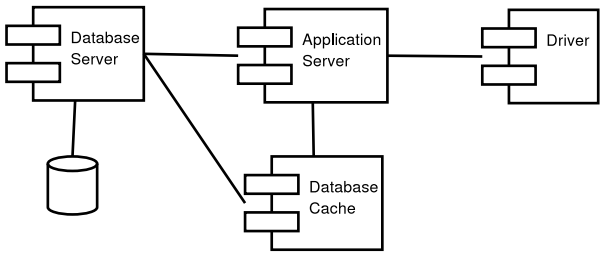


Figure 2: DBT-1 Component Diagram

The emulated users behave similarly to the TPC-W emulated browsers by executing interactions that search for products, make orders, display orders, and perform administrative tasks. There are a total of fourteen interactions that can occur, as shown in Table 1 with their frequency of execution. Each interaction, except Customer Registration, causes read-only or read-write I/O activity to occur. The overall effect is that 80% of the interactions executed cause read-only I/O activity while the remaining 20% of the interactions execute also cause writing to occur. Admin Request, Best Sellers, Home, New Products, Order Inquiry, Order Display, Product Detail, Search Request, and Search Results are the read-only interactions. Admin Confirm, Buy Request, Buy Confirm, and Shopping Cart are the read-write interactions. Customer registration does not interact with the database. It is maintained in the workload to keep the interaction mix close to the TPC-W specification.

An emulated user maintains state between interactions to simulate a browser session. A session lasts an average of fifteen minutes, which can be tester defined, over a negative exponential distribution. The state maintained includes the emulated user's identification and a shopping cart identifier. Each emulated user also randomly picks a think time from a negative exponential distribution, with a specified average, to determine how long to sleep between

Interaction	Executed
Admin Confirm	0.09 %
Admin Request	0.10 %
Best Sellers	11.00 %
Buy Confirm	0.69 %
Buy Request	0.75 %
Customer Registration	0.82 %
Home	29.00 %
New Products	11.00 %
Order Display	0.25 %
Order Inquiry	0.30 %
Product Detail	21.00 %
Search Request	12.00 %
Search Results	11.00 %
Shopping Cart	2.00 %

Table 1: DBT-1 Database Interactions Mix

interactions.

This workload generally exhibits high processor and memory activity mixed with with networking and low to medium I/O activity that is mostly read-only. The parameters that can be controlled to alter the characteristics of the workload are the scale factor of the database, the number of connections the Application Server opens to the database, the number of emulated users created by the driver, and the think time between interaction requests.

The following results for DBT-1[10] are collected from a four-processor Pentium III Xeon* system with 1 MB of L2 cache and 4 GB of memory in the OSDL STP environment. The system is configured in a one-tier environment with SAP DB using a total of eleven raw disk devices and to run against Linux 2.5.67. A database with 10,000 items and 600 users is created, where 400 users are emulated using an average think think of 1.6 seconds

Table 2 displays the top 20 functions called sorted by the frequency of clock ticks. The profile data is collected using readprofile where the processors in the system become 100% utilized, as shown in Figure 3. Other than the scheduler, we can see that TCP network functions are called most frequently in this workload.

Function	Ticks
default_idle	944946
schedule	15835
__wake_up	7376
tcp_v4_rcv	6871
__copy_to_user_ll	6456
tcp_sendmsg	6386
mod_timer	4666
__copy_from_user_ll	4482
tcp_recvmsg	4297
__copy_user_intel	3602
tcp_transmit_skb	3369
ip_queue_xmit	3230
dev_queue_xmit	3105
ip_output	2936
__copy_user_zeroing_intel	2806
tcp_data_wait	2711
tcp_rcv_established	2675
generic_file_aio_write_nolock	2614
fget	2536
do_gettimeofday	2420
...	...
total	1124848

Table 2: DBT-1 Profile Ticks

Table 3 displays the top 20 functions with the highest normalized load, which is calculated by dividing the number of ticks a function has recorded by the length of the address space the function occupies in memory. By comparing Table 2 and Table 3, we can see that `__copy_to_user_ll` and `__copy_from_user_ll` appear in both tables. This implies that the user address space is accessed frequently in this workload.

[10] http://khack.osdl.org/stp/271067/

Function	Load
default_idle	14764.7812
__wake_up	153.6667
__copy_to_user_ll	57.6429
system_call	52.5682
get_offset_tsc	45.9688
syscall_exit	40.1818
__copy_from_user_ll	40.0179
fget	31.7000
restore_fpu	30.6875
fput	26.4062
ipc_lock	24.6250
__copy_user_intel	22.5125
sock_wfree	19.6094
__copy_user_zeroing_intel	17.5375
local_bh_enable	16.7396
mod_timer	16.2014
schedule	15.4639
do_gettimeofday	15.1250
sockfd_lookup	14.8393
device_not_available	13.4146

Table 3: DBT-1 Normalized Profile Load

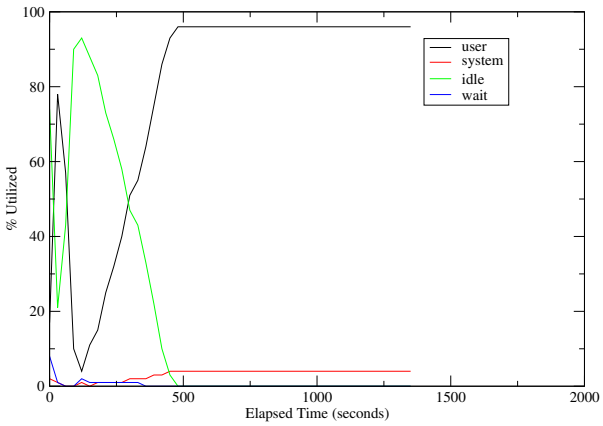


Figure 3: DBT-1 Processor Utilization

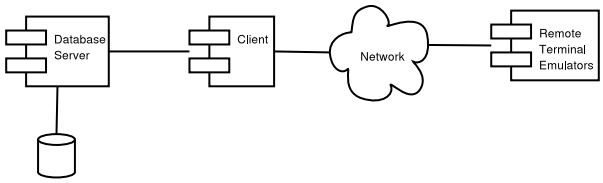


Figure 4: TPC-C Component Diagram 1

2.2 Database Test 2 (DBT-2)

Database Test 2 (DBT-2) is derived from the TPC-C Specification [TPCC], which typically consists of a database server and a transaction manager used to access the database server. There is also an array of systems required to support the benchmark by simulating terminals accessing the database. This benchmark can be run in one of two configurations, as shown in Figure 4 and Figure 5. The only difference between these two configuration is that the emulated terminals access the database through a transaction manager, labeled as the *Client* in Figure 4, while the emulated terminals access the database directly in the Figure 5.

DBT-2 can be also be configured to run in one of two ways. The first way is shown in Figure 6 where the Driver, a multi-threaded program that creates a thread for every terminal emulated, accesses the database through the Client program, a transaction manager that is a multi-threaded program that manages a pool of database connections. The second way, shown in Figure 7, combines the functionality of the Client program into the Driver program so that the driver can connect directly to the database. In either case, the workload can be run in a single- or multi-tier environment.

DBT-2 consists of five transactions that create orders, display order information, pay for orders, deliver orders, and examine stock levels. Table 4 lists each transaction and the frequency that each is executed by the emulated terminals. The Deliver, New-Order, and Payment transactions are read-write transactions, while the Order-Status and Stock-Level transactions are read-only transactions.

This workload can be customized so that the

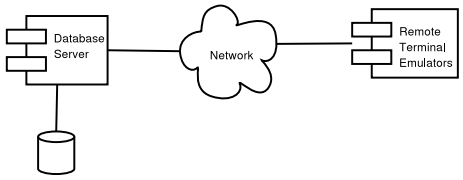


Figure 5: TPC-C Component Diagram 2

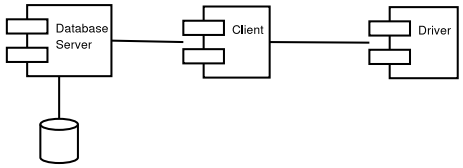


Figure 6: DBT-2 Component Diagram 1

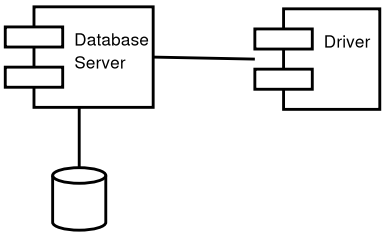


Figure 7: DBT-2 Component Diagram 2

Transaction	Executed
Delivery	4.0 %
New-Order	45.0 %
Order-Status	4.0 %
Payment	43.0 %
Stock-Level	4.0 %

Table 4: DBT-2 Database Transaction Mix

working set of data is contained completely in memory. If the working set of data is cached completely in memory, the system puts a heavy load on processors and memory usage. In situations where the working set of data is not completely cached, random I/O activity increases, while in both cases, sequential writes to the database logging device occurs throughout the test.

There are several parameters that can be customized to alter the characteristics of the workload. There are constant keying and thinking times between interactions that can be tester defined. The keying time simulates the time taken to enter information into a terminal and the thinking time simulates the time taken for a tester to determine the next transaction to execute.

By default, every terminal that is emulated is assigned a district and a warehouse to work out of. This can be changed so that a terminal randomly picks a warehouse and district in a specified range for every transaction. The effect this has on the workload is that a single emulated terminal is likely to access a greater amount of data in the database over the course of a test. Given the same amount of memory, this would create a workload less likely to be cached in memory and more likely to incur an increased amount of I/O activity. It would also create more lock contention in the database. By reducing the number of emulated terminals and by limiting the range of data an emulated terminal accesses in the database, the workload can be cached into memory, effectively creating a workload that only performs synchronous writes on the database logging device.

The following results for DBT-2[11] are collected from a four-processor Pentium III Xeon system with 1 MB of L2 cache and 4 GB of memory in the OSDL STP environment. The system is configured in a one-tier environment with SAP DB using twelve raw disk devices to run against Linux 2.5.67. Sixteen terminals are emulated to randomly select a district across six distinct warehouses with a keying and thinking time of zero seconds.

Table 5 displays the top 20 functions called

[11] http://khack.osdl.org/stp/271071/

sorted by the frequency of clock ticks and Table 6 displays the top 20 functions with the highest normalized load. If we compare these tables like we did with the results from DBT-1, we see that `bounce_copy_vec`, `__blk_queue_bounce`, `scsi_request_fn`, `scsi_end_request`, and `page_address` appear in both tables. The `default_idle` function also appears at the top of both tables. This implies that the processors on the system are not fully utilized and that the system may be stressing the storage subsystem. Figure 8 confirms that the processors are approximately 5% to 10% idle and are approximately 40% to 50% busy waiting for I/O.

Function	Ticks
default_idle	5695427
bounce_copy_vec	84136
schedule	55663
__blk_queue_bounce	28391
scsi_request_fn	23052
do_softirq	21760
__make_request	20124
try_to_wake_up	10511
scsi_end_request	10161
system_call	9734
dio_bio_end_io	9241
scsi_queue_next_request	9066
ipc_lock	6856
sys_semtimedop	5858
do_anonymous_page	5693
kmem_cache_free	5587
free_hot_cold_page	4768
page_address	4752
buffered_rmqueue	4648
try_atomic_semop	4406
...	...
total	6211231

Table 5: DBT-2 Profile Ticks

Function	Load
default_idle	88991.0469
bounce_copy_vec	1051.7000
system_call	221.2273
syscall_exit	105.5455
do_softirq	104.6154
ipc_lock	85.7000
dio_bio_end_io	82.5089
kmem_cache_free	69.8375
scsi_end_request	63.5063
__wake_up	55.9375
schedule	54.3584
get_offset_tsc	54.1562
__blk_queue_bounce	47.9578
bio_put	46.3542
scsi_request_fn	42.3750
fget	37.5125
restore_fpu	36.7812
page_address	33.0000
generic_unplug_device	29.7143
device_not_available	29.6098

Table 6: DBT-2 Normalized Profile Load

2.3 Database Test 3 (DBT-3)

Database Test 3 (DBT-3) is derived from the TPC-H Specification [TPCH], which typically consists of a single database server that is queried by an application in a host-based or client/server configuration, as shown in Figure 9 and Figure 10.

There are twenty-two queries that provide business analyses for pricing and promotions, supply and demand management, profit and revenue management, customer satisfaction, market share, and shipping management. In addition to the twenty-two queries, there are two refresh functions that load new sales information into the database.

This workload consists of loading a database, running a series of queries against the database,

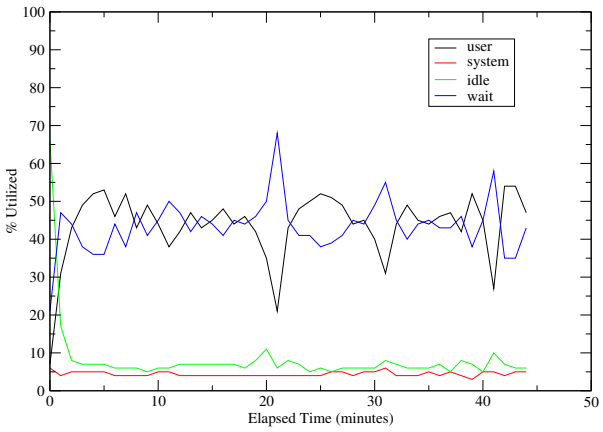


Figure 8: DBT-2 Processor Utilization

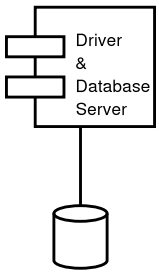


Figure 9: TPC-H Host-Based Component Diagram

and loading new sales information into the database. There are three distinct tests in which these actions occur, the Load Test, Power Test, and Throughput Test. The Load Test creates the database tables and loads data into them. The Power Test executes each of the twenty-two queries and two refresh functions sequentially. The Throughput Test executes a specified number of processes that executes each of the twenty-two queries in parallel and an equal number of processes that executes only the refresh functions.

There are several ways that this workload can be customized. The scale factor of the database can be selected so that at some point during a test, the working set of data becomes cached into memory. The number of streams

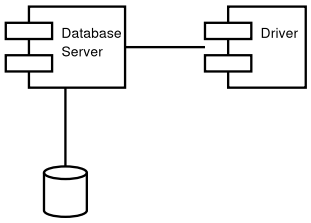


Figure 10: TPC-H Client/Server Component Diagram

for the throughput test may have to be adjusted according to the available system resources. The TPC-H Specification requires a minimum number of streams to be used depending on the scale factor of the database and ideally the number of streams should be selected so that the highest throughput metric can be achieved. However, for DBT-3, selecting the number of streams can be determined by how the Linux kernel is stressed by the workload. In any case, if the working set of data is not cached, large sequential I/O activity occurs in the Power and Throughput Test. Also, each of the twenty-two queries can be modified to meet different needs. For example, a query can be modified to answer another type of business question.

The following results for DBT-3[12] are collected from a four-processor Xeon* system with 256 KB of L2 cache and 4 GB of memory in the OSDL STP environment. The system is configured in a host-based environment running Linux 2.5.67 with hyper-threading enabled and a patch that allows the DAC960 driver to have direct memory access into high memory. A 1 GB database was created and only statistics from a Throughput Test with eight streams are reported here.

Table 7 displays the top 20 functions called sorted by the frequency of clock ticks. Similar to DBT-2, the prominence of

[12] http://khack.osdl.org/stp/271071/

`default_idle` with `__make_request` and `DAC960_BA_InterruptHandler` suggests that the system is also busy waiting for I/O. Table 8 displays the top 20 functions with the highest normalized load and again, similar to DBT-2, seeing `DAC960_BA_InterruptHandler` as one of the more prominent functions on this list also supports the theory that the kernel is spending a significant amount of time attempting to process I/O requests. Figure 11 shows that the processors are waiting for I/O 40% to 80% of time throughout the middle of the Throughput Test.

Function	Ticks
poll_idle	24160697
__make_request	21161
schedule	16354
generic_unplug_device	14701
DAC960_LP_InterruptHandler	12160
system_call	7361
kmap_atomic	3783
fget	3148
get_user_pages	3099
do_direct_IO	2968
dio_await_one	2955
bio_alloc	2850
device_not_available	2767
direct_io_worker	2551
blockdev_direct_IO	2226
find_vma	2020
__generic_file_aio_read	1924
follow_page	1891
__copy_to_user_ll	1817
...	...
total	24322403

Table 7: DBT-3 Profile Ticks

3 PLM and STP

Using the Database Test Suite can be an intimating task for those inexperienced with administering database management systems, or large systems may not be readily available for testing. Rather than implementing one of the Database Test Suite workloads on their own system, Linux kernel developers can test their kernel patches by using the Patch Lifecycle Manager[13] (PLM) and the STP. In order to use PLM or STP, you must sign up as an associate of the OSDL, free of charge, through the Web at `http://www.osdl.org/`.

