Proceedings of the Linux Symposium

Volume Two

July 21st–24th, 2004
Ottawa, Ontario
Canada

Contents

kobjects and krefs: lockless reference counting for kernel structures **295**

Greg Kroah-Hartman

The Cursor Wiggles Faster: Measuring Scheduler Performance **301**

Rick Lindsley

On a Kernel Events Layer and User-space Message Bus System **311**

Robert Love

Linux-tiny and directions for small systems **317**

Matt Mackall

Xen and the Art of Open Source Virtualization **329**

Dan Magenheimer

TIPC: Providing Communication for Linux Clusters **347**

Jon Paul Maloy

Object-based reverse mapping **357**

Dave McCracken

The World of OpenOffice **361**

Michael Meeks

TCPfying the Poor Cousins **367**

Arnaldo Carvalho de Melo

IPv6 IPsec and Mobile IPv6 implementation of Linux **371**

Kazunori Miyazawa

Getting X Off the hardware **381**

Keith Packard

Linux 2.6 performance improvement through readahead optimization **391**

Ram Pai

I would hate user space locking if it weren't that sexy… **403**

Inaky Perez-Gonzalez

Workload Dependent Performance Evaluation of the 2.6 I/O Schedulers **425**

Steven L. Pratt

Creating Cross-Compile Friendly Software **449**

Sam Robb

Page-Flip Technology for use within the Linux Networking Stack **461**

John A. Ronciak

Linux Kernel Hotplug CPU Support **467**

Rusty Russell

Issues with Selected Scalability Features of the 2.6 Kernel **481**

Dipankar Sarma

Achieving CAPP/EAL3+ Security Certification for Linux **495**

Kittur (Doc) S. Shankar

Improving Linux resource control using CKRM **511**

Rik van Riel

Linux on a Digital Camera **525**

Alain Volmat

ct_sync: state replication of ip_conntrack **537**

Harald Marc Welte

Increasing the Appeal of Open Source Projects **547**

Mats Wichmann

Repository-based System Management Using Conary **557**

Matthew S. Wilson

On-demand Linux for Power-aware Embedded Sensors **573**

Carl D. Worth

Virtually Linux **583**

Chris Wright

Conference Organizers

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

Review Committee

Jes Sorensen, *Wild Open Source, Inc.*
Matt Domsch, *Dell*
Gerrit Huizenga, *IBM*
Matthew Wilcox, *Hewlett-Packard*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

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

kobjects and krefs

lockless reference counting for kernel structures

Greg Kroah-Hartman *
Linux Technology Center
IBM Corp.
greg@kroah.com
gregkh@us.ibm.com

Abstract

This paper will describe the current kobject and kref kernel structures in detail. It will cover why they were created, how to use them, and how the internals work. It will also cover a few directions that these structures might be taking in the future.

1 Introduction

The Linux kernel file `Documentation/CodingStyle` has the following statement about reference counting:

> Data structures that have visibility outside the single-threaded environment they are created and destroyed in should always have reference counts. In the kernel, garbage collection doesn't exist (and outside the kernel garbage collection is slow and inefficient), which means that you absolutely _have_ to reference count all your uses.

This requirement of providing proper reference counting for kernel structures has caused developers to create their own logic and functions to implement this feature. During the development of the Linux Kernel Driver model[4], a simple structure, `struct kobject`, was created that provided automatic reference counting for any user of the object. Unfortunately, `struct kobject` is closely tied to the kernel driver model, and for any data structure that does not want to show up in sysfs, and participate in the global kernel "web woven by a spider on drugs"[2], using a struct kobject only for reference counting is a big waste of memory resources and is much more complex than needed. To this end, the data structure, `struct kref`, was created to provide a simple, and hopefully failproof method of adding proper reference counting to any kernel data structure.

2 How to use it

To use the `struct kref` structure, simply embed it within the structure that reference counting is needed for. For example, to add reference counting to a structure called `struct foo` then it would be defined as:

```
struct foo {
    ...
    struct kref kref;
    ...
```

*This work represents the view of the author and does not necessarily represent the view of IBM.

```
};
```

It is not important that the `struct kref` structure be the first or last element of the structure that it is embedded in. The only requirement is that the whole `struct kref` structure be in the structure being reference counted, not a pointer to the a `struct kref` structure.

When the `struct foo` structure is initialized, the `kref` variable must also be initialized before reference counting can be used. This is done with a call to the `kref_init` function:

```
struct foo *foo;
foo = kmalloc(sizeof(*foo),
              GFP_KERNEL);
kref_init(&foo->kref,
          foo_release);
```

The parameter `foo_release` is a pointer The first parameter of `kref_init` is a pointer to the `struct kref` structure that is to be initialized. The second parameter is a pointer to the release function for the structure. This release function is described in detail below.

After the kref structure has been initialized, the internal reference count of the structure is set to 1. Now the reference count can be incremented and decremented at will.

To increment the reference count of a kref structure, the function `kref_get` is called:

```
/* get a new reference to our
   foo structure */
kref_get(&foo->kref);
```

When a user of the structure is finished with it, the `kref_put` function should be called to release the reference:

```
/* finished with this
   foo structure */
kref_put(&foo->kref);
```

This function should also be called after the original creator of the structure that the kref variable is in, is finished with the structure. The `kfree` function must *NOT* be directly called because other portions of the kernel could have valid references to this structure.

After the `kref_put` function is called, the structure can not be referred to by any future code, as the memory for that structure could be now gone.

When the last reference count is released, the function that was passed to the original `kref_init` function is called to release the memory used by the structure. The prototype of this function must accept a pointer to a `struct kref`:

```
void foo_release(struct kref
                 *kref)
{
    struct foo *foo;

    foo = container_of(foo,
                       struct foo,
                       kref);
    kfree(foo);
}
```

As the above example function shows, to get back to the original `struct foo` structure location, the `container_of` macro is used. For a complete description of how the `container_of` macro works, please see[1].

As there are not any locks within the `kref` structure, there are three rules that need to be followed when using this reference counting logic:

- If the code accessing the variable already has a valid reference to the structure, it is

safe, and required to increment that reference with a call to `kref_get` in order to give the variable to any other piece of code.

- If the code accessing the variable already has a valid reference to the structure, then it is safe to release that reference with a call to `kref_put`.
- If the code wanting to access the variable, does not have a valid reference, then it needs to serialize with a place within the code where the last call to `kref_put` put could happen.

This last rule can not be emphasized enough. The only reason that the `struct kref` can work without any internal locks is because a call to `kref_get` can not happen at the same time that `kref_put` is happening. In order to ensure this, a simple lock for the driver or subsystem that owns the specific `struct kref` reference can be used.

An example of using such a lock can be seen in Figure 1.

So, with the three simple functions, `kref_init`, `kref_get`, and `kref_put`, combined with a release function that the caller provides, complete reference counting can be added to any kernel structure.

3 How it works

`struct kref` is a very tiny structure with only two elements:

```
struct kref {
    atomic_t refcount;
    void (*release)(struct kref *kref);
};
```

The `refcount` variable is an atomic counter that is used to hold the reference count of the structure. The `release` variable is a pointer to a function that will be called when the last user of the structure is finished with the structure.

The `kref_init` function is a mere three lines long:

```
void kref_init(struct kref *kref,
               void (*release)
               (struct kref *kref))
{
    WARN_ON(release == NULL);
    atomic_set(&kref->refcount,1);
    kref->release = release;
}
```

First a warning is printed out to the syslog if a `release` callback is not provided, as this is not allowed. Then the `refcount` variable is initialized to 1 as the structure needs to have a single initial reference count. After that the `release` function pointer is stored in the `release` variable in the structure.

The `kref_get` function is also only three lines of code:

```
struct kref *kref_get(struct kref *kref)
{
    WARN_ON(!atomic_read(&kref->refcount));
    atomic_inc(&kref->refcount);
    return kref;
}
```

Again, a warning is printed out to the syslog if the `refcount` variable is zero. This catches the very common error of calling `kref_get` without first calling `kref_init`. After that, the `refcount` variable is incremented, and then a pointer to the same structure is returned. This return type makes it easier for code to do things pass the result of `kref_get` as a function parameter:

```
do_foo(kref_get(my_kref));
```

Keeping with the tradition of tiny functions, the `kref_put` function weighs in at a whopping two lines:

```
/* prevent races between open() and disconnect() */
static DECLARE_MUTEX (disconnect_sem);

static int skel_open(struct inode *inode, struct file *file)
{
    struct usb_skel *dev;
    struct usb_interface *interface;

    /* prevent disconnects */
    down (&disconnect_sem);

    interface = usb_find_interface(&skel_driver, iminor(inode));
    dev = usb_get_intfdata(interface);

    /* increment our usage count for the device */
    kref_get(&dev->kref);
    up(&disconnect_sem);

    ...
}

static void skel_disconnect(struct usb_interface *interface)
{
    struct usb_skel *dev;
    int minor = interface->minor;

    /* prevent skel_open() from racing skel_disconnect() */
    down (&disconnect_sem);

    dev = usb_get_intfdata(interface);
    usb_set_intfdata(interface, NULL);

    /* give back our minor */
    usb_deregister_dev(interface, &skel_class);

    /* decrement our usage count */
    kref_put(&dev->kref);

    up(&disconnect_sem);
}
```

Figure 1: Using a lock to ensure safe access to `kref_put`

```
void kref_put(struct kref *kref)            kref->release(kref);
{                                       }
    if (atomic_dec_and_test
              (&kref->refcount))
```

This function decrements the value stored in the `refcount` variable, and if the result is zero, this was the last reference to the structure, so the function stored in the `release` variable is called to clean up the memory used by this structure.

4 kref vs. kobject

This paper has focused on on how `struct kref` works, and ignored `struct kobject`. For the most part, both structures work identically, with the following minor differences:

- `struct kobject` does not contain a `release` function. When a `struct kobject`'s last reference count is decremented, the release function of the `struct kset` that is associated with the `struct kobject` is called. For more details on how `struct kobject` and `struct kset` is related, please see [3].
- A `struct kobject` can be initialized with two different functions, `kobject_register` or `kobject_init`. `kobject_register` calls `kobject_init` and then calls `kobject_add` to add the kobject to the sysfs hierarchy. If a `struct kobject` is to not be used within the sysfs hierarchy, then `kobject_add` should never be called.
- A `struct kobject` can have its reference count incremented with a call to `kobject_get` and decremented with a call to `kobject_put`. But if the kobject was initialized with the sysfs core with a call to either `kobject_add` or `kobject_register`, then it needs to be removed from it with a call to `kobject_del`, which will also call `kobject_put` on the `struct kobject`. After a `struct kobject` has had `kobject_del` called for it, the `kboject_get` function can not be called on the variable without having a previous reference count already on the variable. This is the same as the previously mentioned issue for calling `kref_put` without serializing the access.
- Before using a `struct kobject`, the structure must be initialized to zero by using `memset` before `kobject_init` or `kobject_register` is called. If not, a warning will be printed out to the syslog.

5 Future

In future releases of the Linux kernel, the `struct kobject` will probably loose its internal reference count and use the `struct kref` instead. If this happens, `struct kref` might have to be changed in order to support passing the `release` callback as a parameter to the `kref_put` function, in order to save the storage size of the function pointer from the structure.

Other kernel uses of a `atomic_t` variable will probably be converted to use the `struct kref` interface instead of providing their own logic to handle reference counting.

6 Legal Statement

References

[1] Greg Kroah-Hartman. The Driver Model Core, part 1. `http://www.linuxjournal.com/modules.php?op=modload&name=NS-lj-issues/is%sue110&file=6717s2`, June 2003.

[2] Linux Weekly News - Driver porting: Device model overview. `http://lwn.net/Articles/31185/`.

[3] Linux Weekly News - The zen of kobjects. `http://lwn.net/Articles/51437/`.

[4] Patrick Mochel. The Linux Kernel Device Model. In *Linux.conf.au*, Perth, Australia, January 2003.

The Cursor Wiggles Faster: Measuring Scheduler Performance

Rick Lindsley
IBM Linux Technology Center
ricklind@us.ibm.com

Abstract

Trying to pin down whether changes to the 2.5 and 2.6 scheduler have helped or hurt performance, especially on interactive programs, has been both difficult to quantify and very subjective. One favored test has been to create your favorite load and then move your cursor around and observe how slow or fast it is. Another one is to drag a window across your desktop and see how quickly it gets redrawn. And I would certainly be skewered if I didn't mention what is probably the favorite: playing your favorite music while under load and listening intently for skips.

Unfortunately, all these measurements are subjective, and even, at times, argumentative. With scheduler statistics installed, one can accurately measure such things as the amount of time processes are spending on the processor or the amount of time they are waiting for the processor. This means that on SMP and NUMA machines, load balancing efforts can be objectively evaluated, and process migration decisions more effectively reviewed. And all of this can be done with no measurable impact to the system.

This paper will describe what information can be captured, use that information to characterize some simple loads, and describe how that same information may be coordinated with other system measurements both to characterize new loads, and to more clearly identify scheduler shortcomings.

1 Introduction

As the 2.5 code revisions came out in mid- to late 2003, the scheduler, like much of the 2.5 release, became more and more stable. True, there was still work to be done in some areas, like SMP and NUMA. Although an increasing number of dual-CPU desktops and even laptops introduced more users to the world of SMP, it was the high end users with 16, 32, 128, or even more CPUs that really were stretching the existing SMP and NUMA code. The increasing load on the existing infrastructure was causing developers to realize that code paths they previously thought "impossible" were really "rarely," and paths deemed "infrequent" were unfortunately morphing to "once or twice a day."

And an odd thing happened on the way to better code for the high end machines. Those pesky desktop and laptop users got in the way. With every fix that would demonstrably improve the situation for the big iron, dozens of desktop and laptop owners would immediately pick up the new code, try it out, and more often than not, pronounce it faulty. Why? Because *their* 2-proc SMP machines were used very differently than the file servers and web servers that the 128-proc systems had become.

The testing and measurements that had gone into verifying the patch did not test the system the same way the desktop users did. Consequently, these desktop users saw very different results, and formed very different opinions about the correctness and usefulness of these high-end SMP fixes.

And while their opinions mattered, of course, addressing their concerns was difficult. They were using human eyes and ears—notoriously unreliable biological components known to be fraught with frequent failure and highly subjective readouts—to detect problems with code. These observations needed to be backed up with numbers somehow.

2 Why is the wiggle so important?

So why weren't the big iron folks seeing the same problems as the desktop people if they were both utilizing the same code? The answer lay in usage patterns. People with laptops and desktops did not run two dozen instances of a server daemon that depended on ultra fast cache and great amounts of parallelism. They did not have petabytes of disk, and typically did not have gigabytes of memory either. They didn't read terabytes of disk per minute, nor expect to fully utilize their bus bandwidth on a regular basis.

These folks browsed the web, sorted mail, and compiled kernels while, in the background, they listened to their favorite playlist. While doing this, they would notice that with the new scheduler mods, their windows took longer to redraw. Or their cursor moved more sluggishly under this relatively heavy load. Or their music skipped now and then because their music player didn't get back on the CPU soon enough to catch the next few notes.

That's not to make light of their complaints; they were uncovering real problems that existing testing was inadequate to find. In fact, there were two main problems that needed to be solved. One was to close the testing hole by reliably repeating the tests that the desktop users were running, and repeating them on as wide a variety of hardware as the original patches had been run on. The other was that even the desktop users quibbled among themselves, sometimes, about whether wiggles, skips, and redraws had degraded. It was important to find a way to measure this "wiggle effect" in some quantifiable, objective way so you could reliably tell whether a new patch worsened it or improved it.

Server software, for its part, didn't need music to function, didn't need cursors to point with, and it sure didn't care how fast windows were redrawn. These highly interactive activities had no place in server evaluations. It was typically all about *throughput*, and placing stress on some subsystem or another: disk, memory, or network, typically. Stress on the scheduler was a given. Even though dozens of benchmarks exist for measuring the throughput of high-end machines, producing megabytes or even gigabytes of analysis and data, there was no easy way to automate the type of subjective human observation that desktop users were using. There was no way to have weekly regression tests pick it up, nor any way to precisely duplicate the environment in which these observations were being made. In short, there was no way to quantify the observations being made, so no existing tests could detect regressions in this area.

Previous scheduler modifications had labeled applications that tended to spend a lot of time waiting for I/O as "interactive," and attempted to give scheduler bonuses to those tasks when the I/O they had been waiting for completed. This was *supposed* to provide the exact behavior the desktops were *not* seeing. The suspicion was that either these types of applications were

not being correctly recognized, or they were not being given sufficient bonuses.

3 Isolating the wiggle

The first part of the solution was recognizing that the "wiggle effect" comes from tasks not regaining the CPU fast enough. The second part was recognizing that the audible stutter from a music player, or the delay in redrawing a window, were showing the same problem as wiggling the cursor.

In the case of a cursor, coordinates from a serial mouse are presented as a stream of input to the windowing system. If the task that moves the cursor is not brought to a CPU quickly enough, there will be a lag between the time the movement is initiated and the time it appears on the screen. With all the input consumed, the task again goes to sleep even though a split second later more input appears as the mouse continues to move. While this is an efficient way to handle a serial mouse, it is dependent on hitting the processor quickly enough to guarantee the input stream doesn't back up too much. If the consuming task does not get to run quickly enough, the cursor will appear to move across the screen in a staccato fashion, even though the mouse itself is being moved smoothly.

In the case of a music player, the application (say, *xmms*) will read a certain amount of input from a file, but it will take longer to play it to the speaker. Even though this is, in general, a very I/O-intensive task, there are times when *xmms* will go to sleep either waiting for output to drain to the speaker or input to come from the file. Waking up too slowly from these self-imposed interruptions is what causes the music to pause or stutter.

Slow window redrawing is a case of applications taking too long after notification to wake up and redraw. This *might* also be attributed to slow interprocess communication or slow signal delivery, but it should be easy to rule out these causes if we were to measure the time a task spent in a queue waiting for a processor.

A patch for scheduler statistics has been available since 2.5.59[1]. However, it was with the 2.6.0-test5 release in September of 2003 that it was updated to include code to measure task latency. The task is given a new timestamp when it is placed in a run queue, placed on a processor, or removed from a processor. This makes it trivial to determine how long the task spent in the run queue before making it to the processor. It has the side effect of allowing us to also measure, on average, how long a task remains on the processor before relinquishing it, usually voluntarily. This allows us to easily characterize the kind of load a benchmark may place on a system.

Adding statistics counting to the scheduler path was a dicey task. This is one of the most heavily used paths in the system, and anything that slows down this path can have a catastrophic effect on the system as a whole. Consequently, the statistics patch tries to do what it can to gather accurate statistics without the use of a lock.

- Per-CPU counters are used, and incremented only by their respective CPU. This makes update collisions (and loss of data) impossible.
- Even so, when possible, these counters are incremented while a per-CPU runqueue lock is already acquired.
- Counters are only incremented, so minor variations from unflushed caches that may be observed while reading another CPU's counters can be safely ignored. (The

[1] http://oss.software.ibm.com/developerworks/opensource/linux/patches/?patch_id=730

counters are declared unsigned long, so user-level utilities on 32-bit architectures must take note that the counters could wrap. While theoretically possible on 64-bit machines, wrapping is far less likely than on 32-bit machines.)

Measurements were taken across several different releases using several different benchmarks to see if any statistical impact could be found on the benchmarks when scheduler statistics were utilized. To date, none have been found.

After the patch is applied, the counters can be obtained by reading `/proc/schedstat`. A full description of the statistics collected can be found in `Documentation/schedstats.txt` in the kernel source. The patch itself introduces a config option SCHEDSTATS that is on by default; if it is turned off, all the additional code is compiled out. There are three important fields:

timestamp *N*
This line indicates a timestamp, in jiffies, of when this output was produced. The statistics are most effectively utilized when collected at small regular intervals, since this allows you to more accurately see how the behavior of a load or benchmark may change over its lifetime. Any process reading this file, however, is subject to the same scheduler delays it is trying to measure. Consequently, a simple script like

```
while true
do
    sleep 10
    cat /proc/schedstats >> \
        /tmp/stat.out
done
```

may find it collects statistics roughly every 10 seconds when the system is lightly loaded, but every 15-20 seconds or more when the system is heavily loaded. The code to note the timestamp is just a few lines before the data is totaled in the kernel, and on a non-preemptible kernel is an inexpensive way of identifying the time at which the snapshot was *actually* taken.

cpu*N n n n n n n n n ...*
These are the values of the counters for cpu ***N***. The precise meaning of these counters will vary depending on the version of scheduler statistics being utilized. A few examples of data collected are:

1. number of times some functions were called
2. number of times certain functions were called under certain circumstances (i.e., were the runqueues unbalanced? was this processor idle?)
3. total number of milliseconds that tasks on this processor have used, not including the current one
4. total number of milliseconds that tasks that ran here had to wait in queue

version *N*
identifies the version of output being produced. Since the meaning of fields (and the number of fields) in the **cpu*N*** line, above, can vary in different versions of scheduler statistics, this allows tools to be as flexible or inflexible as desired when processing input.

A sample of the output from `/proc/schedstat` is provided in Appendix A.

4 What would I use statistics for?

Scheduler statistics can serve three basic purposes. In many cases, they are doing no more than providing some detailed code path and profiling data. Knowing, for instance, that a particular function was called 50,000 times during a benchmark run may be key if it is expected to be called a dozen times—or a million. Similarly, knowing that 22,000 of those calls were made while the processor was idle, or made on just one of eight CPUs, may also be quite informative. About half the counters provide this sort of information, and it must be coupled with a knowledge of what to expect given your workload in order to detect anomalies.

Another purpose is to provide information beyond just counting. There is a counter that sums the imbalance found when queues are inspected. Combine this with the number of times you called this function and you can determine the average imbalance between runqueues. In most cases you wouldn't want this to exceed 1. Truth is, though, that a flurry of forking or even I/O completions might suddenly cause a processor to suddenly find itself with significantly more runnable tasks than other processors. Seeing where these spikes happen during the test run, and how often they happen, may help to suggest better "default" behavior in the scheduler or even tuning in the benchmark itself.

The last purpose has already been mentioned—task latency. We already need to note when a task is queued on a processor and when it acquires a processor. By noting one more thing—when it leaves the processor—we can also determine what I call the *runslice*.

The runslice is the amount of time a task spends *on* the processor before yielding it. In contrast, the *timeslice* alloted by the scheduler indicates how long the task may run before it is *forced* off. Processes are usually given generous timeslices (100 ms is the default) but typically don't use all of them at one shot. A task may need to put itself to sleep, perhaps to wait for input, before it has used up that full 100 ms. It will have any unused amount available to it when the event awakens it, but how long it spends on the processor can be an important characteristic of the system load. If a task spends only a few milliseconds before giving up the processor, it may be I/O-bound. By the same token, if it uses its full timeslice every time before being kicked off, then it is CPU-bound.

While many benchmarks are already characterized as CPU- or I/O-bound, they are rarely that way from beginning to end. Seeing this behavior graphed over a period of time can be very informative to a person trying to tune the system or the benchmark.

5 Diagnostic examples

The data that the scheduler statistics collect can be utilized in several different ways.

5.1 Using the function counts to characterize behavior

Recently a colleague remarked that he was running a benchmark that he expected to fully load a machine; yet profiling was reporting that the system was in the idle routine 50% of the time. He increased the load significantly on the machine and idle time only dropped to 49%. He couldn't believe the machine still had spare cycles, so we used the scheduler statistics to determine what was happening.

From the beginning of the benchmark, we captured the counters in `/proc/schedstat` every 10 seconds with a shell script. When the benchmark exited, we killed the shell script.

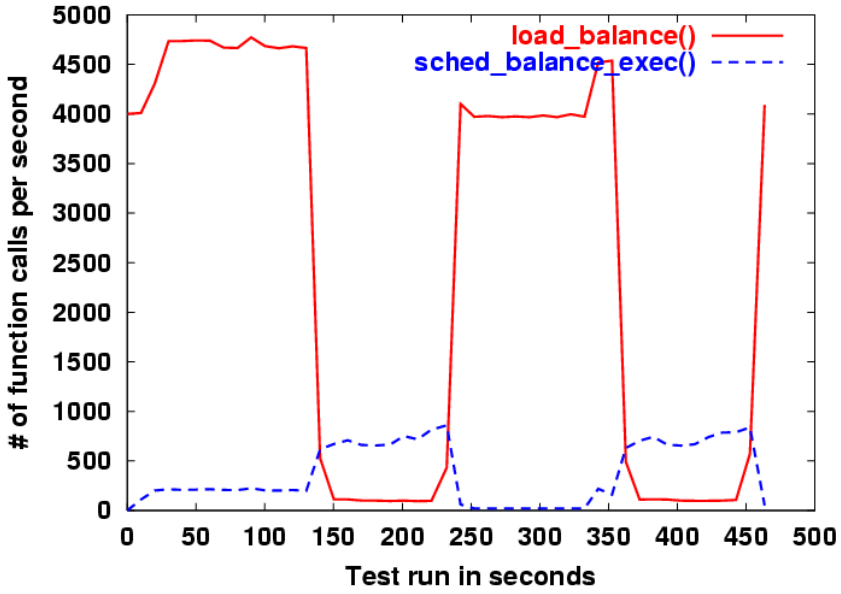


Figure 1: `load_balance()` and `sched_balance_exec()` counts

The two pieces of information that proved most useful were the number of calls per second (*cps*) for `load_balance()` and `sched_balance_exec()`. In Figure 1, you can see that the *cps* for `load_balance()` varies markedly between plateaus of around 4000-4500, and valleys of 100-200. When the system is idle, it calls `load_balance()` as often as once a millisecond to try to find work. When it is busy, it backs off to five times a second. The graph here is clearly indicating that this benchmark has at least two periods of about 100 seconds each out of about 450 seconds total where it is largely idle.

At about the same time that the *cps* for `load_balance()` is high, the *cps* for `sched_balance_exec()` is low. This function is called when tasks issue the `exec()` system call, and is used to do some opportunistic rebalancing. We observed that just as the system starts to get busy, `sched_balance_exec()` tails off.

The data suggested that this benchmark had a notable rampup and cooldown period. With this information in hand, simple observation of *top(1)* while running the benchmark confirmed what the scheduler statistics suggested. The benchmark had a fairly lengthy single-threaded setup: creating log files, making directories for results, and compiling short programs it would use. It then forked many tasks and set them all running to actually start the benchmark. When the test was over, there was again a single threaded task that collected the data created before several tasks organized the data.

5.2 Using latency and runslice information

In another situation, a disk-intensive benchmark was doing much worse with a different version of the scheduler. Figure 2 shows a measurement of the latency from the two runs.

In the “broken” run, the latencies were nearly twice that of the “working” run. Tasks were taking longer to reach the CPU in the broken case. Yet the runslice information shows comparable (and very short) times spent on the CPUs. If tasks were running very short periods of time, but waiting longer to run, what could

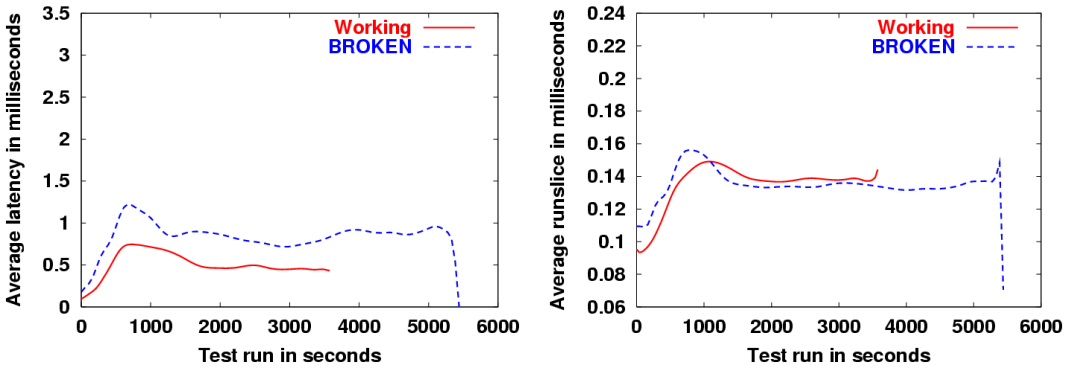


Figure 2: Latency and runslice duration

have been the cause?

Enlightenment was finally attained by viewing the average imbalance (Figure 3) during each of the runs. On the average, the imbalance was twice as great in the broken run as in the working run. Since the runslice was so small, this suggested that tasks were becoming runnable quickly but simply not being balanced often enough. Some queues were getting quite long while others (presumably) were staying short. Additional debugging showed that tasks were indeed awakening (probably by completed I/O) quite frequently but most of the balancing was happening only when one CPU fell idle and went looking for work. These longer queues in the broken run were persisting longer than those in the working run, and tasks stuck in them were waiting a fraction of a millisecond longer than before.

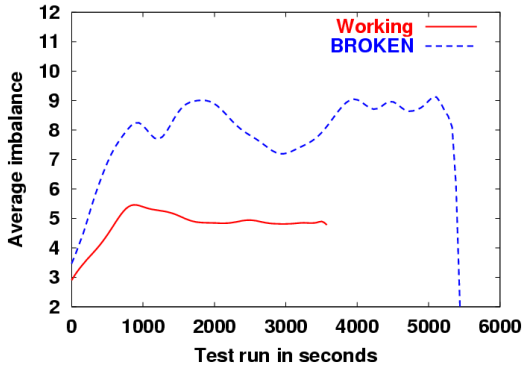


Figure 3: Average load imbalances

6 Conclusion

There is still work to do.

Recent scheduler changes present in Andrew Morton's -mm tree will dramatically change what is important to measure in the scheduler. Additionally, these same changes introduce some self-tuning characteristics which may benefit from statistics describing how often they are retuned.

There is also some evidence that NUMA machines may benefit from device, task, or memory affinitization strategies which try to keep data from crossing NUMA node boundaries. Scheduler statistics can be used to reliably demonstrate whether these strategies are being effective.

Lastly, the data provided by scheduler statistics probably ought to be moved out of /proc eventually, as there is an ongoing effort to return /proc to its original task of just listing processes.

Scheduler statistics provide a quantifiable means of measuring scheduler changes. Much as disk statistics can be used to a variety

of ends—measuring disk utilization, throughput rates, and transfer rates, for example—scheduler statistics can help with analysis of a variety of situations. The latest revisions go to lengths to avoid creating "Heisenbugs," or bugs which disappear when you try to examine them closely. Perhaps best of all, developers need not rely on mice and windowing systems to measure their test results. Latency numbers, in particular, provide a key way of measuring scheduler success, and runslice figures can help characterize the load that tests create so that the best set of tests can be chosen to test a particular feature or system. Cursor wiggles and audible skips can be set aside until they are needed again.

Disclaimer

Appendix A

Table 1 is a sample of what `/proc/schedstat` might look like for a 2-proc machine. The actual format and number of counters will vary between different versions. For purposes of this example, the last three lines are artificially folded for readability, but in actual output, each would be one long line.

This is a brief description of each of the 23 counters for version 4 output. Applications can check the `version` field to make sure they look for and correctly interpret the counters. Note that all counters may wrap back to zero, and applications using these counters should be prepared to deal with that. Since all counters start at zero at boot time, the most useful way to use them is to get periodic snapshots of the counters, then subtract one set from a previously obtained one to obtain the delta. All counters are per-processor.

1. in `sched_yield()`, number of times both the active and the expired queue were empty
2. in `sched_yield()`, number of times just the active queue was empty
3. in `sched_yield()`, number of times just the expired queue was empty
4. in `sched_yield()`, number of times `sched_yield()` was called
5. in `schedule()`, number of times the active queue had at least one other task on it
6. in `schedule()`, number of times we switched to the expired queue and reused it
7. number of times `schedule()` was called
8. number of times `load_balance()` was called at an idle tick
9. number of times `load_balance()` was called at a busy tick
10. number of times `load_balance()` was called from `schedule()`
11. number of times `load_balance()` was called
12. sum of imbalances discovered (if any) with each call to `load_balance()`
13. number of times `load_balance()` was called when we did not find a "busiest" queue

```
version 4
timestamp 4295814751
cpu0 8909 9103 612 11869 264585 9821 392921 1065335 406 140662 1206403 62905
1192940 0 13440 13469 0 0 0 0 82278 1497607 264615
cpu1 5138 5328 577 8126 265205 6270 402877 943453 1005 149999 1094457 77670
1074828 0 13469 13440 0 0 0 0 200998 448842 265175
totals 14047 14431 1189 19995 529790 16091 795798 2008788 1411 290661 2300860
140575 2267768 0 26909 26909 0 0 0 0
```

Table 1: Sample output from `/proc/schedstat`

14. number of times `load_balance()` was called from `balance_node()`
15. number of times `pull_task()` moved a task to this cpu
16. number of times `pull_task()` stole a task from this cpu
17. number of times `pull_task()` moved a task to this cpu from another node (requires `CONFIG_NUMA`)
18. number of times `pull_task()` stole a task from this cpu for another node (requires `CONFIG_NUMA`)
19. number of times `balance_node()` was called
20. number of times `balance_node()` was called at an idle tick
21. sum of all time spent running by tasks (in ms)
22. sum of all time spent waiting by tasks (in ms)
23. number of tasks (not necessarily unique) given to the processor

The last three make it possible to find the average latency on a particular runqueue or, if taken from the `totals` fields, the overall system. Given two points in time, A and B, $(22B - 22A)/(23B - 23A)$ will give you the average time tasks had to wait after being scheduled to run but before actually running.

`/proc/<pid>/stat`

This version of the patch also changes the `stat` output of *individual tasks* to include the same latency and runslice information described above. Three new fields, corresponding to the last three fields described above, are added to the end of the per-task `stat` file, but apply only for that task rather than a whole processor.

On a Kernel Events Layer and User-space Message Bus System

Robert Love
Novell
rml@ximian.com

Abstract

Various Linux usage scenarios, particularly the widely accepted server and the rapidly growing desktop, require a lightweight, simple, asynchronous mechanism for kernel to user-space communication. Such a mechanism is crucial for the transmissions of events to user-space in a type-safe and clean manner. Further, a system-level messaging bus, which can deliver messages up the system stack on both a system-wide and per-user level, is required to further the integration of the Linux system.