Function	Load
poll_idle	383503.127
system_call	167.2955
generic_unplug_device	140.0095
DAC960_LP_InterruptHandler	74.1463
device_not_available	67.4878
fget	44.9714
kmap_atomic	34.3909
fput	31.1154
find_vma	24.3373
unlock_page	20.9059
restore_fpu	20.4857
get_offset_tsc	19.6667
syscall_call	19.4545
io_schedule	18.8542
__make_request	18.7431
dio_await_one	18.5849
current_kernel_time	18.5303
math_state_restore	17.7846
mempool_alloc_slab	15.9524
kmem_cache_alloc	15.8158

Table 8: DBT-3 Normalized Profile Load

PLM can be used to store patches for the Linux kernel that can be used by STP for testing. Currently, PLM automatically copies Linus Torvalds's tree as well as Andrew Morton's, Martin Bligh's, Alan Cox's, and the ia64 patch sets. PLM also executes filters against each patch entered into the system, to verify the patch ap-

[13] PLM can be accessed through the Web at http://www.osdl.org/cgi-bin/plm/

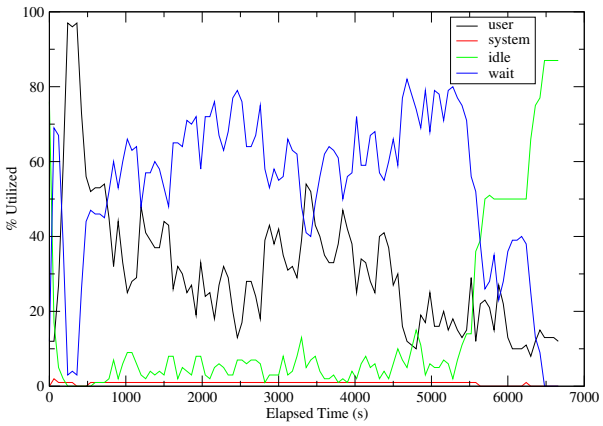


Figure 11: DBT-3 Throughput Test Processor Utilization

plies to a kernel or another patch, and verifies that the kernel can still be compiled with that patch.

STP currently implements all three of the workloads in the Database Test Suite[14]. DBT-1 can be run on systems with 2, 4 or 8 processors, DBT-2 and DBT-3 can be run on systems with 4 processor. The 4 and 8 processor systems also have arrays of external hard drives attached. Each test in STP generates a Web page of results with links to raw data and charts, as well as profile data if desired. E-mail notification is also sent to the test requester when a test has completed.

4 Comparing Results

While the OSDL Database Test Suite is derived from TPC benchmarks, results from the Database Test Suite are in no way comparable to results published from TPC benchmarks. Such comparisons should be reported to the TPC (`admin@tpc.org`) and to the OSDL (`wookie@osdl.org`).

[14]DBT-3 is currently being developed for STP and should be available by the time this paper is published.

References

[TPCC] *TPC Benchmark C Standard Specification Revision 5.0*, February 26, 2001.

[TPCH] *TPC Benchmark H Standard Specification Revision 1.5.0*, 2002.

[TPCH10000GB] *NCR 5350 Using Teradata V2R5.0 Executive Summary*, Teradata a division of NCR, March 12, 2003.

[TPCCRESULT] *ProLiant DL760-900-256P Client/Server Executive Summary*, Compaq Computer Corporation, September 19, 2001.

[TPCW] *TPC Benchmark W Specification Version 1.6*, August 14, 2001.

[TPCW10K] *Netfinity 5600 with Netfinity 6000R using Microsoft SQL Server 2000 Executive Summary*, International Business Machines, Inc., July 1, 2000.

Trademarks

Xr: Cross-device Rendering for Vector Graphics

Carl Worth
USC, Information Sciences Institute
cworth@isi.edu

Keith Packard
Cambridge Research Laboratory, HP Labs, HP
keithp@keithp.com

Abstract

Xr provides a vector-based rendering API with output support for the X Window System and local image buffers. PostScript and PDF file output is planned. Xr is designed to produce identical output on all output media while taking advantage of display hardware acceleration through the X Render Extension.

Xr provides a stateful user-level API with support for the PDF 1.4 imaging model. Xr provides operations including stroking and filling Bézier cubic splines, transforming and compositing translucent images, and antialiased text rendering. The PostScript drawing model has been adapted for use within C applications. Extensions needed to support much of the PDF 1.4 imaging operations have been included. This integration of the familiar PostScript operational model within the native application language environment provides a simple and powerful new tool for graphics application development.

1 Introduction

The design of the Xr library is motivated by the desire to provide a high-quality rendering interface for all areas of application presentation, from labels and shading on buttons to the central image manipulation in a drawing or painting program. Xr targets displays, printers and local image buffers with a uniform rendering model so that applications can use the same API to present information regardless of the media.

The Xr library provides a device-independent API, and can currently drive X Window System[10] applications as well as manipulate images in the application address space. It can take advantage of the X Render Extension[7] where available but does not require it. The intent is to add support for Xr to produce PostScript[1] and PDF 1.4[5] output.

Moving from the primitive original graphics system available in the X Window System to a complete device-independent rendering environment should serve to drive future application development in exciting directions.

1.1 Vector Graphics

On modern display hardware, an application's desire to present information using abstract geometric objects must be translated to physical pixels at some point in the process. The later this transition occurs in the rendering process the fewer pixelization artifacts will appear as a result of additional transformation operations

on pixel-based data.

Existing application artwork is often generated in pixel format because the rendering operations available to the application at runtime are a mere shadow of those provided in a typical image manipulation program. Providing sufficient rendering functionality within the application environment allows artwork to be provided in vector form which presents high quality results at a wide range of sizes.

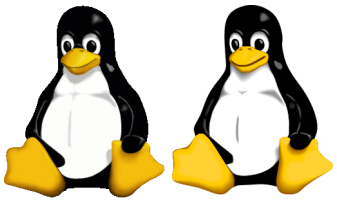


Figure 1: Raster and vector images at original size (artwork courtesy of Larry Ewing and Simon Budig)

Figures 1-3 illustrate the benefits of vector artwork. The penguin on the left of Figure 1 is the familiar image as originally drawn by Larry Ewing[3]. The penguin on the right is an Xr rendering of vector-based artwork by Simon Budig[2] intended to match Ewing's artwork as closely as possible. At the original scale of the raster artwork, the two images are quite comparable.

Figure 2: Raster image scaled 400%

However, when the images are scaled up, the differences between raster and vector artwork become apparent. Figure 2 shows a portion of the original raster image scaled by a factor of 4 with the GIMP [6]. Artifacts from the scaling are apparent, primarily in the jaggies around the contour of the image. The GIMP did apply an interpolating filter to reduce these artifacts but this comes at the cost of blurring the image. Compare this to Figure 3 where Xr has been used to draw the vector artwork at 4 times the original scale. Since the vector artwork is resolution independent, the artifacts of jaggies and blurring are not present in this image.

1.2 Vector Rendering Model

The two-dimensional graphics world is fortunate to have one dominant rendering model. With the introduction of desktop publishing and the PostScript printer, application developers converged on that model. Recent extensions to that model have been incorporated in PDF 1.4, but the basic architecture remains the same. PostScript provides a simple painters model; each rendering operation places new paint on top of the contents of the surface. PDF 1.4 extends this model to include Porter/Duff image compositing [9] and other image manipulation operations which serve to bring the basic PostScript rendering model in line with modern application demands.

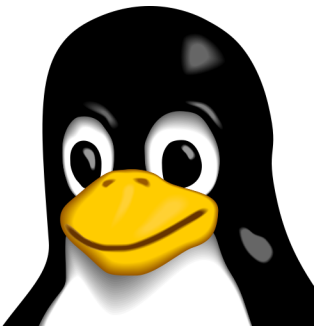


Figure 3: Vector image scaled 400%

PostScript and PDF draw geometric shapes by constructing arbitrary paths of lines and cubic Bézier splines. The coordinates used for the construction can be transformed with an affine matrix. This provides a powerful compositing technique as the transformation may be set before a complex object is drawn to position and scale it appropriately. Text is treated as prebuilt path sections which couples it tightly and cleanly with the rest of the model.

1.3 Xr Programming Interface

While the goal of the Xr library is to provide a PDF 1.4 imaging model, PDF doesn't provide any programming language interface. Xr borrows its imperative immediate mode model from PostScript operators. However, instead of proposing a complete new programming language to encapsulate these operators, Xr uses C functions for the operations and expects the developer to use C instead of PostScript to implement the application part of the rendering system. This dramatically reduces the number of operations needed by the library as only those directly involved in graphics need be provided. The large number of PostScript operators that support a complete language are more than adequately replaced by the C programming language.

PostScript encapsulates rendering state in a global opaque object and provides simple operators to change various aspects of that state, from color to line width and dash patterns. Because global objects can cause various problems in C library interfaces, the graphics state in Xr is held in a structure that is passed to each of the library functions.

The translation from PostScript operators to the Xr interface is straightforward. For example, the lineto operator translates to the XrLineTo function. The coordinates of the line endpoint needed by the operator are preceded by the graphics state object in the Xr interface.

2 API and Examples

This section provides a tour of the application programming interface (API) provided by Xr. Major features of the API are demonstrated in illustrations, and the source code for each illustration is provided in Appendix A.

2.1 Xr Initialization

```
#include <Xr.h>

#define WIDTH 600
#define HEIGHT 600
#define STRIDE (WIDTH * 4)

char image[STRIDE*HEIGHT];

int
main (void)
{
    XrState *xrs;

    xrs = XrCreate ();

    XrSetTargetImage (xrs, image,
                      XrFormatARGB32,
                      WIDTH, HEIGHT, STRIDE);

    /* draw things using xrs ... */

    XrDestroy (xrs);

    /* do something useful with image
       (eg. write to a file) */

    return 0;
}
```

Figure 4: Minimal program using Xr

Figure 4 shows a minimal program using Xr. This program does not actually do useful work—it never draws anything, but it demonstrates the initialization and cleanup procedures required for using Xr.

After including the Xr header file, the first Xr function a program must call is XrCreate. This function returns a pointer to an XrState object, which is used by Xr to store its data. The

XrState pointer is passed as the first argument to almost all other Xr functions.

Before any drawing functions may be called, Xr must be provided with a target surface to receive the resulting graphics. The backend of Xr has support for multiple types of graphics targets. Currently, Xr has support for rendering to in-memory images as well as to any X Window System "drawable", (eg. a window or a pixmap).

The program calls XrSetTargetImage to direct graphics to an array of bytes arranged as 4-byte ARGB pixels. A similar call, XrSetTargetDrawable, is available to direct graphics to an X drawable.

When the program is done using Xr, it signifies this by calling XrDestroy. During XrDestroy, all data is released from the XrState object. It is then invalid for the program to use the value of the XrState pointer until a new object is created by calling XrCreate. The results of any graphics operations are still available on the target surface, and the program can access that surface as appropriate, (eg. write the image to a file, display the graphics on the screen, etc.).

2.2 Transformations

All coordinates passed from user code to Xr are in a coordinate system known as "user space". These coordinates are then transformed to "device space" which corresponds to the device grid of the target surface. This transformation is controlled by the current transformation matrix (CTM) within Xr.

The initial CTM is established such that one user unit maps to an integer number of device pixels as close as possible to 3780 user units per meter (˜96 DPI) of physical device. This approach attempts to balance the competing desires of having a predictable real-world interpretation for user units and having the ability to draw elements on exact device pixel boundaries. Ideally, device pixels would be so small that the user could ignore pixel boundaries, but with current display pixel sizes of about 100 DPI, the pixel boundaries are still significant.

The CTM can be modified by the user to position, scale, or rotate subsequent objects to be drawn. These operations are performed by the functions XrTranslate, XrScale, and XrRotate. Additionally, XrConcatMatrix will compose a given matrix into the current CTM and XrSetMatrix will directly set the CTM to a given matrix. The XrDefaultMatrix function can be used to restore the CTM to its original state.

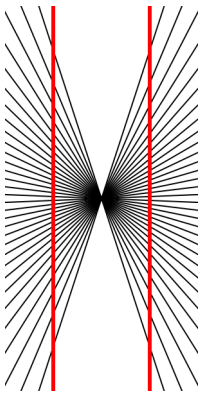


Figure 5: Hering illusion (originally discovered by Ewald Hering in 1861)[11]. The radial lines were positioned with XrTranslate and XrRotate

In Figure 5, each of the radial lines was drawn using identical path coordinates. The different angles were achieved by calling XrRotate before drawing each line. The source code for this image is in Figure 11.

2.3 Save/Restore of Graphics State

Programs using a structured approach to drawing will modify graphics state parameters in a hierarchical fashion. For example, while

traversing a tree of objects to be drawn a program may modify the CTM, current color, line width, etc. at each level of the hierarchy.

Xr supports this hierarchical approach to graphics by maintaining a stack of graphics state objects within the XrState object. The XrSave function pushes a copy of the current graphics state onto the top of the stack. Modifications to the graphics state are made only to the object on the top of the stack. The XrRestore function pops a graphics state object off of the stack, restoring all graphics parameters to their state before the last XrSave operation.

This model has proven effective within structured C programs. Most drawing functions can be written with the following style, wrapping the body of the function with calls to XrSave and XrRestore:

```
void
draw_something (XrState *xrs)
{
    XrSave (xrs);
    /* draw something here */
    XrRestore (xrs);
}
```

This approach has the benefit that modifications to the graphics state within the function will not be visible outside the function, leading to more readily reusable code. Sometimes a single function will contain multiple sections of code framed by XrSave/XrRestore calls. Some find it more readable to include a new indented block between the XrSave/XrRestore calls in this case. Figure 12 contains an example of this style.

2.4 Path Construction

One of the primary elements of the Xr graphics state is the current path. A path consists of one or more independent subpaths, each of which is an ordered set of straight or curved segments. Any non-empty path has a "current point", the final coordinate in the final segment of the current subpath. Path construction functions may read and update the current point.

Xr provides several functions for constructing paths. XrNewPath installs an empty path, discarding any previously defined path. The first path construction called after XrNewPath should be XrMoveTo which simply moves the current point to the point specified. It is also valid to call XrMoveTo when the current path is non-empty in order to begin a new subpath.

XrLineTo adds a straight line segment to the current path, from the current point to the point specified. XrCurveTo adds a cubic Bézier spline with a control polygon defined by the current point as well as the three points specified.

XrClosePath closes the current subpath. This operation involves adding a straight line segment from the current point to the initial point of the current subpath, (ie. the point specified by the most recent call to XrMoveTo). Calling XrClosePath is not equivalent to adding the corresponding line segment with XrLineTo. The distinction is that a closed subpath will have a join at the junction of the final coincident point while an unclosed path will have caps on either end of the path, (even if the two ends happen to be coincident). See Section 2.5 for more discussion of caps and joins.

It is often convenient to specify path coordinates as relative offsets from the current point rather than as absolute coordinates. To allow this, Xr provides XrRelMoveTo, XrRelLineTo, and XrRelCurveTo. Figure 6 shows a rendering of a path constructed with one call to XrMoveTo and four calls to XrRelLineTo in a loop. The source code for this figure can be seen in Figure 13.

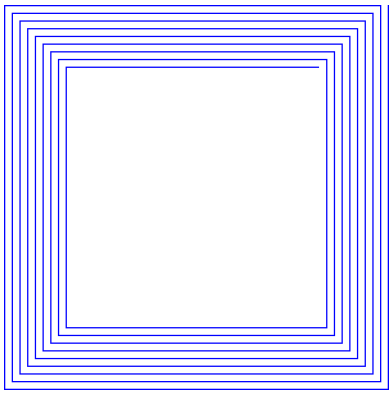


Figure 6: Nested box illusion (after a figure by Al Seckel[11]). Constructed with XrMoveTo and XrRelLineTo

As rectangular paths are commonly used, Xr provides a convenience function for adding a rectangular subpath to the current path. A call to `XrRectangle(xrs, x, y, width, height)` is equivalent to the following sequence of calls:

```
XrMoveTo    (xrs,      x, y);
XrRelLineTo (xrs,  width, 0);
XrRelLineTo (xrs, 0, height);
XrRelLineTo (xrs, -width, 0);
XrClosePath (xrs);
```

After a path is constructed, it can be drawn in one of two ways: stroking its outline (XrStroke) or filling its interior (XrFill).

2.5 Path Stroking

XrStroke draws the outline formed by stroking the path with a pen that in user space is circular with a radius of the current line width, (as set by XrSetLineWidth). The specification of the XrStroke operator is based on the convolution of polygonal tracings as set forth by Guibas, Ramshaw and Stolfi [4]. Convolution lends itself to efficient implementation as the outline of the stroke can be computed within an arbitrarily small error bound by simply using piece-wise linear approximations of the path and the pen.

As subsequent segments within a subpath are drawn, they are connected according to one of three different join styles, (bevel, miter, or round), as set by XrSetLineJoin. Closed subpaths are also joined at the closure point. Unclosed subpaths have one of three different cap styles, (butt, square, or round), applied at either end of the path. The cap style is set with the XrSetLineCap function.

Figure 7 demonstrates the three possible cap and join styles. The source code for this figure (Figure 12) demonstrates the use of XrSetLineJoin and XrSetLineCap as well as XrTranslate, XrSave, and XrRestore.

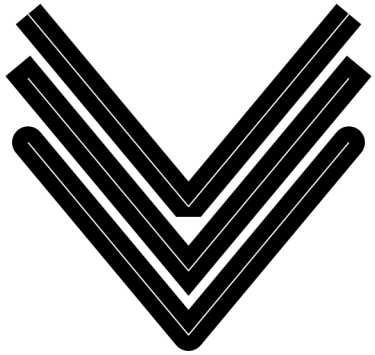


Figure 7: Demonstration of cap and join styles

2.6 Path Filling

XrFill fills the area on the “inside” of the current path. Xr can apply either the winding rule or the even-odd rule to determine the meaning of “inside”. This behavior is controlled by calling XrSetFillRule with a value of either XrFillRuleWinding or XrFillRuleEvenOdd.

Figure 8 demonstrates the effect of the fill rule given a star-shaped path. With the winding rule the entire star is filled in, while with the even-

odd rule the center of the star is considered outside the path and is not filled. Figure 15 contains the source code for this example.

Figure 8: Demonstration of the effect of the fill rule

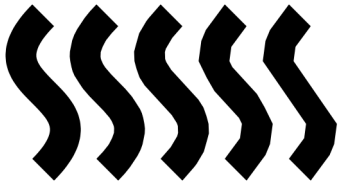


Figure 9: Splines drawn with tolerance values of .1, .5, 1, 5, and 10

2.7 Controlling Accuracy

The graphics rendering of Xr is carefully implemented to allow all rendering approximations to be performed within a user-specified error tolerance, (within the limits of machine arithmetic of course). The XrSetTolerance function allows the user to specify a maximum error in units of device pixels.

The tolerance value has a strong impact on the quality of rendered splines. Empirical testing with modern displays reveals that errors larger than 0.1 device pixels are observable. The default tolerance value in Xr is therefore 0.1 device pixels.

The user can increase the tolerance value to tradeoff rendering accuracy for performance. Figure 9 displays the same curved path rendered several times with increasing tolerance values. Figure 14 contains the source code for this figure.

2.8 Paint

The example renderings shown so far have all used opaque "paint" as the source color for all drawing operations. The color of this paint can be controlled with the XrSetRGBColor function.

Xr supports more interesting possibilities for the paint used in graphics operations. First, the source color need not be opaque; the XrSetAlpha function establishes an opacity level for the source paint. The alpha value ranges from 0 (transparent) to 1 (opaque).

When Xr graphics operations combine translucent surfaces, there are a number of different ways in which the source and destination colors can be combined. Xr provides support for all of the Porter/Duff compositing operators as well as the extended operators defined in the X Render Extension. The desired operator is selected by calling XrSetOperator before compositing. The default operator value is XrOperatorOver corresponding to the Porter/Duff OVER operator.

Finally, the XrSetPattern function allows any XrSurface to be installed as a static or repeating pattern to be used as the "paint" for subsequent graphics operations. The pattern surface may have been initialized from an external image source or may have been the result of previous Xr graphics operations.

Figure 10 was created by first drawing small, vertical black and white rectangles onto a 3X2 surface. This surface was then scaled, filtered, and used as the pattern for 3 XrFill operations. This demonstrates an efficient means of generating linear gradients within Xr.

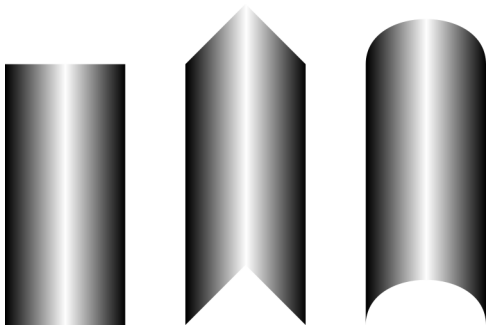


Figure 10: Outline affects perception of depth from shading, (after an illustration by Isao Watanabe[14]). This example uses XrFill with XrSetPattern

2.9 Images

In addition to the vector path support, Xr also supports bitmapped images as primitive objects. Images are transformed, (and optionally filtered), by the CTM in the same manner as all other primitives. In order to display an image, an XrSurface object must first be created for the image, then the surface can be displayed with the XrShowSurface function. XrShowSurface places an image of the given width and height at the origin in user space, so XrTranslate can be used to position the surface.

In addition to the CTM, each surface also has its own matrix providing a transformation from user space to image space. This matrix can be used to transform a surface independently from the CTM.

The XrShowSurface function has another important use besides allowing the display of external images. When using the Porter/Duff compositing operators, it is often desirable to combine several graphics primitives on an intermediate surface before compositing the result onto the target surface. This functionality is similar to the notion of transparency groups in PDF 1.4 and can be achieved with the following idiom:

```
XrSave (xrs);
XrSetTargetSurface (xrs, surface);
/* draw to intermediate surface with
   any Xr functions */
XrRestore (xrs);
XrShowSurface (xrs, surface);
```

In this example an intermediate surface is installed as the target surface, and then graphics are drawn on the intermediate surface. When XrRestore is called, the original target surface is restored and the resulting graphics from the intermediate surface are composited onto the original target.

This technique can be applied recursively with any number of levels of intermediate surfaces each receiving the results of its “child” surfaces before being composited onto its “parent” surface.

Alternatively, images can be constructed from data external to the Xr environment, acquired from image files, external devices or even the window system. Because the image formats used within Xr are exposed to applications, this kind of manipulation is easy and efficient.

3 Implementation

As currently implemented, Xr has good support for all functions described here. The major aspects of the PostScript imaging model that have not been discussed are text/font support, clipping, and color management. Xr does include some level of experimental support for text and clipping already, but these areas need further development.

The Xr system is implemented as 3 major library components: libXr, libXc, and libIc. LibXr provides the user-level API described in detail already.

LibXc is the backend of the Xr system. It provides a uniform, abstract interface to sev-

eral different low-level graphics systems. Currently, libXc provides support for drawing to the X Window System or to in-memory images. The semantics of the libXc interface are consistent with the X Render Extension so it is used directly whenever available.

LibIc is an image compositing library that is used by libXc when drawing to in-memory images. LibIc can also be used to provide support for a low-level system whose semantics do not match the libXc interface. In this case, libIc is used to draw everything to an in-memory image and then the resulting image is provided to the low-level system. This is the approach libXc uses to draw to an X server that does not support the X Render Extension.

The libIc code is based on the original code for the software fallback in the reference implementation of the X Render Extension. It would be useful to convert any X server using that implementation to instead use libIc.

These three libraries are implemented in approximately 7000 lines of C code.

4 Related Work

Of the many existing graphics systems, several relate directly to this new work.

4.1 PostScript and Display PostScript

As described in the introduction, Xr adopts (and extends) the PostScript rendering model. However, PostScript is not just a rendering model as it includes a complete programming language. Display PostScript embeds a PostScript interpreter inside the window system. Drawing is done by generating PostScript programs and delivering them to the window system.

One obvious benefit of using PostScript everywhere is that printing and display can easily be done with the same rendering code, as long as the printer supports PostScript. A disadvantage is that images are never generated within the application address space making it more difficult to use where PostScript is not available.

Using the full PostScript language as an intermediate representation means that a significant fraction of the overall application development will be done in this primitive language. In addition, the PostScript portion is executed asynchronously with respect to the remaining code, further complicating development. Integrating the powerful PostScript rendering model into the regular application development language provides a coherent and efficient infrastructure.

4.2 Portable Document Format

PDF provides a very powerful rendering model, but no application interface. Generating PDF directly from an application would require some kind of PDF API along with a PDF interpreter. The goal for Xr is to be able to generate PDF output files while providing a clean application interface.

A secondary goal is to allow PDF interpreters to be implemented on top of Xr. As Xr is missing some of the less important PDF operations, those will need to be emulated within the interpreter. An important feature within Xr is that such emulation be reasonably efficient.

4.3 OpenGL

OpenGL[12] provides an API with much the same flavor as Xr; immediate mode functions with an underlying stateful library. OpenGL doesn't provide the PostScript rendering model, and doesn't purport to support printing or the local generation of images.

As Xr provides an abstract interface atop many

graphics architectures, it should be possible to layer Xr on OpenGL.

5 Future Work

The Xr library is in active development. Everything described in this paper is currently working, but much work remains to make the library generally useful for application development.

5.1 Text Support

Much of the current design effort has been focused on the high-level drawing model and some low-level rendering implementation for geometric primitives. This design effort was simplified by the adoption of the PostScript model. PostScript offers a few useful suggestions about handling text, but applications require significantly more information about fonts and layout. The current plan is to require applications to use the FreeType [13] library for font access and the Fontconfig [8] library for font selection and matching. That should leave Xr needing only relatively primitive support for positioning glyphs and will push issues of layout back on the application.

5.2 Printing Backend

Xr is currently able to target the X Window System, (with or without the X Render Extension), as well as local images. Still missing is the ability to generate PostScript or PDF output files. Getting this working is important not only so that applications can print, but also because there may be unintended limitations in both the implementation and specification caused by the essential similarity between the two existing backends.

One of the goals of Xr is to have identical output across all output devices. This will require that Xr embed glyph images along with the document output to ensure font matching across all PostScript or PDF interpreters. Embedding TrueType and Type1 fonts in the output file should help solve this problem.

5.3 Color Management

Xr currently supports only the RGB color space. This simplifies many aspects of the library interface and implementation. While it might become necessary to add support for more sophisticated color management, such development will certainly await a compelling need. One simple thing to do in the meantime would be to reinterpret the device-dependent RGB values currently provided as sRGB instead. Using ICC color profiles would permit reasonable color matching across devices while not adding significant burden to the API or implementation.

6 Availability

Xr is free software released under an MIT license from `http://xr.xwin.org`.

7 Disclaimer

Portions of this effort sponsored by the Defense Advanced Research Projects Agency (DARPA) under agreement number F30602-99-1-0529. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

References

[1] Adobe Systems Incorporated. *PostScript Language Reference Manual*. Addison Wesley, 1985.

[2] Simon Budig. The linux-pinguin again. `http://www.home.unix-ag.org/simon/penguin`.

[3] Larry Ewing. Linux 2.0 penguins. `http://www.isc.tamu.edu/~lewing/linux`.

[4] Leo Guibas, Lyle Ramshaw, and Jorge Stolfi. A kinetic framework for computational geometry. In *Proceedings of the IEEE 1983 24th Annual Symposium on the Foundations of Computer Science*, pages 100–111. IEEE Computer Society Press, 1983.

[5] Adobe Systems Incorporated, editor. *PDF Reference: Version 1.4*. Addison-Wesley, 3rd edition, 2001.

[6] Peter Mattis, Spencer Kimball, and the GIMP developers. The GIMP: The GNU image manipulation program. `http://www.gimp.org`.

[7] Keith Packard. Design and Implementation of the X Rendering Extension. In *FREENIX Track, 2001 Usenix Annual Technical Conference*, Boston, MA, June 2001. USENIX.

[8] Keith Packard. Font Configuration and Customization for Open Source Systems. In *2002 Gnome User's and Developers European Conference*, Seville, Spain, April 2002. Gnome.

[9] Thomas Porter and Tom Duff. Compositing Digital Images. *Computer Graphics*, 18(3):253–259, July 1984.

[10] Robert W. Scheifler and James Gettys. *X Window System*. Digital Press, third edition, 1992.

[11] Al Seckel. *The Great Book of Optical Illusions*. Firefly Books Ltd., 2002.

[12] Mark Segal, Kurt Akeley, and Jon Leach (ed). *The OpenGL Graphics System: A Specification*. SGI, 1999.

[13] David Turner and The FreeType Development Team. The design of FreeType 2, 2000. `http://www.freetype.org/freetype2/docs/design/`.

[14] Isao Watanabe. 3-d shape and outline. `http://www.let.kumamoto-u.ac.jp/watanabe/Watanabe-E/Illus-E/3D-E/index.%html`.

A Example Source Code

This appendix contains the source code that was used to draw each figure in Section 2. Each example contains a top-level "draw" function that accepts an XrState pointer, a width, and a height. The examples here can be made into complete programs by adding the code from the example program of Figure 4 and inserting a call to the appropriate "draw" function.

```
void
draw_hering (XrState *xrs,
             int width, int height)
{
#define LINES 32.0
#define MAX_THETA (.80 * M_PI_2)
#define THETA (2 * MAX_THETA / (LINES-1))

  int i;

  XrSetRGBColor (xrs, 0, 0, 0);
  XrSetLineWidth (xrs, 2.0);

  XrSave (xrs);
  {
    XrTranslate (xrs, width / 2, height / 2);
    XrRotate (xrs, MAX_THETA);

    for (i=0; i < LINES; i++) {
      XrMoveTo (xrs, -2 * width, 0);
      XrLineTo (xrs, 2 * width, 0);
      XrStroke (xrs);

      XrRotate (xrs, - THETA);
    }
  }
  XrRestore (xrs);

  XrSetLineWidth (xrs, 6);
  XrSetRGBColor (xrs, 1, 0, 0);

  XrMoveTo (xrs, width / 4, 0);
  XrRelLineTo (xrs, 0, height);
  XrStroke (xrs);

  XrMoveTo (xrs, 3 * width / 4, 0);
  XrRelLineTo (xrs, 0, height);
  XrStroke (xrs);
}
```

Figure 11: Source for Hering illusion of Figure 5

```
void
draw_caps_joins (XrState *xrs,
                 int width, int height)
{
    static double dashes[2] = {10, 20};
    int line_width = height / 12 & (~1);

    XrSetLineWidth (xrs, line_width);
    XrSetRGBColor (xrs, 0, 0, 0);

    XrTranslate (xrs, line_width, line_width);
    width -= 2 *line_width;

    XrSetLineJoin (xrs, XrLineJoinBevel);
    XrSetLineCap (xrs, XrLineCapButt);
    stroke_v_twice (xrs, width, height);

    XrTranslate (xrs, 0, height/4-line_width);
    XrSetLineJoin (xrs, XrLineJoinMiter);
    XrSetLineCap (xrs, XrLineCapSquare);
    stroke_v_twice (xrs, width, height);

    XrTranslate (xrs, 0, height/4-line_width);
    XrSetLineJoin (xrs, XrLineJoinRound);
    XrSetLineCap (xrs, XrLineCapRound);
    stroke_v_twice (xrs, width, height);
}

void
stroke_v_twice (XrState *xrs,
                int width, int height)
{
    XrMoveTo (xrs, 0, 0);
    XrRelLineTo (xrs, width/2,  height/2);
    XrRelLineTo (xrs, width/2, -height/2);

    XrSave (xrs);
    XrStroke (xrs);
    XrRestore (xrs);

    XrSave (xrs);
    {
        XrSetLineWidth (xrs, 2.0);
        XrSetLineCap (xrs, XrLineCapButt);
        XrSetRGBColor (xrs, 1, 1, 1);
        XrStroke (xrs);
    }
    XrRestore (xrs);

    XrNewPath (xrs);
}
```

Figure 12: Source for cap and join demonstration of Figure 7

```
void
draw_spiral (XrState *xrs,
             int width, int height)
{
  int wd = .02 * width;
  int hd = .02 * height;
  int i;

  width -= 2;
  height -= 2;

  XrMoveTo (xrs, width - 1, -hd - 1);
  for (i=0; i < 9; i++) {
    XrRelLineTo (xrs, 0, height-hd*(2*i-1));
    XrRelLineTo (xrs, -(width-wd*(2*i)), 0);
    XrRelLineTo (xrs, 0,-(height-hd*(2*i)));
    XrRelLineTo (xrs,  width-wd*(2*i+1), 0);
  }

  XrSetRGBColor (xrs, 0, 0, 1);
  XrStroke (xrs);
}
```

Figure 13: Source for nested box illusion of Figure 6

```
void
draw_splines (XrState *xrs,
              int width, int height)
{
    int i;
    double tolerance[5] = {.1,.5,1,5,10};
    double line_width = .08 * width;
    double gap = width / 6;

    XrSetRGBColor (xrs, 0, 0, 0);
    XrSetLineWidth (xrs, line_width);

    XrTranslate (xrs, gap, 0);
    for (i=0; i < 5; i++) {
        XrSetTolerance (xrs, tolerance[i]);
        draw_spline (xrs, height);
        XrTranslate (xrs, gap, 0);
    }
}

void
draw_spline (XrState *xrs, double height)
{
    XrMoveTo (xrs, 0, .1 * height);
    height = .8 * height;
    XrRelCurveTo (xrs,
                  -height/2, height/2,
                  height/2, height/2,
                  0, height);
    XrStroke (xrs);
}
```