This talk will discuss the design and implementation for two specific solutions, the Kernel Events Layer and D-BUS, to these two problems. Finally, useful solutions built on the sum of these technologies will be discussed—such as a fully integrated Linux desktop, from the kernel up through the GNOME desktop.

1 Introduction

Usually considered a plus of open source development, the Linux system is developed piece-meal, resulting in cleanly separated layers and properly defined interfaces. This separation, however, also results in a lack of integration among the various components comprising the system stack. In particular, the lack of integration is readily manifest between the lower levels of the stack—kernel and system-level components—and the upper levels of the system, such as the desktop environment on desktop machines.

A particularly important, but missing, component of the Linux system is an ubiquitous IPC mechanism and events system. Such a component would facilitate the dissemination of information up the system stack, better integrating the Linux system from the kernel up through the system layers, the desktop, and the end user applications and daemons. With well defined interfaces, such integration could occur while continuing the current separation and interoperability of Linux components.

What would such an IPC mechanism and event system allow? Quite a bit. Photo applications could start automatically in response to camera insertion. The volume of your music player could automatically lower in response to your phone ringing. System shutdown, reboot, and suspend messages could be trasmitted up the stack. HA applications could receive instant notifications from the kernel. No longer need components in the system live separate lives from the kernel, the layers below them, and themselves. Now, applications can communicate, listen, and evolve.

Such a system may be broken into three requirements:

- Kernel support implementing a kernel-to-user event mechanism

- A user-space message transport and IPC mechanism
- Applications sending and receiving such messages

This paper will discuss two specific implementations of these requirements:

- The Kernel Events Layer
- D-BUS

2 The Kernel Events Layer

2.1 Goals and Design

Current user-space grokking of the kernel typically requires some combination of periodic polling, parsing of unformatted text files from `/proc`, and luck. The Linux kernel currently lacks a mechanism for kernel to user-space communication.

The requirements for such a system include:

- simple and clean
- low overhead and scalable
- asynchronous transport accessible without polling
- type-safe
- generic enough for use in multiple usage scenarios
- support for formalized sender interfaces, allowing standardized messaging

Event systems have been proposed and even implemented, but they generally receive minimal community buyin, presumably due to a lack of one or moe of these requirements (more than likely, the “simple” bit).

2.2 Implementation

The Kernel Events Layer implements an event system satisfying these requirements.

Usage is simple:

```
send_event(int type, char
*interface, char *fmt, ...)
```

The `type` parameter specifies a constant value representing the type of message being sent. The `interface` value specifies the originator of the message. It is used to provide an interface object for object-based component and IPC systems such as CORBA and D-BUS. Finally, `fmt` and any following arguments provide the usual `va_list` of format and arguments.

Example:

```
send_event (DBUS_NORMAL,
"org.kernel.arch.cpu",
"overheating")
```

This specifies a message from the `org.kernel.arch.cpu` interface with a value of `overheating`.

The actual implementation of the Kernel Events Layer uses netlink. In fact, the Kernel Event Layer is simply specific netlink socket into user-space in which the event is formated and then reconstructed by user-space. Netlink is fast, simple, and already in the kernel. Thus it was a natural choice.

The Kernel Events Layer code uses `netlink_broadcast()` internally.

2.3 Real World Usage

The Kernel Events Layer is independent of any specific user-space transport mecha-

nism. The assumed use case is to create a new daemon (or modify an existing daemon, like the D-BUS system message bus, `dbus-system-1`). This daemon listens on the netlink socket, reading each event as it occurs. The events are parsed and reconstructed into the format native to the user-space transport mechanism.

In the case of D-BUS, the `dbus-system-1` daemon sends the kernel events out the system message bus. Components up the system stack may then receive the kernel events right off the D-BUS system bus, along with other system-wide messages.

3 D-BUS

D-BUS is a user-space IPC system.

D-BUS varies from other IPC mechanisms in that it provides a bus system (as opposed to point-to-point) over which messages (as opposed to byte streams) are transported. Messages include a header containing metadata about the message itself and a body containing the data. The bus system is created by forming a point-to-point connection between the D-BUS daemon and each listener. The daemon acts as the hub and the listeners as the spokes of a wheel.

D-BUS provides both a system-wide and a per-user session bus. The system-wide bus is used to dissemenate information on a machine-global scale. A single system daemon provides this service, allowing applications up the stack to receive messages from components down the stack. A security system implements access control.

The per-user session bus exists on a per-user basis, with one daemon created for each user session. The per-user daemon is used for general application IPC and is physically separate from the system-wide bus. The per-user daemon is generally used for traditional point-to-point IPC.

D-BUS is the name given to this system. It is composed of several architectural layers:

- The message bus daemon
- The D-BUS library, `libdbus`, which connects to applications together
- Wrapper libraries and bindings that wrap `libdbus` for direct use on various application frameworks, such as Glib or QT, and various languages, such as C# and Python. The wrapper libraries and bindings provide the API that most programmers should use as they both simplify the rather low-level `libdbus` API and provide an API more familiar and fit for that particular environment.

3.1 D-BUS Concepts

D-BUS introduces various concepts that comprise the IPC system.

- The **bus** is either the system-wide global bus or the per-user session bus.
- **Objects** represent an instance of a specific listener of a D-BUS message. Objects are contained within the applications that use D-BUS, and generally map to objects in object-oriented languages. Because D-BUS would not find using a pointer or reference to identify an object very friendly, it introduces a name for each object. The name resembles a UNIX filesystem path, such as /org/kernel/fs/filesystem.
- **Interfaces** represent methods or signals implemented on an object. Each object supports at least one interface.

- **Messages** are sent to and from a defined method or signal. D-BUS supports multiple message types: method invocation, method return, error message, and signal.

3.2 Use of D-BUS

D-BUS's simplicity, performance, and use of the message and bus paradigm set it up for use across the entire Linux system and make it a perfect replacement for CORBA, DCOP, and other IPC mechanisms.

Multiple projects are taking advantage of D-BUS. They include:

- Project Utopia uses D-BUS as the IPC mechanism to link the kernel, udev, HAL, and the GNOME desktop.
- A CUPS patch uses D-BUS to transmit information about the printer spool.
- Jamboree uses D-BUS to automatically mute the volume.
- A Gconf patch uses D-BUS as the Gconf transport mechanism.

4 The Kernel Events Layer, D-BUS, and Project Utopia

D-BUS is used as the backbone of Project Utopia, an umbrella project aiming to bring improved hardware management and system integration to the Linux system and GNOME desktop. Project Utopia uses D-BUS to link the kernel, up through hotplug, udev and HAL to the rest of the system. Libraries utilizing D-BUS and built on top of HAL provide enhanced hardware support. Applications at the desktop level can then reap the benefits.

4.1 Example: libinput

`libinput` is a simple library for managing input devices that sits on top of HAL and communicates to HAL beneath it and the applications above it via D-BUS. `libinput` is used to enumerate all input devices on the system. `libinput` also provides an interface for applications to register callbacks, and integrate these callbacks into its mainloop. The callbacks are invoked when input devices are added to or removed from the system.

Sample usage of enumerating all input devices on the system:

```
struct input *devices;

if (input_init ())
        /* error ... */

devices = input_devices_get ();
while (devices) {
        /* ... */
        devices = devices->next;
}
input_devices_put (devices);
```

Given a specific `struct input`, the library provides wrappers for opening and closing the device via `open`(2) and `close`(2). This is not strictly required, but furthers the abstracting of device nodes not only from the user but even from the application.

Example:

```
fd = input_device_open (device, 0);

/* ... */

input_device_close (device);
```

Registering of the callbacks is also easy:

```
void my_mainloop
```

```
(DBusConnection *dbus_connection)
{
  dbus_connection_setup_with_g_main
  (dbus_connection, NULL);
}

void my_added
(struct input *device)
{
  printf
  ("%s was just "
          "hotplugged!\n",
          device->product);
}

void my_removed
(struct input *device)
{
  printf
  ("%s was just "
         "hotunplugged!\n",
         device->product);
}

  /* ... */
  input_init_with_callbacks
        (&my_mainloop,
         &my_added,
         &my_removed);

  gtk_main ();
```

When an input device is added or removed from the system, `my_added` and `my_removed` are invoked as appropriate.

The goals behind such a library are twofold:

- Abstract away concepts of device nodes and low-level system-specific behavior and allow application developers to search for enumerate the devices on a system through simple interfaces.

- Allow asynchronous poll-free hack-free callbacks into the application to notify the program of changes in events, such as a new joystick on the system.

5 Conclusion

The Kernel Events Layer and D-BUS are two crucial components in better unifying and integrating the Linux system. They provide the infrastructure required for a future rich with information exchange. Where all levels of the desktop can communicate—talking, listening, evolving.

Linux-tiny And Directions For Small Systems

Matt Mackall
Digeo, Inc.
mpm@digeo.com

Abstract

Linux-tiny is a project to reduce the memory and storage footprint of the 2.6 Linux kernel for embedded, handheld, legacy, and other small systems. I describe strategies for kernel size reduction, some of the major areas already investigated and the results achieved, as well as some avenues for further exploration.

1 Introduction

Historically, Linux had a reputation for running on very modest systems. My first dedicated Linux box, running a 0.99 kernel circa 1994, provided mail, FTP, web, dial-in, and shell services on a 16MHz 386SX with a mere 4 megabytes of RAM. In the 10 years since then, Linux has grown to the point where it runs on machines with over a thousand processors and a terabyte of RAM. Not surprisingly, a modern Linux distribution can have difficulty getting to a shell prompt on machines with less than 8 megabytes of RAM, let alone doing useful work.

1.1 What happened?

In the time between the 0.99 and 2.6 kernels, we've seen Linux become a serious commercial endeavor, we've seen kernel hackers get jobs (and get big machines on their desks), and we've seen a massive boom in Internet use and personal computing. Linux developers have been targeting high end computing and rising demand for hardware has seen prices drop tremendously.

But there are still small machines! Hand-helds and embedded systems are perennially pressed for space to match their desktop counterparts and many people throughout the world still rely on legacy machines to get their work done. What can be done to recapture the 'small is beautiful' utility of those early systems?

1.2 Where is the growth?

The process by which any large software project grows can aptly be described as *death by a thousand cuts*. The accumulation of bloat occurs change by change and creeps in from several different directions.

Perhaps the most visible is the addition of new **features**, which generally requires the introduction of wholly-new code. Frequently features are considered so small or so essential that no thought is given to making them optional. As the median system size grows, this new code tends to be more verbose and less concerned with space issues.

The next, more subtle culprit is **performance**. Given the fundamental importance of kernel performance to overall system performance, trade-offs of size for speed are easy to justify. Unfortunately the accumulation of many such trade-offs can leave us with a system that no longer boots. Ironically, the evolution of pro-

cessors has brought us to a point where cache footprint can be critical to performance so a lot of the choices that have been made in this area bear rethinking.

Next we have **compatibility** and **correctness**. Every time the system is extended to better support a slightly different piece of hardware or work around another corner case, more code is added. Occassionally cleanups and unifications make some of this code redundant, but this is the exception. A related phenomenon is the evolution of the kernel APIs and the accumulation of obsolete code for the sake of backward compatibility.

2 Linux-Tiny for the small system niche

There have been numerous efforts to address the above phenomena for various components of Linux systems, but most of the attention has been addressed at userspace (arguably the biggest offender). Experiments with pre-2.6.0 kernels however suggested it was time to pay some more attention to the kernel itself. So in December of 2003, I decided to create a new 2.6-based tree dedicated to small systems which I named Linux-Tiny [3] (someone had already borrowed my initials for their tree).

2.1 Methodology

With stated targets of embedded, hand-held, and legacy machines, the -tiny tree attempts to tailor the kernel to the needs of small systems. The tree is maintained as a series of small patches stacked on top of mainline kernel releases, managed with the quilt tool [1] (previously with Andrew Morton's patch scripts [4]).

Patches try to observe the following criteria:

- **configurable**: changes that are not clearly wins for all systems should be configurable so that users can make their own trade-offs
- **non-invasive**: patches should be small, self-contained, and largely independent so that integrators can cherrypick the patches they'd like to use
- **mergeable**: while not mandatory, patches should try to be acceptable to the mainline kernel in both style and approach; merging to mainline is a priority

In addition to patches focusing on reducing kernel footprint, I've also added a number of patches to do debugging and auditing including netconsole, kgdb, and kgdb-over-ethernet support.

2.2 Setting goals

Everyone has a different set of functionality requirements in mind for small systems. The features needed on a handheld are very different from those needed for a network appliance or a kiosk. Thus, choosing a subset of features to develop towards is tricky.

The approach I've taken is to choose a series of targets to optimize, and the first is a minimal x86 kernel with filesystem, console, and TCP/IP support. How small can we make this kernel? This puts a focus on the most of the common core functionality of Linux and provides a useful benchmark for progress.

3 Finding bloat

As mentioned above, there are many sources of bloat. There are also several forms it can take: as superfluous code, statically or dynamically allocated data, inline functions or macros, compiler mis-optimizations, or cut-n-paste coding.

Given that the kernel is on the order of several hundred kilobytes, tackling bloat is going to be a matter of trimming several kilobytes here and a couple kilobytes there. While one could simply pick any source file and read through it searching for cleanup opportunities, there are some more straightforward ways of finding the "low-hanging fruit".

3.1 Using nm(1) and size(1)

The easiest place to begin is by using the `nm` tool to find large functions and data structures. Comparing the (hexadecimal) numbers from `nm(1)` with `size(1)` gives us a good start at understanding the relative sizes of some of the major subsystems and their components compared to the kernel as a whole. For instance, we can see by comparing Table 1 and Table 2 that the static `ide_hwifs` data structure alone takes 15360 bytes, over 2% of the data portion of the default kernel.

3.2 Measuring function inlining

Function inlining and macro expansion present a special problem for our bloat detection efforts. In the early 1990s, inlining was a very popular performance technique to avoid function call branches. A great number of key functions are marked for inlining in the kernel and their usage and size impact is obscured because they become a seamless part of the functions that use them. Auditing their usage becomes a matter of convincing the compiler to tell us when inlines are being instantiated in a build and then estimating how large these functions are when expanded inline.

Rather than modifying the compiler itself, the first part of this puzzle was hacked around by redefining `inline` to include the GCC extension `__attribute__((deprecated))`. This causes a very useful warning like the following to be generated:

```
arch/i386/kernel/semaphore.c:58:
warning: 'get_current' is
deprecated (declared at
include/asm/current.h:16)
```

By post-processing these voluminous warning messages, we can determine which inline functions are instantiated directly in C files as well as which are called as parts of other inlines and finally calculate the total number of direct or indirect instantiations of each (see Table 3).

The second part of this puzzle was more challenging. While we know in which modules and how often inlines are instantiated, we cannot yet calculate their sizes. I made several attempts to generate approximate size data by looking at GCC's symbolic debugging output, but this tended to be easily confused by inlining and was too inaccurate for use.

Recently Denis Vlasenko took another stab at this and wrote a set of scripts called inline_hunter [5] to generate a set of dummy functions wrapping single calls to inlines. While these sizes won't directly reflect the size of inline instantiations due to function call overhead and lost optimization opportunities, for larger inline functions, it has proven fairly representative. Some of the larger inlines found with this approach are shown in Table 4.

3.3 Tracking dynamic allocations

Of course much of the kernel's memory footprint is from dynamic allocations. Memory used for page tables, tracking running processes, indexing hashes and so forth is allocated at runtime and can vary with the size of the load. A number of these are hash tables to increase look-up performance, which for small systems can be less important than simply fitting in memory.

There are several important allocators in the kernel. First, the bootmem allocator which

```
2.6.5\$ nm --size -r vmlinux | head -20
00008000 b __log_buf
00007000 D irq_desc
00004e78 d pci_vendor_list
00004000 b bh_wait_queue_heads
00003c00 B ide_hwifs
0000213a T vt_ioctl
00002000 D init_thread_union
00001880 D contig_page_data
0000163b T journal_commit_transaction
00001500 b irq_2_pin
000012f5 T tcp_sendmsg
00001162 t n_tty_receive_buf
00001080 d per_cpu__tvec_bases
00001000 t translation_table
00001000 b sd_index_bits
00001000 D init_tss
00001000 b doublefault_stack
00001000 B con_buf
00001000 b cache_defer_hash
00000fe0 T cdrom_ioctl
```

Table 1: nm output for 2.6.5 default config

handles a number of critical allocations at startup. As there are not terribly many of these, they can be audited very simply with `printk()` techniques.

Second, the SLAB allocator is used to quickly allocate sets of objects of the same size and type. The kernel provides a way to track these allocations with `/proc/slabinfo`.

The more general `kmalloc()` allocator has been rebuilt on top of the aforementioned SLAB allocator, translating kmalloc requests into requests from a set of ascending generic SLAB sizes. Thus all `kmalloc()` allocations are lumped together by size in the `/proc/slabinfo` output. That can be helpful if you know what you're looking for, but doesn't give many hints as to which parts of the kernel are using that memory.

To address this deficiency, I've created a small footprint tool for tracking allocations via `/proc/kmalloc` (see Table 5). This works by tracking the address of each allocation along with the address of the allocating function in a simple hash table. Also tracked are net and gross allocation sizes and counts per caller. When a `kfree()` call is made, it is matched up to its caller for accounting purposes and removed from the hash. Thus it is possible not only to determine how much dynamic memory is used by each function but also to easily identify memory leaks.

4 Some notable opportunities for code trimming

The above methods have revealed numerous opportunities for cutting back the kernel's

```
2.6.5\$ size vmlinux */built-in.o
   text    data     bss     dec     hex filename
3366220  673296  166824 4206340  402f04 vmlinux
1181276  250808   48000 1480084  169594 drivers/built-in.o
 735152   32593   30628  798373   c2ea5 fs/built-in.o
  18151    1120    1316   20587    506b init/built-in.o
  21841     172     204   22217    56c9 ipc/built-in.o
 159632   16115   42402  218149   35425 kernel/built-in.o
   2870       0       0    2870     b36 lib/built-in.o
 129669    9068    2884  141621   22935 mm/built-in.o
 580407   33816   18856  633079   9a8f7 net/built-in.o
   1869       0       0    1869     74d security/built-in.o
 325923   11114    3016  340053   53055 sound/built-in.o
    134       0       0     134      86 usr/built-in.o
```

Table 2: size output for 2.6.5 default config

memory footprint, many of which remain to be examined. What follows are some of the more notable areas that have been explored.

4.1 Debugging data

The kernel has numerous facilities for trapping and reporting problem conditions and other status information, including `printk()`, `bug()`, `warn()`, `panic()`, and friends. In ideal circumstances, these facilities go unexercised. And in the extreme, embedded boxes may have no means of reporting this data, due to lack of a display, writable storage, or the like. Unfortunately, not only do these facilities use a substantial amount of code, their users need extra space for error message strings, filenames, and line numbers.

Linux-tiny has a set of configuration options to compile out most of this code and remove the debugging strings and data from the kernel. Disabling support for `printk()` saves well over 100K. Independent options control the inclusion of the `bug()` infrastructure and support for trapping panics and doublefaults.

4.2 Optional interfaces

For systems with well-defined application requirements, many of the kernel's APIs are unnecessary. Cutting-edge, obsolete, or obscure features are obvious candidates for configurable removal.

- **sysfs**: The new sysfs filesystem makes substantial memory demands (which can be more than half a megabyte even on the smallest systems) but its features may well not be essential to current systems. The -tiny tree was a testbed for options to entirely remove sysfs or to use a lighter "backing store" version.
- **ptrace, aio, posix-timers**: These features are among those that are only used by a small set of applications. These and other Linux-tiny options are enabled under the CONFIG_EMBEDDED menu, which marks them as making the kernel non-standard.
- **uid16, vm86**: Some of the many legacy interfaces in the kernel. Modern applications and libraries use 32-bit user and

group IDs and vm86 support is used to run 16-bit code for emulators like DOSEMU and Wine and for some video drivers used by X.

- **ethtool, tcpdiag, igmp, rtnetlink**: One of the most complicated parts of the kernel is the networking layer, which has grown a variety of APIs to gain access to its many features. But for most users, the interfaces used by the classic `ifconfig(8)` and `route(8)` tools are sufficient.

4.3 4K stacks

During the 2.1 kernel series (circa 1998), the x86 kernel increased the size of the per-task kernel stacks from 4K to 8K to work around issues with stack depth. In addition to the obvious increase in overhead for every userspace process, several new kernel daemons have been added, all with their own stacks. Another issue is that finding pairs of contiguous pages to build an 8K stack can be very difficult on a machine with memory pressure and especially so on machines with a small number of total pages.

Many of the problems that made 4K stacks problematic have since been addressed and 4K stacks are now practical for most applications. Linux-tiny has served as an early testbed for reintroducing 4K stack support to the mainline 2.6 kernel and includes a developer tool called `checkstack` that will automatically disassemble a kernel to find the most extreme stack space users.

4.4 The SLOB allocator

Most memory in the kernel is managed either directly or indirectly through the SLAB allocator. SLAB maintains separate caches for objects of given sizes and types and can very quickly manage allocations for them. In some cases, it can even arrange for objects to be pre-initialized without any additional overhead. SLAB also has some resistance to troublesome memory fragmentation issues. While simple in principle, the SLAB code ends up being quite complex from its efforts to squeeze the maximum possible performance out of the allocator.

The primary downside to SLAB is that because it maintains a collection of independent caches which are all one or more pages, it ends up leaving quite a bit of unused space in each SLAB cache. In addition, as `kmalloc` is implemented on top of SLAB using a set of preset object size SLABs, there is quite a bit of extra space allocated for the average `kmalloc` call. Measurements with the previously described `/proc/kmalloc` tool report that extra overhead can amount to 25-30% of the total memory allocated by `kmalloc`.

Linux-tiny provides an optional replacement for SLAB that I've dubbed *SLOB* (simple list of blocks). SLOB trades performance for space efficiency by implementing a more traditional list-based allocator that also understands requests for objects with particular alignments. The APIs used by SLAB and `kmalloc()` are provided by a small emulation layer.

SLOB manages all objects at a granularity of 8 bytes so overhead for odd object sizes is minimized. It also does away with the numerous partly-used caches of the SLOB approach. Finally, the SLOB code is much simpler and takes up less than one tenth of the space of the standard SLAB allocator.

4.5 TinyVT

As you can see from Table 1, the largest single function in the default kernel is `vt_ioctl()`, which manages many of the special features of the Linux console. As most early Linux

users didn't have the memory for running a full-fledged X desktop, the native Linux text console is very powerful, with support for scrollback, selection, virtual console switching, Unicode translation and character sets, screen blanking, and so on.

These features can be very handy for some users, but on a palmtop or kiosk running a GUI, or for a minimal rescue disk, they're dead weight. Linux-tiny includes a heavily trimmed down replacement for the standard console code which drops many of these features and can trim a couple percent off the size of the kernel image.

5 Results

Recent releases of Linux-tiny contain the above options and numerous others. My test configuration, with support for a text console, IDE disks, the Ext2 filesystem, TCP/IP, and a PCI-based network card results in a 363K compressed kernel image. Other users of Linux-tiny have reported kernel configurations resulting in images as small as 191K.

Booting the test configuration with `mem=2M`, which gives a total of of 1664K after accounting for BIOS memory holes, still leaves ample room for a lightweight userspace (see Table 6). A similarly configured mainline kernel without the -tiny patches compiles to a kernel image of over 500K and has difficulty booting with `mem=4M`.

For comparison, the earliest Linux distribution kernel I've been able to locate, a 0.99pl15 kernel from Slackware 1.1.2 circa 1994, is a mere 301K. Modern `highly-modularized` 2.6 kernels from Fedora Core 2 and SuSE 9.1 weigh in at 1.2M and 1.5M respectively while the default 2.6.5 kernel config builds a 1.9M compressed kernel.

6 Further directions

There are many further avenues to pursue and subsystems to trim. Some of the more aggressive ideas on the to-do list include:

- A lightweight replacement network stack: Minimal TCP stacks like uIP [2] have sufficient functionality for simple network applications and have extremely small footprints.
- Replacements for fixed-sized hash tables: Existing kernel hash tables have difficulty scaling with workloads and memory sizes. Other approaches like radix trees might be better in some areas and avoid wasted memory when the indexes are empty.
- Support for bunzip2: Linux-tiny now has a simplified interface to the boot-time decompressor and allows for replacements to be easily dropped in. While bzip2 compression won't save any memory at runtime, it will save valuable storage space on embedded systems.
- Pageable kernel memory: Following an approach similar to the `__init` approach in current kernels, it should be possible to mark specific functions and data in the kernel core as pageable, provided they meet some specific requirements.
- Tracking kernel growth: Using automated tools to track the size of kernel functions and subsystems from release to release will help catch new bloat when it appears.

Of course, as most of the bloat in the kernel has been introduced in small increments, most of the improvements will be of the same variety. Contributions are encouraged!

References

[1] patchwork quilt patch management tools. `http://savannah.nongnu.org/projects/quilt`.

[2] Adam Dunkels. The uip tcp/ip stack for embedded microcontrollers. `http://www.sics.se/~adam/uip/index.html`.

[3] Matt Mackall. The linux-tiny homepage. `http://www.selenic.com/tiny-about/`.

[4] Andrew Morton. Patch-scripts. `http://www.zip.com.au/~akpm/linux/patches/`.

[5] Denis Vlasenko. inline_hunter 2.0 and its results, 2004. `http://lkml.org/lkml/2004/4/16/191`.

```
1560  get_current (1294 in *.c)
calls:
callers: <other>(336) capable(122) unlock_kernel(44) lock_kernel(33)
flush_tlb_page(11) flush_tlb_mm(10) find_process_by_pid(6)
flush_tlb_range(4) current_is_kswapd(4) current_is_pdflush(3)
rwsem_down_failed_common(2) on_sig_stack(2) do_mmap2(2) __exit_mm(2)
walk_init_root(1) scm_check_creds(1) save_i387_fsave(1)
sas_ss_flags(1) restore_i387_fsave(1) read_zero_pagealigned(1)
handle_group_stop(1) get_close_on_exec(1) fork_traceflag(1)
ext2_init_acl(1) exec_permission_lite(1) dup_mmap(1) do_tty_write(1)
de_thread(1) copy_signal(1) copy_sighand(1) copy_fs(1) check_sticky(1)
cap_set_all(1) cap_emulate_setxuid(1) arch_get_unmapped_area(1)

546  current_thread_info (286 in *.c)
calls:
callers: <other>(207) copy_to_user(95) copy_from_user(86)
tcp_set_state(22) test_thread_flag(20) verify_area(13)
tcp_enter_memory_pressure(6) sock_orphan(3) icmp_xmit_lock(2)
csum_and_copy_to_user(2) tcp_v4_lookup(1) sock_graft(1)
set_thread_flag(1) neigh_update_hhs(1) ip_finish_output2(1) gfp_any(1)
fn_flush_list(1) do_getname(1) clear_thread_flag(1) alloc_buf(1)
activate_task(1)

413  atomic_dec_and_test (55 in *.c)
calls:
callers: put_page(103) kfree_skb(101) <other>(47) mntput(34)
in_dev_put(23) neigh_release(19) tcp_tw_put(18) fib_info_put(17)
sock_put(15) put_namespace(6) mmdrop(6) __put_fs_struct(4)
tcp_listen_unlock(3) ipq_put(3) finish_task_switch(2) __detach_pid(2)
task_state(1) de_thread(1)

255  tcp_sk (134 in *.c)
calls:
callers: <other>(117) tcp_reset_xmit_timer(30) tcp_set_state(22)
tcp_current_mss(13) tcp_initialize_rcv_mss(6) tcp_free_skb(6)
tcp_check_space(6) tcp_data_snd_check(5) tcp_clear_xmit_timer(5)
tcp_synq_removed(3) tcp_select_window(3) westwood_update_rttmin(2)
westwood_acked(2) tcp_synq_len(2) tcp_synq_drop(2)
tcp_ack_snd_check(2) __tcp_inherit_port(2) tcp_use_frto(1)
tcp_synq_young(1) tcp_synq_is_full(1) tcp_synq_added(1)
tcp_prequeue(1) tcp_listen_poll(1) tcp_event_ack_sent(1)
tcp_connect_init(1) tcp_acceptq_queue(1) do_pmtu_discovery(1)
```

Table 3: Some large inline counts and users for 2.6.5-tiny1

Size	Uses	Wasted	Name and definition	
56	461	16560	copy_from_user	include/asm/uaccess.h
122	119	12036	skb_dequeue	include/linux/skbuff.h
164	78	11088	skb_queue_purge	include/linux/skbuff.h
97	141	10780	netif_wake_queue	include/linux/netdevice.h
43	468	10741	copy_to_user	include/asm/uaccess.h
43	461	10580	copy_from_user	include/asm/uaccess.h
145	77	9500	put_page	include/linux/mm.h
49	313	9048	skb_put	include/linux/skbuff.h
109	101	8900	skb_queue_tail	include/linux/skbuff.h
381	21	7220	sock_queue_rcv_skb	include/net/sock.h
55	191	6650	init_MUTEX	include/asm/semaphore.h
61	163	6642	unlock_kernel	include/linux/smp_lock.h
59	165	6396	lock_kernel	include/linux/smp_lock.h
127	59	6206	dev_kfree_skb_any	include/linux/netdevice.h
41	289	6048	list_del	include/linux/list.h
73	83	4346	dev_kfree_skb_irq	include/linux/netdevice.h
131	39	4218	netif_device_attach	include/linux/netdevice.h
110	44	3870	skb_queue_head	include/linux/skbuff.h
84	59	3712	seq_puts	include/linux/seq_file.h
57	75	2738	skb_trim	include/linux/skbuff.h
45	96	2375	skb_queue_head_init	include/linux/skbuff.h
41	111	2310	list_del_init	include/linux/list.h
102	23	1804	__nlmsg_put	include/linux/netlink.h

Table 4: Size estimates found by inline_hunter

```
# cat /proc/kmalloc
total bytes allocated:   266848
slack bytes allocated:    37774
net bytes allocated:     145568
number of allocs:           732
number of frees:            282
number of callers:           71
lost callers:                 0
lost allocs:                  0
unknown frees:                0

   total    slack      net alloc/free  caller
     256      203      256     8/0     alloc_vfsmnt+0x73
    8192     3648     4096     2/1     atkbd_connect+0x1b
     192       48       64     3/2     seq_open+0x10
   12288        0     4096     3/2     seq_read+0x53
    8192        0        0     2/2     alloc_skb+0x3b
     960        0        0    10/10    load_elf_interp+0xa1
    1920      288        0    10/10    load_elf_binary+0x100
     320      130        0    10/10    load_elf_binary+0x1d8
     192       48       96     6/3     request_irq+0x22
    7200     1254     7200    75/0     proc_create+0x74
      64       43       64     2/0     proc_symlink+0x40
    4096      984        0     1/1     check_partition+0x1b
   69632        0    45056    17/6     dup_task_struct+0x38
     128       48      128     2/0     netlink_create+0x84
     128       20      128     1/0     ext2_fill_super+0x2f
      32       28       32     1/0     ext2_fill_super+0x385
      32       31       32     1/0     ext2_fill_super+0x3b6
     608       76      384    19/7     __request_region+0x18
      64       32       64     2/0     rand_initialize_disk+0xd
    8192     2016     8192     2/0     alloc_tty_struct+0x10
     128       56      128     2/0     init_dev+0xba
     128       56      128     2/0     init_dev+0xf3
     128        0      128     2/0     create_workqueue+0x28
    8960     1680     8960    70/0     tty_add_class_device+0x20
    2048      960     2048     4/0     alloc_tty_driver+0x10
    9280     2332     9280     4/0     tty_register_driver+0x2d
     288        0      288     9/0     mempool_create+0x16
    1280      196     1280     9/0     mempool_create+0x41
    1536      384     1536     8/0     mempool_create+0x8f
      64       28       64     1/0     kbd_connect+0x3e
     928      348        0    29/29    kmem_cache_create+0x235
   28288     1448    28288    81/0     do_tune_cpucache+0x2c
...
```

Table 5: Tracking usage of kmalloc/kfree in -tiny

```
Uncompressing Linux... Ok, booting the kernel.
# mount /proc
# cat /proc/meminfo
MemTotal:          980 kB
MemFree:           312 kB
Buffers:            32 kB
Cached:            296 kB
SwapCached:          0 kB
Active:            400 kB
Inactive:           48 kB
HighTotal:           0 kB
HighFree:            0 kB
LowTotal:          980 kB
LowFree:           312 kB
SwapTotal:           0 kB
SwapFree:            0 kB
Dirty:               0 kB
Writeback:           0 kB
Mapped:            380 kB
Slab:                0 kB
Committed_AS:      132 kB
PageTables:         24 kB
VmallocTotal:  1032172 kB
VmallocUsed:         0 kB
VmallocChunk:  1032172 kB
#
```

Table 6: Boot log for a 2.6.5-tiny1 test configuration with mem=2m

Xen and the Art of Open Source Virtualization

Keir Fraser, Steven Hand, Christian Limpach, Ian Pratt
University of Cambridge Computer Laboratory
{first.last}@cl.cam.ac.uk

Dan Magenheimer
Hewlett-Packard Laboratories
{first.last}@hp.com

Abstract

Virtual machine (VM) technology has been around for 40 years and has been experiencing a resurgence with commodity machines. VMs have been shown to improve system and network flexibility, availability, and security in a variety of novel ways. This paper introduces Xen, an efficient secure open source VM monitor, to the Linux community.

Key features of Xen are:

1. supports different OSes (e.g. Linux 2.4, 2.6, NetBSD, FreeBSD, etc.)
2. provides secure protection between VMs
3. allows flexible partitioning of resources between VMs (CPU, memory, network bandwidth, disk space, and bandwidth)
4. very low overhead, even for demanding server applications
5. support for seamless, low-latency migration of running VMs within a cluster

We discuss the interface that Xen/x86 exports to guest operating systems, and the kernel changes that were required to Linux to port it to Xen. We compare Xen/Linux to User Mode Linux as well as existing commercial VM products.