Figure 14: Source for splines drawn with varying tolerance as in Figure 9

```
void
draw_stars (XrState *xrs,
            int width, int height)
{
  XrSetRGBColor (xrs, 0, 0, 0);

  XrSave (xrs);
  {
    XrTranslate (xrs, 5, height/2.6);
    XrScale (xrs, height, height);
    star_path (xrs);
    XrSetFillRule (xrs, XrFillRuleWinding);
    XrFill (xrs);
  }
  XrRestore (xrs);

  XrSave (xrs);
  {
    XrTranslate (xrs,
                 width-height-5, height/2.6);
    XrScale (xrs, height, height);
    star_path (xrs);
    XrSetFillRule (xrs, XrFillRuleEvenOdd);
    XrFill (xrs);
  }
  XrRestore (xrs);
}

void
star_path (XrState *xrs)
{
    int i;
    double theta = 4 * M_PI / 5.0;

    XrMoveTo (xrs, 0, 0);
    for (i=0; i < 4; i++) {
        XrRelLineTo (xrs, 1.0, 0);
        XrRotate (xrs, theta);
    }
    XrClosePath (xrs);
}
```

Figure 15: Source for stars to demonstrate fill rule as in Figure 8

```
void
draw_gradients (XrState *xrs,
                int img_width, int img_height)
{
    XrSurface *gradient;
    double width, height, pad;

    width = img_width / 4.0;
    pad = (img_width - (3 * width)) / 2.0;
    height = img_height;

    gradient=make_gradient(xrs,width,height);

    XrSetPattern (xrs, gradient);
    draw_flat (xrs, width, height);
    XrTranslate (xrs, width + pad, 0);
    XrSetPattern (xrs, gradient);
    draw_tent (xrs, width, height);
    XrTranslate (xrs, width + pad, 0);
    XrSetPattern (xrs, gradient);
    draw_cylinder (xrs, width, height);

    XrRestore (xrs);

    XrSurfaceDestroy (gradient);
}

XrSurface *
make_gradient (XrState *xrs,
               double width, double height)
{
    XrSurface *g;
    XrMatrix *matrix;

    XrSave (xrs);

    g = XrSurfaceCreateNextTo (
                XrGetTargetSurface (xrs),
                XrFormatARGB32, 3, 2);
    XrSetTargetSurface (xrs, g);

    XrSetRGBColor (xrs, 0, 0, 0);
    XrRectangle (xrs, 0, 0, 1, 2);
    XrFill (xrs);

    XrSetRGBColor (xrs, 1, 1, 1);
    XrRectangle (xrs, 1, 0, 1, 2);
    XrFill (xrs);

    XrSetRGBColor (xrs, 0, 0, 0);
    XrRectangle (xrs, 2, 0, 1, 2);
    XrFill (xrs);

    XrRestore (xrs);

    matrix = XrMatrixCreate ();
    XrMatrixScale (matrix,
                   2.0/width, 1.0/height);
    XrSurfaceSetMatrix (g, matrix);
    XrSurfaceSetFilter (g, XrFilterBilinear);
    XrMatrixDestroy (matrix);

    return g;
}
```

```
void
draw_flat (XrState *xrs, double w, double h)
{
    double hw = w / 2.0;

    XrRectangle (xrs, 0, hw, w, h - hw);

    XrFill (xrs);
}

void
draw_tent (XrState *xrs, double w, double h)
{
    double hw = w / 2.0;

    XrMoveTo    (xrs,   0,  hw);
    XrRelLineTo (xrs,  hw, -hw);
    XrRelLineTo (xrs,  hw,  hw);
    XrRelLineTo (xrs,   0,  h - hw);
    XrRelLineTo (xrs, -hw, -hw);
    XrRelLineTo (xrs, -hw,  hw);
    XrClosePath (xrs);

    XrFill (xrs);
}

void
draw_cylinder (XrState *xrs, double w, double h)
{
    double hw = w / 2.0;

    XrMoveTo (xrs, 0, hw);
    XrRelCurveTo (xrs,   0, -hw,
                  w, -hw,   w, 0);
    XrRelLineTo (xrs, 0, h - hw);
    XrRelCurveTo (xrs,   0, -hw,
                  -w, -hw,  -w, 0);
    XrClosePath (xrs);

    XrFill (xrs);
}
```

Figure 16: Source for 3 gradient-filled shapes of Figure 10

relayfs: An Efficient Unified Approach for Transmitting Data from Kernel to User Space

Tom Zanussi zanussi@us.ibm.com
Karim Yaghmour karim@opersys.com
Robert Wisniewski bob@watson.ibm.com
Richard Moore richardj_moore@uk.ibm.com
Michel Dagenais michel.dagenais@polymtl.ca

Abstract

Linux has several mechanisms for relaying information about the system and applications to the user. Some examples include printk and other syslog events, evlog, ltt, oprofile, etc. Each subsystem has its own method for relaying information from the kernel to user space. Some of these mechanisms have difficulties, e.g. logging of printk messages is unreliable. In addition to selected difficulties, the replication of code and maintenance is undesirable. In this paper we describe a high-speed data relay filesystem that satisfies the buffering requirements of the above subsystems while providing a unified, efficient, and reliable relay mechanism. relayfs allows subsystems to log data efficiently and safely using lockless technology that is designed to scale well on multiprocessor systems. relayfs includes the flexibility to be expanded should other subsystems need additional services, but has a simple design intended to meet the needs of currently available subsystems. In this paper we discuss the architecture, implementation, and usage of relayfs. relayfs uses channels that allow data to be directed to a suitable buffer or buffers for the subsystems that allocated the channel. We describe the kernel API and file naming conventions, address init-time issues, and discuss performance trade-offs available using relayfs. Finally we demonstrate how existing subsystems use relayfs to log their data.

1 Introduction

Sharing data between the kernel and user-space applications requires buffering. For relatively small transfers, such as passing variables or small data arrays, the normal system call API is sufficient; the overhead required for safely transferring data across the kernel boundary is acceptable. For larger data transfers, the subsystem generating the data is responsible for providing a buffering and transfer mechanism to deliver the data to user space. With increasing system speed and growing demand for more information regarding the kernel's operation, many of the conventional buffering mechanisms have reached their limits. Further, the buffering and transfer code for each of the subsystems is replicated and needs to be independently maintained.

To address these challenges, we designed and implemented relayfs. relayfs is a unified, reliable, efficient, and simple mechanism for transferring large amounts of data between the kernel and user space. The algorithms used for relayfs have been designed to handle both high frequency and large data applications such as kernel tracing. To satisfy these demand-

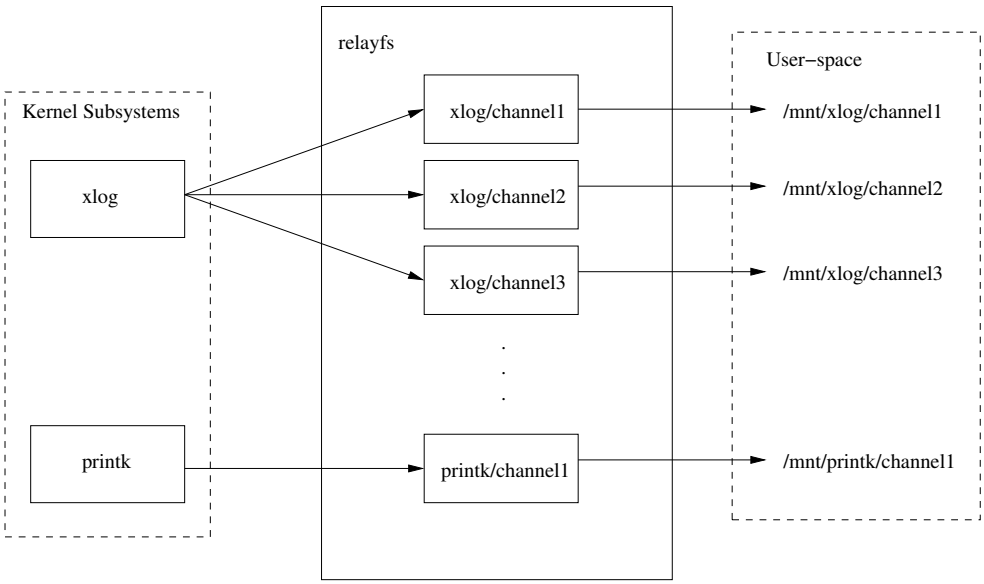


Figure 1: relayfs architecture

ing clients, relayfs is a high-speed, low-impact data relay filesystem. Data logged by a subsystem using relayfs appears under the directory where the filesystem is mounted. For example, if relayfs is mounted under /mnt, printk data may appear in /mnt/printk/data. Although this paper focuses on the use of relayfs for kernel subsystems, user-space clients can also be served by relayfs.

relayfs provides the abstraction of a *channel* to the subsystem using its services. Channels and files have a 1-to-1 mapping. A given subsystem may log data to multiple channels. For example, a tracing subsystem may log the majority of its data to one channel and create a control channel for less frequent but informative events such as new process creation. printk could, for example, use different channels to log information from devices versus the memory system. Alternatively, a kernel developer could create a separate channel for just the events in newly written (and currently being debugged) code, so as to view only those.

relayfs has a simple design and mechanisms intended to meet the demands of current subsystems while affording flexibility in meeting new requirements of future subsystems. relayfs provides the option of using locking or lockless data logging. The lockless option is very efficient but introduces potential difficulties as discussed later. relayfs provides the choice of using per-processor buffers or a single buffer shared across the machine. It also provides choice between block or packet delivery of events. The trade-offs of these options are discussed later as well.

In addition to these options, relayfs reduces code replication among subsystems by providing mechanisms common to the subsystems. These include handling overflow issues, providing efficient timestamping on events, providing efficient delivery to user space, and handling dynamic allocation and de-allocation of memory needed to provide the buffering. Tests conducted using relayfs have shown it can handle significant amounts of data with very low overhead.

The rest of the paper is structured as follows.

Section 2 describes the channel architecture and the lockless algorithm. Section 3 describes how the code is structured and where to find the implementation files. Section 4 describes the interface to relayfs, the different options available, and how it handles overflows. Section 5 describes the impact and performance of relayfs. Section 6 describes how to modify some example subsystems to use relayfs. Section 7 concludes.

2 Architecture

Figure 1 presents the relayfs architecture. Kernel subsystems create and use different relayfs channels to log their data, while user-space applications see those same channels as files located in the mounted relayfs filesystem.

2.1 Channels

The main building block of relayfs is a channel. To relay data to user-space applications, kernel subsystems allocate channels to transfer their data. Multiple channels can be used to implement any data multiplexing desired, including using one channel per CPU to implement per-CPU buffering.

The buffering implemented by relayfs is transparent to the client subsystem. Writing to the channel requires that the following be specified: channel ID, pointer to the data, and size of data. relayfs does not parse the data being relayed; it just transfers bytes. Because some relayfs clients may implement a particular data protocol (for example, special markings on buffer ends), relayfs provides callback functions for its clients when significant changes in the channel buffers occur. The callbacks are optional.

From user space, channels are accessed as files. relayfs implements the standard operations required for normal file manipulation. For example, to gain access to a relayfs file, applications can perform read() operation on the file specifying a number of bytes. Alternatively the application can mmap() in the file and reference the contents of the file via memory pointer.

2.2 Lockless Event Logging

An important feature of relayfs is its ability to write data to buffers without requiring locks. Previous lockless logging schemes[2] used fixed-length events with valid bits. There are several advantages to using variable-length events (assuming the random access problem is solved, see Section 4.2). The algorithm integrated into relayfs allows variable-length events to be logged without locking.

Each process attempts to *reserve* enough space in the buffer immediately after the current index for the event it wants to log. Once the process makes a successful reservation, it may proceed to log its data. To reserve space, each process attempts to atomically increment the current index using a `compare and store`. The process that successfully increments (as determined by the return value of the `compare and store` operation) the index has the right to proceed to log data into the buffer; failing processes retry. Figure 2 shows on the left, in step 1, two processes, A and B, attempting to log events of different lengths after the current index from the initial configuration in step 0. Each process attempts to increment the current index by the size of the event being logged. The process that succeeds, in this case B, will log the event immediately following the old current index (see step 2). This will be followed by process A's data, assuming no other competing processes attempt to log more data (see step 3). Because it is important to guarantee monotonically increasing timestamps, processes must re-determine the timestamp during each attempt to atomically

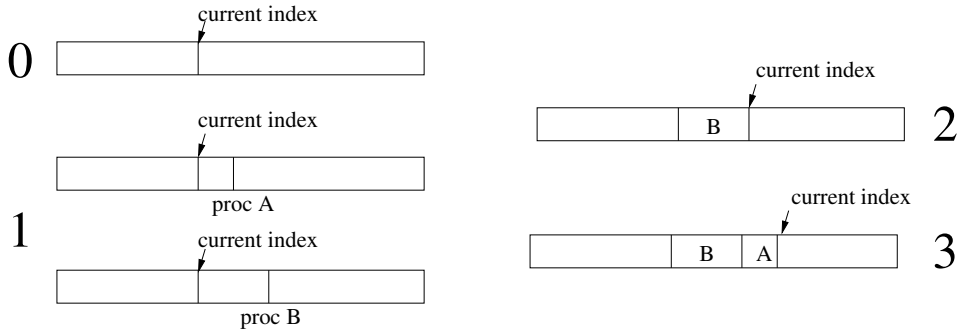


Figure 2: Illustration of Lockless Event Logging

```
eventReserve(length, *indexPtr, *timestampPtr)
  integer: oldIndex, newIndex
  EvtCtl: *evtCtlPtr
    update evtCtlPtr
    do
      oldIndex = evtCtlPtr->index
      newIndex = oldIndex + length
      if (newIndex >= buffer end)
        eventReserveSlow(length, indexPtr, timestampPtr)
          // generates filler event, sets timestamp, moves to new buffer
        return
      *timestampPtr = getTimestamp()
    while (!CompareAndStore(&(evtCtlPtr->index), oldIndex, newIndex))
    *indexPtr = oldIndex & INDEXMASK  // confine index to buffer bounds

eventLog(majorID, minorID, data)
  integer: index, timestamp, length
    length = length of data
    eventReserve(length, &index, &timestamp)
    evtArray[index] = logEvtHeader(timestamp, length, majorID, minorID)
    evtArray[index+1 ... index+length] = data
    eventCommit(index, length) // optional, see explanation in text
```

Figure 3: Pseudo code for lockless event logging

increment the index.

The memory used for logging is logically divided into buffers. Once a buffer is full, the logging facility proceeds to the subsequent buffer and the previous buffer is available to be written out. The pseudo code appears in Figure 3 and complete C code can be obtained by downloading relayfs from the relayfs web site[1].

Despite having good performance, complications can arise from using the lockless algorithm. A process's execution may be interrupted after it has reserved space to log an event, but before it actually performs the log. The interruption can occur because the process is preempted, blocks for a long time, or is killed. Depending on where (in the sequence of code in Figure 3) the process is interrupted, different problems occur. If the process has had a chance to write the event header, but not the data, then only the data will be unrecoverable. If, however, the process has not yet logged the event header then it is possible the

rest of the buffer will be uninterpretable. Only by locking, making the kernel perform the log, and disabling interrupts can this problem be prevented (in practice there are low-level kernel events that would still exhibit the problem). There are methods that avoid these difficulties.

If the reason the process's execution was interrupted was due to preemption, then it is likely the process will run again soon and finish filling in the event before another entity notices, thus posing no real problem. If the reason for the process's interruption was because the process was killed, then the data will never finish being logged. The last line of pseudo-code detects this situation. The `eventCommit` function updates a per-buffer count of the amount of data that has been logged to that buffer. The count is zeroed during the `start new buffer` code. When the code responsible for writing the data (to a network stream, file, etc.) writes this buffer, it can compare the amount of data logged to this buffer with the buffer's size and report an anomaly if they do not match. If the reason the process was interrupted was because of a long blocking operation, it is possible that both the current buffer will not have enough data logged, and that the same buffer, when reused in the future, will have too much (because the long-blocked process was unblocked and logged data into a recycled buffer). Again the per-buffer counts can detect this situation.

Besides a per-buffer count there are other possible ways to detect or minimize the occurrence of corrupted data. For example, it is possible to use a flag in a per-process data structure to indicate to the kernel that a process should not be killed while the flag is set. Other possibilities include zero-filling a buffer before use, or keeping a side array of valid bits for the header data. In practice the probability of corrupting a buffer, and the ease with which tools can handle the situation, reduces the issue's importance except perhaps in setups where recreating the situation generating the logged data is very difficult. In those cases the locking version of the event logging may be the best option.

3 Implementation

The relayfs code is structured as follows: the public API and common relay code are contained in fs/relayfs/relay.c, with the scheme-specific code in fs/relayfs/relay_lockless.c and fs/relayfs/relay_locking.c. The file fs/relayfs/inode.c implements the VFS layer on top of the relay channel code.

relayfs can be compiled either directly into the kernel or as a kernel module.

4 Interface and Use

relayfs is used both as a temporary repository for logged data and as a filesystem from which user-space clients may retrieve logged data. The first of these is supported by set of kernel-space APIs. For the second, relayfs is mounted as a filesystem:

```
mount -t relayfs relayfs /mnt/relay
```

Kernel subsystems (also referred to as kernel clients) create and write to channels via the kernel API described below. The contents of these channels are available to user-space programs via a standard file abstraction that can be read using mmap() or read(). relayfs provides automatic support for locking or lockless logging, overflow handling, data delivery, and timestamping.

4.1 Interface

This section describes the basic usage of the kernel and user APIs. Com-

plete details can be found in Documentation/filesystems/relayfs.txt. To initiate data logging, a kernel client creates a channel relative to the mountpoint of the relayfs filesystem via relay_open():

```
int channel_id =
      relay_open("file", ...);
```

This would cause the creation of a relayfs file named /mnt/relay/file, assuming relayfs was mounted at /mnt/relay.

relayfs does not impose any namespace conventions; clients may choose names as they wish. We recommend the adoption of the convention whereby a client specifies a top-level directory name that is closely associated with the corresponding subsystem. For example, the Linux Trace Toolkit would manage the namespace under /mnt/relay/trace, printk would use /mnt/relay/printk, driver debugging channels might use /mnt/relay/debug/drivers/mydriver, etc.

A kernel client can then log a variable-length data item to the channel via relay_write(), given the channel id.

```
relay_write(channel_id, data,
            count,...);
```

In user space, a program can open the relayfs file and wait in a read() loop waiting for data.

```
fd = open("/mnt/relay/file", ...);

while(1) {
  n = read(fd, buf, sizeof(buf));
  if(n <= 0) {
     close(fd);
     break;
  }
}
```

Alternatively, the file can be mmap()'ed and directly accessed via a pointer to the mapped buffer when data is ready.

```
fd = open("/mnt/relay/file", ...);
char *map = mmap(..., fd, ...);

void on_ready(count) {
   write(diskfile, map, count);
}
```

There are five callbacks that can optionally be registered by the kernel client when a channel is opened. These are used to notify the client when significant events occur (buffer start, buffer end, event deliver, buffers full, buffer resize) and are described below.

4.2 Channel and data management schemes

The channel data for a given channel is internally managed via one of two schemes defined at channel creation. They are *lockless* or *locking*. The lockless scheme is described in Section 2. The locking scheme is a simple two-buffer ping-pong scheme. One of the buffers is the current write buffer, into which events are written, and the other is the current read buffer, from which events are read (by, for instance, a user daemon). When the current write buffer is filled, it becomes the current read buffer and the current read buffer becomes the new write buffer. The key feature of the locking scheme is that the channel is locked (the exact semantics are described below) while space is allocated for the event and for the duration of the write.

The reason two schemes exist is that while it would be ideal to have all channels managed by the lockless scheme, the availability of the lockless scheme depends on the availability of a cmpxchg instruction or a generic equivalent, which does not exist on all Linux platforms. Thus, the locking scheme is available as a fallback scheme for those platforms that cannot support the lockless scheme.

relayfs channels are implemented as circular buffers divided into a number of sub-buffers. If a scheme has not been explicitly specified, the channel creation code will choose lockless. The number and size of these sub-buffers

are specified at channel creation (or with certain restrictions can be dynamically sized afterward). For efficiency reasons, in the lockless code, both of these are a power of 2. Having each buffer size be a power of 2 allows a cheap logical and comparison to determine if the end of a buffer has been reached. Having the number of buffers be a power of 2 allows a cheaper operation to be performed to ensure the index remains within the memory allocated for the buffers. Both of these operations occur on the fast path of every event log and thus are critical. The index value could be interpreted as a straight index into a single circular buffer, but for other reasons multiple buffers are preferable. For example, breaking up the buffers into multiple sub-buffers rather than a single large buffer allows flexibility in buffer processing, allows more timely delivery of events to user space, and minimizes the impact of potential data corruption.

One aspect of using the lockless scheme is the use of variable-length events. There are tradeoffs between using fixed-length or variable-length events. Fixed-size events allow for simpler logging and reading out as the consumer of events always knows the starting point of an event. This allows validity bits to be used, and allows invalid events to be skipped. Fixed-length events allow easy random access to the data stream, aiding reading and displaying large files. The disadvantages of fixed-length events are that they waste space, they take longer to write (to disk or network) because extra data needs to be written for short events, and they make it complicated to log data that is larger than the fixed size. The lockless scheme obtains the benefits of each by ensuring that events never cross medium-scale alignment boundaries. We insert filler events as necessary to align the event stream. Data analysis tools can skip to any of the alignment points in a large event buffer and can begin interpreting events from that point. This

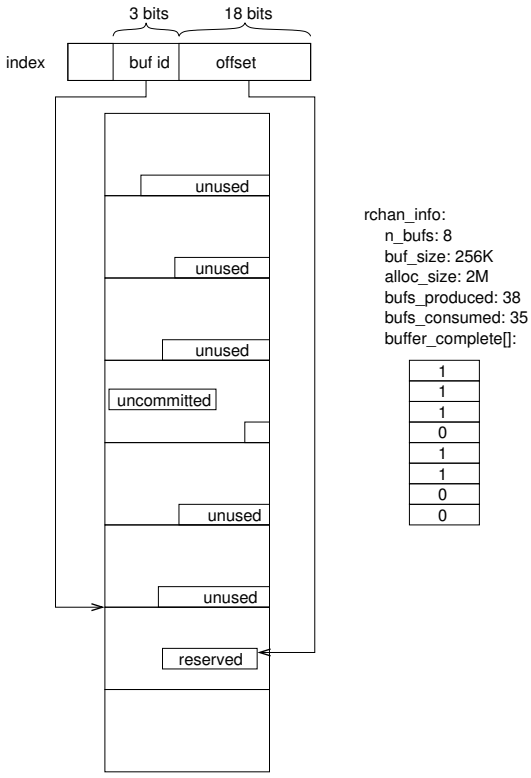


Figure 4: Buffer layout.

technique provides the advantages of variable-length events and still allows fast access to all parts of a large file. A filler event is a header with a length equal to the remainder of the current buffer; no data need be logged. For the clients we have studied this alignment wastes little space. Other applications that have few large events and whose events frequently end on buffer boundaries will exhibit similar behavior. This technique does not provide completely random access, but is a close enough approximation that it allows post-processing tools to make it appear to the end user that the stream is completely random access.

Figure 4 shows a 2M buffer divided into 8 sub-buffers of 256K (in this case, 256K would be the alignment boundary described above). This splits the index logically into a 3-bit portion containing the buffer id and 18 bits containing the current offset within the buffer. The rchan_info struct, retrieved via relay_info(),

contains a snapshot of the current state of the channel. The data in Figure 4 shows that 35 sub-buffers have been consumed and 38 have been produced. The buffer_complete array shows the completeness state of all of the sub-buffers not including the current one. As Figure 4 shows, sub-buffer 3 is not yet complete as there is still a pending write. This is also the reason buffers_consumed has not caught up with buffers_produced. The user-space client suspends processing the sub-buffers when it encounters the incomplete buffer until such time as the buffer becomes complete, or is "lapped", in which case the buffer contents would be lost.

4.3 Buffer start/end callbacks

Two of the five channel callbacks registered by the kernel client exist to allow the client to be notified when sub-buffer boundaries are crossed, i.e. when buffer switches occur. These callbacks are invoked in the slow path of event logging, which is executed when an event write would overflow the current sub-buffer. The buffer_end() callback is invoked to allow the kernel subsystem the opportunity to perform end-of-buffer processing on the just-filled buffer. To allow space for data even when a buffer is exactly filled, there needs to be space reserved at the end of the buffer into which the client can write an unused count. This is the purpose of the end_reserve parameter to relay_open. It specifies the number of bytes at the end of each sub-buffer that should be left alone by the logging algorithm and left available for the client to write data. Similarly, the buffer_start() callback is invoked to give the client an opportunity to write header data at the beginning of a sub-buffer. The start_reserve parameter to relay_open() allows the client to specify a value for this purpose. One of the parameters of the buffer_start() callback is the buffer id, which has a value of zero for the very first buffer in the channel. Clients can check for this value and optionally write channel header data in the position reserved for it by the channel_start_reserve parameter of relay_open().

4.4 Channel attributes

The characteristics of a given channel are derived from a set of channel attributes specified when the channel is opened. These are:

- **scheme**: lockless or locking. Indicates which of the logging algorithms to use. If 'any' is specified, relayfs attempts to use the lockless scheme; if unavailable it reverts to the locking scheme.
- **SMP usage**: global or SMP. Applies only if the locking scheme is being used. Global indicates that the channel is being shared across multiple processors. In this case lock acquisition involves a spinlock_irqsave/restore. The SMP option indicates per-processor channels and thus lock acquisition can use the cheaper local_irqsave/restore.
- **delivery mode**: bulk or packet. If bulk delivery mode is specified, the kernel client is notified via the delivery callback when complete buffers become available. If packet delivery mode is specified, the delivery callback is invoked after each write. Bulk delivery is suited for clients logging large volumes of data. They would be noticeably affected by callbacks after each event, but still may be interested in the beginnings and ends of buffers. Less demanding clients may require notification for each event logged. Clients specify callbacks to, for example, signal a user-level program indicating data is ready. This is a typical mode of operation for a bulk data client and is somewhat less typical for packet client as its user-space pro-

gram is often polling for the next piece of data. Either type of client may be interested in filtering the data and/or dispatching events to other channels or subsystems. Typically, packet clients would be more interested in data filtering due to the more manageable volume of events.

- **timestamping**: TSC or gettimeofday() deltas. This attribute allows the client to choose the granularity and cost of timestamping. Timestamping is optional and timestamps are not written to a channel unless explicitly requested. Events are timestamped by relay_write() timestamp events using either the efficient TSC (or equivalent cheap clock on other architectures), or a slower but globally consistent gettimeofday() time delta method. gettimeofday() is also a fallback in the cases where the TSC, or an equivalent clock counter, is unavailable on a given architecture. In brief, part of the task of writing an event involves obtaining the current TSC, or the current gettimeofday() value. If the gettimeofday() option is chosen each timestamp is the difference between the current time and the time when the buffer was started. The value logged is either the TSC value or this difference and is written at the offset within the event slot specified by the td_offset parameter of relay_write() (if the parameter value is negative, the timestamp is not written). The start and end buffer gettimeofday() and TSC (if applicable) values are available as parameters to the buffer_start() and buffer_end() kernel client callbacks. If a client is interested in timestamping, it can write these values into the reserved space for later inter-buffer correlation.

4.5 Channel overflow handling

As with any buffering scheme, data may be written into a channel faster than the channel's clients can read it out. relayfs channel clients have three options for dealing with an overflow:

- do nothing, writers overwrite old data (flight recorder mode)
- suspend writing into the channel, causing loss of new events
- resize the channel, making more space for writers

The first two options are controlled by the return value of the buffers_full() callback. This callback provides the client with the option of what action to take if the consumer has not kept pace with the logging. A value of 0 directs relayfs to continue logging events, overwriting the oldest data. A value of 1 directs relayfs to discard subsequent events until the overflow situation has been resolved. In this case, an events_lost count is kept and is available via relay_info(). Once the consumer (usually the user-space daemon reading from the channel) has caught up, the relayfs client can call relay_resume() to allow the channel to continue logging events. To implement this, the client needs to keep the channel informed of how many buffers it has read. It increments the count of buffers consumed by calling relay_buffers_consumed(n_buffers). This value is compared with the channel's count of buffers produced (tracked on the buffer-switch slow path) to determine whether a buffers-full condition exists. If the difference is greater than or equal to the number of sub-buffers in the channel, the buffers are considered full and the callback is invoked.

The third option, resizing the channel, is available to clients that have specified non-zero values for the resize_min and resize_max parameters to relay_open() when the channel was created. If (during the buffer-switch slow path)

relayfs detects that the channel is almost full (if 3/4 of the sub-buffers remain unread, by default), the needs_resize() callback is invoked with parameter values containing a suggested new sub-buffer size and/or sub-buffer count, which can be used to expand the buffer space. The client can use these values (or ignore them and supply its own) to allocate a new buffer for the channel via the API function relay_resize_channel(). This function can block, so it should not be called with spinlocks held. If called from user context, it directly allocates the new buffer, which is available upon return. If called from within interrupt context, the allocation is put onto a work queue, and the client is notified upon completion via another call to the needs_resize() callback. Once the new buffer is allocated, the client can call relay_replace_buffer() to replace the channel's buffer. This function can be called from any context. Clients call it when they can guarantee the replacement does not interfere with other channel activity, such as outstanding writes. Reducing the buffer size follows a similar path. When relayfs detects that the "almost-full" condition has not existed for a period of time (1 minute by default), the needs_resize() callback is invoked with the new suggested (smaller) sub-buffer values. Buffer reduction is handled by the client in a similar manner to buffer expansion. Clients can choose to ignore the details of buffer resizing. To do so, they specify a non-zero value for the autoresize parameter to relay_open() causing buffer re-allocation requests to be placed onto a work queue. The resizing strategy reflects empirical observations that channel traffic tends to be bursty in nature with sudden activity creating immediate short-term need for increased buffer capacity, which after a short period is no longer needed.

Config.	avg time per run (in seconds)	difference
1	756.1	
2	766.7	+1.40%
3	771.3	+2.01%

Figure 5: LTT on relayfs test results.

5 Testing

To test the efficiency of relayfs, we ran LTT, a very demanding client of relayfs, while performing 10 kernel compiles under each of the following conditions:

1. not tracing anything (baseline)
2. tracing everything, daemon not writing to disk
3. tracing everything, daemon writing to disk

Testing was performed on a 4-way 700MHz Pentium III system. The tests generated large amounts of data for relayfs to process. Approximately 200 million events comprising about 2 gigabytes were generated during each 10-compile run. As can be seen from the results in Figure 5, the overhead of relayfs was at most 1.4 percent (a portion of this overhead is in fact due to tracing code), demonstrating the efficiency of relayfs.

6 Example client subsystems

In this section we examine some practical uses of relayfs. In particular, we discuss how relayfs can be used as an engine for printk and LTT, how it can be used for driver debugging, and how it could become a replacement for rvmalloc and rvfree.

6.1 printk

We have developed a version of printk that replaces the static printk buffer with a dynamically resizeable relayfs channel. This solves the lost printk problem by providing reliable printk logging services. A dynamically resizeable channel prevents lost printk messages in normal usage, but it can also prevent the loss of init-time printk messages. At init-time (before free_initmem()), the contents of the static printk buffer are copied into the relayfs channel created for printk. The static printk buffer used at init-time is marked as __initdata and is subsequently discarded. A benefit of this scheme is that the printk kernel buffer can be made relatively large. In fact, it can be large enough so that it does not overflow even when copious boot messages are printed on a large system. The relayfs version of printk modifies the syslog(3) system call to read from the printk channel instead of from the static kernel buffer. Since /proc/kmsg also uses syslog(3) to retrieve data from the kernel buffer, user-space programs that read from the printk buffer do not need to be modified to use the relayfs version of printk. The relayfs version of printk adds commands to syslog(3) allowing user-mode clients to manually resize the printk channel; these of course must be coded for. See the relayfs web page[1] for code and status.

6.2 Linux Trace Toolkit

LTT creates a separate bulk-delivery channel for each processor. The LTT kernel client replicates the inner workings of relay_write by using the relayfs low-level API described briefly in Documentation/filesystems/relayfs.txt. More detail can be found by examining trace() in kernel/trace.c and the relay_write() implementation. It uses the low-level API to obtain maximum performance from relayfs, because system tracing is one of the most demanding clients. We would expect most subsystems to function acceptably using the described APIs, but the low-level API is available for those requiring maximum performance. The low-level API allows a client to write directly into the reserved slot in the channel, rather than passing the address of a buffer to relay_write for it to copy. By doing so, LTT avoids the overhead of forcing trace() to collate its fields into a separate buffer before passing it to relay_write, and avoids putting the event data on the stack (in reality this would not be possible because some events can be 8K in length). This is more convenient than having trace() manage a staging buffer between potentially multiple writers.

```
rchan = rchan_get(channel_handle);
if (rchan == NULL)
   return -ENODEV;

/* this is a nop for lockless */
relay_lock_channel(rchan, flags);

reserved =
   relay_reserve(rchan,
        data_size, &time_stamp,
        &time_delta, &reserve_code,
        &interrupting);

if (reserved == NULL)
   goto check_buffer_switch_signal;

bytes_written +=
   relay_write_direct(reserved,
       &event_id, sizeof(event_id));
bytes_written +=
   relay_write_direct(reserved,
         &data, sizeof(data));

relay_commit(rchan, reserved,
      bytes_written, reserve_code,
      interrupting);

relay_unlock_channel(rchan, flags);
rchan_put(rchan);
```

In the above code, the first few tasks are bookkeeping tasks. These involve getting the channel structure backing the channel handle, and locking the channel (if applicable—for the

lockless scheme, this is effectively a no-op). We then reserve a slot, and using the special relay_write_direct() low-level method (essentially a memcpy), write fields directly into the channel buffer. When we are done writing the fields, we indicate to the channel that the data is ready by 'committing' the slot, and then finish up with a couple more bookkeeping tasks.

See the relayfs web page[1] for code and status.