1 Introduction

Modern computers are sufficiently powerful to use virtualization to present the illusion of many smaller virtual machines (VMs), each running a separate operating system instance. This has led to a resurgence of interest in VM technology. In this paper we present Xen, a high performance resource-managed virtual machine monitor (VMM) which enables applications such as server consolidation, co-located hosting facilities, distributed web services, secure computing platforms, and application mobility.

Successful partitioning of a machine to support the concurrent execution of multiple operating systems poses several challenges. Firstly, virtual machines must be isolated from one another: it is not acceptable for the execution of one to adversely affect the performance of another. This is particularly true when virtual machines are owned by mutually untrusting users. Secondly, it is necessary to support a variety of different operating systems to accommodate the heterogeneity of popular applications. Thirdly, the performance overhead introduced by virtualization should be small.

Xen hosts commodity operating systems, albeit with some source modifications. The prototype described and evaluated in this paper can support multiple concurrent instances of our XenLinux guest operating system; each instance exports an application binary interface identical to a non-virtualized Linux 2.6. Xen ports of NetBSD and FreeBSD have been completed, along with a proof of concept port of Windows XP.[1]

There are a number of ways to build a system to host multiple applications and servers on a shared machine. Perhaps the simplest is to deploy one or more hosts running a standard operating system such as Linux or Windows, and then to allow users to install files and start processes—protection between applications being provided by conventional OS techniques. Experience shows that system administration can quickly become a time-consuming task due to complex configuration interactions between supposedly disjoint applications.

More importantly, such systems do not adequately support performance isolation; the scheduling priority, memory demand, network traffic and disk accesses of one process impact the performance of others. This may be acceptable when there is adequate provisioning and a closed user group (such as in the case of computational grids, or the experimental PlanetLab platform [11]), but not when resources are oversubscribed, or users uncooperative.

One way to address this problem is to retrofit support for performance isolation to the operating system, but a difficulty with such approaches is ensuring that *all* resource usage is accounted to the correct process—consider, for example, the complex interactions between applications due to buffer cache or page replacement algorithms. Performing multiplexing at a low level can mitigate this problem; unintentional or undesired interactions between tasks are minimized. Xen multiplexes physical resources at the granularity of an entire operating system and is able to provide performance isolation between them. This allows a range of guest operating systems to gracefully coexist rather than mandating a specific application binary interface. There is a price to pay for this flexibility—running a full OS is more heavyweight than running a process, both in terms of initialization (e.g. booting or resuming an OS instance versus `fork`/`exec`), and in terms of resource consumption.

For our target of 10-100 hosted OS instances, we believe this price is worth paying: It allows individual users to run unmodified binaries, or collections of binaries, in a resource controlled fashion (for instance an Apache server along with a PostgreSQL backend). Furthermore it provides an extremely high level of flexibility since the user can dynamically create the precise execution environment their software requires. Unfortunate configuration interactions between various services and applications are avoided (for example, each Windows instance maintains its own registry).

Experience with deployed Xen systems suggests that the initialization overheads and additional resource requirements are in practice quite low: An operating system image may be resumed from an on-disk snapshot in typically just over a second (depending on image memory size), and although multiple copies of the operating system code and data are stored in memory, the memory requirements are typically small compared to those of the applications that will run on them. As we shall show later in the paper, the performance overhead of the virtualization provided by Xen is low, typically just a few percent, even for the most demanding applications.

[1] The Windows XP port required access to Microsoft source code, and hence distribution is currently restricted, even in binary form.

2 XEN: Approach & Overview

In a traditional VMM the virtual hardware exposed is functionally identical to the underlying machine [14]. Although *full virtualization* has the obvious benefit of allowing unmodified operating systems to be hosted, it also has a number of drawbacks. This is particularly true for the prevalent Intel *x86* architecture.

Support for full virtualization was never part of the x86 architectural design. Certain supervisor instructions must be handled by the VMM for correct virtualization, but executing these with insufficient privilege fails silently rather than causing a convenient trap [13]. Efficiently virtualizing the x86 MMU is also difficult. These problems can be solved, but only at the cost of increased complexity and reduced performance. VMware's ESX Server [3] dynamically rewrites portions of the hosted machine code to insert traps wherever VMM intervention might be required. This translation is applied to the entire guest OS kernel (with associated translation, execution, and caching costs) since all non-trapping privileged instructions must be caught and handled. ESX Server implements shadow versions of system structures such as page tables and maintains consistency with the virtual tables by trapping every update attempt—this approach has a high cost for update-intensive operations such as creating a new application process.

Notwithstanding the intricacies of the x86, there are other arguments against full virtualization. In particular, there are situations in which it is desirable for the hosted operating systems to see real as well as virtual resources: providing both real and virtual time allows a guest OS to better support time-sensitive tasks, and to correctly handle TCP timeouts and RTT estimates, while exposing real machine addresses allows a guest OS to improve performance by using superpages [10] or page coloring [7].

We avoid the drawbacks of full virtualization by presenting a virtual machine abstraction that is similar but not identical to the underlying hardware—an approach which has been dubbed *paravirtualization* [17]. This promises improved performance, although it does require modifications to the guest operating system. It is important to note, however, that we do not require changes to the application binary interface (ABI), and hence no modifications are required to guest *applications*.

We distill the discussion so far into a set of design principles:

1. Support for unmodified application binaries is essential, or users will not transition to Xen. Hence we must virtualize all architectural features required by existing standard ABIs.
2. Supporting full multi-application operating systems is important, as this allows complex server configurations to be virtualized within a single guest OS instance.
3. Paravirtualization is necessary to obtain high performance and strong resource isolation on uncooperative machine architectures such as x86.
4. Even on cooperative machine architectures, completely hiding the effects of resource virtualization from guest OSes risks both correctness and performance.

In the following section we describe the virtual machine abstraction exported by Xen and discuss how a guest OS must be modified to conform to this. Note that in this paper we reserve the term *guest operating system* to refer to one of the OSes that Xen can host and we use the term *domain* to refer to a running virtual machine within which a guest OS executes; the

distinction is analogous to that between a *program* and a *process* in a conventional system. We call Xen itself the *hypervisor* since it operates at a higher privilege level than the supervisor code of the guest operating systems that it hosts.

2.1 The Virtual Machine Interface

The paravirtualized x86 interface can be factored into three broad aspects of the system: memory management, the CPU, and device I/O. In the following we address each machine subsystem in turn, and discuss how each is presented in our paravirtualized architecture. Note that although certain parts of our implementation, such as memory management, are specific to the x86, many aspects (such as our virtual CPU and I/O devices) can be readily applied to other machine architectures. Furthermore, x86 represents a *worst case* in the areas where it differs significantly from RISC-style processors—for example, efficiently virtualizing hardware page tables is more difficult than virtualizing a software-managed TLB.

2.1.1 Memory management

Virtualizing memory is undoubtedly the most difficult part of paravirtualizing an architecture, both in terms of the mechanisms required in the hypervisor and modifications required to port each guest OS. The task is easier if the architecture provides a software-managed TLB as these can be efficiently virtualized in a simple manner [5]. A tagged TLB is another useful feature supported by most server-class RISC architectures, including Alpha, MIPS and SPARC. Associating an address-space identifier tag with each TLB entry allows the hypervisor and each guest OS to efficiently coexist in separate address spaces because there is no need to flush the entire TLB when transferring execution.

Unfortunately, x86 does not have a software-managed TLB; instead TLB misses are serviced automatically by the processor by walking the page table structure in hardware. Thus to achieve the best possible performance, all valid page translations for the current address space should be present in the hardware-accessible page table. Moreover, because the TLB is not tagged, address space switches typically require a complete TLB flush. Given these limitations, we made two decisions: (i) guest OSes are responsible for allocating and managing the hardware page tables, with minimal involvement from Xen to ensure safety and isolation; and (ii) Xen exists in a 64MB section at the top of every address space, thus avoiding a TLB flush when entering and leaving the hypervisor.

Each time a guest OS requires a new page table, perhaps because a new process is being created, it allocates and initializes a page from its own memory reservation and registers it with Xen. At this point the OS must relinquish direct write privileges to the page-table memory: all subsequent updates must be validated by Xen. This restricts updates in a number of ways, including only allowing an OS to map pages that it owns, and disallowing writable mappings of page tables. Guest OSes may *batch* update requests to amortize the overhead of entering the hypervisor. The top 64MB region of each address space, which is reserved for Xen, is not accessible or remappable by guest OSes. This address region is not used by any of the common x86 ABIs however, so this restriction does not break application compatibility.

Segmentation is virtualized in a similar way, by validating updates to hardware segment descriptor tables. The only restrictions on x86 segment descriptors are: (i) they must have

lower privilege than Xen, and (ii) they may not allow any access to the Xen-reserved portion of the address space.

2.1.2 CPU

Virtualizing the CPU has several implications for guest OSes. Principally, the insertion of a hypervisor below the operating system violates the usual assumption that the OS is the most privileged entity in the system. In order to protect the hypervisor from OS misbehavior (and domains from one another) guest OSes must be modified to run at a lower privilege level.

Efficient virtualizion of privilege levels is possible on x86 because it supports four distinct privilege levels in hardware. The x86 privilege levels are generally described as *rings*, and are numbered from zero (most privileged) to three (least privileged). OS code typically executes in ring 0 because no other ring can execute privileged instructions, while ring 3 is generally used for application code. To our knowledge, rings 1 and 2 have not been used by any well-known x86 OS since OS/2. Any OS which follows this common arrangement can be ported to Xen by modifying it to execute in ring 1. This prevents the guest OS from directly executing privileged instructions, yet it remains safely isolated from applications running in ring 3.

Privileged instructions are paravirtualized by requiring them to be validated and executed within Xen—this applies to operations such as installing a new page table, or yielding the processor when idle (rather than attempting to `hlt` it). Any guest OS attempt to directly execute a privileged instruction is failed by the processor, either silently or by taking a fault, since only Xen executes at a sufficiently privileged level.

Exceptions, including memory faults and software traps, are virtualized on x86 very straightforwardly. A table describing the handler for each type of exception is registered with Xen for validation. The handlers specified in this table are generally identical to those for real x86 hardware; this is possible because the exception stack frames are unmodified in our paravirtualized architecture. The sole modification is to the page fault handler, which would normally read the faulting address from a privileged processor register (`CR2`); since this is not possible, we write it into an extended stack frame[2]. When an exception occurs while executing outside ring 0, Xen's handler creates a copy of the exception stack frame on the guest OS stack and returns control to the appropriate registered handler.

Typically only two types of exception occur frequently enough to affect system performance: system calls (which are usually implemented via a software exception), and page faults. We improve the performance of system calls by allowing each guest OS to register a 'fast' exception handler which is accessed directly by the processor without indirecting via ring 0; this handler is validated before installing it in the hardware exception table. Unfortunately it is not possible to apply the same technique to the page fault handler because only code executing in ring 0 can read the faulting address from register `CR2`; page faults must therefore always be delivered via Xen so that this register value can be saved for access in ring 1.

Safety is ensured by validating exception handlers when they are presented to Xen. The only required check is that the handler's code segment does not specify execution in ring 0. Since no guest OS can create such a segment,

[2] In hindsight, writing the value into a pre-agreed shared memory location rather than modifying the stack frame would have simplified the XP port.

it suffices to compare the specified segment selector to a small number of static values which are reserved by Xen. Apart from this, any other handler problems are fixed up during exception propagation—for example, if the handler's code segment is not present or if the handler is not paged into memory then an appropriate fault will be taken when Xen executes the `iret` instruction which returns to the handler. Xen detects these "double faults" by checking the faulting program counter value: if the address resides within the exception-virtualizing code then the offending guest OS is terminated.

Note that this "lazy" checking is safe even for the direct system-call handler: access faults will occur when the CPU attempts to directly jump to the guest OS handler. In this case the faulting address will be outside Xen (since Xen will never execute a guest OS system call) and so the fault is virtualized in the normal way. If propagation of the fault causes a further "double fault" then the guest OS is terminated as described above.

2.1.3 Device I/O

Rather than emulating existing hardware devices, as is typically done in fully-virtualized environments, Xen exposes a set of clean and simple device abstractions. This allows us to design an interface that is both efficient and satisfies our requirements for protection and isolation. To this end, I/O data is transferred to and from each domain via Xen, using shared-memory, asynchronous buffer-descriptor rings. These provide a high-performance communication mechanism for passing buffer information vertically through the system, while allowing Xen to efficiently perform validation checks (for example, checking that buffers are contained within a domain's memory reservation).

Linux subsection	# lines
Architecture-independent	78
Virtual network driver	484
Virtual block-device driver	1070
Xen-specific (non-driver)	1363
Total	**2995**
Portion of total x86 code base	**1.36%**

Table 1: The simplicity of porting commodity OSes to Xen.

Similar to hardware interrupts, Xen supports a lightweight event-delivery mechanism which is used for sending asynchronous notifications to a domain. These notifications are made by updating a bitmap of pending event types and, optionally, by calling an event handler specified by the guest OS. These callbacks can be 'held off' at the discretion of the guest OS—to avoid extra costs incurred by frequent wake-up notifications, for example.

2.2 The Cost of Porting an OS to Xen

Table 1 demonstrates the cost, in lines of code, of porting commodity operating systems to Xen's paravirtualized x86 environment.

The architecture-specific sections are effectively a port of the x86 code to our paravirtualized architecture. This involved rewriting routines which used privileged instructions, and removing a large amount of low-level system initialization code.

2.3 Control and Management

Throughout the design and implementation of Xen, a goal has been to separate policy from mechanism wherever possible. Although the hypervisor must be involved in data-path aspects (for example, scheduling the CPU between domains, filtering network packets before transmission, or enforcing access control

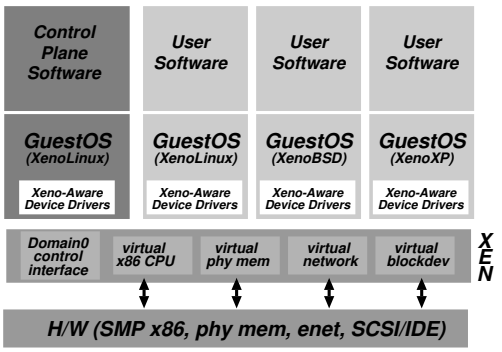


Figure 1: The structure of a machine running the Xen hypervisor, hosting a number of different guest operating systems, including *Domain0* running control software in a XenLinux environment.

when reading data blocks), there is no need for it to be involved in, or even aware of, higher level issues such as how the CPU is to be shared, or which kinds of packet each domain may transmit.

The resulting architecture is one in which the hypervisor itself provides only basic control operations. These are exported through an interface accessible from authorized domains; potentially complex policy decisions, such as admission control, are best performed by management software running over a guest OS rather than in privileged hypervisor code.

The overall system structure is illustrated in Figure 1. Note that a domain is created at boot time which is permitted to use the *control interface*. This initial domain, termed *Domain0*, is responsible for hosting the application-level management software. The control interface provides the ability to create and terminate other domains and to control their associated scheduling parameters, physical memory allocations and the access they are given to the machine's physical disks and network devices.

In addition to processor and memory resources, the control interface supports the creation and deletion of virtual network interfaces (VIFs) and block devices (VBDs). These virtual I/O devices have associated access-control information which determines which domains can access them, and with what restrictions (for example, a read-only VBD may be created, or a VIF may filter IP packets to prevent source-address spoofing or apply traffic shaping).

This control interface, together with profiling statistics on the current state of the system, is exported to a suite of application-level management software running in *Domain0*. This complement of administrative tools allows convenient management of the entire server: current tools can create and destroy domains, set network filters and routing rules, monitor per-domain network activity at packet and flow granularity, and create and delete virtual network interfaces and virtual block devices.

Snapshots of a domains' state may be captured and saved to disk, enabling rapid deployment of applications by bypassing the normal boot delay. Further, Xen supports *live migration* which enables running VMs to be moved dynamically between different Xen servers, with execution interrupted only for a few milliseconds. We are in the process of developing higher-level tools to further automate the application of administrative policy, for example, load balancing VMs among a cluster of Xen servers.

3 Detailed Design

In this section we introduce the design of the major subsystems that make up a Xen-based server. In each case we present both Xen and guest OS functionality for clarity of exposition. In this paper, we focus on the XenLinux guest OS; the *BSD and Windows XP ports use the Xen interface in a similar manner.

3.1 Control Transfer: Hypercalls and Events

Two mechanisms exist for control interactions between Xen and an overlying domain: synchronous calls from a domain to Xen may be made using a *hypercall*, while notifications are delivered to domains from Xen using an asynchronous event mechanism.

The hypercall interface allows domains to perform a synchronous software trap into the hypervisor to perform a privileged operation, analogous to the use of system calls in conventional operating systems. An example use of a hypercall is to request a set of page-table updates, in which Xen validates and applies a list of updates, returning control to the calling domain when this is completed.

Communication from Xen to a domain is provided through an asynchronous event mechanism, which replaces the usual delivery mechanisms for device interrupts and allows lightweight notification of important events such as domain-termination requests. Akin to traditional Unix signals, there are only a small number of events, each acting to flag a particular type of occurrence. For instance, events are used to indicate that new data has been received over the network, or that a virtual disk request has completed.

Pending events are stored in a per-domain bitmask which is updated by Xen before invoking an event-callback handler specified by the guest OS. The callback handler is responsible for resetting the set of pending events, and responding to the notifications in an appropriate manner. A domain may explicitly defer event handling by setting a Xen-readable software flag: this is analogous to disabling interrupts on a real processor.

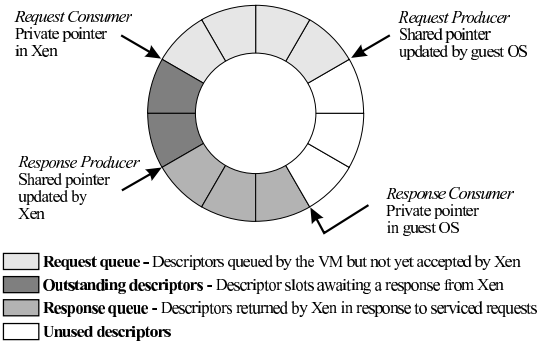


Figure 2: The structure of asynchronous I/O rings, which are used for data transfer between Xen and guest OSes.

3.2 Data Transfer: I/O Rings

The presence of a hypervisor means there is an additional protection domain between guest OSes and I/O devices, so it is crucial that a data transfer mechanism be provided that allows data to move vertically through the system with as little overhead as possible.

Two main factors have shaped the design of our I/O-transfer mechanism: resource management and event notification. For resource accountability, we attempt to minimize the work required to demultiplex data to a specific domain when an interrupt is received from a device—the overhead of managing buffers is carried out later where computation may be accounted to the appropriate domain. Similarly, memory committed to device I/O is provided by the relevant domains wherever possible to prevent the crosstalk inherent in shared buffer pools; I/O buffers are protected during data transfer by pinning the underlying page frames within Xen.

Figure 2 shows the structure of our I/O descriptor rings. A ring is a circular queue of descriptors allocated by a domain but accessible from within Xen. Descriptors do not directly contain I/O data; instead, I/O data buffers are al-

located out-of-band by the guest OS and indirectly referenced by I/O descriptors. Access to each ring is based around two pairs of producer-consumer pointers: domains place requests on a ring, advancing a request producer pointer, and Xen removes these requests for handling, advancing an associated request consumer pointer. Responses are placed back on the ring similarly, save with Xen as the producer and the guest OS as the consumer. There is no requirement that requests be processed in order: the guest OS associates a unique identifier with each request which is reproduced in the associated response. This allows Xen to unambiguously reorder I/O operations due to scheduling or priority considerations.

This structure is sufficiently generic to support a number of different device paradigms. For example, a set of 'requests' can provide buffers for network packet reception; subsequent 'responses' then signal the arrival of packets into these buffers. Reordering is useful when dealing with disk requests as it allows them to be scheduled within Xen for efficiency, and the use of descriptors with out-of-band buffers makes implementing zero-copy transfer easy.

We decouple the production of requests or responses from the notification of the other party: in the case of requests, a domain may enqueue multiple entries before invoking a hypercall to alert Xen; in the case of responses, a domain can defer delivery of a notification event by specifying a threshold number of responses. This allows each domain to trade-off latency and throughput requirements, similarly to the flow-aware interrupt dispatch in the ArseNIC Gigabit Ethernet interface [12].

3.3 Subsystem Virtualization

The control and data transfer mechanisms described are used in our virtualization of the various subsystems. In the following, we discuss how this virtualization is achieved for CPU, timers, memory, network and disk.

3.3.1 CPU scheduling

Xen currently schedules domains according to the Borrowed Virtual Time (BVT) scheduling algorithm [4]. We chose this particular algorithms since it is both work-conserving and has a special mechanism for low-latency wake-up (or *dispatch*) of a domain when it receives an event. Fast dispatch is particularly important to minimize the effect of virtualization on OS subsystems that are designed to run in a timely fashion; for example, TCP relies on the timely delivery of acknowledgments to correctly estimate network round-trip times. BVT provides low-latency dispatch by using virtual-time warping, a mechanism which temporarily violates 'ideal' fair sharing to favor recently-woken domains. However, other scheduling algorithms could be trivially implemented over our generic scheduler abstraction. Per-domain scheduling parameters can be adjusted by management software running in *Domain0*.

3.3.2 Time and timers

Xen provides guest OSes with notions of real time, virtual time and wall-clock time. Real time is expressed in nanoseconds passed since machine boot and is maintained to the accuracy of the processor's cycle counter and can be frequency-locked to an external time source (for example, via NTP). A domain's virtual time only advances while it is executing: this is typically used by the guest OS scheduler to ensure correct sharing of its timeslice between application processes. Finally, wall-clock time is specified as an offset to be added to the current real time. This allows the wall-clock time to be adjusted without affecting the forward

progress of real time.

Each guest OS can program a pair of alarm timers, one for real time and the other for virtual time. Guest OSes are expected to maintain internal timer queues and use the Xen-provided alarm timers to trigger the earliest timeout. Timeouts are delivered using Xen's event mechanism.

3.3.3 Virtual address translation

As with other subsystems, Xen attempts to virtualize memory access with as little overhead as possible. As discussed in Section 2.1.1, this goal is made somewhat more difficult by the x86 architecture's use of hardware page tables. The approach taken by VMware is to provide each guest OS with a virtual page table, not visible to the memory-management unit (MMU) [3]. The hypervisor is then responsible for trapping accesses to the virtual page table, validating updates, and propagating changes back and forth between it and the MMU-visible 'shadow' page table. This greatly increases the cost of certain guest OS operations, such as creating new virtual address spaces, and requires explicit propagation of hardware updates to 'accessed' and 'dirty' bits.

Although full virtualization forces the use of shadow page tables, to give the illusion of contiguous physical memory, Xen is not so constrained. Indeed, Xen need only be involved in page table *updates*, to prevent guest OSes from making unacceptable changes. Thus we avoid the overhead and additional complexity associated with the use of shadow page tables—the approach in Xen is to register guest OS page tables directly with the MMU, and restrict guest OSes to read-only access. Page table updates are passed to Xen via a hypercall; to ensure safety, requests are *validated* before being applied.

To aid validation, we associate a type and reference count with each machine page frame. A frame may have any one of the following mutually-exclusive types at any point in time: page directory (PD), page table (PT), local descriptor table (LDT), global descriptor table (GDT), or writable (RW). Note that a guest OS may always create readable mappings to its own page frames, regardless of their current types. A frame may only safely be retasked when its reference count is zero. This mechanism is used to maintain the invariants required for safety; for example, a domain cannot have a writable mapping to any part of a page table as this would require the frame concerned to simultaneously be of types PT and RW.

The type system is also used to track which frames have already been validated for use in page tables. To this end, guest OSes indicate when a frame is allocated for page-table use—this requires a one-off validation of every entry in the frame by Xen, after which its type is pinned to PD or PT as appropriate, until a subsequent unpin request from the guest OS. This is particularly useful when changing the page table base pointer, as it obviates the need to validate the new page table on every context switch. Note that a frame cannot be retasked until it is both unpinned and its reference count has reduced to zero – this prevents guest OSes from using unpin requests to circumvent the reference-counting mechanism.

3.3.4 Physical memory

The initial memory allocation, or *reservation*, for each domain is specified at the time of its creation; memory is thus statically partitioned between domains, providing strong isolation. A maximum-allowable reservation may also be specified: if memory pressure within a domain increases, it may then attempt to claim additional memory pages from Xen, up

to this reservation limit. Conversely, if a domain wishes to save resources, perhaps to avoid incurring unnecessary costs, it can reduce its memory reservation by releasing memory pages back to Xen.

XenLinux implements a *balloon driver* [16], which adjusts a domain's memory usage by passing memory pages back and forth between Xen and XenLinux's page allocator. Although we could modify Linux's memory-management routines directly, the balloon driver makes adjustments by using existing OS functions, thus simplifying the Linux porting effort. However, paravirtualization can be used to extend the capabilities of the balloon driver; for example, the out-of-memory handling mechanism in the guest OS can be modified to automatically alleviate memory pressure by requesting more memory from Xen.

Most operating systems assume that memory comprises at most a few large contiguous extents. Because Xen does not guarantee to allocate contiguous regions of memory, guest OSes will typically create for themselves the illusion of contiguous *physical memory*, even though their underlying allocation of *hardware memory* is sparse. Mapping from physical to hardware addresses is entirely the responsibility of the guest OS, which can simply maintain an array indexed by physical page frame number. Xen supports efficient hardware-to-physical mapping by providing a shared translation array that is directly readable by all domains – updates to this array are validated by Xen to ensure that the OS concerned owns the relevant hardware page frames.

Note that even if a guest OS chooses to ignore hardware addresses in most cases, it must use the translation tables when accessing its page tables (which necessarily use hardware addresses). Hardware addresses may also be exposed to limited parts of the OS's memory-management system to optimize memory access. For example, a guest OS might allocate particular hardware pages so as to optimize placement within a physically indexed cache [7], or map naturally aligned contiguous portions of hardware memory using superpages [10].

3.3.5 Network

Xen provides the abstraction of a virtual firewall-router (VFR), where each domain has one or more network interfaces (VIFs) logically attached to the VFR. A VIF looks somewhat like a modern network interface card: there are two I/O rings of buffer descriptors, one for transmit and one for receive. Each direction also has a list of associated rules of the form (<*pattern*>, <*action*>)—if the *pattern* matches then the associated *action* is applied.

Domain0 is responsible for inserting and removing rules. In typical cases, rules will be installed to prevent IP source address spoofing, and to ensure correct demultiplexing based on destination IP address and port. Rules may also be associated with hardware interfaces on the VFR. In particular, we may install rules to perform traditional firewalling functions such as preventing incoming connection attempts on insecure ports.

To transmit a packet, the guest OS simply enqueues a buffer descriptor onto the transmit ring. Xen copies the descriptor and, to ensure safety, then copies the packet header and executes any matching filter rules. The packet payload is not copied since we use scatter-gather DMA; however note that the relevant page frames must be pinned until transmission is complete. To ensure fairness, Xen implements a simple round-robin packet scheduler.

To efficiently implement packet reception, we

require the guest OS to exchange an unused page frame for each packet it receives; this avoids the need to copy the packet between Xen and the guest OS, although it requires that page-aligned receive buffers be queued at the network interface. When a packet is received, Xen immediately checks the set of receive rules to determine the destination VIF, and exchanges the packet buffer for a page frame on the relevant receive ring. If no frame is available, the packet is dropped.

3.3.6 Disk

Only *Domain0* has direct unchecked access to physical (IDE and SCSI) disks. All other domains access persistent storage through the abstraction of virtual block devices (VBDs), which are created and configured by management software running within *Domain0*. Allowing *Domain0* to manage the VBDs keeps the mechanisms within Xen very simple and avoids more intricate solutions such as the UDFs used by the Exokernel [6].

A VBD comprises a list of extents with associated ownership and access control information, and is accessed via the I/O ring mechanism. A typical guest OS disk scheduling algorithm will reorder requests prior to enqueuing them on the ring in an attempt to reduce response time, and to apply differentiated service (for example, it may choose to aggressively schedule synchronous metadata requests at the expense of speculative readahead requests). However, because Xen has more complete knowledge of the actual disk layout, we also support reordering within Xen, and so responses may be returned out of order. A VBD thus appears to the guest OS somewhat like a SCSI disk.

A translation table is maintained within the hypervisor for each VBD; the entries within this table are installed and managed by *Domain0* via a privileged control interface. On receiving a disk request, Xen inspects the VBD identifier and offset and produces the corresponding sector address and physical device. Permission checks also take place at this time. Zero-copy data transfer takes place using DMA between the disk and pinned memory pages in the requesting domain.

Xen services *batches* of requests from competing domains in a simple round-robin fashion; these are then passed to a standard elevator scheduler before reaching the disk hardware. Domains may explicitly pass down *reorder barriers* to prevent reordering when this is necessary to maintain higher level semantics (e.g. when using a write-ahead log). The low-level scheduling gives us good throughput, while the batching of requests provides reasonably fair access. Future work will investigate providing more predictable isolation and differentiated service, perhaps using existing techniques and schedulers [15].

4 Evaluation

In this section we present a subset of our evaluation of Xen against a number of alternative virtualization techniques. A more complete evaluation, as well as detailed configuration and benchmark specs, can be found in [1] For these measurements, we used our 2.4.21-based XenLinux port as, at the time of this writing, the 2.6-port was not stable enough for a full battery of tests.

There are a number of preexisting solutions for running multiple copies of Linux on the same machine. VMware offers several commercial products that provide virtual x86 machines on which unmodified copies of Linux may be booted. The most commonly used version is VMware Workstation, which consists

of a set of privileged kernel extensions to a 'host' operating system. Both Windows and Linux hosts are supported. VMware also offer an enhanced product called ESX Server which replaces the host OS with a dedicated kernel. By doing so, it gains some performance benefit over the workstation product. We have subjected ESX Server to the benchmark suites described below, but sadly are prevented from reporting quantitative results due to the terms of the product's End User License Agreement. Instead we present results from VMware Workstation 3.2, running on top of a Linux host OS, as it is the most recent VMware product without that benchmark publication restriction. ESX Server takes advantage of its native architecture to equal or outperform VMware Workstation and its hosted architecture. While Xen of course requires guest OSes to be ported, it takes advantage of paravirtualization to noticeably outperform ESX Server.

We also present results for User-mode Linux (UML), an increasingly popular platform for virtual hosting. UML is a port of Linux to run as a user-space process on a Linux host. Like XenLinux, the changes required are restricted to the architecture dependent code base. However, the UML code bears little similarity to the native x86 port due to the very different nature of the execution environments. Although UML can run on an unmodified Linux host, we present results for the 'Single Kernel Address Space' (skas3) variant that exploits patches to the host OS to improve performance.

We also investigated three other virtualization techniques for running ported versions of Linux on the same x86 machine. Connectix's Virtual PC and forthcoming Virtual Server products (now acquired by Microsoft) are similar in design to VMware's, providing full x86 virtualization. Since all versions of Virtual PC have benchmarking restrictions in their license agreements we did not subject them to closer analysis. UMLinux is similar in concept to UML but is a different code base and has yet to achieve the same level of performance, so we omit the results. Work to improve the performance of UMLinux through host OS modifications is ongoing [8]. Although Plex86 was originally a general purpose x86 VMM, it has now been retargeted to support just Linux guest OSes. The guest OS must be specially compiled to run on Plex86, but the source changes from native x86 are trivial. The performance of Plex86 is currently well below the other techniques.

4.1 Relative Performance

The first cluster of bars in Figure 3 represents a relatively easy scenario for the VMMs. The SPEC CPU suite contains a series of long-running computationally-intensive applications intended to measure the performance of a system's processor, memory system, and compiler quality. The suite performs little I/O and has little interaction with the OS. With almost all CPU time spent executing in user-space code, all three VMMs exhibit low overhead.

The next set of bars show the total elapsed time taken to build a default configuration of the Linux 2.4.21 kernel on a local ext3 file system with gcc 2.96. Native Linux spends about 7% of the CPU time in the OS, mainly performing file I/O, scheduling and memory management. In the case of the VMMs, this 'system time' is expanded to a greater or lesser degree: whereas Xen incurs a mere 3% overhead, the other VMMs experience a more significant slowdown.

Two experiments were performed using the PostgreSQL 7.1.3 database, exercised by the Open Source Database Benchmark suite (OSDB) in its default configuration. We present results for the multi-user Information

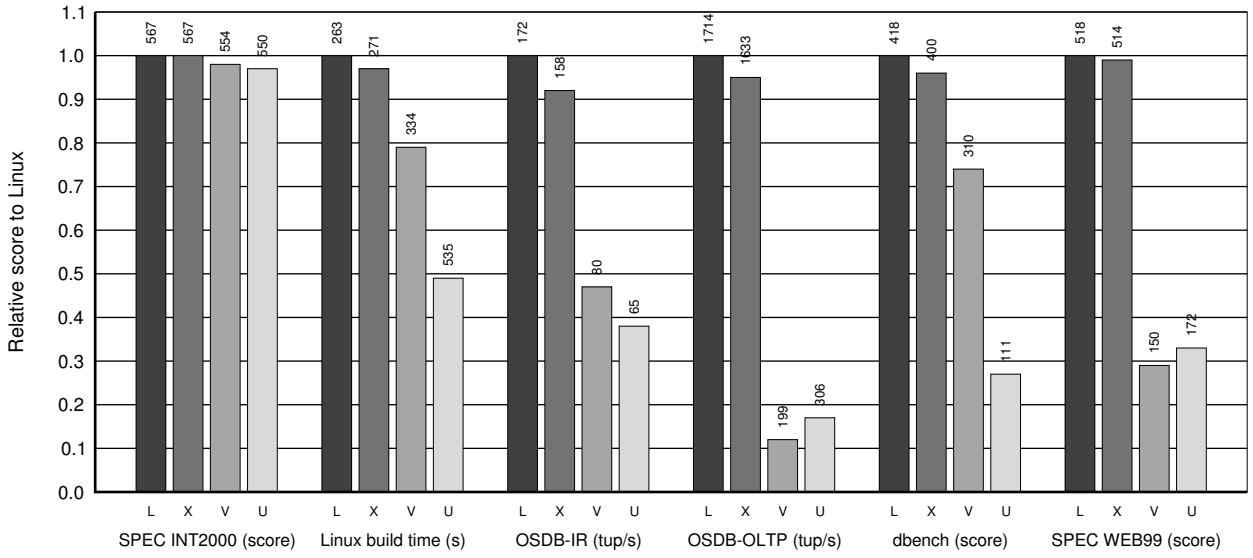


Figure 3: Relative performance of native Linux (L), XenLinux (X), VMware workstation 3.2 (V) and User-Mode Linux (U).