6.3 Driver debugging

It is straightforward to create and use a relayfs channel for driver tracing/debugging. Open a channel with the desired_attributes:

```
channel = relay_open("channel", ...);
```

and write to it from within your driver:

```
relay_write(channel, ...)
```

Then write a simple user-space program that loops around read():

```
fd = open("/mnt/relay/channel", ...);

for(;;)
   read(fd, ...);
```

6.4 rvmalloc/rvfree replacement

At last count there were 9 drivers with separate implementations of rvmalloc/rvfree. There have been discussions in the past of exporting a single 'blessed' instance of rvmalloc/rvfree, but so far that has not happened. Since relayfs channels are based on rvmalloc/rvfree, relayfs provides the equivalent of a public rvmalloc/rvfree by providing clients a means to create rvmalloc'ed buffers using degenerate values to relay_open() e.g. a large frame buffer could be allocated via relay_open with most parameters 0/NULL:

```
int channel_id =
  relay_open(
    "framebuffer",
    bufsize, /* size of sub-buffer */
    1, /* one sub-buffer only */
    0, /* no flags in this case */,
    NULL, /* no callbacks */
    0, /* no reserve */
    0, /* no reserve */
    0, /* no reserve */
    0, /* no resize_min */
    0, /* no resize_max */
    0, /* don't autoresize */
    NULL); /* no special file ops */
```

The above opens a relay channel with a filename of /mnt/relay/framebuffer, assuming relayfs is mounted at /mnt/relay.

Information about the channel can be retrieved using relay_info().

```
relay_info(channel_id, &rchan_info);
```

The data contained in rchan_info includes the virtual address of the buffer allocated for the channel via rvmalloc. This can be directly used to write into the buffer memory from the kernel side.

The file created for the channel can then subsequently be mmap()'ed into user space:

```
int framebuf_file =
  open("/mnt/relay/framebuffer",
       ...);
char * framebuf = mmap(NULL, bufsize,
  PROT_READ|PROT_WRITE, MAP_PRIVATE,
  framebuf_file);
```

Note that if a channel is mmap()'ed, it cannot be resized, but a client wishing to resize the channel can unmap the file, resize it, then remap it.

Finally, closing the channel frees the buffer via rvfree():

```
relay_close(channel_id);
```

7 Conclusion

We have presented relayfs, an efficient and unified mechanism for transferring data from kernel to user space. relayfs addresses the need

to have a reliable mechanism that can be used across various kernel subsystems, without having to replicate and separately maintain the functionality for each subsystem. The lockless and per-processor techniques allow efficient logging on multiprocessor systems, and the channels provide flexibility to the clients. Test results show that relayfs performs well even under demanding workloads. We presented several example kernel subsystems that use relayfs including printk, LTT, and driver debugging. As relayfs becomes more widely accepted, other kernel subsystems will be able to use it, reducing code replication and improving reliability of the kernel subsystems.

References

[1] relayfs home page, `http://www.opersys.com/relayfs`.

[2] Robert W. Wisniewski and Luis F. Stevens. A model and tools for supporting parallel real-time applications in unix environments. In *Proceedings of The 12th IEEE Real-Time Technology and Applications Symposium*, pages 126–133, Chicago Illinois, May 15-17 1995.

Linux IPv6 Networking

Past, Present, and Future

Hideaki Yoshifuji
The University of Tokyo
yoshfuji@linux-ipv6.org

Kazunori Miyazawa
Yokogawa Electric Corporation
miyazawa@linux-ipv6.org

Yuji Sekiya
The University of Tokyo
sekiya@linux-ipv6.org

Hiroshi Esaki
The University of Tokyo
hiroshi@wide.ad.jp

Jun Murai
Keio University
jun@wide.ad.jp

Abstract

In order to deploy high-quality IPv6 protocol stack, we, USAGI Project[13], have analyzed and addressed issues on Linux IPv6 implementation. In this paper / in our talk in OLS2003, we describe the analysis of Linux IPv6 protocol stack, improvements and implementation of the IPv6 and IPsec protocol stack and patches which are integrated into the mainline kernel. We will explain the impacts of our API improvements on network applications.

We want to discuss on missing pieces and direction for future development.

As a demonstration we would like to provide IPv6 network connectivity to the OLS2003 meeting venue.

1 Introduction

Establishment of IPv6, as a next-generation Internet protocol to IPv4, started from the beginning of the 1990's. The aspect of IPv6 is on providing the solution to the protocol scalability, the greatest problem IPv4 facing as the Internet growing larger. In detail, IPv6 differ from IPv4 in following ways.

- 128bit address space.
- Forbidding of packet fragmentation in intermediate routers.
- Flexible feature extension using extension headers.
- Supporting security features by default.
- Supporting Plug & Play features by default.

Currently, IPv6 is at the final phase of standardization. Fundamental specifications are almost fixed and commercial products with IPv6 support are being deployed in the market. International leased lines for IPv6 are out as well. IPv6 has expanded the existing Internet by providing solutions to protocol scalability and beginning to grow as a standard for connecting everything, not just existing computers.

2 The Dawn of Linux IPv6 Stack

Linux IPv6 implementation was originally developed by Pedro Roque and integrated into mainline kernel at the end of 1996 in early 2.1 days and this was the one of the earliest implementation of IPv6 stack in the world.

In 1998, Linux IPv6 Users JP, which is a group of Linux IPv6 users in Japan, examined the status of IPv6 implementation in Linux and recognized the several grave issues[1].

- lack of scope in socket API; for example Linux does not have `sin6_scope_id` member in `sockaddr_in6{}`.
- So many bugs (Table 1), found by the TAHI [11] IPv6 Conformance Test Suite, especially in Neighbor Discovery and Stateless Address Auto-configuration.
- "default routes" are ignored on routers
- many missing features such as IPsec, Mobile IP.

These were because the stack had not been well-maintained / developed since 2.1 because there were not so widely used by Linux hackers. Thus, there had been few new features. Implementation had not followed the specification even the spec had been changed, and then, conformity to Specification became very low.

In 2.3 days, they, the Linux IPv6 Users JP, developed `sin6_scope_id` support. Their code was integrated into mainline kernel. There, however, were very few change in 2.3 days other than this.

Considering above circumstances, USAGI Project was lunched in October, 2000. USAGI Project is a project which aims to provide improved IPv6 stack on Linux; It seems to be required (almost) full-time task-force which commits Linux IPv6 development. There are similar organization called KAME [6], which provides IPv6 stack on BSD Operating systems such as FreeBSD, NetBSD, OpenBSD, and BSD/OS. However, KAME Project does not target their development on Linux. It is important to provide high-quality IPv6 stack on Linux, which is one of the most popular free open-source operating systems in the world, and widely used in embedded systems, for IPv6 to propagate.

[1] We will discuss them later.

Table 1: Summary of TAHI Conformance Test (linux-2.2.15, %)

Test Series	Pass	Warn	Fail
Spec.	94	6	0
ICMPv6	100	0	0
Neighbor Discovery	34	0	66
Autoconf	4	0	96
PMTU	50	0	50
IPv6/IPv4 Tunnel	100	0	0
Robustness	100	0	0

3 USAGI Challenges in Linux IPv6 Stack

Since USAGI Project started, we have continued analyzing issues we faced. In this section, we describes issues we found and our challenges to solve them.

3.1 ND and Addrconf

Neighbor Discovery (ND [8]) and Stateless Address Auto-configuration (Addrconf, [12]) are ones of the core features of IPv6. They take very important role to keep stable communication. However, the results of the Conformance Tests of Linux IPv6 stack were bad.

We've tried to fix the problems in the following way.

- Reinforcing checking illegal ND Messages
- Improving control times for ND state transition and address validation.

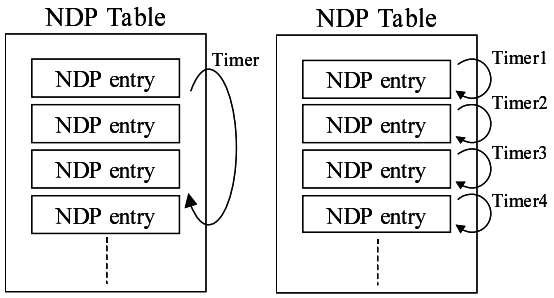


Figure 1: NDP Table: Linux vs USAGI

- Fixing ND state transition

3.1.1 Improving Timers for ND

The state of a neighbor is changed by events such as incoming Neighbor Advertisement message and timer expiration. It is required to manage timer accurately.

However, the existing Linux IPv6 protocol stack checks reachability of neighbor nodes with a single kernel timer(Figure 1 (Left)). Consequently, reachability were checked in constant intervals, regardless of the status for each node.

Therefore, USAGI Project improved this kernel timer to check each NDP entry independently as shown in Figure 1 (Right). Thus, resource management for a neighbor including mutual exception is simplified, and it is possible to enable and disable timer separately for each NDP entry, and prevent check made to unnecessary NDP entries. Moreover, it is possible to exchange messages correspondent to the status of each NDP entry as defined in the NDP specifications.

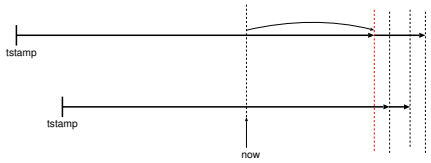


Figure 2: Dynamic Address Validation Timer

3.1.2 Improving Timers for Address Validation

As ND is, the state of an address is changed by events such as incoming Router Advertisement and time expiration. It is required to manage timer accurately, especially for Privacy Extensions [7].

However, the existing Linux IPv6 protocol stack performs validity checks with a long-term, constant, single kernel timer.

USAGI Project introduced new dynamic timer. When the timer expires, the timeout function visits each address for validation and determining the next timeout (Figure 2) with minimum and maximum interval between timeouts. Thus, accuracy of timer is improved. It is usual that several addresses request next timeout at (almost) the same time, introducing minimum interval between operations aggregates them and suppresses load of timer events.

Table 2 shows the results of these improvements and other minor fixes. Neighbor Discovery and Autoconf are significantly improved.

3.2 Routing Restructuring

3.2.1 Default Route Support on Routers

In routing table using radix tree[9], the top of the tree is the host which possesses the information regarding “default route.” However, as shown in Figure 3, Linux IPv6

Table 2: Summary of TAHI Conformance Test (usagi24-s20020401, %)

Test Series	Pass	Warn	Fail
Spec.	100	0	0
ICMPv6	100	0	0
Neighbor Discovery	79	5	15
Autoconf	98	2	0
PMTU	50	0	50
IPv6/IPv4 Tunnel	100	0	0
Robustness	100	0	0

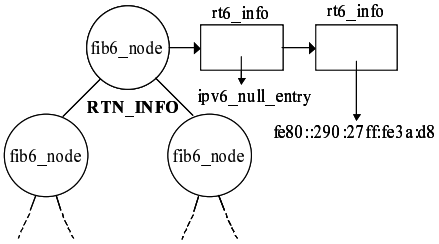


Figure 3: Linux IPv6 Routing Table Structure

protocol stack has a radix tree with fixed node information on top and it points to `ipv6_null_entry`. Therefore, when default route is added, the information is attached next to the `rt6_info{}` structure which contains `ipv6_null_entry`. This causes default route not to be referred.

In USAGI implementation, we replace the `ipv6_null_entry` with the new entry when adding a new routing entry on the top level root of the tree (Figure 4). When the last route is being deleted from the the top level root of the tree, we re-insert `ipv6_null_entry`. Thus, we can insert and remove the "default route" entries properly to/from the routing table.

3.3 Improvements on Router Selection

We pick one default router from the default router list and round-robin the de-

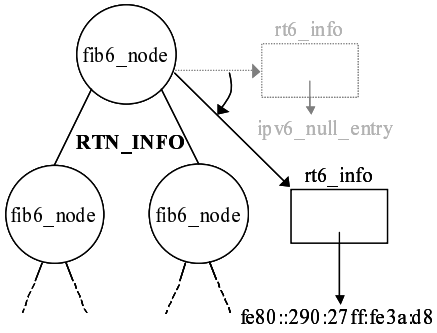


Figure 4: USAGI IPv6 Routing Table Structure

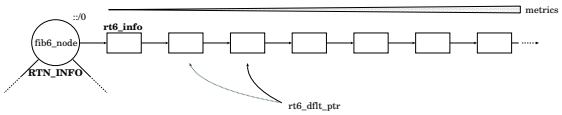


Figure 5: Default Routers in Linux

fault router list when it becomes unreachable. The default router is pointed by the `rt6_dflt_pointer`, which is guarded by `rt6_dflt_lock`, and default routers are stored on the top level root node of the routing tree (Figure 5). In this implementation, there were several issues.

- `rt6_dflt_pointer` is reset when routing is modified; this happens very often and routers are not equally selected.
- We did not regard the metrics; we could not force using routes with smaller metrics (which is probably added manually.)

"Default Router Preferences, More-Specific Routes, and Load Sharing" [1] improves the ability of hosts to pick an appropriate router, especially when the host is multi-homed and the routers are on different links, and mandates load-share between routers with same "preferences."

To implement this specification, we stores preference (2 bits) of routes into the flags of the

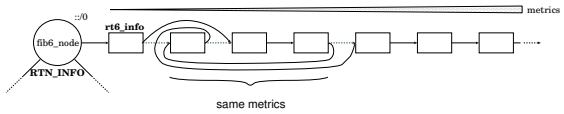


Figure 6: New Method for Route Round-robin

routing informations instead of reflecting it to the metrics; We would have to fix up routing table when receiving RA.

We also make a new generic round-robin code for the routes with same metrics (Figure 6). We use this for all routes and the `rt6_dflt_pointer` and `rt6_dflt_lock` are eliminated. Now we are free from above issues.

4 Linux IPv6 in 2.6

In this section, we describes key changes of IPv6 networking code between 2.4 and 2.6 [2] Then we try to describes how IPsec works.

4.1 Key Changes in 2.6.x

We have been developing IPv6 actively since end of 2.3.x era. However, only several selected changes were integrated into the mainline tree. One reason was that we were obscure and novice on kernel development.

After experiencing about two years of kernel development, we started integrating our efforts to the mainline kernel from the fall of 2002 more aggressively than before.

We have been being fixing several bugs such as:

- Verify ND options properly
- Refine IPv6 Address Validation Timer
- Fixing source address of MLD messages
- Avoiding garbage sin6_scope_id for MSG_ERRQUEUE message

In addition to these bug fixing, we've integrated following new features:

IPsec for IPv6
: This is based on IPsec for IPv4, developed by David S. Miller et.al. See section 4.2.

Default route support on router
: See section 3.2.1.

IPV6_V6ONLY support
: See section 5.2.4.

ICMP6 rate limit support
: Added rate-limit sysctl for ICMPv6 like for ICMP.

Privacy Extensions [7]
: Assign randomized interface identifier to improve privacy.

AF-independent XFRM Infrastructure
: Split up XFRM subsystem into af-independent portion and af-specific portion. Section 4.2.3.

Per-interface Statistics Infrastructure
: Make a new infrastructure to provide per-interface statistics information.

4.2 IPsec

IP security provides security functionality for IP layer. An implementation of IPv4 IPsec by FreeS/WAN is available for years, however, the code was never merged into the mainline kernel. In 2000, IABG Project provided IPv6 support patch for FreeS/WAN. It, however, was "patched" and also unlikely to be merged into the mainline kernel.

[2] Some changes will be appeared in 2.4.21 (or later 2.4.x series).

We redesigned the architecture for multi-protocol, both IPv4 and IPv6, extensible IPsec. In our design, IPv4 and IPv6 share the Security Policy Database (SPD) and Security Association Database (SAD). CryptoAPI and its variants are used for cryptographic, digesting and compression/decompression algorithms.

4.2.1 Stackable Destination and XFRM

A new framework for processing IP packets has been introduced into linux-2.5.x. It is called "stackable destination" and XFRM.

Stackable destination is like a linked list of `dst{}`, which is made temporally and cached. We are able to insert another `dst{}` to original `dst{}` and make a stack of the `dst{}` structure. `dst{}` normally has a pointer to `xfrm_state{}`, whose output provides some functionality, i.e. transformation, for the packet.

XFRM stands for transformer. `xfrm_policy{}` and `xfrm_state{}` represent IPsec policy and IPsec SA respectively. `xfrm_state{}` is associated with `xfrm_policy{}` by `xfrm_tmpl{}`. SPD consists of `xfrm_policy{}`. SAD also consist of `xfrm_state{}`.

4.2.2 Packet Processing

The output process of IPsec fully uses this architecture. The order of primal functions are `xfrm_lookup()`, `xfrm_tmpl_resolve()`, `xfrm_bundle_create()` and `dst_output()`. `xfrm_lookup()` looks up `xfrm_policy{}` in SPD after routing resolution. At the moment the parameter `dst{}` in the stack points original `dst{}` structure. `xfrm_tmpl_resolve()` is called in `xfrm_lookup()` to resolve `xfrm_tmpl{}` in `xfrm_policy{}` which represents how the packet is processed and find `xfrm_state{}` matched up with `xfrm_tmpl{}`. This process is equivalent to looking up IPsec SA or IPsec SA bundle matched with IPsec policy. `xfrm_bundle_create()` creates the stackable destination and IPsec SA bundle if multiple SA are needed. These functions are called at routing resolution. `dst_output()` is called after building up the packet. Each output routine specified by the function pointer in the `dst{}` is called along with the chain of `dst{}`. This pointer points e.g `esp6_output()`. The output function is able to use `xfrm_state{}` from `dst{}` pointer in `sk_buff{}`.

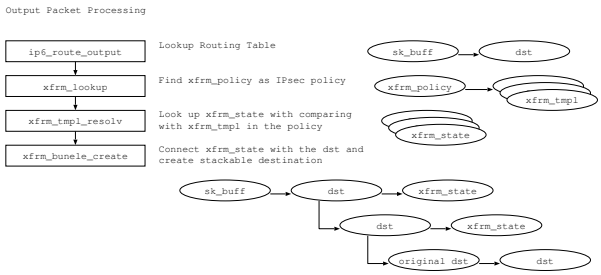


Figure 7: IPsec output process

The input process for IPsec is more simple than output. AH and ESP process routines are registered to `inet6_protos[]` at initiation. The kernel parse a packet and call the routines when protocol is AH or ESP. To unified extension header processing, all header types and handlers are registered in `inet6_protos[]` like upper layer protocol. IPsec packet process is looking up `xfrm_state{}` and process it. When it succeeds, used `xfrm_state{}` pointer keep in `sec_path{}` in `sk_buff{}` which contains the packet. After processing IPsec, the kernel call `xfrm_policy_check()` at entrance of upper layer process. In `xfrm_policy_check()` the kernel match up `xfrm_tmpl{}` in `xfrm_policy{}` and

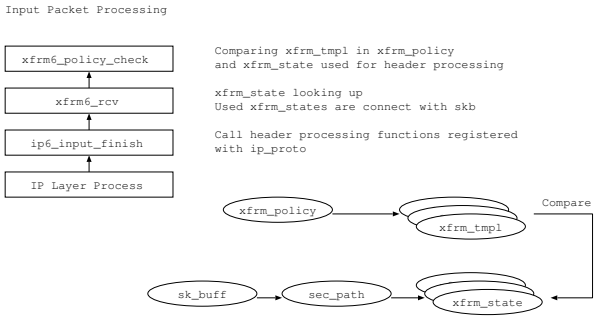


Figure 8: IPsec input process

`xfrm_state{}` kept in `sec_path{}`.

4.2.3 AF Indenendent XFRM Infrastructure

Since core functionality of the XFRM engine is common among address families, AF independent XFRM infrastructure has been introduced.

Address family specific XFRM functions are registered via address family information tables, e.g. `xfrm_policy_afinfo{}` and `xfrm_state_afinfo{}`. Common variables are also passed via the tables.

4.2.4 Key And Policy Management Interface

`PF_KEY` and `netlink(7)` interface are provided as IPsec interfaces. `PF_KEY` provides interface to maintain SAD and SPD. `PF_KEY` protocol is defined in RFC2367 but it is not so enough to maintain IPsec that implementation of `PF_KEY` is ordinary extended. The extension is different each implementation. Linux-2.5.x `PF_KEY` is compatible with KAME.

4.2.5 Test Results

On 24th April, 2003, Tom Lendacky reported to netdev mailing list that Test results of Linux-2.5 IPsec are very excellent(Table 3).

We have tried to fix the bugs in IPv6 IPsec fragmentation, and they should be fixed for now.

Test Series	Pass	Warn	Fail
ipsec	95	2	3
ipsec4	98	2	0
ipsec4-udp	96	4	0

Table 3: Summary of TAHI Conformance Test (linux-2.5.58, %)

5 Modern Programming Style for Network Applications

It is requested that applications should support both IPv4 and IPv6. In this section, we try to describe modern programming style for network applications.

5.1 Socket API and Protocol Independency

The Socket API is the framework of programming for communication including networking via the Internet. It was designed to be protocol independent. Communication is abstracted by the socket descriptor, and endpoint information, which is protocol dependent, is passed via opaque pointers to the generic socket address structure `sockaddr`.

IPv6 networking is also supported in this framework. New address family `AF_INET6` and IPv6 socket address structure `sockaddr_in6` are defined.

5.2 Protocol Independent Programming

The framework of the Socket API between the kernel and the user space is basically protocol independent. However, since the socket address structure and the naming space of the protocol depend on the protocol (or address), it was protocol dependent to lookup the name/address and to setup the protocol specific socket address structure. This prevented application from the protocol independency.

In RFC2133[3], new two name-lookup functions are defined: `getaddrinfo()` and `getnameinfo()`. It abstracts translation between name/service representation and socket address structure.

5.2.1 getaddrinfo(3)

`getaddrinfo(3)` [2] is the protocol independent function for forward lookup (name to address). This function looks up the "node" and "service" on condition that is specified by the "hints." It returns dynamically allocated linked list of `addrinfo{}`. Each `addrinfo{}` includes information for `socket(2)`, `connect(2)` (or `bind(2)`, if `AI_PASSIVE` flag is specified in hints).

Thus, application is not required to know the details of socket address structure, now. A client application walks though the list trying to create a socket and trying to connecting remote host until one of the attempt succeeds. Likewise, a server application walks through the list trying to create a socket and trying to binding local address.

5.2.2 getnameinfo(3)

`getnameinfo(3)` [2] is the protocol independent function for reverse lookup (address to name). This function takes socket address structure and looks up node name and service name on condition specified by the flags. By using this function, application is not required to know the details of each socket address structure for extracting addresses and / or port number.

For example, to extract numeric service number from the socket address structure, use `getnameinfo(3)` with `NI_NUMERICSERV`, and convert the resulting service number using `atoul(3)`.

Sample programs using `getaddrinfo(3)` and `getnameinfo(3)` are provided in Appendix.

5.2.3 getifaddrs(3)

Other issue to support IPv6 in application how we know addresses on the node on which it is running. SIOCGIFADDR is used for IPv4, however, `ifreq{}` is not enough to store IPv6 socket address structure. There might be possibility to introduce new `ioctl(2)` to manage lager addresses, however, it is nasty to get information via buffer of fixed length. Thus, `getifaddrs(3)` was invented.[3] This function grubs network address information including netmask etc. on the node. MAC, IPv4 and IPv6 are supported for now. Application walks through the linked list returned from this function, looking for appropriate information using the family, flags etc.

Sample programs using `getifaddrs(3)` is provided in Appendix.

[3] BSDI's invention; this is not standardized yet.

5.2.4 IPV6_V6ONLY Socket Option

IPv6 sockets may be used for both IPv4 and IPv6 communications. IPv4-mapped IPv6 address is defined [5] to enable IPv6 application IPv4 address of an IPv4 node is represented as an IPv4-mapped IPv6 address in such applications.

Linux supports this feature and port space of TCP (or UDP) has been completely shared between IPv4 and IPv6 [4].

However, some applications may want to restrict their use of an IPv6 socket to IPv6 communication only. For these applications, `IPV6_V6ONLY` is defined in RFC3493 [2].

In 2.6, `IPV6_V6ONLY` socket option is supported. In this implementation, the 'IPv4-mapped" feature is enabled by default as before, and as spec says. If the `IPV6_V6ONLY` socket option is set to the IPv6 socket, the socket will not care about IPv4 address space at all.

As mentioned before, spec says this "IPv4-mapped" feature is enable by default. However, there are OSes, such as NetBSD, which do not enable that feature by default, or even which do not support that feature at all. These OSes are not RFC compliant, but, unfortunately, it is the real. So, application would need to take care of this situation. For example:

- Try to setup both IPv6 and IPv4 sockets.
- Set `IPV6_V6ONLY` socket option to the IPv6 socket, prior to perform `bind(2)`.
- It is not fatal to failed to set `IPV6_V6ONLY` socket option.
- Don't take it fatal unless all socket creation resulted in error.

The sample server provided in the Appendix is written in this manner.

[4] In some OSes, such as FreeBSD 4.x, IPv4 socket can override IPv6 socket of the same port. We believe that this fashion is vulnerable to "binding closer" type attacks.

6 Future Plans

Finally, we list our future plans. Here's our future plan, especially for this year.

- stabilizing IPv6 IPsec
- introducing IP Mobility to mainline
- completing porting ND fix to (pre-)linux-2.6 and submitting patches to mainline
- introducing generic IPv4,6 tunnel interface
- examining and stabilizing IPv6 Netfilter
- completing Prefix Delegation
- improving API support [2, 10]
- implementing IPv6 Multicast Routing

As of writing this paper, we're working hard stabilizing IPv6 and IPv6 IPsec stack in (pre-)linux-2.6.

We're also discussing how the Mobile IP should be implemented with the maintainers and HUT [4] people.

XFRM is flexible and promising framework in netwroking. We are able to and going to implement Mobile IP, generic tunnel etc.

We have been waiting for new documents for the APIs. It has taken long time to publish the new documents for APIs, however, new version are (about to) available. We'll follow them.

Multicast routing is probably the biggest missing piece; we will try to implement this, too.

References

[1] R. Drave and R. Hinden. Default router preferences, more-specific routes, and load sharing. Work in Progress, June 2002.

[2] R. Gilligan, S. Thomson, J. Bound, J. McCann, and W. Stevens. Basic Socket Interface Extensions for IPv6. RFC3493, March 2003.

[3] R. Gilligan, S. Thomson, J. Bound, and W. Stevens. Basic Socket Interface Extensions for IPv6. RFC2133, April 1997.

[4] GO Project. MIPL Mobile IPv6 for Linux. http://www.mipl.mediapoli.com/.

[5] R. Hinden and S. Deering. Ip version 6 addressing architecture. RFC2373, July 1998.

[6] KAME Project. KAME Project Web Page. http://www.kame.net.

[7] T. Narten and R. Draves. Privacy extensions for stateless address autoconfiguration in ipv6. RFC3041, January 2001.

[8] T. Narten, E. Nordmark, and W. Simpson. Neighbor Discovery for IP Version 6 (IPv6). RFC2461, December 1998.

[9] Keith Sklower. A tree-based packet routing table for berkeley unix. In *USENIX Winter*, pages 93–104, 1991.

[10] W. Stevens, M. Thomas, E. Nordmark, and T. Jinmei. Advanced sockets api for ipv6. Work in Progress, March 2003.

[11] TAHI Project. Test and Verification for IPv6. http://www.tahi.org.

[12] S. Thomson and T. Narten. IPv6 Stateless Address Autoconfiguration. RFC2462, December 1998.

[13] USAGI Project. USAGI Project Web Page. http://www.linux-ipv6.org.

7 Appendix: Sample Application Written in Modern Manner

7.1 Client

```
/*
 * Sample Modern Client
 *
 * Usage:
 * % ./modern-client host.example.com daytime
 *
 * $Id: modern-client.c,v 1.1 2003/05/13 20:06:58 yoshfuji Exp $
 */

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>

int main(int argc, char **argv) {
    char *host, *port;
    struct addrinfo hints, *ai0, *ai;
    int s;
    int gai;

    /* check arguments */
    if (argc != 2 && argc != 3) {
        fprintf(stderr, "Usage: %s [host] portnum\n", argv[0]);
        exit(1);
    }

    if (argc == 3) {
        host = argv[1];
        port = argv[2];
    } else {
        host = NULL;    /* loopback address */
        port = argv[1];
    }

    /* look-up name */
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;
    hints.ai_flags = 0;

    gai = getaddrinfo(host, port, &hints, &ai0);
    if (gai) {
        fprintf(stderr,
                "getaddrinfo(): %s port %s: %s\n",
                host, port, gai_strerror(gai));
```

```
    exit(1);
}

/* loop connecting remote entity */
s = -1;
for (ai = ai0; ai; ai = ai->ai_next) {
    /* create a socket */
    s = socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
    if (s == -1)
        continue;

    /* connect */
    if (connect(s, ai->ai_addr, ai->ai_addrlen) == 0)
        break;

    close(s);
    s = -1;
}

/* free address information */
freeaddrinfo(ai0);

/* check if we have failed */
if (s == -1) {
    fprintf(stderr, "Cannot connect to %s port %s\n",
            host != NULL ? host : "(null)",
            port);
    exit(1);
}

/* process loop */
while (1) {
    ssize_t cc;
    char buf[1024];

    /* read from remote host */
    cc = read(s, buf, sizeof(buf));
    if (cc == -1) {
        perror("read");
        close(s);
        exit(1);
    } else if (cc == 0) {
        break;
    }

    /* output response */
    if (write(STDOUT_FILENO, buf, cc) == -1) {
        perror("write");
        close(s);
        exit(1);
    }
}

close(s);
```

```
    exit(0);
}
```

7.2 Server

```
/*
 * Sample Modern Server
 *
 * Usage:
 * % ./modern-server :: 12345
 *
 * $Id: modern-server.tex,v 1.1 2003/05/15 03:54:13 yoshfuji Exp $
 */

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <stdlib.h>
#include <sys/select.h>

#ifndef MAX_SOCKNUM
# define MAX_SOCKNUM FD_SETSIZE
#endif

static const char *message = "Hello, world!\n";

int main(int argc, char **argv) {
    char *host, *port;
    struct addrinfo hints, *ai0, *ai;
    int gai;
    int socknum = 0, *socklist = NULL;
    int maxfd = -1;
    fd_set fds_init, fds;
    int i;

    /* check arguments */
    if (argc != 2 && argc != 3) {
        fprintf(stderr, "Usage: %s [host] portnum\n", argv[0]);
        exit(1);
    }

    if (argc == 3) {
        host = argv[1];
        port = argv[2];
    } else {
        host = NULL;    /* unspecified address */
        port = argv[1];
    }
```

```
    /* resolve address */
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;
    hints.ai_flags = AI_PASSIVE;

    gai = getaddrinfo(host, port, &hints, &ai0);
    if (gai) {
        fprintf(stderr,
                "getaddrinfo(): %s port %s: %s\n",
                host != NULL ? host : "(null)",
                port,
                gai_strerror(gai));
        exit(1);
    }

    /* initialize fd_set for select(2) */
    FD_ZERO(&fds_init);

    /* loop waiting for connection */
    for (ai = ai0; ai; ai = ai->ai_next) {
        int s;
        int *newlist;
#ifdef IPV6_V6ONLY
        int on = 1;
#endif

        /* create a socket */
        s = socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
        if (s == -1)
            continue;

#ifdef IPV6_V6ONLY
        if (ai->ai_family == AF_INET6 &&
            setsockopt(s,
                       IPPROTO_IPV6, IPV6_V6ONLY,
                       &on, sizeof(on)) == -1) {
            perror("setsockopt(IPV6_V6ONLY)");
            /*
             * Some systems do not support his option;
             * This error should no be fatal.
             */
        }
#endif

        /* listen */
        if (bind(s, ai->ai_addr, ai->ai_addrlen) == -1) {
            close(s);
            continue;
        }

        if (listen(s, 5) == -1) {
            close(s);
```

```
            continue;
        }

        if (s >= FD_SETSIZE || socknum >= MAX_SOCKNUM) {
            close(s);
            fprintf(stderr, "too many file/socket descriptors\n");
            break;
        }

        /* re-allocate list of socket */
        newlist = realloc(socklist, sizeof(int)*(socknum+1));
        if (newlist == NULL) {
            perror("realloc");
            close(s);
            break;      /* XXX: terminate immidiately? */
        }

        socklist = newlist;
        socklist[socknum++] = s;

        /* set fd_set */
        FD_SET(s, &fds_init);

        if (maxfd < s)
            maxfd = s;
    }

    /* free address information */
    freeaddrinfo(ai0);

    /* check if we have failed */
    if (socknum == 0) {
        fprintf(stderr,
                "Cannot allocate any listen sockets on %s port %s\n",
                host != NULL ? host : "(null)",
                port);
        exit(1);
    }

    while (1) {
      int i;

      fds = fds_init;

      if (select(maxfd + 1, &fds, NULL, NULL, NULL) == -1) {
          perror("select");
          continue;
      }

      for (i = 0; i < socknum; i++) {
          int sock = socklist[i];

          /* look up listener.
           * XXX: this is not fair between listers
```

```
             */
            if (FD_ISSET(sock, &fds)) {
                int newfd;
                struct sockaddr_storage ss;
                socklen_t sslen;
                ssize_t cc;
                char hostbuf[NI_MAXHOST];
                int gni;

                sslen = sizeof(ss);
                newfd = accept(sock, (struct sockaddr *)&ss, &sslen);
                if (newfd == -1) {
                    perror("accept");
                    continue;
                }

                gni = getnameinfo((struct sockaddr *)&ss, sslen,
                                  hostbuf, sizeof(hostbuf),
                                  NULL, 0,
                                  NI_NUMERICHOST);
                if (gni)
                    strcpy(hostbuf, "???");       /*FIXME!*/

                printf("accept from %s\n", hostbuf);

                cc = write(newfd, message, strlen(message));
                if (cc == -1) {
                    perror("write");
                } else if (cc != strlen(message)) {
                    fprintf(stderr,
                            "write returned %d "
                            "while %d is expected.\n",
                            cc, strlen(message));
                }

                close(newfd);
            }
          }
        }

        /* we should not reache here */
        for (i = 0; i < socknum; i++)
            close(socklist[i]);
        free(socklist);

        exit(0);
    }
```

7.3 getifaddrs(3)

```
    #include <stdlib.h>
    #include <ifaddrs.h>
```

```
int main() {
    struct ifaddrs *ifa0, *ifa;
    int ret;

    ret = getifaddrs(&ifa0);
    if (ret) {
        perror("getifaddrs()");
        exit(1);
    }

    for (ifa = ifa0; ifa; ifa = ifa->ifa_next) {
        if (!ifa->ifa_addr)
            continue;
        switch(ifa->ifa_addr->sa_family) {
          case AF_INET:
            /* ifa->ifa_addr points sockaddr_in{} */
            /* ... */
            break;
          case AF_INET6:
            /* ifa->ifa_addr points sockaddr_in6{} */
            /* ... */
            break;
#if defined(AF_PACKET)
          case AF_PACKET:
            /* ifa->ifa_addr points sockaddr_ll{} */
            /* ... */
            break;
#endif
#if defined(AF_LINK)
          case AF_LINK:
            /* ifa->ifa_addr points sockaddr_dl{} */
            /* ... */
            break;
#endif
          default:
            /* not supported */
            ;
        }
    }
    freeifaddrs(ifa0);
    exit(0);
}
```

Fault Injection Test Harness

a tool for validating driver robustness

Louis Zhuang
Intel Corp.
louis.zhuang@intel.com,
louis.zhuang@acm.org

Stanley Wang
Intel Corp.
stanley.wang@intel.com

Kevin Gao
Intel Corp.
kevin.gao@intel.com

Abstract

FITH (Fault Injection Test Harness) is a tool for validating driver robustness. Without changing existing code, it can intercept arbitrary MMIO/PIO access and IRQ handler in driver.