Retrieval (IR) and On-Line Transaction Processing (OLTP) workloads, both measured in tuples per second. PostgreSQL places considerable load on the operating system, and this is reflected in the substantial virtualization overheads experienced by VMware and UML. In particular, the OLTP benchmark requires many synchronous disk operations, resulting in many protection domain transitions.

The `dbench` program is a file system benchmark derived from the industry-standard 'NetBench'. It emulates the load placed on a file server by Windows 95 clients. Here, we examine the throughput experienced by a single client performing around 90,000 file system operations.

SPEC WEB99 is a complex application-level benchmark for evaluating web servers and the systems that host them. The benchmark is CPU-bound, and a significant proportion of the time is spent within the guest OS kernel, performing network stack processing, file system operations, and scheduling between the many `httpd` processes that Apache needs to handle the offered load. XenLinux fares well, achieving within 1% of native Linux performance. VMware and UML both struggle, supporting less than a third of the number of clients of the native Linux system.

4.2 Operating System Benchmarks

To more precisely measure the areas of overhead within Xen and the other VMMs, we performed a number of smaller experiments targeting particular subsystems. We examined the overhead of virtualization as measured by McVoy's *lmbench* program [9]. The OS performance subset of the lmbench suite consist of 37 microbenchmarks.

In 24 of the 37 microbenchmarks, XenLinux performs similarly to native Linux, tracking the Linux kernel performance closely. In Tables 2 to 4 we show results which exhibit interesting performance variations among the test systems; particularly large penalties for Xen are shown in bold face.

In the process microbenchmarks (Table 2), Xen

Config	null call	null I/O	open close	slct TCP	sig inst	sig hndl	fork proc	exec proc	sh proc
Linux	0.45	0.50	1.92	5.70	0.68	2.49	110	530	4k0
Xen	0.46	0.50	1.88	5.69	0.69	1.75	**198**	**768**	**4k8**
VMW	0.73	0.83	2.99	11.1	1.02	4.63	874	2k3	10k
UML	24.7	25.1	62.8	39.9	26.0	46.0	21k	33k	58k

Table 2: `lmbench`: Processes - times in μs

Config	2p 0K	2p 16K	2p 64K	8p 16K	8p 64K	16p 16K	16p 64K
Linux	0.77	0.91	1.06	1.03	24.3	3.61	37.6
Xen	**1.97**	**2.22**	**2.67**	**3.07**	**28.7**	**7.08**	39.4
VMW	18.1	17.6	21.3	22.4	51.6	41.7	72.2
UML	15.5	14.6	14.4	16.3	36.8	23.6	52.0

Table 3: `lmbench`: Context switching times in μs

Config	0K File create	0K File delete	10K File create	10K File delete	Mmap lat	Prot fault	Page fault
Linux	32.1	6.08	66.0	12.5	68.0	1.06	1.42
Xen	32.5	5.86	68.2	13.6	**139**	1.40	**2.73**
VMW	35.3	9.3	85.6	21.4	620	7.53	12.4
UML	130	65.7	250	113	1k4	21.8	26.3

Table 4: `lmbench`: File & VM system latencies in μs

exhibits slower *fork*, *exec*, and *sh* performance than native Linux. This is expected, since these operations require large numbers of page table updates which must all be verified by Xen. However, the paravirtualization approach allows XenLinux to batch update requests. Creating new page tables presents an ideal case: because there is no reason to commit pending updates sooner, XenLinux can amortize each hypercall across 2048 updates (the maximum size of its batch buffer). Hence each update hypercall constructs 8MB of address space.

Table 3 shows context switch times between different numbers of processes with different working set sizes. Xen incurs an extra overhead between 1μs and 3μs, as it executes a hypercall to change the page table base. However, context switch results for larger working set sizes (perhaps more representative of real applications) show that the overhead is small compared with cache effects. Unusually, VMware Workstation is inferior to UML on these microbenchmarks; however, this is one area where enhancements in ESX Server are able to reduce the overhead.

The *mmap latency* and *page fault latency* results shown in Table 4 are interesting since they require two transitions into Xen per page: one to take the hardware fault and pass the details to the guest OS, and a second to install the updated page table entry on the guest OS's behalf. Despite this, the overhead is relatively modest.

One small anomaly in Table 2 is that XenLinux has lower signal-handling latency than native Linux. This benchmark does not require any calls into Xen at all, and the 0.75μs (30%) speedup is presumably due to a fortuitous cache alignment in XenLinux, hence underlining the dangers of taking microbenchmarks too seriously.

4.3 Additional Benchmarks

We have also conducted comprehensive experiments that: evaluate the overhead of virtualizing the network; compare the performance of running multiple applications in their own guest OS against running them on the same native operating system; demonstrate performance isolation provided by Xen; and examine Xen's ability to scale to its target of 100 domains. All of the experiments showed promising results and details have been separately published [1].

5 Conclusion

We have presented the Xen hypervisor which partitions the resources of a computer between domains running guest operating systems. Our

paravirtualizing design places a particular emphasis on protection, performance and resource management. We have also described and evaluated XenLinux, a fully-featured port of the Linux kernel that runs over Xen.

Xen and the 2.4-based XenLinux are sufficiently stable to be useful to a wide audience. Indeed, some web hosting providers are already selling Xen-based virtual servers. Sources, documentation, and a demo ISO can be found on our project page[3].

Although the 2.4-based XenLinux was the basis of our performance evaluation, a 2.6-based port is well underway. In this port, much care is been given to minimizing and isolating the necessary changes to the Linux kernel and measuring the changes against benchmark results. As paravirtualization techniques become more prevalent, kernel changes would ideally be part of the main tree. We have experimented with various source structures including a separate architecture, *a la* UML, a subarchitecture, and a CONFIG option. We eagerly solicit input and discussion from the kernel developers to guide our approach. We also have considered transparent paravirtualization [2] techniques to allow a single distro image to adapt dynamically between a VMM-based configuration and bare metal.

As well as further guest OS ports, Xen itself is being ported to other architectures. An x86_64 port is well underway, and we are keen to see Xen ported to RISC-style architectures (such as PPC) where virtual memory virtualization will be much easier due to the software-managed TLB.

Much new functionality has been added since the first public availability of Xen last October. Of particular note are a completely revamped I/O subsystem capable of directly utilizing Linux driver source, suspend/resume and live migration features, much improved console access, etc. Though final implementation, testing, and documentation was not complete at the deadline for this paper, we hope to describe these in more detail at the symposium and in future publications.

As always, there are more tasks to do than there are resources to do them. We would like to grow Xen into the premier open source virtualization solution, with breadth and features that rival proprietary commercial products.

We enthusiastically welcome the help and contributions of the Linux community.

[3] `http://www.cl.cam.ac.uk/netos/xen`

Acknowledgments

Xen has been a big team effort. In particular, we would like to thank Zachary Amsden, Andrew Chung, Richard Coles, Boris Dragovich, Evangelos Kotsovinos, Tim Harris, Alex Ho, Kip Macy, Rolf Neugebauer, Bin Ren, Russ Ross, James Scott, Steven Smith, Andrew Warfield, Mark Williamson, and Mike Wray. Work on Xen has been supported by a UK EPSRC grant, Intel Research, Microsoft Research, and Hewlett-Packard Labs.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM SIGOPS Symposium on Operating Systems Principles*, volume 37(5) of *ACM Operating Systems Review*, pages 164–177, Bolton Landing, NY, USA, Dec. 2003.

[2] D. Magenheimer and T. Christian. vBlades: Optimized Paravirtualization for the Itanium Processor Family. In *Proceedings of the*

USENIX 3rd Virtual Machine Research and Technology Symposium, May 2004.

[3] S. Devine, E. Bugnion, and M. Rosenblum. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. *US Patent*, 6397242, Oct. 1998.

[4] K. J. Duda and D. R. Cheriton. Borrowed-Virtual-Time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *Proceedings of the 17th ACM SIGOPS Symposium on Operating Systems Principles*, volume 33(5) of *ACM Operating Systems Review*, pages 261–276, Kiawah Island Resort, SC, USA, Dec. 1999.

[5] D. Engler, S. K. Gupta, and F. Kaashoek. AVM: Application-level virtual memory. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems*, pages 72–77, May 1995.

[6] M. F. Kaashoek, D. R. Engler, G. R. Granger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on Exokernel systems. In *Proceedings of the 16th ACM SIGOPS Symposium on Operating Systems Principles*, volume 31(5) of *ACM Operating Systems Review*, pages 52–65, Oct. 1997.

[7] R. Kessler and M. Hill. Page placement algorithms for large real-indexed caches. *ACM Transaction on Computer Systems*, 10(4):338–359, Nov. 1992.

[8] S. T. King, G. W. Dunlap, and P. M. Chen. Operating System Support for Virtual Machines. In *Proceedings of the 2003 Annual USENIX Technical Conference*, Jun 2003.

[9] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the USENIX Annual Technical Conference*, pages 279–294, Berkeley, Jan. 1996. Usenix Association.

[10] J. Navarro, S. Iyer, P. Druschel, and A. Cox. Practical, transparent operating system support for superpages. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, ACM Operating Systems Review, Winter 2002 Special Issue, pages 89–104, Boston, MA, USA, Dec. 2002.

[11] L. Peterson, D. Culler, T. Anderson, and T. Roscoe. A blueprint for introducing disruptive technology into the internet. In *Proceedings of the 1st Workshop on Hot Topics in Networks (HotNets-I)*, Princeton, NJ, USA, Oct. 2002.

[12] I. Pratt and K. Fraser. Arsenic: A user-accessible gigabit ethernet interface. In *Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM-01)*, pages 67–76, Los Alamitos, CA, USA, Apr. 22–26 2001. IEEE Computer Society.

[13] J. S. Robin and C. E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *Proceedings of the 9th USENIX Security Symposium, Denver, CO, USA*, pages 129–144, Aug. 2000.

[14] L. Seawright and R. MacKinnon. VM/370 – a study of multiplicity and usefulness. *IBM Systems Journal*, pages 4–17, 1979.

[15] P. Shenoy and H. Vin. Cello: A Disk Scheduling Framework for Next-generation Operating Systems. In *Proceedings of ACM SIGMETRICS'98, the International Conference on Measurement and Modeling of Computer Systems*, pages 44–55, June 1998.

[16] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, ACM Operating Systems Review, Winter 2002 Special Issue, pages 181–194, Boston, MA, USA, Dec. 2002.

[17] A. Whitaker, M. Shaw, and S. D. Gribble. Denali: Lightweight Virtual Machines for Distributed and Networked Applications. Technical Report 02-02-01, University of Washington, 2002.

TIPC: Providing Communication for Linux Clusters

Jon Paul Maloy
Ericsson Research, Montreal
jon.maloy@ericsson.com

Abstract

Transparent Inter Process Communication (TIPC) is a protocol specially designed for efficient intra cluster communication, leveraging the particular conditions present within clusters of loosely coupled nodes.

TIPC provides a powerful infrastructure for designing distributed, site-independent, scalable, highly- available and high-performing applications, as well as a good support for cluster, network and software management functionality. In this paper, we will discuss the motives for developing TIPC and describe its architecture. Then, we will present the most important features of TIPC, such as its functional, location transparent, addressing scheme, lightweight reactive connections, reliable multicast, signalling link protocol, topology subscription services and more. We will also discuss the various design decisions that influenced the implementation of these features. We conclude by describing the current implementation status and our planned roadmap for TIPC.

1 Introduction

For the last six years, telecom equipment vendor Ericsson has been developing and deploying a tailor-made reliable communication protocol, TIPC,for their cluster-based products. This protocol has recently undergone a significant redesign, and is now available as a portable source code package of about 12,500 lines of C code. The code implements a Linux kernel driver, a design that has made it possible to improve performance (35% faster than TCP) and minimize code footprint.

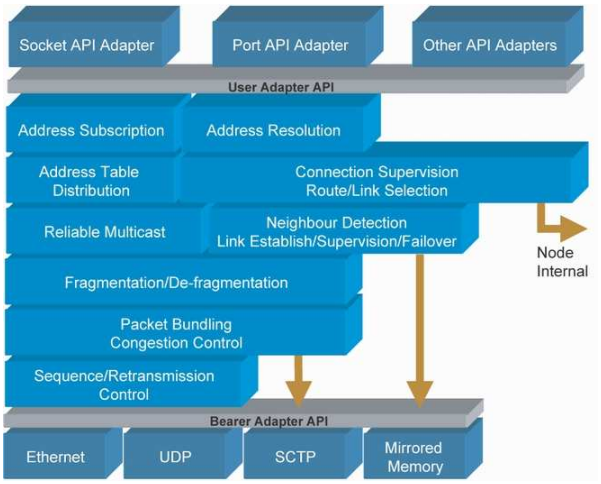


Figure 1: *Functional View of TIPC*

The current version is available under a dual BSD/GPL license from [1]. TIPC is supported on Linux 2.4 and 2.6; and several proprietary portations to other OS'es (OSE, True64, VxWorks) also exist.

TIPC offers an interesting combination of features, some of them quite unique, to achieve the overall goal: to make the cluster act as one single computer from a communication viewpoint, while helping applications to keep track of and adapt to topology changes. Figure 1 illustrates a functional view of TIPC.

2 Motivation

There are no standard protocols available today that fully satisfy the special needs of application programs working within highly available, dynamic cluster environments. Clusters may grow or shrink by orders of magnitude; member nodes may crash and restart, routers may fail and be replaced, services may be moved around due to load balancing considerations, etc. All this must be possibe to handle without significant disturbances of the service offered by the cluster. In order to minimize the effort by the application programmers to deal with such situations, and to maximize the chance that they are handled in a correct and optimal way, the cluster internal communication service should provide special support, helping the applications to adapt to changes in the cluster. It should also, when possible, leverage the special conditions present within cluster environments to present a more efficient and fault-tolerant communication service than more general protocols are capable of.

2.1 Existing Protocols

TCP [2] has the advantage of being ubiquitous, proven, and wellknown by most programmers. Its most significant shortcomings in a real-time cluster environment are the following:

- TCP lacks any notion of functional addressing and addressing transparency. Mechanisms exist (DNS, CORBA Naming Service) for transparent and dynamic lookup of the correct IP-adress of a destination, but those are in general too static and too inefficient to be useful in a dynamic, real-time environment.

- Performance is not as good as it could be, especially for intra-node communication and for short messages in general. For intra-node communication, other more efficient mechanisms are available, at least on Unix, but then the location of the destination process has to be assumed, and can not be changed. It is desirable to have a protocol working efficiently for both intra-node and inter-node messaging, without forcing the user to distinguish between these cases in his code.

- The heavy connection setup/shutdown scheme of TCP is a disadvantage in a dynamic environment. The minimum number of packets exchanged for even the shortest TCP transaction is nine (SYN, SYNACK, etc.), while with TIPC this can be reduced to two, or even to one if connectionless mode is used.

- The connection-oriented nature of TCP makes it impossible to support true multicast.

Stream Control Transmission Protocol (SCTP) [3] is message oriented; it provides some level of user connection supervision, message bundling, loss-free changeover, and a few more features that may make it more suitable than TCP as an intra-cluster protocol. Otherwise, it has all the drawbacks of TCP already listed above.

Apart from these weaknesses, neither TCP nor SCTP provide any topology information/subscription service, something that has proven very useful both for applications and for management functionality operating within cluster environments.

Both TCP and SCTP are general purpose protocols, in the sense that they can be used safely over the Internet as well as within a closed cluster. This virtual advantage is also their major weakness: they require funtionality and header space to deal with situations that will

never happen, or only infrequently, within clusters.

2.2 Assumptions

The TIPC design is based on the following assumptions, empirically known to be valid within most clusters.

- Most messages cross only one direct hop.
- Transfer time for most messages is short.
- Most messages are passed over intra-cluster connections.
- Packet loss rate is normally low; retransmission is infrequent.
- Available bandwidth and memory volume is normally high.
- For all relevant bearers packets are checksummed by hardware.
- The number of inter-communicating nodes is relatively static and limited at any moment in time.
- Security is a less crucial issue in closed clusters than on the Internet.

Because of the above one can use a simple, traffic-driven, fixed-size sliding window protocol located at the signalling link level, rather than a timer-driven transport level protocol. This in turn gives a lot of other advantages, such as earlier release of transmission buffers, earlier packet loss detection and retransmission, and earlier detection of node unavailability, only to mention some. Of course, situations with long transfer delays, high loss rates, long messages, security issues, etc. must be dealt with as well, but rather from the viewpoint of being exceptions than as the general rule.

3 Five-Layer Network Topology

From a TIPC viewpoint the network is organized in a five-layer structure (Figure 2).

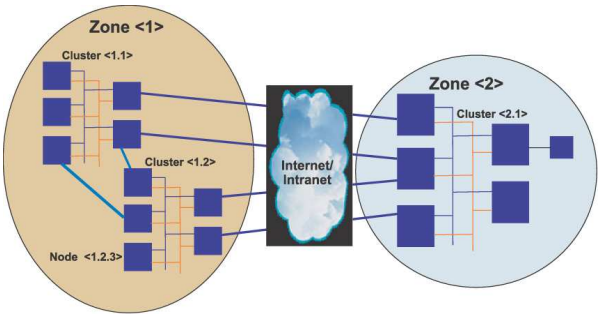


Figure 2: *TIPC Network Topology*

The top level is the *TIPC network*. This is the ensemble of all computers (nodes) interconnected via TIPC, i.e., the domain where any node can reach any other node by using a TIPC network address. A TIPC network is distringuished from other networks by its *network identity*, a 32-bit value that is known by all nodes.

The next level in the hierarchy is an entity called *zone*. This "cluster of clusters" is the maximum scope of location transparency within a network, i.e., the domain where any process can reach any other process by using a functional address rather than a network addresses.

The third level is what we call the *cluster*. This is a group of nodes interconnected all-to-all via one or two TIPC links.

The fourth level is the individual *system node*, or just *node*. There may be up to 2047 system nodes in a cluster.

The lowest level is the *slave node*. Slave nodes provide the same properties regarding location transparency and availability as system nodes, but they don't need full physical connectivity

to the rest of the cluster. One link to one system node is sufficient, although there may be more for redundancy reasons.

All entities within a TIPC network are accessed using a TIPC network address, a 32-bit value subdivided into a zone, cluster, and node field. This address is internally mapped to the address type for the communication media actually used, e.g., an Ethernet address or an IP-address/port number tuplet.

4 Location-Transparent Functional Addressing

To present a cluster as one computer, the addressing scheme used must hide the physical location of a requested service to its users. To achieve this, TIPC provides a functional address type, called *port name*, to be used both for connectionless messaging and connection setup calls. Binding a socket to a port name corresponds to binding it to a port number in other protocols, except that the port name is unique and has validity for the whole cluster, not only the local node. A caller wanting to set up a connection needs only to specify this address, and the TIPC internal translation service ensures that the request ends up in the right socket, on the right node.

A port name consists of two 32-bit fields. The first field is called the *name type* and typically identifies a certain service type or function. The second field is the *name instance* and is used as a key for accessing a certain instance of the requested service. This address structure gives excellent support for both service partitioning and service load sharing.

Further support for service partitioning is provided by an address type called *port name sequence*. This is a three-integer structure defining a range of port names, i.e., a name type plus the lower and upper boundary of the instance range. By allowing a socket to bind to a sequence, instead of just an individual port name, it is possible to partition a service's scope of responsibility into sub-ranges, without having to create a vast number of sockets to do so.

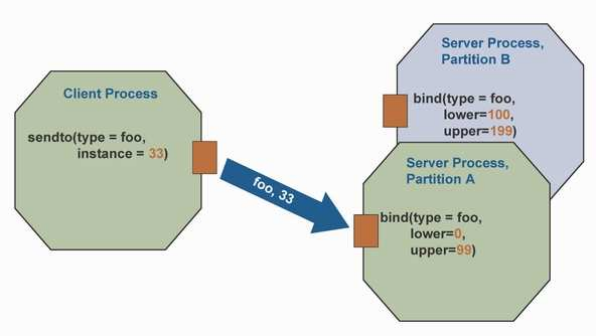


Figure 3: *Functional Addressing*

This addressing scheme is illustrated by the example in Figure 3. Two processes, partition A and partition B of the service *foo*, bind their sockets to the port name sequences *[foo,0,99]* and *[foo,100,199]* respectively (*foo* represents the name type part of the sequence). A process wanting to send a message to instance number 33 of that service, uses the port name *[foo,33]* as destination address. The TIPC name translation function will find that the indicated instance is within the range bound to by partition A, and directs the message to A's socket.

There are very few limitations on how name sequences may be bound to sockets. One may bind many different sequences, or many instances of the same sequence, to the same socket, to different sockets on the same node, or to different sockets anywhere in the cluster.

4.1 Binding Scope

Although complete location transparency is desirable and sufficient for most applications,

there must be ways to control this property for those who may need to do so. Hence, when binding a name sequence to a socket, it's possible to qualify it with a *binding scope* parameter, indicating how far the knowledge of the binding should be distributed in the network. The typical behavior is to spread it to the nodes in the binder's cluster, but it is possible to extend the scope to the whole zone, or limit it to the local node.

4.2 Lookup Domain

Similarly, a client may indicate a *lookup domain* for a message or connection setup request. This is a TIPC network address not only indicating where the lookup, i.e., the translation from a port name to socket address, should first be done, but implicitly even the lookup algorithm to be used.

Two such algorithms are available: 1) *round-robin* lookup is used when the lookup domain is non-zero and there is more than one matching server. Internally TIPC selects the server from a circular list; which root entry is stepped between each lookup. 2) *Closest-first* lookup is used when the lookup domain is zero. Here, the translation is always performed at the client's node and will first look for a matching socket on the local node. If none such is found, the algorithm will successively look for matches elsewhere in the cluster and finally in the whole zone.

5 Reliable Functional Multicast

Functional addressing is also used to provide a *reliable multicast* service. If the sender of a message indicates a port name sequence instead of a port name as destination, a replica of the message is sent to all sockets bound to a name sequence fully or partially overlapping with that sequence (Figure 4).

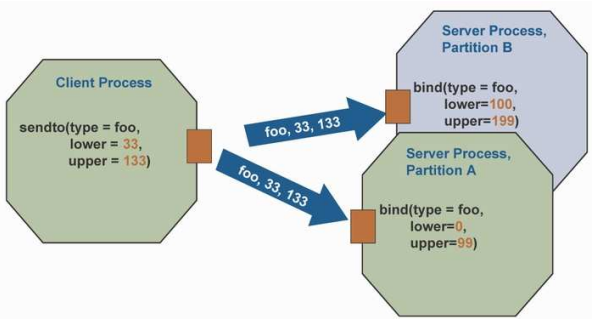


Figure 4: *Reliable Functional Multicast*

Only one replica of the message is sent to each identified target port, even if it is bound to more than one matching sequence. Whenever possible, this function will make use of the multicast/broadcast properties of the carrying media. In such cases, reliability is ensured by a special *reliable cluster broadcast* [4][5] protocol implemented internally in TIPC.

6 Name Translation Table

Translation from port name to socket addresses is performed transparently and on-the-fly via an internal translation table, replicated on each node. When a socket is bound to a port name sequence, a corresponding table entry is distributed to all nodes within the binding scope, i.e., the local cluster in most cases.

7 Topology Services

TIPC also provides a mechanism for inquiring or subscribing for the availability of port names or ranges of port names.

7.1 Functional Topology Service

This *functional topology service* is built on and uses the contents of the local instance of the name translation table.

To access this service, a user makes a blocking or nonblocking request to TIPC, asking it to indicate when a name sequence within the requested range is bound to or unbound. The request is associated with a timer, giving the duration of the subscription. A timer value of zero causes the call to return or issue a subscription event immediately, making it a pure inquiry, while a value of -1 makes it stay forever, indicating every change pertaining to the requested name sequence.

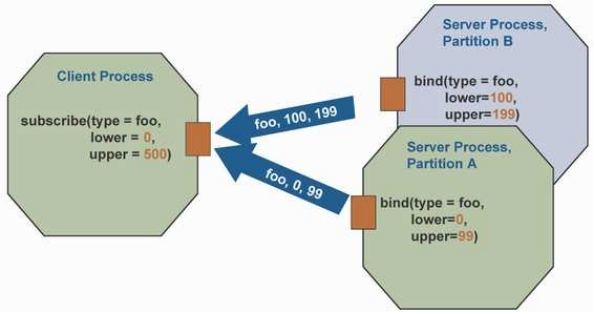


Figure 5: *Functional Topology Subscription*

Figure 5 illustrates this service: If the client process (see also example in Figure 3) wants to syncronize itself with the servers before starting any communication he issues a *subscribe()* call to TIPC, telling it to indicate when a server overlapping with the subscribed range becomes available. Since both ranges of partition A and B are within the given range *[foo,0,500]*, the client will receive two such indications, informing about the exact range of the new bindings. If there is only a partial overlap, e.g., if the client should subscribe for *[foo,0,150]* instead, he will only be informed about the actual overlap, i.e., *[foo,100,150]* for partition B.

7.2 Physical Topology Service

The physical network topology may be considered a special case of the functional topology, and can be kept track of in the same way. Hence, to subscribe for the availability/disappearance of a specific node, a group of nodes, or a whole cluster, the user specifies a dedicated port name sequence, representing this function and the range of nodes he wants to subscribe for. A special name type (zero) is used for this purpose, while the lower and upper boundaries are represented by TIPC network addresses—as described earlier, those are in reality 32-bit numbers.

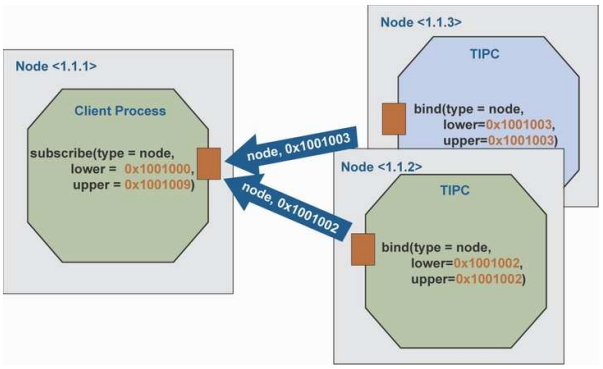


Figure 6: *Physical Topology Subscription*

In the example in Figure 6, the client process subscribes for the node range [0,9] within zone number 1, cluster number 1. Hence, when node <1.1.3> (i.e., zone 1, cluster 1, node 3) establishes a link to the client's node, the client will immediately be informed about this. For this particular service, TIPC will by itself bind/unbind the corresponding port name as soon as it discovers or loses contact with a node.

8 Lightweight Connections

The number of active user connections within a big cluster may be extremely large, and each cluster node must be able to establish and terminate thousands of such connections per second.

8.1 Simple Setup/Shutdown

To deal with this dynamism, TIPC connections are made very lighweight, in reality leaving the user to decide the setup/shutdown sequence. The protocol as such does not specify how connections are established and shut down, so an application caring about performance is free to use its own scheme, e.g., only exchanging payload-carrying messages.

For convenience an alternative, TCP-style connection type is also provided on Linux, with exchange of hidden protocol messages and stream-oriented data exchange.

8.2 Reactive Connections

TIPC connections are highly reactive and give the users almost immediate failure indication if anything should happen at the endpoints, or to the media between them. This is due to a connection supervision and abortion mechanism, which takes advantage of the properties of the local operating system to detect process crashes, or the status of the concerned links to detect node crashes or carrier failure. When any of this happens, a special *connection shutdown* message is spontaneously generated by TIPC and sent to the affected endpoint or endpoints, along with an appropriate error code. This error code delivered up to the user in the failure indication. In some cases, when the failure is detected due to inability to deliver a message, the original message is returned to the sender along with the error code, to further enable him to analyze the situation and take proper action. This *message rejection* mechanism is also used when connection-less messages are undeliverable.

9 Link-Level Protocol

Assuming that most clusters are relatively static in size, some of the tasks normally performed at the transport protocol level have been moved down to the signalling link level.

9.1 Link-level Retransmission

Implementing the retransmission protocol at this level has several advantages. First, it gives better resource utilization since all packets, connectionless and connection oriented, are funneled into one single packet sequence per node pair. Each packet can hence carry the acknowledge of many received packets, regardless of their origin, and we need not keep transmission buffers longer than strictly necessary. Second, packet losses can be detected and restransmission performed earlier than would otherwise be the case. Third, packet delivery and sequentiality guaranteed at the link level eliminates any need for per packet timers at the transport level—a background timer per link is sufficient to ensure those properties. As a result, we obtain a packet flow that is both smoother and more “traffic driven” than with corresponding transport level protocols, which often rely on timers to keep traffic running.

9.2 Link-level Node Supervision

Internode connectivity is also ensured at the link level. First, a background timer for each link endpoint supervises the traffic flow on the link and initiates a probing procedure if the peer is silent too long. Second, if a link is found to have failed after probing, there is a mechanism to steer its traffic over to the remaining link to the same node, if there is one.

9.3 Link-level Redundancy and Load Sharing

In fact, having two links and two carriers between each node pair is considered the normal configuration when using TIPC, as it eliminates any single point of failure in the communication service. The failover procedure used on such occations is completely transparent to the users, and complies to the same QOS as is guaranteed by each individual link: no message losses, no duplicates, and in-sequence delivery. The relationship between dual links is configurable; while full load sharing is the default behavior, an active-standby scheme is also supported.

Detection time for a failed link, and consequently for a crashed node, is configurable and is by default set to 1500 ms in the current implementation.

10 Automatic Neighbour Detection

Signalling links may be configured manually, but this is a tedious task if the size of a cluster runs up to dozens or even hundreds of nodes. Therefore, TIPC uses a designated neighbour detection protocol to establish links between nodes. Within a cluster this protocol is very simple. Each starting node uses the multicast or broadcast capability of the carrying media to tell about its existence, and expects a corresponding unicast response from all nodes recognizing it as part of the cluster.

Between clusters, both multicast and a unicast "pilot" link may be used, and results in a link pattern where each node in one cluster has links to a configurable (default two) number of nodes in the other cluster.

11 Performance

The performance figures we have are from the Linux-2.4 version of TIPC. We have not yet been able to do code optimizations and corresponding measurements on the Linux-2.6 version.

Performance was measured by letting a set of 16 process pairs on two nodes exchange messages in a ping-pong like manner at full speed. This ensures that the CPUs always runs at 100% load, and we can assume that almost all execution time is spent on transferring TIPC messages. We measured the time it took to exchange a message of a certain size 16 X 10 000 000 times, and divided the obtained value with number of messages. The result gives pure CPU execution time per message, automatically excluding latency times on the network and in the OS's sceduling queues, which is anyway the same for all protocols. For comparison, a similar measurement sequence was done for TCP, on the same OS and hardware.

Table 1 shows measured execution time for transferring a message process-to-process between two 750 Mhz Pentium III based nodes. The communication media used was two parallel 100 Mb Fast Ethernet switches.

Msg Size [bytes]	TIPC [μs]	TCP [μs]
64	25	38
256	29	42
1024	44	52
4096	176	178
16384	704	716
65408	3200	2800

Table 1: *Inter Node Execution time (send + receive) for TIPC and TCP messages*

The overall result shows that TIPC is around 35% faster than TCP for inter-node messages

smaller than Ethernet MTU, while performance is about the same for larger messages. A similar measurement, where all processes were kept on the same node, showed that TIPC is about four times faster (6 μs vs 25 μs) than TCP for 64 byte intra-node messages; the difference decreasing linearly with message size. At 64 Kbyte messages performance was even here almost the same.

12 Implementation

12.1 Source Code

The latest implementation on Linux is available as a source code package of 12,500 lines of C-code from [1]. It compiles into a loadable module of 167 Kbyte for the Linux-2.6 kernel, and it requires no kernel patches to be installed. This version, just as an earlier one for Linux-2.4, is stable, but still has some limitations. Most notably, only single-cluster communication is supported for now; it is not possible to set up links between nodes in different clusters or different zones.

12.2 Standardization

Open Source Development Lab (OSDL) has defined TIPC as a cornerstone in their Carrier Grade Linux (CGL) strategy, and people from OSDL are contributing actively to the code. TIPC meet several Priority 1 requirements and many Priority 2 requirements in the clustering specifications of Carrier Grade Linux version 2.0 [7]. Within IETF, the ForCES Work Group is considering TIPC to be used as transport protocol between forwarding and control elements in distributed routers. An IETF-draft [4] with a complete specification was presented for the WG at IETF-59 for this purpose.

12.3 Roadmap

The goal is to have TIPC accepted as an integrated part of the Linux kernel in future releases (2.7/2.8). Before the end of 2004, we also want to have it accepted as the preferred protocol for intra cluster transport of the ForCES protocol. Also, before the end of this year, we plan to have developed full support for inter-cluster and inter-zone communication, as well as a redesigned slave node communication framework.

13 Conclusion

Within Ericsson, TIPC has proven to be a very useful toolbox for design of high-availability clusters. It is our hope that this experience will be repeated by others now as the potential of advanced clustering is becoming more widely recognized.