Firstly I'll first list the requirements and design for Fault Injection. Next, we discuss a couple of new generally useful implementation in FITH

1. KMMIO - the ability to dynamically hook into arbitrary MMIO operations.

2. KIRQ - the ability to hook into an arbitrary IRQ handler,

Then I'll demonstrate how the FITH can help developers to trace and identify tricky issues in their driver. Performance benchmark is also provided to show our efforts in minimizing the impact to system performance. At last, I'll elaborate on current and future efforts and conclude.

1 Introduction

High-availability (HA) systems must respond gracefully to fault conditions and remain operational during unexpected software and hardware failures. Each layer of the software stack of a HA system must be fault tolerant, producing acceptable output or results when encountering system, software or hardware faults, including faults that theoretically should not occur. An empirical study [2] shows that 60-70% of kernel space defects can be attributed to device driver software. Some defect conditions (such as hardware failure, system resource shortages, and so forth) seldom happen, however, it is difficult to simulate and reproduce without special assistant hardware, such as an In-Circuit Emulator. In these situations, it is difficult to predict what would happen should such a fault occur at some time in the future. Consequently, device drivers that are highly available or hardened are designed to minimize the impact of failures to a system's overall functionality.

Developing hardened drivers requires employing fault avoidance software development techniques early in the development phase. To

eliminate faults during development and confirm a driver's level of hardening, a developer can test a device driver by injecting or simulating fault events or conditions. The focus of this paper is on the injection or simulation of hardware faults. Injection of software faults will be considered in future version.

FITH simulates hardware-induced software errors without modifying the original driver. It offers flexible customization of hardware fault simulation as well as provides command-line tool for facilitating test development. FITH can also provide the ability to log the route of an injected fault, thereby enabling driver developers to diagnose the tested driver.

2 Requirements

This section describes some requirements for FITH; we derived as part of the development.

The only behavioral requirement is that FITH should not impact functionality of the tested driver. The tested driver should work as if there is no FITH at all.

There are various functionality requirements that need to be considered. Most center around the interception of resources access. FITH needs to have capability to intercept accesses for MMIO, IO, IRQ and PCI configuration. The other major requirement is about handling after interception. FITH needs to have capability to support the complex and customized post-handling, such as tracing hardware status, emulating fake hardware register, injecting error data and logging.

With respect to performance, there are basically two overriding goals:

- minimize the impact to system performance when the tested driver doesn't enable FITH.
- minimize the number of instructions in critical kernel path, such as exception and interrupt part.

3 Architecture

FITH consists of four components: interceptors, faultsets, configuration tools, and a fault injection manager. The configuration tools provide the command-line utilities needed to customize a faultset for a driver. Three interceptors will be implemented to catch IO, MMIO and IRQ access. When a driver tries to access a hardware resource, the IO interceptor captures this access and asks the fault injection manager if there is a corresponding item in the faultset. If there is, the fault injection manager determines how to inject the appropriate fault according to the associated properties defined in the faultset. The fault injection manager then returns this information to the interceptor, so the interceptor injects the actual fault into the hardware.

IRQ fault injection is somewhat different from the other types of fault injection. Hardware triggers the IRQ, and a kernel IRQ interrupt handler delivers this event to the IRQ interceptor. The IRQ interceptor then checks the faultset to determine whether a specific fault is available to inject into the event. 1 illustrates how the interceptor interacts with the other subsystems.

The interceptor sits between the hardware and the device driver and modifies information based on certain conditions in the driver's namespace. An interceptor for one driver does not affect the interceptor for others. As a matter of fact, the hardware that the driver observes is the hardware that our interceptor wraps.

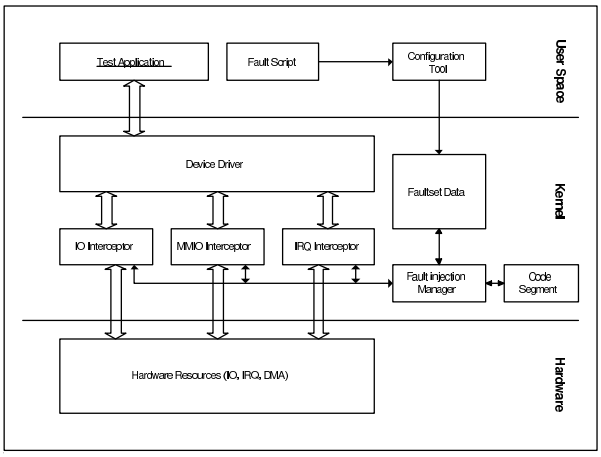


Figure 1: Architecture of FITH

4 KMMIO—Interceptor of MMIO access

One of those requirementsof FITH is the ability to hook to a specific memory mapped IO region before the user of the region gets access. A fault injection test case may need to just note when a given region is being read/written, take some action before the caller returns from the read or write operation, or change the value that is being read or written.

4.1 Approaches for capturing MMIO accesses

There are several hardware/software approaches for capaturing MMIO accesses.

- Overriding MMIO functions.

  Memory mapped IO access can be captured by overriding MMIO functions, such as `readb()` and `writeb()`. The major advantage is this method is platform-independent because all Linux platforms support these MMIO functions. Disadvantages are

  1. Any driver that accesses MMIO without using the standard MMIO functions cannot be intercepted.
  2. A special FITH header file needs to be added to the driver code, and the driver needs to be recompiled.
  3. There are some differences between the “released driver” and the “driver with FITH.” The driver that is validated and verified is the driver with FITH rather than the released driver.

- Setting Watch Points.

  IA-32 architecture provides extensive debugging facilities for debugging code and monitoring code execution. These facilities can also be used to intercept memory access. The major advantages are

  1. The driver does not need to be recompiled.
  2. There have been a good patch to support it.[3]

  On the other hand, there are several disadvantages:

  1. In IA32 architecture, this is a trap type of exception, which means that the processor generates the exception after the IO instruction has been executed, so this method cannot do fault injection in write operation.
  2. There are only four watch points that can be used. This may be not enough in a complex environment.

- Trapping MMIO access by using Page-Fault Exceptions.

  Like normal memory, MMIO is handled by a page-protection mechanism. Therefore, MMIO access can be intercepted by capturing page faults. The method clears the PE (PRESENT) bit of the PTE (Page Table Entry) of the MMIO address so that

the processor triggers a page-fault exception when MMIO is accessed. The major advantages are

1. In IA32 architecture, this is a fault type of exception, which means that the processor generates an exception before MMIO access is executed, so this method can do fault injection in write operation.
2. The driver does not need to be recompiled. There is a disadvantage in the method—because the unit of intercepted MMIO is the size of a page (4k in IA-32), an adjacent MMIO access may unnecessarily trigger an exception, system performance might be impacted.

Based on FITH requirements and our analysis above, we implemented a page-fault method to capture MMIO.

```
diff −Nru a/arch/i386/mm/fault.c
b/arch/i386/mm/fault.c
−−− a/arch/i386/mm/fault.c Thu May 15
15:52:08 2003
+++ b/arch/i386/mm/fault.c Thu May 15 15:52:08
2003
@@ −80,6 +81,9 @@
  /* get the address */
  __asm__("movl %%cr2,%0":"=r"
          (address));

+ if (is_kmmio_active() && kmmio_handler(regs,
address))
+     return;
+
  /* It's safe to allow irq's after cr2 has been saved */
  if (regs−>eflags & X86_EFLAGS_IF)
      local_irq_enable();
```

Figure 2: Patch against fault.c

4.2 Implementation

We also followed the same usage style like what kprobes provides, with a `register_kmmio_probe()` function for adding the probe, and a `unregister_kmmio_probe()` function for removing the probe. The `register_kmmio_probe` adds the probe into internal list and set the page which the probe is on as UNPRESENT. After `register_kmmio_probe`, any access on the page will trigger a page-fault exception and fall into KMMIO core.

To get the control in page-fault exception, we need some tweaks to `faults.c`. KMMIO needs to add an additional path here. When the page-fault falls into KMMIO, KMMIO looks up the fault address in probe hash list. If the fault address is one of probes, the `pre_handler` of the probe is called.

Then, KMMIO tries to recover normal execution. It sets the page as PRESENT. But KMMIO needs to do more than this. Because the probe should be re-enabled after current instruction which trigger the page-fault exception, KMMIO enables single-step before exiting page-fault exception. Similar to the change to `faults.c`, KMMIO needs to patch `traps.c` to get the control when the single-step exception is triggered.

After the instruction, which triggers the page-fault execption, is executed, a single-step execption is triggered and falls into KMMIO again. If there is a probe on the fault address, KMMIO calls the `post_handler` of the probe. Then KMMIO sets the page as UNPRESENT again to enable the probe on the page.

```
diff −Nru a/arch/i386/kernel/traps.c
b/arch/i386/kernel/traps.c
−−− a/arch/i386/kernel/traps.c Thu May 15
15:52:08 2003
+++ b/arch/i386/kernel/traps.c Thu May 15
15:52:08 2003
@@ −524,6 +525,9 @@

   __asm__ __volatile__("movl %%db6,%0" : "=r"
                        (condition));

+  if (post_kmmio_handler(condition, regs))
+      return;
+
   /* It's safe to allow irq's after DR6 has been saved */
   if (regs−>eflags & X86_EFLAGS_IF)
       local_irq_enable();
```

Figure 3: Patch against traps.c

5 KIRQ—Interceptor of IRQ handler

Placing hooks into IRQ handler of devices is a straight task. KIRQ stores IRQ handler of the device into `struct kirq` and modifies the IRQ chain in kernel to replace IRQ handler of the device with KIRQ's handler. When the device's interrupt falls into KIRQ, it calls `handler` of the hook. Based on return value of `handler` of hook, KIRQ calls the original handler of the device in turn.

6 Example

The serial device driver in Linux kernel was used in our fault injection trials. Four steps were involved in developing the fault injection tests:

1. Identify resources that needs to be fault injected.
2. Prepare the faultset data source.
3. Set up the test environment.
4. Run the workload and analyze the results.

6.1 Preparing the Faultset Data Source

FITH supports faultset scripts and action code segments. They can be used for customizing For example, a transmission error fault would modify data when the driver received the hardware status from the register. The faultset description looks like the following:

```
<?xml version="1.0" encoding="UTF-8" ?>
<fsml
 xmlns=
 "http://fault-injection.sourceforge.net/FSML/">
  <trigger id="2"
         type="r"
         len="1"
         addr="0x3FD"
         bitmask="0"
         min="0"
         max="0"
         skip="0"
         protection_mask="0"
         hz="0">
        <action code−segment="cs_001" />
  </trigger>
</fsml>
```

Figure 4: Faultset description example

The corresponding code segment looked like the following:

6.2 Setting Up the Test Environment

There are three steps:

1. load FITH kernel modules.

```
#include <fith/state_machine.h>

unsigned long pointer=0;
int inject_faults(struct
  context *cur) {
  //translate the bus address into linear address
  line_addr = fith_bus2line(pointer);

  //inject errors in data by going though
  //the device special
  //structure data
  // ...

  return 0;
};


/* cs_001 is called by trigger "001" in FSML
 * script when IO port 0x3FD
 * (Command Register) is written. */
int cs_001(struct state_machine *sm,
  struct context *cur) {
  unsigned long line_addr;
  if (cur->data==,a,) {
    // check if it is 'a' character
    inject_faults(cur);
  }
  return 0;
};
```

Figure 5: Code segment example

2. set up the faultsets by Fault Injection Command-Line (ficl) configuration tool.

3. load the serial driver.

6.3 Running the Workload and Analyzing the Results

To validate the serial driver, a well-chosen workload was run to stress the driver when it accessed the device. FITH injected faults between the driver and device and logged these operations. The fault-induced results and injected faults were later analyzed.

7 System Performance Impact

In this section we assess the performance impact of current implementation of FITH.

7.1 LMbench

LMbench is a general OS benchmark designed to measure all sides of OS from application view. This is useful for generating a set of apples to apples systems comparisons between pure Linux kernel and FITH-enabled Linux kernel.

All experiments were performed on a dual Pentium-III 933, 512K L2 Cache, 512 MB RAM system. The “52-pure” data was obtained by running on a vanilla 2.5.52 linux kernel. The “cs@1901” data was obtained by running on a patched 2.5.52 Linux kernel (which contained KMMIO KIRQ etc. patches). There are no active probes in “cs@1901” experiment.

Because FITH patches Linux kernel in page-fault exception path, the potential impacts should be in memory management subsystem. Current FITH implementation, however, has minimized the impact when there are no active probes. Differences between two experiments are so small that they are buried by testing noise.

8 Acknowledgements

We specially thanks following persons for their kind help and feedback:

David Edwards (for initial prototype and design), Frank Wang and Elton Yang (for project plan and support), Fleming Feng (for feedback

LMBENCH 2.0 SUMMARY

Basic system parameters			
Host	OS	Description	Mhz
52-pure	Linux 2.5.52	i686-pc-linux-gnu	932
cs@1901	Linux 2.5.52	i686-pc-linux-gnu	932

Processor, Processes - times in microseconds - smaller is better												
Host	OS	Mhz	null call	null I/O	stat	open clos	selct TCP	sig inst	sig hndl	fork proc	exec proc	sh proc
52-pure	Linux 2.5.52	932	0.39	0.68	22.9	24.4	27.2	1.09	4.47	210.	996.	5050
cs@1901	Linux 2.5.52	932	0.37	0.68	22.7	24.0	31.5	1.06	4.51	221.	1052.	5202

Local Communication latencies in microseconds - smaller is better									
Host	OS	2p/0K ctxsw	Pipe	AF UNIX	UDP	RPC/ UDP	TCP	RPC/ TCP	TCP conn
52-pure	Linux 2.5.52		7.109	33.4	41.7	61.6	81.1	110.2	110.
cs@1901	Linux 2.5.52		7.137	15.2	41.8	61.1	81.1	109.8	134.

File & VM system latencies in microseconds - smaller is better								
Host	OS	0K File		10K File		Mmap	Prot	Page
		Create	Delete	Create	Delete	Latency	Fault	Fault
52-pure	Linux 2.5.52	92.5	46.0	225.6	75.3	2053.0	0.885	2.00000
cs@1901	Linux 2.5.52	93.4	46.5	229.2	77.1	2063.0	0.765	2.00000

Local Communication bandwidths in MB/s - bigger is better										
Host	OS	Pipe	AF UNIX	TCP	File reread	Mmap reread	Bcopy (libc)	Bcopy (hand)	Mem read	Mem write
52-pure	Linux 2.5.52			40.7						
cs@1901	Linux 2.5.52			41.0						

Figure 6: LM Benchmark

and requirement), Rusty Lynch (for sysfs interface in FITH).

`http://www-124.ibm.com/linux/projects/kprobes/`

References

[1] David A. Edwards, “An Approach to Injecting Faults into Hardened Software,” Proceedings of the Ottawa Linux Symposium, `http://wwww.linuxsymposium.org/2002`

[2] Andy Chou, Junfeng Yang, Benjamin Chelf etc., “An Empirical Study of Operating System Errors,” 2001, `http://www.stanford.edu/~engler/metrics-sosp-01.ps`

[3] Vamsi Krishna, Rusty Russell etc., “Kernel Probes for Linux,”