References

[1] TIPC code and documentation
`http://tipc.sourceforge.net`

[2] Postel, J., Transmission Control Protocol, RFC 793
`http://www.ietf.org/rfc/rfc0793.txt`

[3] Stream Control Transmission Protocol, RFC 2960
`http://www.ietf.org/rfc/rfc2960.txt`

[4] Maloy, J., Transparent Inter Process Communication, IETF Draft
`http://www.ietf.org/internet-drafts/draft-maloy-tipc-00.txt`

[5] Guo Min: TIPC Reliable Multicast Design

http://www.linux-ericsson.ca/papers/tipc-multicast/tipc_multicast.pdf

[6] Maloy, J., Make Clustering Easy With TIPC, Linux World Journal April 2004 http://www.linux.ericsson.ca/papers/tipc_lwm/index.shtml

[7] OSDL CGL Requirement Definition 2.0 http://www.osdl.org/lab_activities/carrier_grade_linux/documents.html

Object-based Reverse Mapping

Dave McCracken
IBM
dmccr@us.ibm.com

Abstract

Physical to virtual translation of user addresses (reverse mapping) has long been sought after to improve the pageout algorithms of the VM. An implementation was added to 2.6 that uses back pointers from each page to its mapping (pte chains). While pte chains do work, they add significant spaceoverhead and significant time overhead during page mapping/unmapping and fork/exit.

I will describe an alternative method of reverse mapping based on the object each page belongs to. I will discuss the partial implementation I did last year as well as the work done by Hugh Dickins and Andrea Arcangelli to complete it. I will describe the current implementations, their relative strengths and weaknesses, and what plans if any there are for solutions to the remaining issues.

1 Introduction

Up through version 2.4, the Linux® kernel had no mechanism for translating physical addresses to user virtual addresses, commonly called reverse mapping, or rmap. This meant it was not possible for the memory management subsystem to point to a physical page and remove all its mappings. There was a mechanism that walked through each process's mappings and selected pages to unmap. Only after all a page's mappings were removed could it be selected for pageout.

Many in the memory management community considered this very inefficient. Page aging and removal could be made much more efficient if the page could be directly unmapped when it was ready to be removed. Some form of rmap was clearly needed for this to work.

2 PTE Chains

Rik van Riel implemented an rmap mechanism that added a chain of pointers to each page back to all its mappings, commonly called `pte_chains`. It works by adding a linked list to the control structure for each physical page (struct page) which points to all the page table entries that map that page. His code was accepted into mainline early in the 2.5 development cycle.

Once this rmap implementation was in place the page aging and removal algorithm was changed to use it, streamlining the code and allowing better tuning.

One negative to the `pte_chain` implementation was a significant performance cost to `fork`, `exec`, and `exit`. The cost to these functions was related to the amount of memory mapped to the process, but was close to an order of magnitude worse.

A second cost was space. In its original form `pte_chains` cost two pointers per mapping. An optimization eliminated the extra structure for singly-mapped pages and another optimiza-

tion added multiple pointers per list entry, but the space taken by the `pte_chain` structures was still significant.

3 A Brief History of Object-based Rmap

Processes do not really map memory one page at a time. They map a range of data from an offset within some object (usually a file) to a range of addresses. The virtual addresses of all pages within that range can be calculated from their offset in that object and the base mapping address of the range.

The kernel has the information to do object-based reverse mapping for files. Each `struct page` for a file has an offset and a pointer to a `struct address_space`, which is the base anchor for all memory associated with a file. Every time a range of data from that file is mapped to a process, a `vm_area_struct` or `vma` is created. The `vma` contains the virtual address of the mapping and the base offset within the file. It is then added to a linked list of all `vma`s in the `address_space` for that file.

The remaining problem in the kernel is anonymous memory. Blocks of anonymous memory have `vma`s but these `vma`s are not connected to any common object that can be used for reverse mapping.

3.1 Partial Object-based Rmap

Given this information, last year I did a sample implementation of object-based rmap for files, but left the `pte_chain` implementation in place for anonymous memory. It works by following the pointer in the `struct page` to the `struct address_space`, then walking the linked list of `vma`s to find all that contain the page. A simple calculation then determines the virtual address of that page and a page table walk finds the page table entry.

This implementation recovers the performance of `fork`, `exec`, and `exit` and eliminates the space penalty used by `pte_chain` structures. It introduces a performance penalty when it walks the linked list of `vma`s, but this is incurred by the page aging code instead of the application code. It could still be significant, however, since it rises linearly with the number of times any part of the file is mapped while with `pte_chains` the cost rises linearly with the number of times that page is mapped.

3.2 First Cut at Full Object-based Rmap

Hugh Dickins took my implemenation and extended it to handle anonymous mappings, eliminating `pte_chains` entirely. He did this by creating an `anonmm` object for each process that all anonymous pages belong to. All `anonmm` structures are linked together by `fork`. A new `anonmm` structure is allocated on `exec`. The offset stored in `struct page` is the virtual address of the page, while the object pointer points to an `anonmm` that the page is mapped in.

Finding all mappings of a page is simple. The pointer in `struct page` is followed to the `anonmm` chain, which is then walked looking for mappings of that page at the virtual address specified in the offset.

Hugh's initial patch ignored the problem of shared anonymous pages that were remapped by an `mremap` call. The problem with `mremap` is that it allows an anonymous page to be at different virtual addresses in different processes, but there is only one offset for the page.

After some initial discussion among the community, both Hugh and I moved on to other things.

3.3 A Second Cut at Full Object-based Rmap

In February of this year Andrea Arcangeli began to investigate what could be done about the problems of `pte_chains`. He took my partial object-based rmap patch and implemented his own solution for anonymous memory, called `anon_vma`.

The basic mechanism of `anon_vma` is the addition of an `anon_vma` structure linked to each `vma` that has anonymous pages. The `anon_vma` structure has a linked list of all `vma`s that map that anonymous range. The pointer in `struct page` points to the `anon_vma` and the index is the offset into the current mapping.

An advantage of Andrea's `anon_vma` structure is that it solves the mremap problem that the `anonmm` structure did not. Since the offset stored in each page is relative to the base of the `vma` that maps it, the region can be remapped without changing the offset. However, since `vma`s can be merged, it is not an an absolutely painless solution.

4 Advancements All Around

In response to Andrea's patch, Hugh resumed work on his `anonmm` patch. Prompted by a discussion among the community and an approach suggested by Linus, Hugh implemented a simple scheme for handling the remap case. For each page, if there is only one reference, that page can simply have its offset changed. If the page is shared, a copy is forced and the new unshared page is mapped at the new address. Since all anonymous pages are already copy-on-write, it is likely that the page would be written to eventually and the copy taken. It is possible that some read-only pages might be duplicated, but to date there is no evidence that any code actually remaps shared read-only anonymous pages.

5 The `vma` List Problem

All these implementations still include the original implementation for file pages, including the need to walk the linked list of `vma`s attached to the `address_space` structure. This has been identified as a possible performance issue for massively mapped files, though few if any real-life examples have been found. A few optimizations have been tried, including sorting the list by start address and making a two level list based on start and end address. Both these solutions share the problem that adding or modifying a `vma` is fairly expensive and holds the associated lock for a long time.

A recent contribution by Rajesh Venkatasubramanian is the use of a `prio_tree`, which is similar to a radix tree but supports sorting objects by both start and end addresses. It adds some complexity to the `vma` list but greatly reduces the potential performance impact of a large number of mappings.

6 The `remap_file_pages` Problem

While object-based rmap appears relatively simple, there is one new feature that greatly complicates the problem. This feature is `remap_file_pages`.

The `remap_file_pages` system call was introduced during the 2.5 development cycle. It works on a range of shared memory mapped from a file, and allows an application to change the memory range to map a different offset within that file. This is done without modifying the `vma` describing the mapping. This means the offsets specified within the `vma` are now

wrong. Since the `address_space` pointer and offset within the `page` structure are intact, the page can still be mapped back to its place in the file, but it is no longer possible to use this information to find its virtual mappings. The `vma` is called a `nonlinear vma` and is put on a special list within the `address_space`.

Andrea and Hugh have provided two different solutions to the problem of what to do when a nonlinear page is called to be unmapped. Andrea's solution is the more draconian in that it walks the list of nonlinear `vma`s and unmaps all pages in them until the page in question has no more mappings. Hugh's solution only unmaps a fixed number of nonlinear pages and makes no attempt to unmap the actual page passed in.

7 Release Status

As of the date this was written, Hugh has been submitting incremental rmap changes to Andrew Morton for the -mm tree over the past couple of months. The early submissions were primarily cleanup, but later patches included first my partial object-based rmap implementation followed by his `anonmm` implementation, which completely removed the `pte_chain` code.

Hugh has just submitted a final set of patches to Andrew that removes his `anonmm` implementation and replaces it with Andrea's `anon_vma` implementation.

The general expectation among the VM developer community is that once this code has been adequately tested in the -mm tree that it will replace the existing `pte_chain` implementation in mainline 2.6.

Legal Statement

This paper represents the views of the author and does not necessarily represent the view of IBM.

The World of OpenOffice

Michael Meeks
Novell, Inc.
mmeeks@novell.com

Abstract

In this talk I will present some of the issues facing OpenOffice.org, particularly related to: performance, interoperability, buildability, ABI / engineering and release practice. We'll look at how to build the beast, the UNO component model, and iterate a quick hack before your eyes. We'll also show some of the flash new features including the Gnome desktop integration work.

1 A friendly giant

The OpenOffice.org source base is one of the largest monolithic Free software projects in existence, even with the pre-compiled mozilla binaries for several architectures stripped out:

Project	Source bz2 (MB)
Mozilla 1.4.1	31
Linux 2.6.7	33
GNOME 2.6.2	108
OO.o 1.1.2	160

OpenOffice.org (OO.o) represents one of the largest single contributions to Free software ever. Given this, it is somewhat incredible that Sun immediately settled on a licensing scheme in that is both liberal and substantially symmetric.

OpenOffice.org is licensed under two licenses:

- `LGPL` – the familiar, and best Lesser GPL.
- `SISSL` – essentially X11 with trip-wires for malicious UNO API, and XML file format compatibility breakage.

While it is necessary to share copyright with Sun by signing the Joint Copyright Assignment (JCA)[2], the use of OO.o code in StarOffice can be considered as being achieved under the SISSL[3] provisions.

Thus there is clearly huge potential for add-ins, integration with proprietary data-feeds, macros, etc.

2 Sun's dilemma

Sun's StarOffice product substantially consists of the OpenOffice.org core, as seen in public CVS, with the addition of a few extra proprietary modules. While this means that all the latest bug fixes are available in public CVS, it creates a number of frustrating artificial problems:

2.1 Release Engineering

- `minor release cycles` – there is a correct separation of commercial updates; of around once per quarter; thus this tends to be the frequency of minor OO.o releases regardless of bugginess.
- `release patch-size` – there is a fixed upper-bound on the size of a cus-

tomer patch download, thus ABI alterations in low-level libraries which would have a large knock-on effect, are forbidden.

- `ultra conservatism` – since customer updates are infrequent there is little incentive to back-port fixes to the stable branch; so many, trivial but high-impact fixes don't make it.
- `major release cycles` – for reasons unknown StarOffice works on an 18-month release cycle, so—at times (given freezes, etc.), it is possible to punt a feature / fix by nearly 2 years.

Clearly many of these problems make the OO.o development process somewhat cumbersome.

2.2 Portability Engineering

In contrast to many Free software project, StarOffice and hence OO.o, is designed to run on a broad spectrum of operating systems and versions. By contrast, e.g. GNOME applications, would typically require the latest version of GNOME to run.

This creates a number of interesting, hard-core engineering issues, and shows up the true state of Linux as a robust platform for ISVs.

For example, for font discovery much Linux software will link to the pleasant fontconfig library, and use purely client-side font rendering. OO.o in contrast has to run on older (or newer) platforms where there is either no fontconfig install, or it has a changed ABI, or it is badly configured. Thus the OO.o font discovery method uses the following heuristics:

- `fontconfig` – since this may not be available, we try to `dlopen` it, hook out various symbols, and extract a simple list of font filenames.
- `chkfontpath` – Red Hat, and others once shipped this tool which dumps a list of font paths; we try to `popen` and parse the output.
- `hard-coded paths` – various directories such as `/usr/X11R6/lib/X11/fonts/truetype` are known to be a good bet, and are scanned for fonts, including several language specific variants.
- `X server query` – the X server is queried to see what it can do, and a load of XLFDs are parsed.
- `internal fonts` – whatever internal fonts, and font-metric files we distribute are added to the mix.

Naturally, after doing all this work, we build a OO.o specific cache of much of the information, to accelerate subsequent startup.

This heavily engineered approach is not constrained to any one API-set, or technology—so, e.g., OO.o will attempt to use either lpr or cups for printing in a dynamic fashion.

Even glibc problems show up in Figure 1.

In addition, the cross-platform nature of OO.o and the unpredictability of the Linux feature-set (particularly the C++ ABI), leads to a large number of software packages being included inside the OO.o build itself. Thus, a stock OOo would include it's own compiles of (at least): `python`, `freetype`, `zlib`, `expat`, `libdb`, `NAS`, `neon`, `curl`, `sane`, `myspell`, `Xrender`.

As is probably obvious, this level of old platform support, and dependency aversion is hard to get enthusiastic about.

```
typedef struct {
    struct { long status; int spinlock; } sem_lock;
    int sem_value;
    void *sem_waiting;
} glibc_21_sem_t;
    /*
     * XXX this a hack of course. since sizeof(sem_t) changed
     * from glibc-2.0.7 to glibc-2.1.x, we have to allocate the
     * larger of both XXX
     */
#ifdef LINUX
    if (sizeof(glibc_21_sem_t) > sizeof(sem_t))
        Semaphore = malloc(sizeof(glibc_21_sem_t));
    else
#endif
        Semaphore = malloc(sizeof(sem_t));
}
```

Figure 1: compatibility with old `glibc` versions

3 Community Issues

In addition to these unusual constraints, the OO.o project is encumbered by acute tooling and collaboration inadequacies.

Perhaps the most serious problem, is that it appears CVS was not designed with 200+ MB of source / binaries in mind. Thus, even basic operations, such as a `cvs tag` can take up to a couple of hours, and are frequently blocked by robots slowly traversing the repository.

Secondarily, the collab.net SourceCast system adds a level of bureaucracy, and lack of responsiveness which when combined with being totally un-fixable makes for an unnecessarily painful experience. It seems likely that SourceCast is ideal for the use of existing, established Free software projects, or even newly formed projects—but it stumbles with OO.o. Furthermore, using closed software for Open Source collaboration is an intrinsically interesting decision.

4 The other side of the coin

4.1 `http://ooo.ximian.com/`

To make up for the existing inadequate web-tools, and documentation we provide several ‘external’ tools of interest.

- `hackers guide` – a Linux focused, hackers guide on how to build, iterate, and some basics of the OO.o code structure.
- `LXR/Bonsai` – basic web tools without which navigating the OO.o source is substantially more difficult.
- `bug filing` – a gateway that demangles the curious user-focused issue filing process, and allows bug filing directly against given code modules.
- `Planet OO.o` – the obligatory RSS aggregator.

4.2 ooo-build

The process of productising OO.o into a Linux package is filled with pain; so to amortise this

a collaboration has coalesced between various Linux vendors: Novell ne Ximian, Debian, Red Hat, SuSE, Ark, and PLD Linux around *ooo-build*.

ooo-build provides a growing set of useful patches many of which may arrive in OO.o in many months time; indeed all our work is intended to go up-stream into OO.o. We also provide a simple patch sub-setting system, to allow vendors to select a suitable set of patches.

Many of the features associated with ooo-build are desktop integration, system integration, and GUI cleanup pieces; e.g.:

- attractive new icons
- native-widget rendering
- GNOME-VFS integration
- ergonomic & aesthetic fixes
- system library usage

The ooo-build wrapper is also intended to make OO.o substantially easier to compile with a familiar `./configure; ./download; make; make install` process.

5 Performance

Performance is an area ripe for substantial improvement in OO.o, however, poor performance is caused by many factors, and identifying the most important of these is not always easy.

5.1 Linking

The linker has a very hard time linking OO.o, and while this can be reduced by pre-linking, the architecture of OO.o—whereby the majority of the code is in shared libraries required not by the main binary—but by other shared component libraries, linked at run-time.

Ulrich's analysis of OO.o [1] shows that 20,000 relocations are performed during startup, which combined with lookups across multiple libraries gives 1,700,000 string comparisons to startup. The sheer size of the symbol tables and the lack of locality of reference in the linking process causes much of this work to fall outside the processors' cache—giving abnormally poor performance.

5.2 C++ issues

Some features of C++ exacerbate the problems of large symbol tables, and poor startup performance. The stripping / re-working of static initialisers has helped accelerate performance—these being replaced with a thread-safe late instantiation based on accessor method local static variables.

C++ is a very symbol-hungry language—particularly with respect to virtual functions, which create an unnecessary burden (Figure 2). Virtual functions, despite resolving to a simple function pointer export a symbol, which is referred to directly to chain to parent implementations. While of course this can often be resolved away at link time, in a cross-library situation it would perhaps be more efficient to dereference a parent vtable function pointer.

Similarly, since in theory at least, a single class can be implemented across multiple shared objects, even 'private:' methods export symbols.

In addition to these problems, a more proactive approach to pruning old, and redundant code has been adopted in the development branch, to reduce code footprint, and symbol count.

```
class Foo : public Baa {
    virtual void VFunc();
  private:
    void ExportsSymbol();
};
...
void VFunc()
{
    ...
    Baa::VFunc();
}
```

Figure 2: C++ virtual functions

5.3 Binary filter code

To shrink the OO.o footprint, a large chunk of creaking binary format code has been extracted, along with compatible chunks of the core. This code pre-dating the XML file formats scattered the process of serialisation across the code, and resulted in a complex, hard-to-maintain and increasingly irrelevant maintenance problem. In OO.o 2.0 it will be used only on the rare occasions it is necessary as a binary to XML filter.

5.4 system libraries

Shrinking the large number of internal libraries, on Linux systems, and increasing the number of libraries shared with the system is an important part of performance improvement in 2.0. It clearly makes little sense to have an internal gtk+ library when the system version is ABI compatible, and better maintained.

Using system libraries—e.g., neon—also reduces the pain of handling security updates in the built-in libraries.

5.5 mmap performance

Possibly the most significant speedup in the 1.0 to 1.1 transition was the process of forcing as much of the OO.o code into memory before attempting to run it. This gave a very noticeable win; this was implemented in a simple fashion with mmap, and a loop reading a byte from each page. Ideally of course, the underling operating system would be able to do better here.

6 Interoperability

In a world where a tiny fraction of people are using Free software, the ability to share documents in a loss-less fashion with other people is crucial to the adoption of OO.o.

Much work has been done in this area for 2.0, of particular note the row-limit in calc has been raised to that of Excel, and much work has been done on form controls.

There are also exciting developments in VBA interoperability. OO.o provides a VBA-like language: StarBasic, and by devious means it has been possible to extract VBA text from Office files for some while. Office for performance reasons however stores VBA in 3 forms: an SRP stream, a compiled form, and a compressed text form. Since these are authoritative in that order (the text providing only a final fallback), it was thought that effective export would entail reverse engineering at least the the compiled form.

However in recent time, yet more devious means have been discovered to export macros as text to Excel and have them run transparently to the user. This it turns out is the foundation of macro interoperability between Office versions 97 through XP. Thus work is ongoing to improve the macro support so crucial for effective Excel interoperability.

7 Desktop integration

Much of the work of ooo-build has been adopted in one form or another up-stream for 2.0, giving the prospect of a highly desktop integrated OO.o experience out-of-the-box.

To achieve this, the lowest levels of OO.o's cross-platform abstraction: the Visual Class Library (VCL) have been virtualised, and now the main-loop, and top-level windows on a GNOME system are handled by the gtk+ toolkit. In order to avoid a complete re-write of the widget system—we use a simplified theming system that virtualises only the rendering of widgets, allowing basic widgets to match the look of the rest of the desktop.

Similarly main-loop integration makes things such as integrating the gtk+ file-selector and other GNOME dialogs fairly simple. The main-loop integration was made substantially more painful by the mis-match between the recursive OO.o toolkit lock, and the non-recursive gtk+ lock. In order to reconcile these and provide a single, comprehensible locking pattern—after considerable thought we added hooks to gtk+ to allow a shared (recursive) lock to be used. This makes gtk+ use in OO.o virtually seamless.

8 UNO component model

OO.o provides a rich, and well documented component model, which is exported for the use of language bindings. The power of this, and its flexibility have resulted in active bindings for StarBasic, Java, and Python.

The UNO model is particularly interesting, since it consumes little overhead beyond a stock C++ virtual function call. In addition each class has associated, small compiled IDL type information. This can be used, to dynamically (at run time) construct bridges to other languages, and allow dynamic method invocation. While this adds a compiler version / ABI dependency to the OO.o core, it avoids the problem of creating stub / skeleton code which ended up consuming many MB before the dynamic approach was adopted.

9 Conclusions

OO.o provides an unusualm and particularly pathalogical case of a gigantic C++ project. This leads us to push the boundaries of the system, showing up several areas for potential improvement.

The ooo-build infrastructure provides a solid base for contributing work to OO.o in a familiar and accessible manner, and seeing the deployed results of your work quickly.

OpenOffice 2.0 will give substantial performance, code-cleanliness and interoperability improvements, in addition to many new features.

References

[1] Ulrich Drepper. How to write shared libraries, 2004. `http://people.redhat.com/drepper/dsohowto.pdf`.

[2] Sun Microsystems, Inc. Joint copyright assignment. `http://www.openoffice.org/licenses/jca.pdf`.

[3] Sun Microsystems, Inc. Sun industry standards source license. `http://www.openoffice.org/licenses/sissl_license.html`.

TCPfying the Poor Cousins

Arnaldo Carvalho de Melo
Conectiva S.A.
acme@conectiva.com.br
http://advogato.org/person/acme
http://www.conectiva.com.br

Abstract

In this paper I will describe the work I am doing on the Linux networking infrastructure, with emphasis on cleaning the code, but with important "side effects" like reduction of core structures already saving over 600 bytes on UDP sockets all over the net in 2.5/2.6 (tcp, etc.), elimination of data dependencies, reduction of the non-mainstream network families maintenance cost by making them use code that now is in `net/ipv4` but can be moved to `net/core`, leaving only the really ipv4-specific code and making LLC use it as a proof of concept (work done in my net-exp tree, pending submission).

TCP code becomes used by the poor cousins, they appreciate that!

1 How This Started

Making IPX uptodate with regards to advances in the core networking infrastructure, to kill `deliver_to_old_ones`, i.e., special cases in the core kernel for protocols that hasn't been converted to shared skbs and multithreading.

In the process I noticed several areas where code was replicated or used a different, older framework, due to the evolution of the core networking infrastructure.

Also de experience of porting the NetBEUI and LLC code released as GPL by Procom Inc. from the 2.0 Linux kernel networking infrastructure to 2.4 and then to 2.5/6, working on a BSD sockets API for `PF_LLC`, initially contributed by Jay Schullist was instrumental in realising the existing similarities in the infrastructure needs required by several protocol families.

2 TCP/IP Evolves Faster

Most of the attention is given, of course, to TCP/IP, and in the process new infrastructure is created, with TCP/IP using it at first and sometimes leaving things like the `deliver_to_old_ones` function to simulate the previously existing big networking lock and the `SOCKOPS_WRAPPED` macro, to allow the other protocol families to continue working, hoping that their maintainers do the necessary work, but this sometimes doesn't happen for a long time.

In other cases code is added to TCP/IP that, upon further inspection, could be moved to `net/core` and be useful for the other protocol families.

Doing this factorization will help make these improvements to TCP/IP be taken advantage of by the other protocol families and will help in realising the ultimate goal of keep the proto-

col families code with just what is completely specific.

3 Trimming `struct sock`

In 2.4, `struct sock` has a big fat union that has most of the private data for each protocol family, so when any change had to be done to a specific protocol family the layout of `struct sock` would change, generating unnecessary recompilation of most of the network related code in the kernel.

In 2.6 this has changed and `struct sock` nowadays is mostly free of details specific to network protocol families.

In the process two ways were devised to store the network protocol private area, one for protocols that have stringent performance requirements, like TCP/IP, using per-protocol slab caches and another one, simpler, that allows protocol families to just allocate a chunk of memory and store its pointer in the `struct sock` member `sk_protinfo`. As most stacks now use helper macros to access its private area, the eventual switch to the slab cache approach is easily done.

With this in place the footprint of the `struct sock`, that was of about 1280 bytes on a UP machine in 2.4 to 308 bytes for the generic sock slabcache in 2.6, with the `tcp_sock` slabcache using 1004 bytes, `udp_sock` slabcache using 484 bytes and finally the `unix_sock` (`PF_UNIX` sockets) using just 356 bytes.

This changes also resulted in a performance gain in the establishment of connections, as was verified with the `lmbench` tool.

Another related change was to diminish the data dependency among `struct sock` and `struct tcp_tw_bucket`, that is a "mini socket" used to represent TCP connections in the `TIME_WAIT` state. To accomplish this, `struct sock_common` was introduced, that is the minimal required set of members common to these structs. With this data layout we will certainly avoid bugs introduced when changing only one of the structs, like has happened at least once to my knowledge.

4 Using `list.h` in the Networking Code

With the advent of the hashed lists (`struct hlist_node`) it turned out to be useful to make the networking code follow the general kernel trend of using the `linux/list.h` macros, replacing the ad-hoc lists present in the networking code.

The work consisted of introducing a set of helper macros to handle `struct sock` list handling, namely `sk_add_node` and `sk_del_node_init`, and bind list variants.

These functions also bump the reference count for the socket, something that was not being done by some protocols, that have since been converted to use this new set of helper macros, thus fixing some bugs in the process.

It should also be noted that `hlist` started using prefetch as part of the process of convincing David Miller, the Linux Networking maintainer, to accept such changes. Performance gains are an important technique in getting code-cleaning patches accepted.

5 Socket Timers Manipulation Helpers

Another area that received attention was the socket timers manipulation routines, that in some protocols aren't always bumping the ref-

erence count as they should and do in the Bluetooth and TCP/IP code.

To abstract this handling the `sk_reset_timer` and `sk_stop_timer` functions were introduced recently to do the `timer_list` handling and deal with `struct sock` reference counting.

6 Factorization of net/ipv4 Code

In the past Alan Cox worked on having datagram code that could be shared among several network families shared at the `net/core/datagram.c` file, moving chunks of code out of the UDP implementation.

Now with this work I'm trying to do the same with the stream code, now moving chunks of TCP code to the core infrastructure.

Initial steps are just moving code around, like `tcp_eat_skb`, that became `sk_eat_skb`; `tcp_data_wait` became `sk_wait_data`; and here we see something interesting, namely the fact that this function correctly sets the `SOCK_ASYNC_WAITDATA` bits in the `struct socket` flags member, something that some protocols aren't doing now but will as soon as they start using `sk_wait_data`.

In my `net-experimental` tree I have introduced some new members to the `struct sock` member `sk_prot`, allowing both TCP and LLC to use common `stream_sendmsg` and `stream_sendpage` functions, that are generalizations of `tcp_sendmsg` and `tcp_sendpage`. Further work is needed to fully determine the performance implications of such changes, but no noticeable performance drop or stability problems have been verified in using this patched kernel in my main machine for over a month.

7 BSD Sockets Layer

There is some duplication of work at the BSD sockets level among the network protocol families implementation. Trying to reduce the code required to implement a protocol family is being investigated, with some proofs-of-concept already implemented, where the functions now used for TCP/IP are being shared with LLC.

The idea here is to to reduce the protocol-specific implementation to just that, i.e., what is absolutely specific to each protocol.

Perhaps this will make it easier to stack protocols, allowing combinations that are possible in other kernels but not on Linux right now.

The extra function pointers in `sk->sk_prot` probably won't be a problem because they will make it possible to eliminate `sock->proto_ops` by calling directly the `sk->sk_prot` functions.

8 Future Developments

With this newly common infrastructure, it may be possible to add features like network async I/O to all protocols. More sharing will be investigated, trying to avoid pitfalls that appeared in similar work done in other kernel subsystems.

9 Conclusion

Looking every other year at how core infrastructures evolve and how the implementations of subsystems attached to those infrastructure evolve is something that should be done, paying off in terms of code clarity, reduction of the cost of maintaining code that has come out of mainstream but are still used in lots of legacy setups.

Another eventual benefit gained is the performance one, as making the code clear and more general is not incompatible with having fast code.

Reuse the code, Luke.

10 Acknowledgments

I'd like to thank David S. Miller for all the support he gives me in continuing this work, reviewing my patches, and providing much valued words of wisdom. And my respect for Andi Kleen, for helping me out in my childhood as a Linux networking wannabe hacker, Alan Cox for throwing me the Procom NetBEUI stack, that was fun! And nasty as well. As well as all the fine kernel networking hackers that spare some of their time to comment on my ideas.

IPv6 IPsec and Mobile IPv6 implementation of Linux

Kazunori MIYAZAWA
USAGI Project/Yokogawa Electric Corporation
kazunori@miyazawa.org

Masahide NAKAMURA
USAGI Project/Hitachi Communication Technologies, Ltd
masahide_nakamura@hitachi-com.co.jp

Abstract

USAGI Project [8] has improved Linux IPv6 [1] stack. IPv6 IPsec is one of the products of our efforts. Linux IPsec [6] stack is implemented based on XFRM architecture which is introduced in linux-2.5. We design and implement Mobile IPv6 (MIPv6) [4] Stack on the architecture. MIPv6 uses IPsec for its secure signaling. Accordingly IPv6 IPsec and MIPv6 closely cooperate each other. In this paper we describe the architecture and how they work.

1 Introduction

IPv6 is the next version of an Internet Protocol. The protocol was developed against IPv4 address exhaustion. It was developed for not only spreading address space but improving some features such as plug and play, aggregatable routing architecture, IPsec native support and smooth transition.

IPsec provides security services which are integrity, authentication, anti-replay attacks and confidentiality. Because IPsec is mandatory in IPv6 specification, we must implement IPsec to conform to it.

MIPv6 provides all IPv6 nodes with mobility service which allows nodes to remain reachable while moving around IPv6 networks. To support mobility, We need some signaling architecture to notify movement and deliver mechanisms to assure reachability. Using MIPv6, we can keep routability to mobile node's home link address and deliver a packet to mobile node wherever it is on the network. Because IPv6 is able to process these extension headers natively, we no longer need to arrange foreign agents to all links where mobile node may move to as Mobile IPv4 does, so that IP mobility is easier to be introduce in IPv6 than IPv4.

Linux supported IPsec at version 2.5.47. However it supporting only IPv4 IPsec, we implemented IPsec stack for IPv6. Linux version 2.6 supports IPsec on both IPv6 and IPv4. XFRM architecture and stackable destination were introduced into the kernel for IPsec packet processing [7]. They can be not only for IPsec packet processing, but also general packet processing such as MIPv6. USAGI Project decided to expand the architecture to implement MIPv6.

To develop Linux MIPv6, we cooperate with GO/Core Project [2] which is proven in linux-

2.4.

2 XFRM and stackable destination

XFRM architecture is mainly consist of three structures which are xfrm_policy, xfrm_state and xfrm_tmpl. xfrm_policy corresponds to IPsec policy and xfrm_state to IPsec SA. xfrm_tmpl is intermediate structure between xfrm_policy and xfrm_state. Each IPsec policy and SA database are realized with list of the structures which are also contained hash database.

The kernel provides three interface to configure xfrm structures about IPsec. One is PF_KEY interface which is standard interface to manipulate IPsec database. another is netlink socket interface. The last is socket option interface.

Stackable destination is architecture for efficient outbound packet processing. It is a link list of dst_entry structure which is cached in xfrm_policy. To create stackable destination, the kernel linearly searches xfrm_policy with flow information for a sending packet after routing looking up. After finding xfrm_policy corresponding to the flow information, the kernel searches and gathers xfrm_state from xfrm_state database by xfrm_tmpl in the xfrm_policy. Gathering xfrm_states, the kernel builds up stackable destination and substitutes it into its own member “bundles” to cache it. Additionally xfrm_policy itself is cache in flow_cache. Therefore the kernel only needs to lookup xfrm_policy after second until xfrm_state expired.

3 IPsec

IPsec functionality is consist of packet processing and key exchanging for automatic keying. In the implementation of Linux packet processing runs in the kernel and key exchange is done by a key exchange daemon in user space.

3.1 IPsec database and packet processing

IPsec packet processing is realized with XFRM architecture and stackable destination. Outbound process is explained in previous section. With searching XFRM database and building stackable destination, the kernel gets list of dst_entry structure. To process each function which are ah6_output, esp6_output and ipcomp6_output, the kernel searches insertion point on a packet because a packet is created including IPv6 header and other extension headers before stackable destination process (Figure 1). The insertion point is before upper layer payload, fragmentable destination options header, IPsec header or fragment header. This is not efficient because the kernel searches the insertion point every time when processing one dst_entry.

Inbound process is simpler than outbound process. When packet containing AH or ESP, the kernel finds xfrm_state corresponding to received packet and keep pointers of used xfrm_state in sec_path of skb structure. After process of IP layer, the kernel checks the packet correctly processed with comparing sec_path and xfrm_policy which is searched with flow information of the packet (Figure 2).

3.2 Interface for user and IKEd

Current linux kernel provides users with PF_KEY interface, which however is specified only for IPsec SA interface and it needs some extension to configure IPsec policy. Because this extension is not standardized, there are some different extensions and it prevents compatibility of IKEd. Linux adopts the extension which is compatible with KAME [5] so that racoon is the IKEd for linux. Racoon is originally product of KAME project and its could not compile on Linux. Fortunately

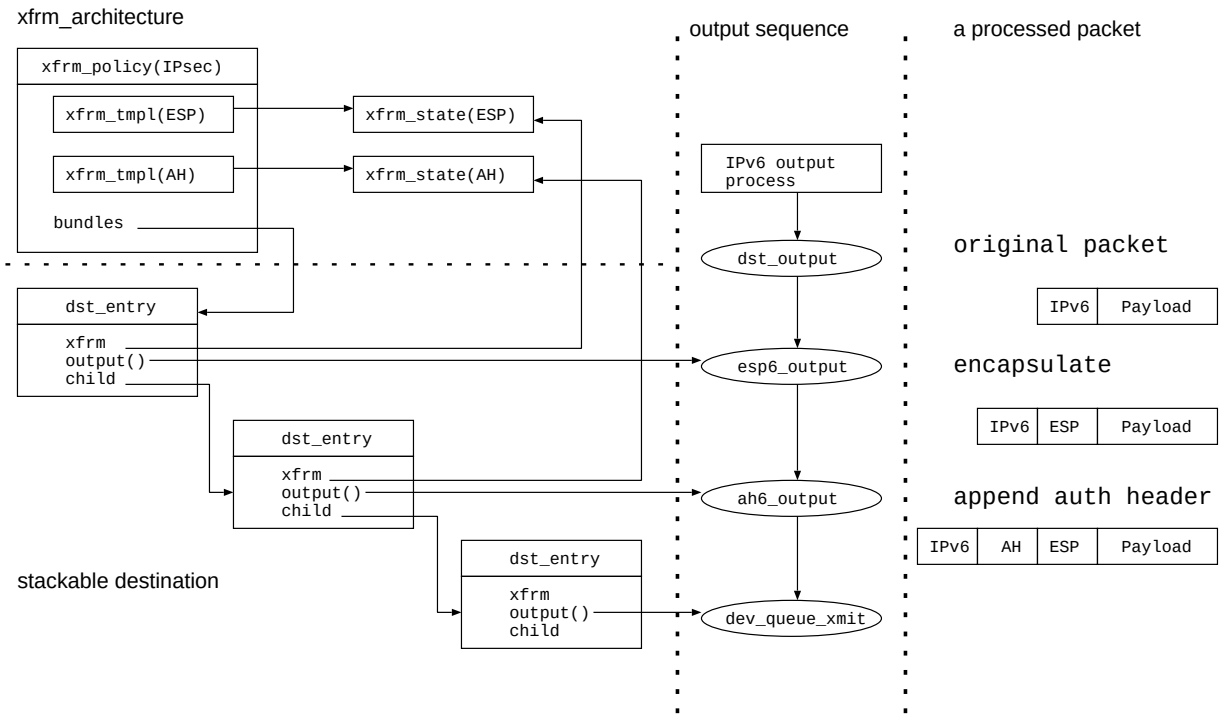


Figure 1: IPsec output process

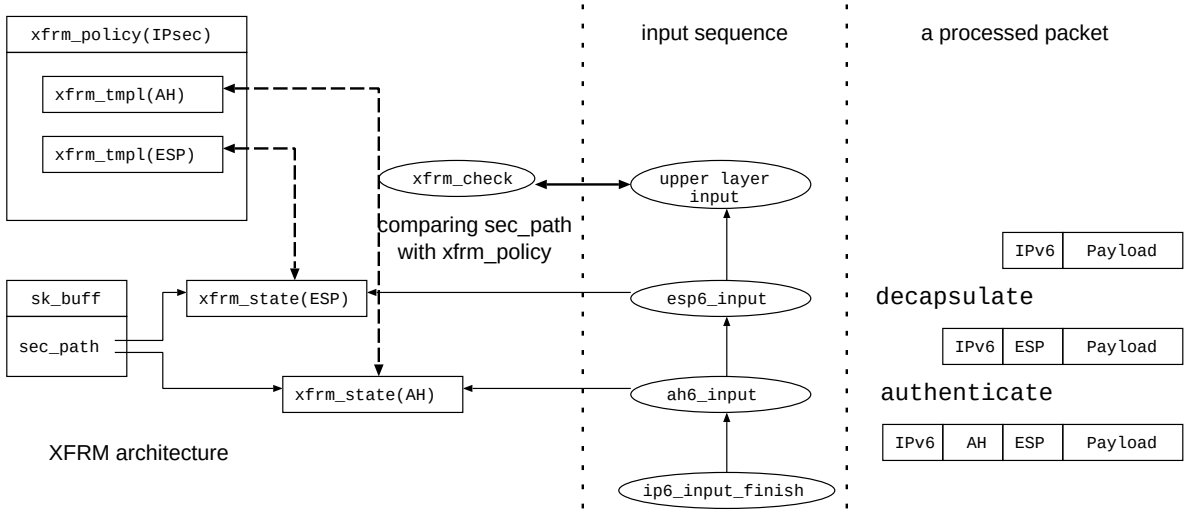


Figure 2: IPsec input process

ported racoon which is provided by ipsec-tools project [3] is available.

4 Mobile IPv6

4.1 Mobile IPv6

In MIPv6, nodes are classified into 3 types. One is a Mobile Node (MN) which moves in the IPv6 Internet bringing its home address (HoA) assigned in a home link which is a base of mobility and in which there is a home agent. Home agent (HA) is another type of node which is a router and manages MN's addresses and supports its signaling and ensures reachability. The other is a correspondent node (CN) which is a node communicating with a MN. CN may be either mobile or stationary.

When MN in a foreign link, it uses a care-of address (CoA) which is the address of a foreign link. MIPv6 accordingly needs to manage relationship between CoA and HoA. A MN sends a packet including HoA in an extension header from CoA.

MIPv6 appends two extension headers and one option for destination options header. Mobility Header (MH) is an extension header for signaling to manage binding cache which is a address list for optimized routing. Type2 routing header (RT2) which is different from routing header in RFC2460 effects destination address in IPv6 header and realizes direct routing according to binding cache. Home Address Option (HAO) is an option carried by destination options header to contain HoA which is an address of a MN in home link and swapped with CoA. HAO effects source address in IPv6 header.

We describe an outline of the procedure taking as an example that MN making binding cache on HA and communicating CN after MN moving to a foreign link (Figure 3). This procedure is divided two steps. First is making IPv6 over IPv6 tunnel between MN and HA (1-4). After this step, HoA of MN becomes routable and MN is able to communicate with all nodes by using HoA via HA through the tunnel. Second is route optimization between MN and CN because MN always communicating via HA (5-8), a packet goes through a superfluous route and communication uses more network resource.

1. MN sends a Binding Update (BU) to HA.
2. HA updates a binding cache and returns Binding Acknowledgment (BA) to MN.
3. MN updates a binding update list.
4. At this time, there is a tunnel between MN and HA.
5. MN sends HoTI to CN through the tunnel and CoTI to CN directly from CoA.
6. CN keeps contents of HoTI and CoTI. CN returns HoT via HA and CoT to CoA.
7. When MN receives HoT and CoT, MN sends BU to CN and updates its own binding list.
8. Then MN and CN have binding between HoA and CoA. They communicate directly with appending HAO and RT2 to packets. They have an optimized route.

4.2 Implementation

We design MIPv6 in Linux consisted with two part. One is packet processing for RT2 and HAO in the kernel and the other is MIPv6 daemon (MIPd) to handle the signaling and manage binding cache and binding update list. It is similar to separation of packet process and IKEd in IPsec.

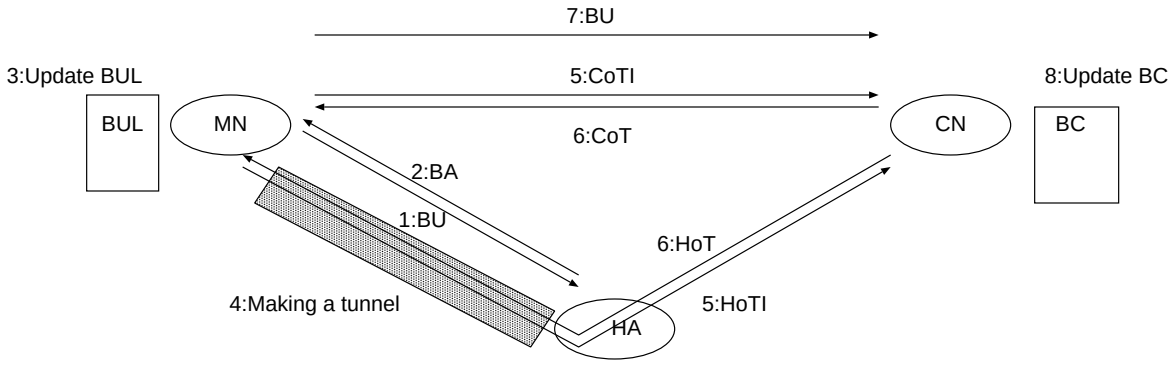


Figure 3: MIPv6 procedure outline

Packet processing for MIPv6 is realized with XFRM and stackable destination architecture, because they are general way to process a packet which matches some selector. Using XFRM, we can avoid to implement duplicate functionality in the kernel. MIPv6 needs to manage a binding cache which specifies an MN address on the network on CN and HA. It also needs to manage a binding update list which is list of sending binding update request for CN on MN. We have two choices to implement this functionality in the kernel or userland. Because we should implement functionalities in userland if it is possible, we consider to basically implement it in userland. Implementing in userland brings us advantages which are easier extension its functionality than implementing in the kernel and reducing the kernel size.

Our MIPd's roles are

- processing a signaling message including an error message
- managing xfrm_policy and xfrm_state of MIPv6 in the kernel through the netlink
- managing binding cache and binding update list
- moving detection and changing CoA when MIPd running on MN

4.3 XFRM operation

In this section, we describe MIPd XFRM operation relating each nodes state with an example which is a phase of binding update to HA and making tunnel for routability. It is called home registration. At first, we initialize MN and HA to send and receive binding message. On MN MIPd sets a xfrm_policy which allows an outbound packet from HoA to HA, proto MH, and type BU with appending HAO and a xfrm_state which appends HOA with CoA to a packet from HoA to HA and including MH of BU. It also set xfrm_policy to receive BA, the policy which allows an inbound packet from HA to HoA including MH of BA with appending RT2 and the inbound xfrm_state which processes RT2. Because MIPd on HA can not expect the source address of BU from MN, it sets a xfrm_policy which allows an inbound packet from Any to HA with MH of BU if it has HAO. It also set xfrm_state which processes HAO included in a packet from ANY to HA with MH of BU. See Figure 6:INITIALIZE.

MIPd on MN sends BU to HA, the packet matches with the xfrm_policy and process with the xfrm_state which appends HAO destination option and swap a source address in IPv6 header with a CoA. HA received the BU from MN. In the kernel the packet matching the xfrm_state, the kernel swaps addresses. Then

MIPd on HA receives BU and updates a binding cache. MIPd configures xfrm_policy and xfrm_state for route optimization with high priority. See Figure 6:Routing Optimization.

At this moment, route optimization is available for all packets between MN and HA. It also sets up a tunnel between MN and HA. After some xfrm_policy and xfrm_state configuration it returns BA with RT2. The kernel of MN receives BA with RT2 and processes it with the inbound xfrm_state and throws up BA packet to MIPd. MIPd on MN updates a binding update list and sets up the tunnel. Each nodes has totally 6 policies at the end of registration.

5 Cooperation of IPsec and MIPv6

MIPv6 uses IPsec for its secure signaling between MN and HA. Our design uses XFRM and stackable destination for both IPsec and MIPv6. MIPv6 needs two kind of IPsec SA one is a transport mode SA which is used for signaling. The other is a tunnel mode SA which is used instead of IPv6 over IPv6 tunnel. We consider two steps to implement MIPv6 with IPsec about IPesc policy and SA management. At first, we implement MIPd to not only manage xfrm_policy and xfrm_state of MIPv6 but also IPsec and a xfrm_policy for MIPv6 holds both MIPv6 and IPsec xfrm_tmpl. This implementation has a couple of issues. One is separation of management of xfrm_policy and xfrm_state of IPsec into MIPv6 and ordinary IPsec. Another issue is interaction between the kernel and IKE daemon. xfrm_policy including a xfrm_tmpls of Mobile IPv6 and IPsec sends a signal for only MIPd. The other is the order of xfrm_policy. When some situation such as configuration done with wrong order, a packet which would be originally applied MIPv6 and IPsec not be applied only IPsec.

For improvement, we will let the kernel hold two xfrm databases and mediate them because it is difficult to manage xfrm_tmpl in a xfrm_policy via userland interface by two management daemons and the xfrm_policies have probably different granularity (Figure 7). In current outbound process, the kernel looks up single xfrm_ policy database and gets a xfrm_policy which includes xfrm_tmpl for IPsec and xfrm_tmpl for MIPv6. However we will change the kernel to separately look up IPsec and MIPv6 xfrm databases and create temporary xfrm_policy which holds xfrm_tmpl gathered from each xfrm_policy. The list of xfrm_tmpl must be serialized as the order of packet processing. For instance, the kernel must put xfrm_state for AH at the end of the list. For inbound process, it is not so difficult, the kernel processes a packet by using xfrm_state which is searched and needs to check sec_path in skb against each xfrm_policy. To make it be efficient, the kernel should use flow_cache for inbound process.

If we could merge two policies correctly, we have another issue. MIPv6 needs two IPsec SA between NM and HA. One is a transport mode SA for signaling and the other is a tunnel mode SA for other packet. Taking outbound SA as an example, a transport mode SA is applied by the policy whose selector is from HoA to HA and protocol MH. On the other hand a tunnel mode SA is applied by the policy whose selector is from HoA to ANY and protocol ANY. The packet should be applied the transport mode SA has possibility to be applied the tunnel mode SA. We can avoid this mismatch by using priority in xfrm_policy.

racoon has a couple of issues as IKE daemon for MIPv6. One is that racoon can not handle multiple peers which have address ANY as peer's address in its configuration. When it behaves as responder on HA, the issue occurs because despite multiple peers being, each configuration has addresses from ANY to HA thus

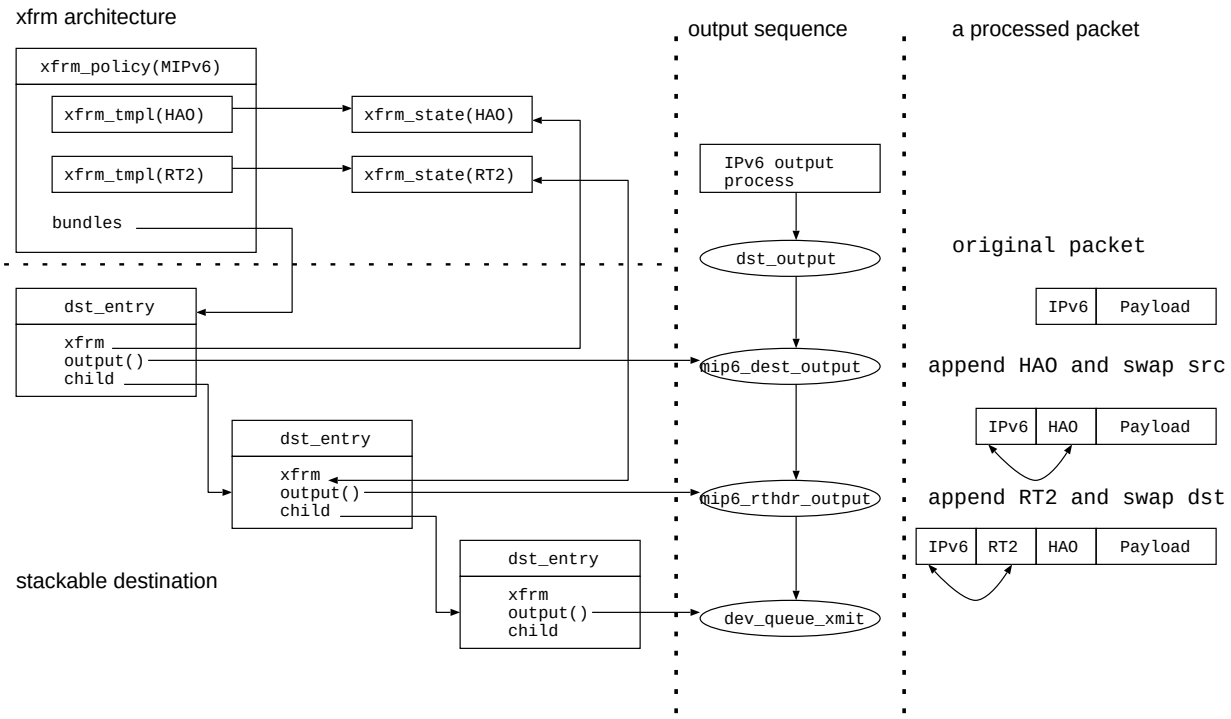


Figure 4: MIPv6 output process

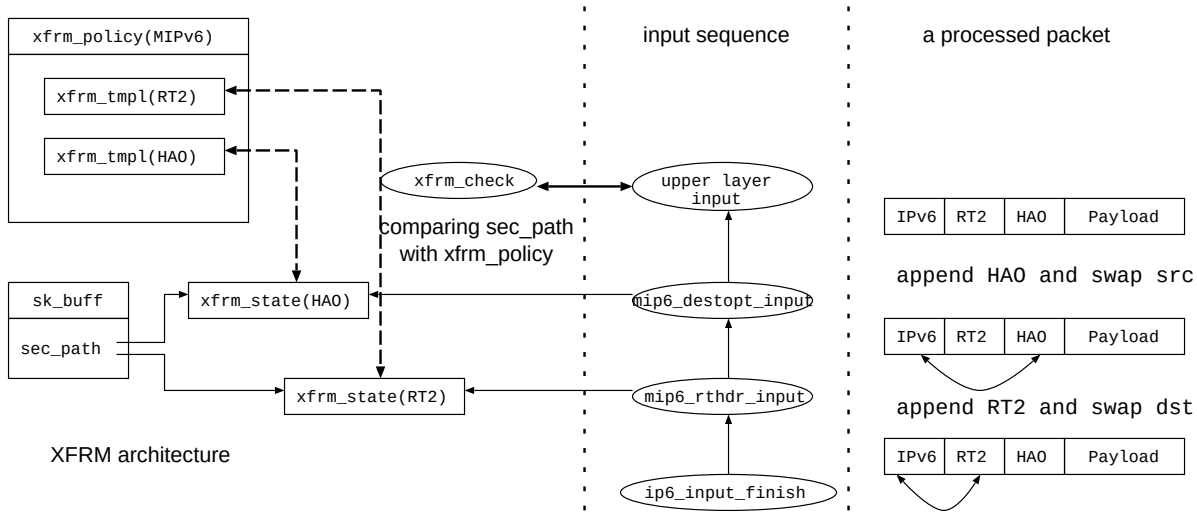


Figure 5: MIPv6 input process

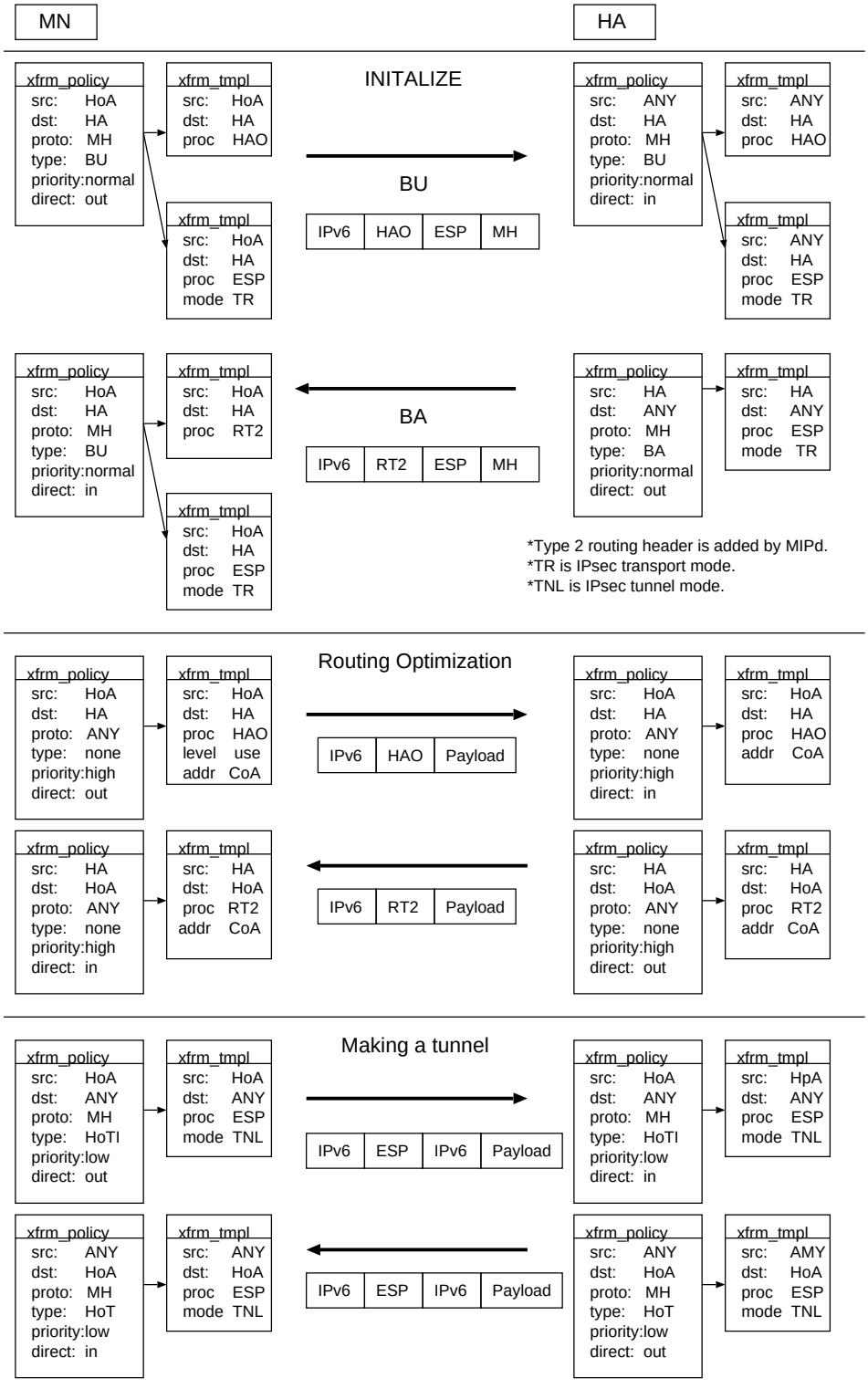


Figure 6: Binding update procedure to Home Agent

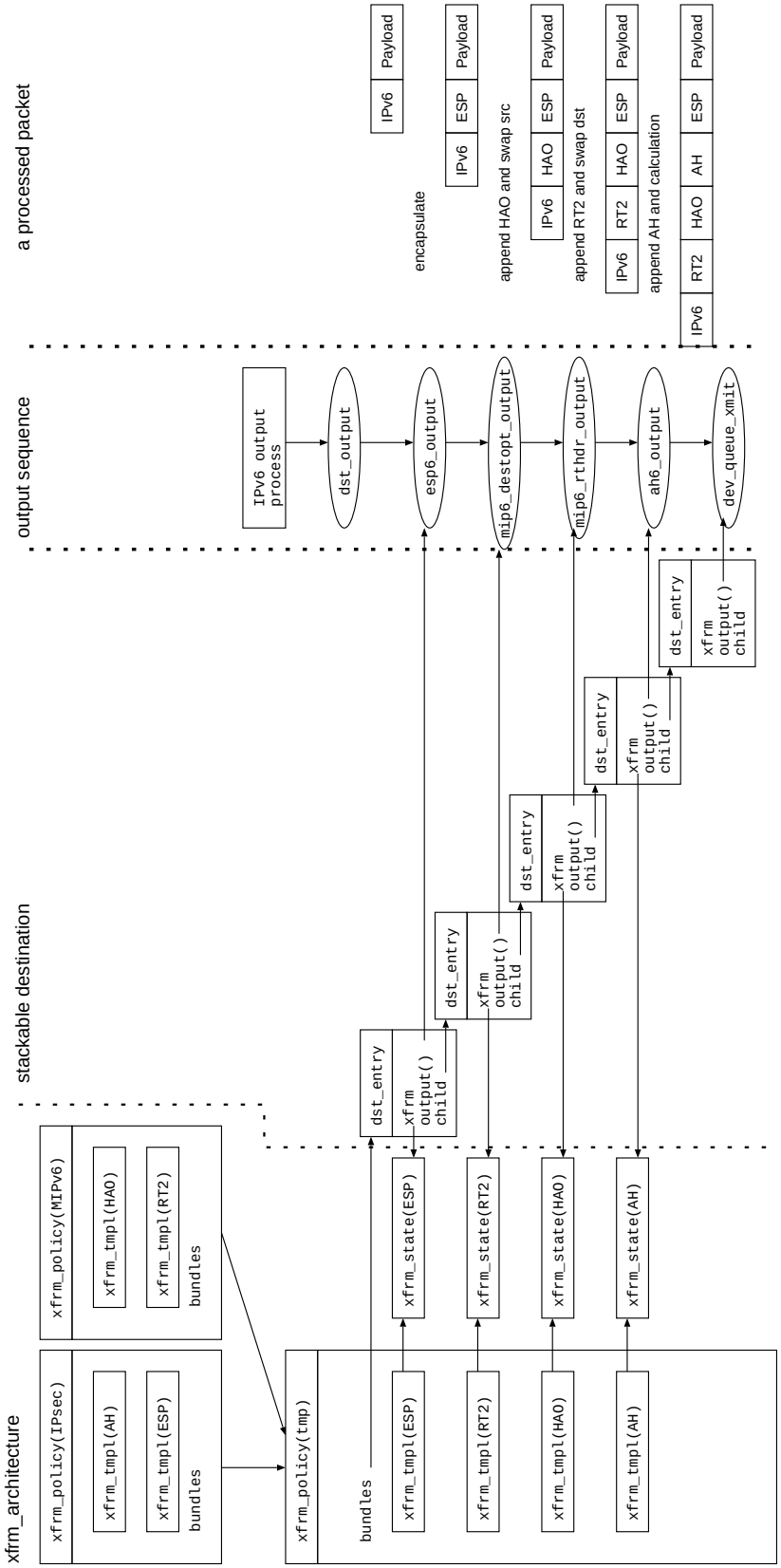


Figure 7: MIPv6 and IPsec output process

racoon can not distinct peer and fails to search proper key. The other issue is update ISAKMP SA end-point address. When MN moves, IKEs on MN and HA need to detect movement in some way and update its ISAKMP SAs because an address of those SAs is CoA. To solve these issues, we will make racoon handle the multiple peers listen netlink socket for the detection and make the kernel notify address changing via netlink socket.

6 Summary

USAGI Project implements IPv6 IPsec and MIPv6 by using XFRM and stackable destination architecture. In this paper we describe our design, implementation and issues. We also describe future design of IPv6 IPsec and MIPv6 which improves flexibility of xfrm configuration.

7 future work

Our future works about MIPv6 are

- implement our new design
- make racoon support MIPv6
- NEMO
- Multihome
- vertical hand-over

Additionally we consider that we should improve or change stackable destination itself because stackable destination runs after building a packet. Thus, IPv6 packet processing is not efficient itself because an IPv6 packet has some extension header and the order of headers is not always same as the order of process so that every process searches correct point on a packet from the head. We should improve its packet processing with keeping xfrm architecture and cache mechanism.

References

[1] S. Deering and R. Hinden. Internet Protocol, Version 6 Specification. RFC2460, December 1998.

[2] GO/Core Project. MIPL Mobile IPv6 for Linux. `http://www.mobile-ipv6.org`.

[3] IPsec Tools. IPsec Tools Web Page. `http://www.ipsec-tools.sourceforge.net/`.

[4] D. Johnson, C. Perkins, and J. Arkko. Mobility Support in IPv6. Work in Progress, June 2003.

[5] KAME Project. KAME Project Web Page. `http://www.kame.net`.

[6] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. RFC2401, November 1998.

[7] Kazunori Miyazawa, Hideaki Yoshifuji, and Yuji Sekiya. Linux IPv6 Networking—Past, Present, and Future. In *Proceedings of the Linux Symposium*, Ottawa, July 2003.

[8] USAGI Project. USAGI Project Web Page. `http://www.linux-ipv6.org`.

Getting X Off the Hardware

Keith Packard
Hewlett-Packard
Cambridge Research Laboratory
keithp@keithp.com

Abstract

The X window system is generally implemented by directly inserting hardware manipulation code into the X server. Mode selection and 2D acceleration code are often executed in user mode and directly communicate with the hardware. The current architecture provides for separate 2D and 3D acceleration code, with the 2D code executed within the X server and the 3D code directly executed by the application, partially in user space and partially in the kernel. Video mode selection remains within the X server, creating an artificial dependency for 3D graphics on the correct operation of the window system. This paper lays out an alternative structure for X within the Linux environment where the responsibility for acceleration lies entirely within the existing 3D user/kernel library, the mode selection is delegated to an external library and the X server becomes a simple application layered on top of both of these. Various technical issues related to this architecture along with a discussion of input device handling will be discussed.

1 History

The X11[SG92] server architecture was designed assuming significant operating assistance for supporting input and output devices. How that has changed over the years will inform the discussion of the design direction proposed in this paper.

1.1 Original Architecture

One of the first 2D accelerated targets for X11 was the Digital QDSS (Dragon) board. The Dragon included a 1024x768 frame buffer with 4 or 8 bits for each pixel. The frame buffer was not addressable by the CPU, rather every graphics operation was performed by the coprocessor. The Dragon board had only a single video mode supporting the monitor supplied with the machine. A primitive terminal emulator in the kernel provided the text mode necessary to boot the machine.

Graphics commands to the processor were queued to a shared DMA buffer. The X server would block in the kernel waiting for space in the buffer when full. This is similar to the architecture used by the DRI project for accelerated 3D graphics today.

Keyboard and mouse support were provided by another shared memory queue between the kernel and X server. Abstract event structures were constructed by the kernel from the raw device data, timestamped and placed in the shared queue. A file descriptor would be signalled when new data were inserted to awaken the X server, and the X server could also directly examine the queue indices which were stored in the shared segment. This low-overhead queue polling was used by the X server to check for new input after every X re-

quest was executed to reduce input latency.

The hardware sprite was handled in the kernel; its movement was directly connected with the mouse driver so that it could be moved at interrupt time, leading to a responsive pointer even in the face of high CPU load within the X server and other applications. The keyboard controller managed the transition from ASCII console mode to key-transition X mode internally; abnormal termination of the X server would leave the underlying console session working normally.

1.2 The Slippery Slope

Early Sun workstations had unaccelerated frame buffers. Like the QDSS above, they used fixed monitors and had no need to support multiple video modes. As the hardware advanced, they did actually gain programmable timing hardware, but that was not configurable from the user mode applications.

The X server simply mapped the frame buffer into its address space and manipulated the pixel values directly. Around 1990, Sun shipped the cgsix frame buffer which included an accelerator. Unlike the QDSS, the cgsix frame buffer could be mapped by the CPU, and the accelerator documentation was not published by Sun. X11R4 included support for this card as a simple dumb frame buffer. As CPU access to the frame buffer was slower than with Sun's earlier unaccelerated frame buffers, the result was a much slower display.

By disassembling the provided SunWindows driver, the author was able to construct an accelerated X driver for X11R5 entirely in user mode. This driver could not block waiting for the accelerator to finish, rather it would spin, polling the accelerator until it indicated it was idle.

Keyboard and mouse support were provided by the kernel as files from which events could be read. The lack of any shared memory mechanism to signal available input meant that the original driver would not notice input events until the X server polled the kernel, something which could take significant time. As there was no kernel support for the pointer sprite, the X server was responsible for updating it as well, leading to poor mouse tracking when the CPU was busy.

To ameliorate the poor mouse tracking, the X server was modified to receive a signal when input was present on the file descriptors and immediately process the input. When supported, the hardware sprite would also be moved at this time, leading to dramatically improved tracking performance. Still, the fact that the X server itself was responsible for connecting the mouse motion to the sprite location meant that under high CPU load, the sprite would noticeably lag the mouse.

Kernel support for the keyboard consisted of a special mode setting which would transform the keyboard from an ASCII input device to reporting raw key transition events. Because the kernel didn't track what state the keyboard was in, the X server had to carefully reset the keyboard on exit back to ASCII mode or the user would no longer be able to interact with the console.

Placing the entire graphics driver in user mode eliminated the need to write a kernel driver, but marginalized overall system performance by forcing the CPU to busy-wait for the graphics engine. Placing responsibility for manging the sprite led to poor tracking, while requiring the X server to always reset the keyboard mode frequently resulted in an unusable system when X terminated abnormally.

Fixing the kernel to address these problems was never even considered; the problems didn't prevent the system from functioning, they only

made it less than ideal.

1.3 The Dancing Bear

With widespread availability of commodity 386-based PC hardware, numerous vendors began shipping Unix (and Unix-like) operating systems for them. These originally did not include the X window system. A disparate group of users ported X to these systems without any support from the operating system vendors.

That these users managed to get X running on the early 386 hardware was an impressive feat, but that they had to do everything without any kernel support only increased the difficulty.

Early PC graphics cards were simple frame buffers as far as graphics operations went, but configuring them to generate correct video timings was far from simple. Because monitors varied greatly, each graphics card could be programmed to generate many different video timings. Incorrect timings could destroy the monitor.

Keyboard support in these early 386-based Unix systems was very much like the Sun operating system; the keyboard was essentially a serial device and could be placed in a mode which translated key transitions into ASCII or placed a mode which would report the raw bytes emitted from the keyboard.

The X server would read these raw bytes and convert them to X events. Again, there was latency here as the X server would not process them except when polling for input across all X clients and input devices. As with the Sun driver, if the X server terminated without switching the keyboard back to translated mode, it would not be usable by the console. This particular problem was eventually fixed in some kernels by adding special key sequences to reset the keyboard to translated mode.

Mouse support really was just a kernel serial driver—PS/2 mice didn't exist, and so bus and serial mice were used. The X server itself would open the device, configure the communication parameters and parse the stream of bytes. As there was no hardware sprite support, the X server would also have to draw the cursor on the screen; that operation had to be synchronized with rendering and so would be delayed until the server was idle.

Because the X server itself was managing video mode configuration, an abnormal X server termination would leave the video card misconfigured and unusable as the console. Similarly, the keyboard driver would be left in untranslated mode, so the user couldn't even operate the computer blind to reboot.

This caused the X server to assume the same reliability requirements as the operating system kernel itself; bugs in the X server would render the system just as unusable as bugs in the kernel.

1.4 The Pit of Despair

With the addition of graphics acceleration to the x86 environment, the X server extended its user-mode operations to include manipulation of the accelerator. As with the Sun GX driver described above, these drivers included no kernel support and were forced to busy-wait for the hardware.

However, unlike the GX hardware, PC graphics hardware would often tie down the PCI bus while transferring data between the CPU and the graphics card. Incorrect manipulation of the hardware would result in the PCI bus locking and the system not even responding to network or disk activity. Unlike the simple keyboard translation problem described above, this cannot be be fixed in the operating system.

Because the graphics devices had no kernel

driver support, there was no operating system management of their address space mappings. If the BIOS included with the system incorrectly mapped the graphics device, it fell to the X server to repair the PCI mapping spaces. Manipulating the PCI address configuration from a user-mode application would work only on systems without any dynamic management within the kernel.

If the machine included multiple graphics devices controlled through the standard VGA addresses, the X server would need to manipulate these PCI mappings on the fly to address the active card.

The overall goal was not to build the best system possible, but rather to make the code as portable as possible, even in the face of obviously incorrect system architecture.

1.5 A Glimmer of Hope

The Mesa project started as a software-only rasterizer for the OpenGL API. By providing a freely available implementation of this widely accepted API, people could run 3D applications on every machine, even those without custom 3D acceleration hardware. Of course, performance was a significant problem, especially as the 3D world moved from simple colored polygons to textures and complex lighting environments.

The Mesa developers started adding hardware support for the few cards for which documentation was available. At first, these were whole-screen drivers, but eventually the DRI project was started to support multiple 3D applications integrated into the X window system. Because of the desire to support secure direct rendering from multiple unprivileged applications, the DRI project had to include a kernel driver. That driver could manage device mappings, DMA and interrupt logic and even clean up the hardware when applications terminated abnormally.

The result is a system which is stable in the face of broken applications, and provides high performance and low CPU overhead.

However, the DRI environment remains reliant on the X server to manage video mode selection and basic device input.

2 Forward to the Past

Given the dramatic changes in system architecture and performance characteristics since the original user-mode X server architecture was promulgated, it makes sense to look at how the system should be constructed from the ground up. Questions about where support for each operation should live will be addressed in turn, first starting with graphics acceleration, then video mode selection and finally (and most briefly) input devices.

3 Graphics Acceleration

X has always directly accessed the lowest levels of the system to accelerate 2D graphics. Even on the QDSS, it constructed the register-level instructions within the X server itself. With the inclusion of OpenGL[SAe99] 3D graphics in some systems, the system requires two separate graphics drivers, one for the X server operating strictly in 2D mode and the other inside the GL library for 3D operations. Improvements to the 3D support have no effect on 2D performance.

As a demonstration of how effectively OpenGL can implement the existing X server graphics operations, Peter Nilsson and David Reveman implemented the Glitz library[NR04] which supports the Render[Pac01] API on top of the OpenGL API. In a few months, they managed

to provide dramatic acceleration for the Cairo graphics library[WP03] on any hardware with an OpenGL implementation. In contrast, the Render implementation within the X sample server using custom 2D drivers has never seen significant acceleration, even three and a half years after the extension was originally designed. Only a few drivers include even half-hearted attempts at acceleration.

The goal here is to have the X server use the OpenGL API for all graphics operations. Eliminating the custom 2D acceleration code will reduce the development burden. Using accelerated OpenGL drivers will provide dramatic performance improvements for important operations now ill-supported in existing X drivers. Work in this area will depend on the availability of stand-alone OpenGL drivers that work in the absence of an underlying window system. Fortunately, the Mesa project is busy developing the necessary infrastructure. Meanwhile, development can progress apace using the existing window-system dependent implementations, with the result that another X server is run just to configure the graphics hardware and set up the GL environment.

For cards without complete OpenGL acceleration, the desired goal is to provide DRI-like kernel functionality to support DMA and interrupts to enable efficient implementation of whatever useful operations the card does support. For 2D graphics, the operations needing acceleration are those limited by memory bandwidth—large area fills and copies. In particular acceleration of image composition results in dramatic performance improvements with minimal amounts of code. The spectacular amounts of code written in the past that provide modest acceleration for corner cases in the X protocol should be removed and those cases left to software to minimize driver implementation effort.

This architecture has been implemented by Eric Anholt in his kdrive-based Xati server[Anh04]. Using the existing DRI driver for the Radeon graphics card, he developed a 2D X driver with reasonable acceleration for common operations, including significant portions of the X render API. The driver uses only a small fraction of the Radeon DRI driver, a significantly smaller kernel driver would suffice for a ground-up implementation.

In summary, graphics cards should be supported in one of two ways:

1. With an OpenGL-based X server
2. With a 2D-only X server based on a simple loadable driver API.

3.1 Implications for Applications

None of the architectural decisions about the internal X server architecture change the nature of the existing X and Render APIs as the fundamental 2D interface for applications. Applications using the existing APIs will simply find them more efficient when the X server provides a better implementation for them. This means that applications needn't migrate to non-X APIs to gain access to reasonable acceleration.

However, applications that wish to use OpenGL should find a wider range of supported hardware as driver writers are given the choice of writing either an OpenGL or 2D driver, and aren't faced with the necessity of starting with a 2D driver just to support X.

In any case, use of the cairo graphics library provides insulation from this decision as it supports X and GL requiring only modest changes in initialization to select between them.

4 Video Mode Configuration

The area of video mode selection involves many different projects and interests; one significant goal of this discussion is to identify which areas are relevant to X and how those can be separated from the larger project.

4.1 Overview of the Problem

Back in 1984 when X was designed, graphics devices were fundamentally fixed in their relationship with the attached monitor. The hardware would be carefully designed to emit video timings compatible with the included monitor; there was no provision for adjusting video timings to adapt to different monitors, each video card had a single monitor connector.

Fast forward to 2004 when common video cards have two or more monitor connectors along with outputs for standard NTSC, SECAM, or PAL video formats. The desire to dynamically adjust the display environment to accommodate different use modes is well supported within the Macintosh and Microsoft environments, but the X window system has remained largely stuck with its 1984 legacy.

4.2 X Attempts to Fix Things

X servers for PC operating systems adapted to simple video mode selection by creating a 'virtual' desktop at least as large as the largest desired mode and making the current mode view a subset of that, panning the display around to keep the mouse on the screen. For users able to accept this metaphor, this provided usable, if less than ideal support. Most of the time, however, having content off of the screen which could only be reached by moving the mouse was confusing. To help address this, the X Resize and Rotate extension (RandR)[GP01] was designed to notify applications of changes in the pixel size of the screen and allow programmatic selection among available video modes.

The RandR extension solved the simple single monitor case well enough, even permitting the set of available modes to change on the fly as monitors were switched. However, it failed to address the wider problem of supporting multiple different video outputs and the dynamic manipulation of content between them.

Statically, the X server can address each video output correctly and even select between a large display spanning a collection of outputs or separate displays on each video screen. However, there is no capability to adjust these configurations dynamically, nor even to automatically adapt to detected changes in the environment.

4.3 X is Only Part of the Universe

With 2D performance no longer a significant marketing tool, graphics hardware vendors have been focusing instead on differentiating their products based on video output (and input) capabilities. This has dramatically extended the options available to the user, and increased the support necessary within the operating system.

As the suite of possible video configuration options continues to expand, it seems impossible to construct a fixed, standard X extension capable of addressing all present and future needs. Therefore, a fully capable mechanism must provide some "back door" through which display drivers and user agents can communicate information about the video environment which is not directly relevant to the window system or applications running within it.

One other problem with the current environment is that video mode selection is not a requirement unique to the X window system. Numerous other graphical systems exist which

are all dependent on this code. Currently, that is implemented separately for each video card supported by each system. The MxN combination of graphics systems and video cards means that only a few systems have support for a wide range of video cards. Support for systems aside from X is pretty sparse.

4.4 Who's in Charge Here, Anyway?

X itself places relatively modest demands on the system. The X server needs to be aware of what video cards are available, what video modes are available for each card and how to select the current mode. Within that mode there may be a wealth of information that is not relevant to the X server; it really only needs to know the pixel dimensions of each frame buffer, the physical dimension of pixels on each monitor and the geometric relationship among monitors. Details about which video port are in use, or how the various ports relate to the frame buffer are not important. Information about video input mechanisms are even less relevant.

As the X server need have no way of interpreting the complexity of the video mode environment, it should have no role in managing it. Rather, an external system should assume complete control and let the X server interact in its own simple way.

This external system could be implemented partially in the kernel and partially in usermode. Doing this would allow the kernel to share the same logic for video mode selection during boot time for systems which don't automatically configure the video card suitably on power-on. In addition, alternate graphics systems would be able to share the same API for their own video mode configuration.

5 Input Device Support

In days of yore, the X environment supported exactly one kind of mouse and one (perhaps of an internationalized family) keyboard. Sadly, this is no longer the case. The wealth of available input devices has caused no small trouble in X configuration and management. Add to that the relative failure of the X Input extension to gain widespread acceptance in applications and the current environment is relegated to emulating that available in 1984.

5.1 Uniform Device Access

The first problem to attack is that of the current hodgepodge device support where the X server itself is responsible for parsing the raw bytestreams coming from the disparate input devices. Fortunately, the kernel has already solved that problem—the new `/dev/input`-based drivers provide a uniform description of devices and standard interface to all. Converting the X server over to those interfaces is straightforward.

However, the `/dev/input/mice` interface has a significant advantage in todays world; it unifies all mouse devices into a single stream so that the X server doesn't have to deal with devices that come and go. So, to switch input mechanisms, the X server must first learn to deal with that.

5.2 Hotplug and HAL

Mice (and even keyboards) can be easily attached and detached from the machine. With USB, the system is even automatically notified about the coming and going of devices. What is missing here is a way of getting that notification delivered to the X server, having the X server connect to the new device (when appropriate), notifying X applications about the

availability of the new device and integrating the devices events into the core pointer or keyboard event stream.

The Hardware Abstraction Layer (HAL)[Zeu] project is designed to act as an intermediary between the Linux Hotplug system and applications interested in following the state of devices connected to the machine. By interposing this mechanism, the complexity of discovering and selecting input devices for the X server can be moved into a separate system, leaving the X server with only the code necessary to read events from the devices specified by the HAL. One open question is whether this should be done by a direct connection between the X server and the HAL daemon or whether an X client could listen to HAL and transmit device state changes through the X protocol to the X server.

One additional change needed is to extend the X Input Extension to include notification of new and departed devices. That extension already permits the list of available devices to change over time, all that it lacks is the mechanism to notify applications when that occurs. Inside the X server implementation, the extension is in for some significantly more challenging changes as the current codebase assumes that the set of available devices is fixed at server initialization time.

6 Migrating Devices

With X was developed, each display consisted of a single keyboard and mouse along with a fixed set of monitors. That collection was used for a single login session, and the input devices never moved. All of that has now changed; input devices come and go, computers get plugged into video projectors, multiple users login to the same display. The dynamic nature of the modern environment requires some changes to the X protocol in the form of new or modified extensions.

6.1 Whose Mouse Is This?

Input devices are generally located in physical proximity to the related output device. In a system with multiple output devices and multiple input devices, there is no existing mechanism to identify which device is where. Perhaps some future hardware advance will include geographic information along with the bus topology.

The best we can probably do for now is to provide a mechanism to encode in the HAL database the logical grouping of input and output devices. That way the X server would receive from the HAL the set of devices to use at startup time and then accept ongoing changes in that as the system was reconfigured.

One problem with this simplistic approach is that it doesn't permit the migration of input devices from one grouping to another; one can easily imagine the user holding a wireless pointing device to attempt to interact with the "wrong" display. Some mechanism for dynamically reconfiguring the association database will need to be included.

6.2 Hotplugging Video Hardware

While most systems have no ability to add or remove graphics cards, it's not unheard of—many handheld computers support CF video adapters. On the other hand, nearly all systems do support "hotplugging" of the actual display device or devices. Many can even detect the presence or absence of a monitor enabling true auto-detection and automatic reconfiguration.

When a new monitor is connected, the X server needs to adapt its configuration to include it. In the case where the set of physical screens are

gathered together as a single logical screen, the change can be reflected by resizing that single screen as supported by the RandR extension. However, if each physical screen is exposed to applications as a separate logical screen, then the X server must somehow adapt to the presence of a new screen and report that information to applications. This will require an extension.

In terms of the existing X server implementation, the changes are rather more dramatic. Again, it has some deep-seated assumptions that the set of hardware under its control will not change after startup. Fixing these will keep developers entertained for some time.

6.3 Virtual Terminal Switching

One capability Linux has had for a long time is the ability to rapidly switch among multiple sessions with "virtual terminals." The X server itself uses this to preserve a system console, running on a separate terminal ensures that the system console can be viewed by simply switching to the appropriate virtual terminal. Given this, multiple X servers can be started on the same hardware, each one on a different virtual terminal and rapidly switched among.

The virtual terminal mechanism manages only the primary graphics device and the system keyboard. Management of other graphics and input devices is purely by convention. The result is that multiple simultaneous X sessions are not easily supported by the standard build of the X server. The X server targeted at a non-primary graphics device needs to avoid configuring the virtual terminal. However, this also eliminates the ability for that device to support multiple sessions; there cannot be virtual terminal switching on a device which is not associated with any virtual terminals.

With the HAL providing some indication of which devices should be affiliated into a single session configuration, the X server can at least select them appropriately. Similarly, the X server should be able to detect which device is the console keyboard and manage virtual terminals from there. Whether the kernel needs to add support for virtual terminals on the other graphics/keyboard devices is not something X needs to answer.

The final problem is that of other input devices; when switching virtual terminals, the X server conventionally drops its connection to the other input devices, presuming that whatever other program is about to run will want to use the same ones. While that does work, it leaves open the possibility that an error in the X server will leave these devices connected and deny other applications access to them. Perhaps it would be better if the kernel was involved in the process and directing input among multiple consumers automatically as VT affiliation changed.

7 Conclusion

Adapting the X window system to work effectively and competently in the modern environment will take some significant changes in architecture, however throughout this process existing applications will continue to operate largely unaffected. If this were not true, the fundamental motivation for the ongoing existence of the window system would be in doubt.

Migrating responsibility for device management out of the X server and back where it belongs inside the kernel will allow for improvements in system stability, power management and correct operation in a dynamic environment. Performance of the resulting system should improve as the kernel can take better advantage of the hardware than is possible in user

mode.

Sharing graphics acceleration between 2D and 3D applications will reduce the effort needed to support new graphics hardware. Migrating the video mode selection will allow all graphics systems to take advantage of it. This should permit some interesting exploration in system architecture.

Significant work remains in defining the precise architecture of the kernel video drivers; these drivers need to support console operations, frame buffer device access and DRI (or other) 3D acceleration. Common memory allocation mechanism seem necessary, along with figuring out a reasonable division of labor between kernel and user mode for video mode selection.

Other work remains to resolve conflicts over sharing devices among multiple sessions and creating a mechanism for associating specific input and output devices together.

The resulting system regains much of the flavor of the original X11 server architecture. The overall picture of a system which provides hardware support at the right level in the architecture appears to have wide support among the relevant projects making the future prospects bright.

References

[Anh04] Eric Anholt. High Performance X Servers in the Kdrive Architecture. In *FREENIX Track, 2004 Usenix Annual Technical Conference*, Boston, MA, July 2004. USENIX.

[GP01] Jim Gettys and Keith Packard. The X Resize and Rotate Extension - RandR. In *FREENIX Track, 2001 Usenix Annual Technical Conference*, Boston, MA, June 2001. USENIX.

[NR04] Peter Nilsson and David Reveman. Glitz: Hardware Accelerated Image Compositing using OpenGL. In *FREENIX Track, 2004 Usenix Annual Technical Conference*, Boston, MA, July 2004. USENIX.

[Pac01] Keith Packard. Design and Implementation of the X Rendering Extension. In *FREENIX Track, 2001 Usenix Annual Technical Conference*, Boston, MA, June 2001. USENIX.

[SAe99] Mark Segal, Kurt Akeley, and Jon Leach (ed). *The OpenGL Graphics System: A Specification*. SGI, 1999.

[SG92] Robert W. Scheifler and James Gettys. *X Window System*. Digital Press, third edition, 1992.

[WP03] Carl Worth and Keith Packard. Xr: Cross-device Rendering for Vector Graphics. In *Proceedings of the Ottawa Linux Symposium*, Ottawa, ON, July 2003. OLS.

[Zeu] David Zeuthen. HAL Specification 0.2. `http://freedesktop.org/~david/hal-0.2/spec/hal-spec.html`.

Linux 2.6 performance improvement through readahead optimization

Ram Pai
IBM Corporation
linuxram@us.ibm.com

Badari Pulavarty
IBM Corporation
badari@us.ibm.com

Mingming Cao
IBM Corporation
mcao@us.ibm.com

Abstract

Readahead design is one of the crucial aspects of filesystem performance. In this paper, we analyze and identify the bottlenecks in the redesigned Linux 2.6 readahead code. Through various benchmarks we identify that 2.6 readahead design handles database workloads inefficiently. We discuss various improvements made to the 2.6 readahead design and their performance implications. These modifications resulted in impressive performance improvements ranging from 25%–100% with various benchmarks. We also take a closer look at our modified 2.6 readahead algorithm and discuss current issues and future improvements.

1 Introduction

Consider an application that reads data sequentially in some fixed-size chunks. The kernel reads data sufficiently enough to satisfy the request from the backing storage and hands it over to the application. In the meantime the application ends up waiting for the data to arrive from the backing store. The next request also takes the same amount of time. This is quite inefficient. What if the kernel anticipated the future requests and cached more data? If it could do so, the next read request could be satisfied much faster, decreasing the overall read latency.

Like all other operating systems, Linux uses this technique called *readahead* to improve read throughput. Although readahead is a great mechanism for improving sequential reads, it can hurt the system performance if used blindly for random reads.

We studied the performance of the readahead algorithm implemented in 2.6.0 and noticed the following behavior for large random read requests.

1. reads smaller chunks of data many times, instead of reading the required size chunk of data once.
2. reads more data than required and hence wasted resources.

In Section 2, we discuss the readahead algorithm implemented in 2.6 and identify and fix the inefficient behavior. We explain the performance benefits achieved through these fixes in Section 3. Finally, we list the limitations of our fixes in Section 4.

2 Readahead Algorithm in 2.6

2.1 Goal

Our initial investigation showed the performance on Linux 2.6 of the Decision Support System (DSS) benchmark on filesystem was

about 58% of the same benchmark run on raw devices. Note that the DSS workload is characterized by large-size random reads. In general, other micro-benchmarks like rawio-bench and aio-stress showed degraded performance with random workloads. The suboptimal readahead behavior contributed significantly toward degraded performance. With these inputs, we set the following goals.

1. Exceed the performance of 2.4 large random workloads.
2. DSS workload on filesystem performs at least 75% as well as the same on raw devices.
3. Maintain or exceed sequential read performance.

2.2 Introduction to the 2.6 readahead algorithm

Figure 1 presents the behavior of 2.6.0 readahead. The `current_window` holds pages that satisfy the current requests. The `readahead_window` holds pages that satisfy the anticipated future request. As more page requests are satisfied by the `current_window` the estimated size of the next `readahead_window` expands. And if page requests miss the `current_window` the estimated size of the `readahead_window` shrinks. As soon as the read request cross `current_window` boundary and steps into the first page of the `readahead_window`, the `readahead_window` becomes the `current_window` and the `readahead_window` is reset. However, if the requested page misses any page in the `current_window` and also the first page in the `readahead_window`, both the `current_window` and the `readahead_window` are reset and a new set of pages are read into the `current_window`. The number of pages read in the current window depends upon the estimated size of the `readahead_window`. If the estimated size of the `readahead_window` drop down to zero, the algorithm stops reading ahead, and enters the slow-read mode till page request pattern become sufficiently contiguous. Once the request pattern become sufficiently contiguous the algorithm re-enters into readahead-mode.

2.3 Optimization For Random Workload

We developed a user-level simulator program that mimicked the behavior of the above readahead algorithm. Using this program we studied the read patterns generated by the algorithm in response to the application's read request pattern.

In the next few subsections we identify the bottlenecks, provide fixes and then explain the results of the fix. As a running example we use a read sequence consisting of 100 random read-requests each of size 16 pages.

2.3.1 First Miss

Using the above read pattern, we noticed that the readahead algorithm generated 1600 requests of size one page. The algorithm penalized the application by shutting down readahead immediately, for not reading from the beginning of the file. It is sub-optimal to assume that application's read pattern is random, just because it did not read the file from the beginning. The offending code is at line 16 in Figure 1. Once shut down, the slow-read mode made readahead to not resume since the `current_window` never becomes large enough. For the ext2/ext3 filesystem, the `current_window` must become 32 pages large, for readahead to resume. Since the application's requests were all 16 pages large, the `current_window` never opened. We re-

```
1 for each page in the current request
2 do
3       if readahead is shutdown
4       then  // read one page at a time (SLOW-READ MODE)
5         if requested page is next to the previously requested page
6         then
7              open the current_window by one more page
8         else
9              close the current_window entirely
10        fi

11        if the current_window opens up by maximum readahead_size
12        then
13             activate readahead  // enter READAHEAD-MODE
14        fi
15        read in the requested page

        else  // read many pages at a time (READAHEAD MODE)
16        if this is the first read request and is for the first page
                          of this open file instance
17              set the estimated readahead_size to half the size of
18                      maximum readahead_size
19        fi

20        if the requested page is within the current_window
21              increase the estimated readahead_size by 2
22              ensure that this size does not exceed maximum
23                       readahead_size
24        else
25              decrease the estimated readahead_size by 2
26              if this estimate becomes zero, shutdown readahead
27        fi

28        if the requested page is the first page in the readahead_window
29        then
30               move the pages in the readahead_window to the
31                     current_window and reset the readahead_window
32               continue
33        fi
34
35        if the requested page is not in the current_window
36        then
37              delete all the page in current_window and readahead_window
38              read the estimated number of readahead pages starting
39                   from the requested page and place them into the current
40                   window.
41              if all these pages already reside in the page cache
42              then
43                   shrink the estimated readahead_size by 1 and
                     shutdown readahead if the estimate touches zero
44              fi
45            else if the readahead_window is reset
46                 then
47                       read the estimated number of readahead pages
48                       starting from the page adjacent to the last page
49                       in the current window and place them in the
50                       readahead_window.
51                       if all these pages already reside in the page cache
52                       then
53                           shrink the estimated readahead_size by 1 and
                             shutdown readahead if the estimate touches zero
54                       fi
55                 fi
56        fi
57      fi
58 done
```

Figure 1: *Readahead algorithm in 2.6.0*

moved the check at line 16 to not expect read access to start from the beginning.

For the same read pattern the simulator showed 99 32-page requests, 99 30-page requests, one 16-page request, and one 18-page request to the block layer. This was a significant improvement over 1600 1-page requests seen without these changes.

However, the DSS workload did not show any significant improvement.

2.3.2 First Hit

The reason why DSS workload did not show significant improvement was that readahead shut down because the accessed pages already resided in the page-cache. This behavior is partly correct by design, because there is no advantage in reading ahead if all the required pages are available in the cache. The corresponding code is at line 43. But shutting down readahead by just confirming that the initial few pages are in the page-cache and assuming that future pages will also be in the page cache, leads to worse performance. We fixed the behavior, to not close the `readahead_window` the first time, even if all the requested pages were in the page-cache. The combination of the above two changes ensured continuous large-size read activity.

The simulator showed the same results as the First-Miss fix.

However, the DSS workload showed 6% improvement.

2.3.3 Extremely Slow Slow-read Mode

We also observed that the slow-read mode of the algorithm expected 32 contiguous page access to resume large size reads. This is not a realistic expectation for random workload. Hence, we changed the behavior at line 9 to shrink the `current_window` by one page if it lost contiguity.

The simulator and DSS workload did not show any better results because the combination of First-Hit and First-Miss fixes ensured that the algorithm did not switch to the slow-read mode. However a request pattern comprising of 10 single page random requests followed by a continuous stream of 4-page random requests can certainly see the benefits of this optimization.

2.3.4 Upfront Readahead

Note that readahead is triggered as soon as some page is accessed in the `current_window`. For random workloads, this is not ideal because none of the pages in the `readahead_window` are accessed. We changed line 45, to ensure that the readahead is triggered only when the last page in the `current_window` is accessed. Essentially, the algorithm waits until the last page in the `current_window` is accessed. This increases the probability that the pages in the `readahead_window` if brought in, will get used.

With these changes, the simulator generated 99 30-page requests, one 32-page request, and one 16-page request.

There was a significant 16% increase in performance with the DSS workload.

2.3.5 Large `current_window`

Ideally, the readahead algorithm must generate around 100 16-page requests. Observe however that almost all the page re-

quests are of size 30 pages. When the algorithm observes that a page request has missed the `current_window`, it scraps both the `current_window` and the `readahead_window`, if one exists. It ends up reading in a new `current_window`, whose size is based on the estimated `readahead_size`. Since all of the pages in a given application's read request are contiguous, the estimated readahead size tends to reach the maximum `readahead_size`. Hence, the size of the new `current_window` is too large; most of the pages in the window tend to be wasted. We ensured that the new `current_window` is as large as the number of pages that were used in the present `current_window`.

With this change, the simulator generated 100 16-page requests, and 100 32-page requests. These results are awful because the last page of the application's request almost always coincides with the last page of the `current_window`. Hence, the readahead is triggered when the last page of the `current_window` is accessed, only to be scrapped.

We further modified the design to read the new `current_window` with one more page than the number of pages accessed in the present `current_window`.

With this change, the simulator for the same read pattern generated 99 17-page requests, one 32-page request, and one 16-page request to the block layer, which is close to ideal!

The DSS workload showed another 4% better performance.

The collective changes were:

1. Miss fix: Do not close readahead if the first access to the file-instance does not start from offset zero.

2. Hit fix: Do not close readahead if the first access to the requested pages are already found in the page cache.

3. Slow-read Fix: In the slow-read path, reduce one page from the `current_window` if the request is not contiguous.

4. Lazy-read: Defer reading the `readahead_window` until the last page in the `current_window` is accessed.

5. Large `current_window` fix: Read one page more than the number of pages accessed in the current window if the request misses the current window.

These collective changes resulted in an impressive 26% performance boost on DSS workload.

2.4 Sequential Workload

The previously described modifications were not without side effects! The sequential workload was badly effected. Trond Myklebust reported 10 times worse performance on sequential reads using the iozone benchmark on an NFS based filesystem. The lazy read optimization broke the pipeline effect designed for sequential workload. For sequential workload, readahead must be triggered as soon as some page in the current window is accessed. The application can crunch through pages in the `current_window` as the new pages get loaded in the `readahead_window`.

The key observation is that upfront readahead helps sequential workload and lazy readahead helps random workload. We developed logic that tracked the average size of the read requests. If the average size is larger than the maximum readahead size, we treat that workload as sequential and adapt the algorithm to do upfront readahead. However, if the average size is less than the maximum `readahead_`

```
1 for each page in the current request ; do
3       if readahead is shutdown
4       then // read one page at a time (SLOW-READ MODE)
5            if requested page is next to the previously requested page
6            then
7                 open the current_window by one more page
8            else
9                 shrink current_window by one page
10           fi
11           if the current_window opens up by maximum readahead_size
12           then
13                  activate readahead  // enter READAHEAD-MODE
14           fi
15           read in the requested page
       else // read many pages at a time (READAHEAD MODE)
16-17        if this is the first read request for this open file-instance ; then
18              set the estimated readahead_size to half the size of maximum readahead_size
19           fi
20           if the requested page is within the current_window
21              increase the estimated readahead_size by 2
22              ensure that this size does not exceed maximum readahead_size
23           else
24               decrease the estimated readahead_size by 2
25               if this estimate becomes zero, shutdown readahead
26           fi
27           if requested page is contiguous to the previously requested page
28           then
29              Increase the size of the present read request by one more page.
30           else
31               Update the average size of the reads with the size of the previous request.
32           fi
33           if the requested page is the first page in the readahead_window
34           then
35                move the pages in current_window to the  readahead_window
36                reset readahead_window
37                continue
38           fi
39-40        if the requested page is not in the current_window ; then
41              delete all pages in current_window and readahead_window
42              if this is not the first access to this file-instance
43              then
44                 set the estimated number of readahead pages to the
                   average size of the read requests.
45              fi
46              read the estimated number of readahead pages starting from
                    the requested page and place them into the current window.
47              if this not the first access to this file instance and
                      all these pages already reside in the page cache
48              then
49                    shrink the estimated readahead_size by 1 and
                            shutdown readahead if the estimate touches zero
50              fi
51           else if the readahead_window is reset and if the average
                    size of the reads is above the maximum readahead_size
52                then
53                     read the readahead_window with the estimated
54                       number of readahead pages starting from the
55                       page adjacent to the last page in the current window.
56                       if all these pages already reside in the page cache
57                       then
58                           shrink the estimated readahead_size by 1 and
                             shutdown readahead if the estimate touches zero
59                       fi
60                fi
61           fi
62-63    fi ; done
```

Figure 2: *Optimized Readahead algorithm*

`size`, we treat that workload as random and adapt the algorithm to do lazy readahead.

This adaptive-readahead fixed the regression seen with sequential workload while sustaining the performance gains of random workload.

Also we ran a sequential read pattern through the simulator and found that it generated large size upfront readahead. For large random workload it hardly read ahead.

2.4.1 Simplification

Andrew Morton rightly noted that reading an extra page in the `current_window` to avoid lazy-readahead was not elegant. Why have lazy-readahead and also try to avoid lazy-readahead by reading one extra page? The logic is convoluted. We simplified the logic through the following modifications.

1. Read ahead only when the average size of the read request exceeds the maximum `readahead_size`. This helped the sequential workload.
2. When the requested page is not in the `current_window`, replace the `current_window`, with a new `current_window` the size of which is equal to the average size of the application's read request.

This simplification produced another percent gain in DSS performance, by trimming down the `current_window` size by a page. More significantly the sequential performance returned back to initial levels. We ran the above modified algorithm on the simulator with various kinds of workload and got close to ideal request patterns submitted to the block layer.

To summarize, the new readahead algorithm has the following modifications.

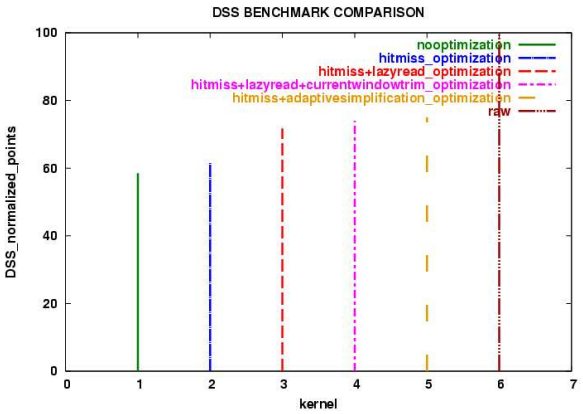


Figure 3: *Progressive improvement in DSS benchmark, normalized with respect to the performance of DSS on raw devices.*

1. Miss fix: Do not close readahead if the first access to the file-instance does not start from offset zero.
2. Hit fix: Do not close readahead if the first access to the requested pages are already found in the page cache.
3. Slow-read Fix: Decrement one page from the `current_window` if the request is not contiguous in the slow-read path.
4. Adaptive readahead: Keep a running count of the average size of the application's read requests. If the average size is above the maximum `readahead_size`, readahead up front. If the request misses the `current_window`, replace it with a new `current_window` whose size is the average size of the application's read requests.

Figure 2 shows the new algorithm with all the optimization incorporated.

Figure 3 illustrates the normalized steady increase in the DSS workload performance with each incremental optimization. The graph is normalized with respect to the performance of

DSS on raw devices. Column 1 is the base performance on filesystem. Column 2 is the performance on filesystem with the hit, miss and slow-read optimization. Column 3 is the performance on filesystem with first-hit, first-miss, slow-read and lazy-read optimization. Column 4 is the performance on filesystem with first-hit, first-miss, slow-read, and large `current_window` optimization. Column 5 is the performance on filesystem with first-hit, first-miss, slow-read, and adaptive read simplification. Column 6 is the performance on raw device.

3 Overall Performance Results

In this section we summarize the results collected through simulator, DSS workload, and iozone benchmark.

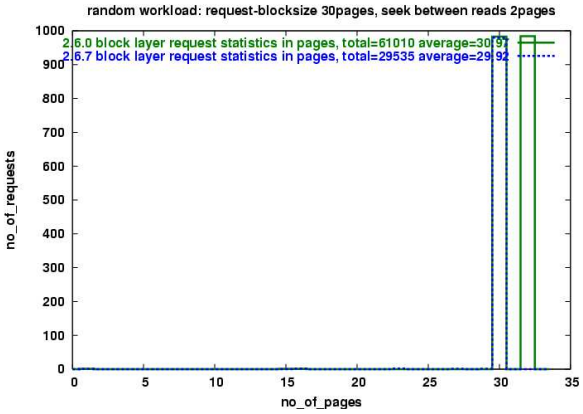
3.1 Results Seen Through Simulator

We generated different types of input read patterns. There is no particular reason behind these particular read pattern. However, we ensured that we get enough coverage. Overall the read requests generated by our optimized readahead algorithm outperformed the original algorithm. The graphs refer to our optimized algorithm as 2.6.7 because all these optimizations are merged in the 2.6.7 release candidate.

Figure 4 shows the output of readahead algorithm with and without optimization for 30-page read request followed by 2-page seek, repeated 984 times.

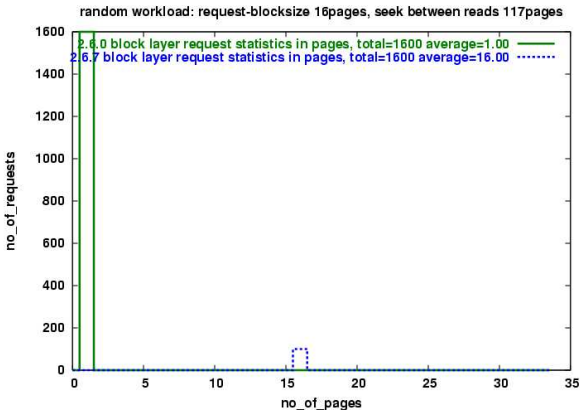
Figure 5 shows the output of readahead algorithm with and without optimization for 16-page read request followed by 117-page seek, repeated 100 times.

Figure 6 shows the output of readahead algorithm with and without optimization for 32-



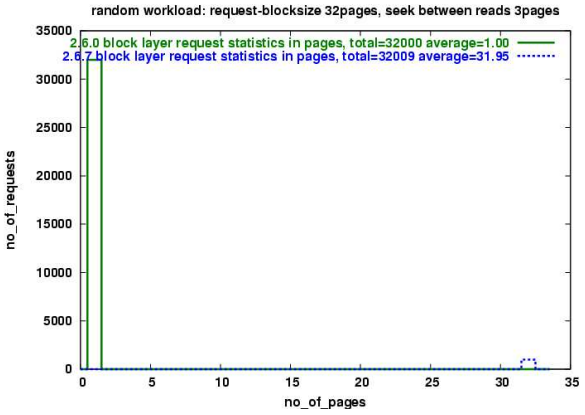
	2.6.0	2.6.7
Average Size	31	30
Pages Read	61010	29535
Wasted Pages	31490	15
No Of Read Requests	1970	987

Figure 4: *Application generates 30-page read request followed by 2-page seek, repeating 984 times. Totally 29520 pages requested.*



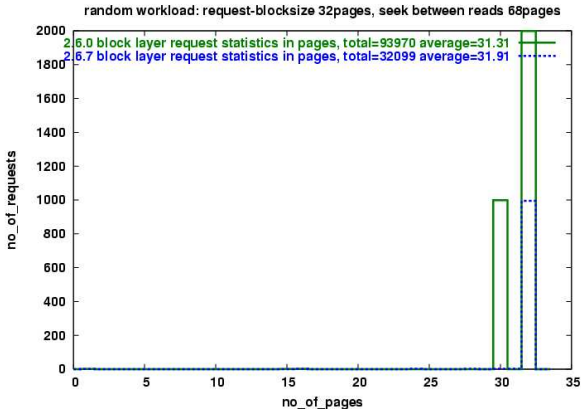
	2.6.0	2.6.7
Average Size	1	16
Pages Read	1600	1600
Wasted Pages	0	0
No Of Read Requests	1600	100

Figure 5: *Application generates 16-page read request followed by 117-page seek, repeating 100 times. Totally 1600 pages requested.*



	2.6.0	2.6.7
Average Size	1	31.95
Pages Read	32000	32009
Wasted Pages	0	9
No Of Read Requests	32000	1002

Figure 6: *Application generates 32-page read request followed by 3-page seek, repeating 1000 times. Totally 32000 pages requested.*



	2.6.0	2.6.7
Average Size	31.31	31.91
Pages Read	93970	32099
Wasted Pages	61970	99
No Of Read Requests	3001	1006

Figure 7: *Application generates 32-page read request followed by 68-page seek, repeating 1000 times. Totally 32000 pages requested.*

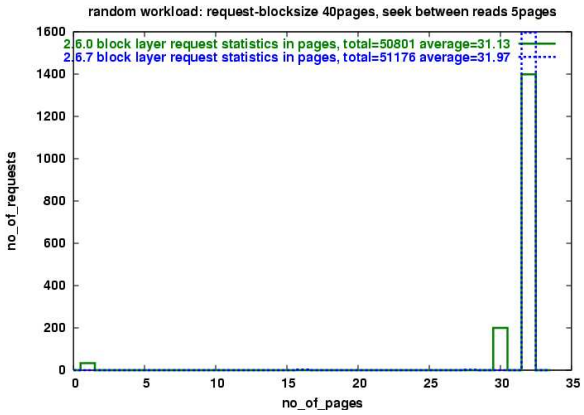
page read request followed by 3-page seek, repeated 1000 times.

Figure 7 shows the output of readahead algorithm with and without optimization for 32-page read request followed by 68-page seek, repeated 1000 times.

Figure 8 shows the output of readahead algorithm with and without optimization for 40-page read request followed by 5-page seek, repeated 1000 times.

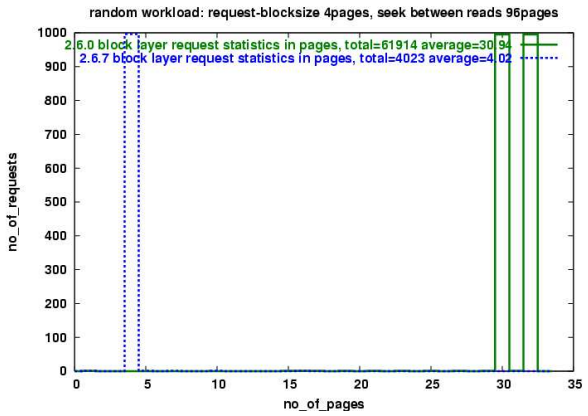
Figure 9 shows the output of readahead algorithm with and without optimization for 4-page read request followed by 96-page seek, repeated 1000 times.

Figure 10 shows the output of readahead algorithm with and without optimization for 16-page read request followed by 0-page seek, repeated 1000 times.



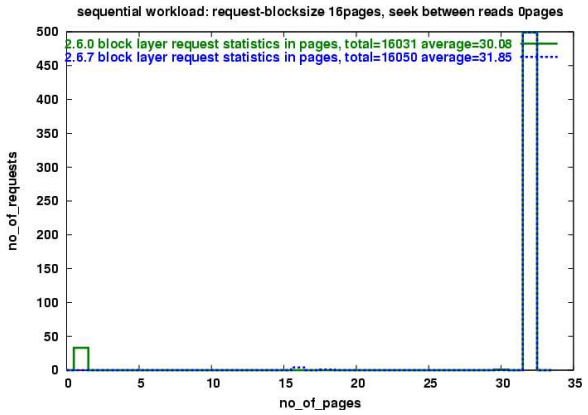
	2.6.0	2.6.7
Average Size	31.13	31.91
Pages Read	50810	51176
Wasted Pages	10801	11176
No Of Read Requests	1631	1601

Figure 8: *Application generates 40-page read request followed by 5-page seek, repeating 1000 times. Totally 40000 pages requested.*



	2.6.0	2.6.7
Average Size	30.94	4.02
Pages Read	61914	4023
Wasted Pages	57914	23
No Of Read Requests	2001	1001

Figure 9: *Application generates 4-page read request followed by 96-page seek, repeating 1000 times. Totally 4000 pages requested.*



	2.6.0	2.6.7
Average Size	30.08	31.85
Pages Read	16031	16050
Wasted Pages	31	50
No Of Read Requests	533	504

Figure 10: *Application generates 16-page read request with no seek, repeating 1000 times. Totally 16000 pages requested.*

3.2 DSS Workload

The configuration of our setup is as follows:

- 8-way Pentium III machine.
- 4GB RAM
- 5 fiber-channel controllers connected to 50 disks.
- 250 partitions in total each containing a ext2 filesystem.
- 30GB Database is striped across all these filesystems. No filesystem contains more than one table.
- Workload is mostly read intensive, generating mostly large 256KB random reads.

With this setup we saw an impressive 26% increase in performance. The DSS workload on filesystems is roughly about 75% to DSS workload on raw disks. There is more work to do, although the bottlenecks may not necessarily be in the readahead algorithm.

3.3 Iozone Results

The iozone benchmark was run a NFS based filesystem. The command used was `iozone -c -t1 -s 4096m -r 128k`. This command creates one thread that reads a file of size 4194304 KB, generating reads of size 128 KB. The results in Table 1 show an impressive 100% improvement on random read workloads. However we do see 0.5% degradation with sequential read workload.

4 Future Work

There are a couple of concerns with the above optimizations. Firstly, we see a small 0.5%

Read Pattern	2.4.20	2.6.0	2.6.0 + optimization
Sequential Read	10846.87	14464.20	13614.49
Sequential Re-read	10865.39	14591.19	13715.94
Reverse Read	10340.34	10125.13	20138.83
Stride Read	10193.87	7210.96	14461.63
Random Read	10839.57	10056.49	19968.79
Random Mix Read	10779.17	10053.37	21565.43
Pread	10863.56	11703.76	13668.21

Table 1: *Iozone benchmark Throughput in KB/sec for different workloads.*

degradation with the sequential workload using the iozone benchmark. The optimized code assumes the given workload to be random to begin with, and then adapts to the workload depending on the read patterns. This behavior can slightly affect the sequential workload, since it takes a few initial sequential reads before the algorithm adapts and does upfront readahead.

The optimizations introduce a subtle change in behavior. The modified algorithm does not correctly handle inherently-sequential clustered read patterns. It wrongly thinks that such read patterns seek after every page-read. The original 2.6 algorithm did accommodate such patterns to some extent. Assume an application with 16 threads reading 16 contiguous pages in parallel, one per thread. Based on how the threads are scheduled, the read patterns could be some combination of those 16 pages. An example pattern could be 1,15,8,12,9,6,2,14,10,7,5,3,4,11,12,13. The original 2.6.0 readahead algorithm did not care which order the page requests came in as long as the pages were in the current-window. With the adaptive readahead, we expect the pages to be read exactly in sequential order.

Issues have been raised regularly that the readahead algorithm should consider the size of the current read request to make intelligent decisions. Currently, the readahead logic bases its readahead decision on the read patterns seen in the past, including the request for the current page without considering the size of the current request. This idea has merit and needs investigation. We probably can ensure that we at least read the requested number of pages if readahead has been shutdown because of page-misses.

5 Conclusion

This work has significantly improved random workloads, but we have not yet reached our goal. We believe we have squeezed as much as possible performance from the readahead algorithm, though there is some work to be done to improve some special case workloads, as mentioned in Section 4. There may be other subsystems that need to be profiled to identify bottlenecks. There is a lot more to do!

6 Legal Statement

I would hate user space locking if it weren't that sexy...

(fusyn+RTNPTL: The making of a real-time synchronization infrastructure for Linux)

`http://developer.osdl.org/dev/robustmutexes/`

Iñaky Pérez-González
inaky.perez-gonzalez@intel.com

Boris Hu
boris.hu@intel.com

Salwan Searty
salwan.searty@intel.com

Adam Li
adam.li@intel.com

David P. Howell
david.p.howell@intel.com

Linux OS & Technology Team, Intel Corporation

Abstract

Linux has seen a lot of new features and developments in the last years in order to accommodate better scalability, interactivity, response time and POSIX compliance. With these changes, Telecom developers began to get serious about using Linux and started porting their systems to it. By doing that they brought new usage models and needs to the community; and among those needs was support for threads, mutual exclusion, priority inversion protection and robust synchronization for mission critical and fault-proof systems on both timesharing and soft real-time environments. This paper describes our experiences trying to meet this need, the current state and where are we headed. We will detail how originally we tried to modify the futex code, but later found we had to abandon that in favor of a similar design based on a layered implementation. This implementation accommodates a kernel and user space locking and synchronization infrastructure that will meet the requirements of those applications needing to use and port complex multithreaded real-time code.

1 A look at the requirements

The Carrier Grade Working group, or CGL, was created under the auspices of the OSDL; it provides a meeting point for all parties who share an interest on Linux use for Telecom: network equipment vendors, Linux distributors and developers, carriers, etc.

It was in this forum where missing features were identified. Carrier Grade Linux needed good soft real-time[1] features, specially with multi-threaded programs. As well, it needed a common feature provided by Solaris' mutexes

[1]For short, we'll use real-time to refer to *soft* real-time.

that was not present in Linux: *robustness*.

This project was started to provide a kernel synchronization infrastructure (*fusyn*) with the indicated characteristics, as well as the proper modifications to the NPTL user space library (*RTNPTL*) for it to use the new infrastructure and provide the new features.

The basic immediate requirements could be summarized in:

- The infrastructure should provide the primitives needed by NPTL to support the following POSIX tags:
  - TPS: thread priority scheduling
  - TPI: priority inheritance in mutexes
  - TPP: priority protection in mutexes

  Or simply: *anything that is needed for soft-real time support*.
- The implementation should support robust mutexes similar to those of Solaris.
- The implementation should provide equivalent features at the kernel level for use by drivers and subsystems.

With this in mind, we aimed to satisfy the following detailed requirements:

1. mutexes and conditional variables must work according to real-time expectancies
   (a) All operations (lock, unlock, priority promotion and demotion, etc.) should be deterministic in time, and *O(1)* when possible (except of course, for waits).
   (b) The order of lock acquisition by waiters (in mutexes) and wake up (in conditional variables) has to be determined by the scheduling properties of each blocked task/thread.
   (c) Minimization of priority inversion (given the importance of this item, it will be treated in its own section):
      i. lock stealing: in SMP systems, during on the acquisition of the lock a lower priority thread can steal the lock from a higher priority thread.
      ii. when a high priority thread A is waiting for a lower priority owner B to relinquish the mutex and B is preempted by a medium priority thread C.
         A. priority protection
         B. priority inheritance
2. Robustness: when a mutex owner dies, the mutex switches to a *dead-owner* state and the first waiter gets ownership with a special error code.
3. Uncontested locks/unlocks must happen without kernel intervention.
4. Deadlock detection

As well, in order to provide the benefits of this infrastructure to all the levels of a Linux system, it must be possible to use it not only by the user space code, but also by the kernel code.

1.1 The real time expectancies

Real-time is all about being **deterministic**, so all algorithm execution times need to be as predictable or bounded as possible. Using $O(1)$ algorithms helps with this [2].

[2] It is possible to be deterministic with a $O(f(N))$ operation, as long as $f(N)$ is known; however, in most, if not all, of the cases involving mutex operation, it is highly impractical or plainly impossible to find out $f(N)$, and thus a possibly simpler implementation has to be replaced with one potentially more complex, but $O(1)$.

POSIX dictates that upon unlock of a mutex, the scheduling policy shall determine who is the next owner. An obvious way of doing this would be to wake up all of the waiters and let them compete for the lock–the scheduler would determine that the highest priority task would get there first.

However, this causes scheduling storms, unnecessary context switches and general avoidable overhead. It is easier and more effective to determine which is the highest priority waiter and only wake that one up. To implement this task in an $O(1)$ way, we need to queue the waiters in a sorted list that provides constant time queuing and unqueuing. On unlock or wake up time, the first waiter in the list will be the highest priority one.

1.2 Priority inversion

This condition happens when a lower priority thread blocks a higher priority one. The most general case (Figure 1) is the lower priority thread that holds a resource needed by the higher priority one–a situation that has to be avoided–as much as possible. As indicated before, we aim to solve three different flavors.

The first is **lock stealing**. For performance reasons, to avoid the convoy phenomenon[3] [1], the *unlock* operation is done by unlocking the mutex and then waking up the first waiter (eg: A). The waiter claims the mutex and then becomes owner. On a single CPU system it can be preempted only by higher priority tasks[4], so lock stealing is not a problem; however, on multi-CPU systems, a lower priority task C running on another CPU could claim the lock just before B had the chance to do it and it would create a priority inversion scenario (see Figure 2).

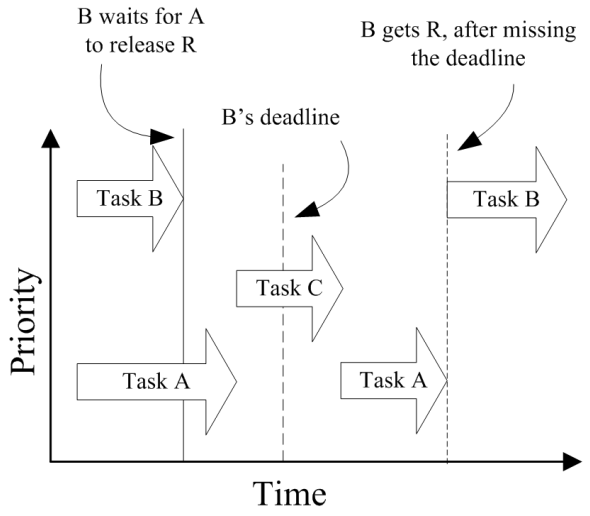


Figure 1: A case of priority inversion: high-priority task B misses its deadline because lower-priority task A holds for too long a resource it needs, as mid-priority task C preempted it. A lower priority task C blocks a high-priority task B.

The solution to this problem is simple: do not unlock the mutex, just transfer the ownership without unlocking it. We call this *serialized* unlock (versus *parallel*, wake and claim). This method severely limits performance in many cases, because it forces a context switch (causing the already mentioned convoy phenomenon). There has to be a compromise between protection and performance and by offering the option to unlock a mutex in either way, a developer can dynamically adapt according to her needs.

The other two cases (of priority inversion) are more complex. They solve the scenario depicted in Figure 1 where task B is waiting for a mutex owned by task A and task C preempts task A. When the priorities are $p(B) > p(C) > p(A)$, we have a priority inversion; task B will miss its deadline because C is blocking A from

[3] Summarizing: if task A (high priority) unlocks by transferring ownership to the first waiter B (lower priority), it forces a context switch to B, and if then A recontends for the lock it will create a convoy of waiters that is difficult to dissolve.

[4] We will use the terms tasks or threads indistinctly to refer to any entity that can acquire a mutex.

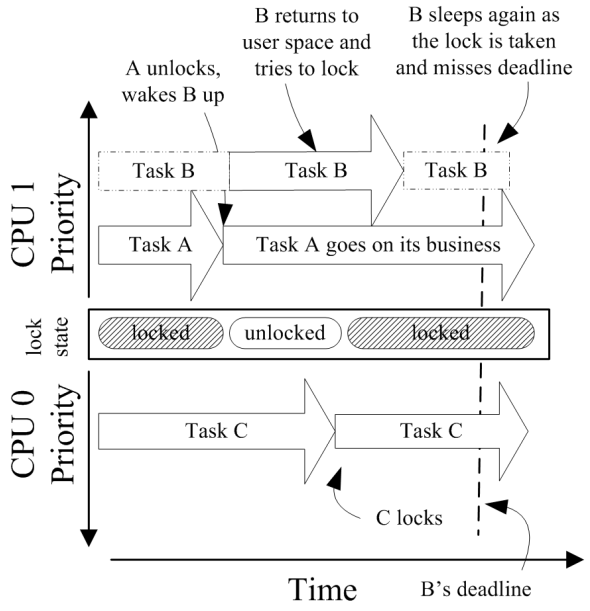


Figure 2: Low priority task C running on CPU0 steals the lock from higher priority task B running on CPU1.

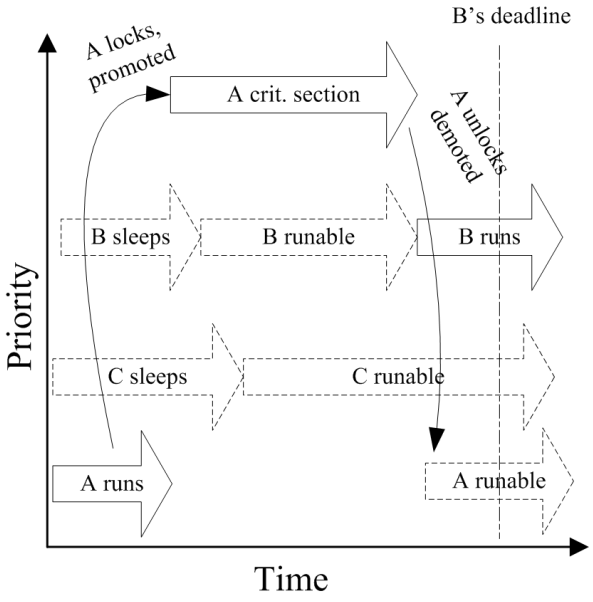


Figure 3: Priority protection: task A locks and its priority is promoted to the prioceiling; task C cannot preempt it and it finishes its critical section (and is demoted) in time for B to meet its deadline.

completing its mutex-protected critical section.

There are different ways to deal with this problem, but the most common involve bumping up the priority of the owner of the lock to a certain value.

In **priority protection** (or PP), a *priority ceiling* is determined as part of the design cycle. This is normally the highest of all the priorities among the threads that will share a given mutex; as soon as it is locked, the priority of the owner is raised to match that of the priority ceiling (see Figure 3. When a thread owns many priority-protected mutexes, its priority is that of the highest ceilings. This approach is simple and guaranteed to be trouble free. However, it is laborious; determining the priority ceiling might not be an easy task at all in a moderately complex system where modules from different parties need to interact.

Enter **priority inheritance** (PI): in this case we have a similar situation, but there is no priority ceiling. What happens in this case is that the priority of the owner is boosted up to that of the highest priority waiter, the first one (see Figure 4). Similarly to the previous case, if a task owns many PI-mutexes, its priority will be the highest of them all. There is no need now to do design-time analysis; the system solves it automatically. Of course, there are drawbacks– it does not come for free. This operation is more expensive, especially in the presence of owner/wait chains[5]. The propagation of the priority boost can be long (and will be $O(N)$ on the depth of the chain) and this can lead to unexpected surprises if the interaction across different threads and mutexes in the system is not kept on a tight leash (see [2] and [3]).

[5]Task A waits for mutex M that is owned by task B that is waiting for mutex N that is owned by task C that is waiting for mutex O…

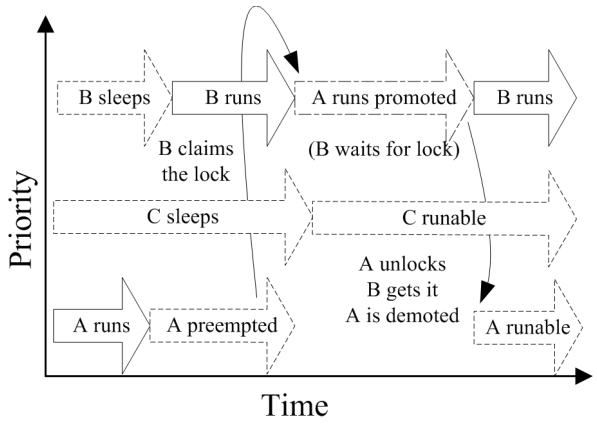


Figure 4: Priority inheritance: Task A (lock owner) is promoted to task B's priority when B waits for the lock; as soon as A unlocks, it gets demoted and B gets the lock. C never has a chance to preempt A.

Priority inheritance needs to be used with care– it is not a straight solution for a system with deadlock problems to make a mutex PI. What if that mutex is being shared with some low priority timesharing task that is not aware of the fact? In these cases, if a task does some kind of CPU spinning, the system is dead. The concept of priority inheritance and the simplicity it gives to designs provides enough rope as to hang oneself, as the effects can propagate way far more than expected.

1.3 Robustness

Mutex robustness is a key feature for implementing systems tolerant to certain kind of failures. A certain task A is holding a normal, *non-robust* mutex M with one or more waiters W_n blocked in the kernel. If it receives a fatal signal and is killed, the mutex will still be locked and the waiters will be never woken up. There are different ways to detect and recover from this situation, but they usually involve painful and complicated designs with watchdogs, timeouts, etc.

Robustness embeds all these in the mutex mechanism. When a task owns a mutex, the mutex knows who is its owner, and asks to be notified if the owner dies. If and when this happens, the mutex will be moved to an special *consistency state*, *dead-owner* and effectively unlocked; this will give control to the first waiter (or remain unlocked as *dead-owner* until somebody else claims it).

Threads claiming a dead mutex will receive ownership with a special error code, `-EOWNERDEAD`. This serves as a warning: *the data protected by this mutex might be inconsistent, it should be fixed*. The new locker can do different things at this point:

- it can ignore it (scary choice!)
- it might be unqualified for the job and pass the responsibility on to somebody else (by unlocking).
- it can try to fix the data and succeed–then it will *heal* the mutex, setting its consistency state back to normal and proceed.
- or it can fail and pass it on...
- or it can fail and deem the state completely broken; to notify about this situation, it can mark the mutex *not-recoverable*, so all waiters and future claimers will get a `-ENOTRECOVERABLE` error code so other recovery strategies can kick in.

The most important aspect to take into account is that the user of the mutex has means to detect this situation instantly without having to rely on timeouts or other overheads.

1.4 Deadlock detection

A situation of deadlock happens when a task A that owns a mutex M tries to lock it again.

This is the simplest case, of course. The general case is:

- task T_1 owns mutex M_1 and tries to lock mutex M_2
- task T_2 owns mutex M_2 and is waiting to lock mutex M_3
- task T_3 owns mutex M_3 and is waiting to lock mutex M_4
- ...
- task T_N owns mutex M_N and is waiting to lock mutex M_1

Construct like these are called *ownership-wait chains*. And it is obvious that in this particular case, this chain would deadlock, as M_1 would never be released if T_1 is allowed to block waiting for M_2.

The only way to detect this situation is, upon lock time, to walk the chain and verify if the task that is about to lock owns any lock on the chain.

By definition this is a linear operation that is going to take time to execute. The best way to avoid this expensive check is to make sure our design uses proper locking techniques (like for example, acquire and release multiple locks in LIFO order).

2 The first try: rtfutex

Once the requirements were laid out, we first tried modifying the futex code in `kernel/futex.c`, adding functionality while maintaining the original futex interface.

In a glimpse, the locking mode used with futexes works like this (see [4]): there is a word in user space that represents the mutex. The fast lock operation is performed entirely in user space; if the word is unlocked, then it becomes locked and work proceeds. If it is locked, the program sets a different value in the user space word and then goes down to the kernel and waits.

When the unlock operation is performed, the unlocker will check the value of the word; if it indicates that only a fast-lock was performed (and thus there are no waiters in the kernel), it will be simply unlocked in user space; otherwise, it will ask the kernel to wake up one or more waiters. These waiters will come back to user space and reclaim the lock; only one will get it, the rest will go back to the kernel to sleep[6].

With this in mind, we performed the following modifications:

- To allow wake-the-highest-priority waiter behavior on a bound time, the hash table model had to be modified.

  One node per waiter was replaced by one node per futex, and each node would have its own priority-ordered list of waiters. Although the lookup of the futex-node in the hash table is $O(N)$, at least the manipulation of the waiter-list (or wait list) can be made $O(1)$.

  This introduced the need of having to allocate the futex-node, as it could not live in the stack of some waiter[7].

- In order to support robustness, deadlock detection and priority inheritance, the

[6] note this means that the lock is actually unlocked for an unspecified amount of time in an unlock to lock transition.

[7] This raises extra issues; allocation can fail and is not time-predictable; it can be slow, so it is needed to cache the nodes (as normally they are frequently reused); caching means a strategy is needed to purge them (garbage collection).

concept of *ownership* had to be added to the futex. This would save *which* task owns the futex on each moment. It also required to note in the task struct which futex was being waited for, as well as a list of owned futexes.

- A different method had to be used for locking in user space, the fast lock and unlock paths.

  The user space word representing the futex would store the PID of the locker while on the fast path and indicate with a bit the presence of waiters in the kernel. This way if a locker died after having done a fast-lock operation in user space (and thus the kernel not having any notion of it), a potential waiter could check if the lock was stale[8]. When a futex went into the *dead-owner* or *not-recoverable* state, the kernel would modify the user space word with special values to mark these states.

  As well, the unlock operation had to always be serialized, with the kernel assigning ownership and modifying the user space word, ensuring robustness[9] and that no lock stealing happened.

This design (and its implementation) was broken: the futexes are designed to be queues, and they cannot be stretched to become mutexes– it is simply not the same. The result was a bloated implementation.

As well, the code itself missed many fine (and not so fine) details:

- it suffered from race conditions: the modification of the different back pointers in the task struct was being done without protection.
- the priority inheritance engine was very limited (to the most simple cases of inheritance) and it didn't support `SCHED_NORMAL` tasks.
- serialized unlocking is slow, it causes the convoy phenomenon, and the code did not provide flexibility to allow the user to balance performance vs. robustness or priority inversion protection depending on the situation.
- it didn't provide the functionality at the kernel level, for usage by kernel code.
- it didn't support changing the priority of a task while it was waiting for a futex while at the same time properly repositioning it on the wait list according to its new priority.

While broken, it was perfect as a prototype– it gave an indication of what was wrong, how things should not be done and hinted which methods were a good idea. It was time to rethink all over again.

3 Trying again: fusyn

With rtfutexes we found that stretched designs are not a good idea, however, experience tells layered designs are a better idea.

The fusyn design follows the same basic principles of the futexes, providing the same service in kernel and user space. Enforcing a strict modularity among the different units that comprise it, it is possible to accomplish much more with less bloat and complexity.

[8] This is a very simple method that cannot guarantee conflicts when PIDs are reused; we implemented a naive task-signature system to try to avoid this case. We didn't realize how broken it was until later.

[9] If waiters coming up from the kernel died before locking again and there were still some others waiting, the kernel would never know about it and the remaining tasks would wait for ever.

The four main blocks that comprise the fusyn architecture are:

- *fuqueues* are the wait queues (very similar to the Linux kernel's waitqueues) and are the basic building block.
- *fulocks* provide the mutex functionality by adding the concept of ownership on top of fuqueues and dealing with all the priority promotion.
- *vlocators* serve as the link between user space words and the kernel objects (fuqueues and fulocks) associated to them.
- *vfulock sync* maintains synchronization between the fulocks and the *vfulocks*, the user space word associated to them. It also is responsible for identifying owners from the cookies stored in the vfulocks.

fuqueues

We start with a simple queue structure, `struct fuqueue`, declared in `linux/fuqueue.h`. It merely contains a priority-sorted list where to register the waiters for the queue, a spinlock and an operations pointer. The operations are for managing a reference count (used when associated to user space), for canceling a task's wait on a fuqueue and notifying the fuqueue of a priority change on a waiter (most functions are defined in `kernel/fuqueue.c`).

A fuqueue can be initialized, waited on with `fuqueue_wait()` or a number of waiters for it can be woken up with `fuqueue_wake()`. All the functions for doing that are conveniently broken up so they can be used by other layers.

Whenever a task waits on a fuqueue, it registers itself by filling up a `struct fuqueue_waiter`; that structure and the fuqueue being waited for are linked to from the task struct (`struct fuqueue_waiter *fuqueue_waiter` and `struct fuqueue fuqueue_wait`), so that the signal delivery code (through `fuqueue_waiter_cancel()`) and the scheduler priority changing functions (through `fuqueue_waiter_chprio()`) can properly locate which fuqueue to act upon. A spinlock protects these pointers in the task structure.

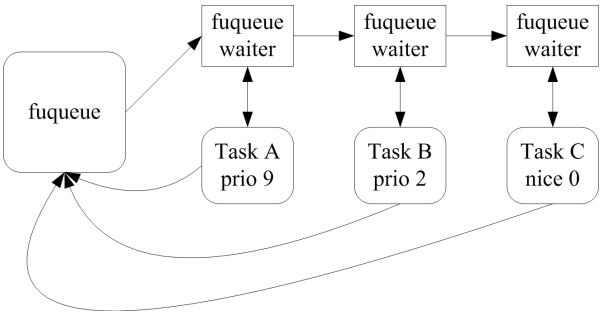


Figure 5: A fuqueue with three waiters, $p(A) > p(B) > p(C)$, showing the different pointers on each structure.

This satisfies the real-time requirements of wake-up order by priority. As well, the addition to the waiters list is bounded in time to the maximum number of different priority levels used–being this 140 for the Linux kernel, that makes the addition operation $O(140) \equiv O(1)$.

Note the fuqueue structure has to be protected, similarly to waitqueues with an IRQ-safe spinlock, as they will be accessed for wake-up from atomic contexts.

fulocks

Once we have a queue structure that is real-time friendly, we can build mutexes on top of them. Adding the concept of ownership, we create a `struct fulock` in `linux/fulock.h` that contains a fuqueue (for the waiters), a

pointer to a task struct (the owner), some flags and a node for a priority-sorted ownership list (to register all the fulocks owned by a task).

Let's ignore for a while the secondary effects of priority inheritance and protection. Come lock time, `fulock_lock()`: if the fulock is unlocked, the current task is assigned ownership by setting the owner pointer in the fulock to point to the task, the fulock is added to the `task->fulock_olist` ownership list through its `olist_node`.

If the fulock is locked (unless just try-locking) the task waits on the fulock's fuqueue; when woken up, depending on the result code of the wake up, it will own the lock (and thus proceed) or try again (serialized vs. parallelized unlocks).

The unlock operation, `fulock_unlock()` is quite simple: if the unlocker desires to perform a serialized wakeup, it just changes the owner to be the first waiter, removes it from the wait list and wakes him up with a 0 result code. If the unlock has to be parallelized, it unlocks the fulock and unqueues and wakes up the first waiter (or the first N waiters) with a `-EAGAIN` code–that will lead the sleeping `__fulock_lock()` call to retry. The unlock mode can be automatically determined based on the policy of the first waiting task: serialized for real-timers, parallelized for timesharers.

All this code is defined in `kernel/fulock.c`.

Robustness

Robustness comes into play with a hook in `kernel/exit.c:do_exit()`. When a process dies, `exit_fulocks()` goes over the list of fulocks owned by the exiting task; for each of them, the operation registered for task `exit` is executed, and that leads to setting the dead flag (`FULOCK_FL_DEAD`) and serially unlocking the fulock to the next waiter with the `-EOWNERDEAD` error code[10].

This introduces the need to have a way for the user to switch the fulock from one state to the other. `fulock_ctl()` provides this capability.

Deadlock detection

The process of checking for deadlocks is done via a hook in the `__fulock_lock()` function that calls `__fulock_check_deadlock()`.

This function will query the owner of the fulock the current task wants to wait for and inquire which fulock this owner is waiting for. If not waiting for anyone, there is no possible deadlock, so all resources are dropped and success is returned.

If it is waiting, the fulock is safely acquired (the ugliest part is to get the spinlocks properly as well as the reference counts); if the owner is the current task, then that is a deadlock; if not, then the operation repeats with the owner of the new fulock.

Priority inheritance and protection

Now let's take priority inheritance and protection into consideration. The key here is that in the priority-sorted list (plist), every node, including the list head, has a priority field, and that in a consistent plist, the priority of the list is that of the head, that in turn is that of the highest priority node queued.

Thus, by virtue of the priority-sorted list, each fuqueue has a *priority*. Fulocks inherit this property and when doing **priority inheritance**, they set that priority on the node for the priority-based ownership list. **Priority protected** fulocks set as priority that of the priority

[10] as well, a warning is issued if the fulock wasn't declared robust.

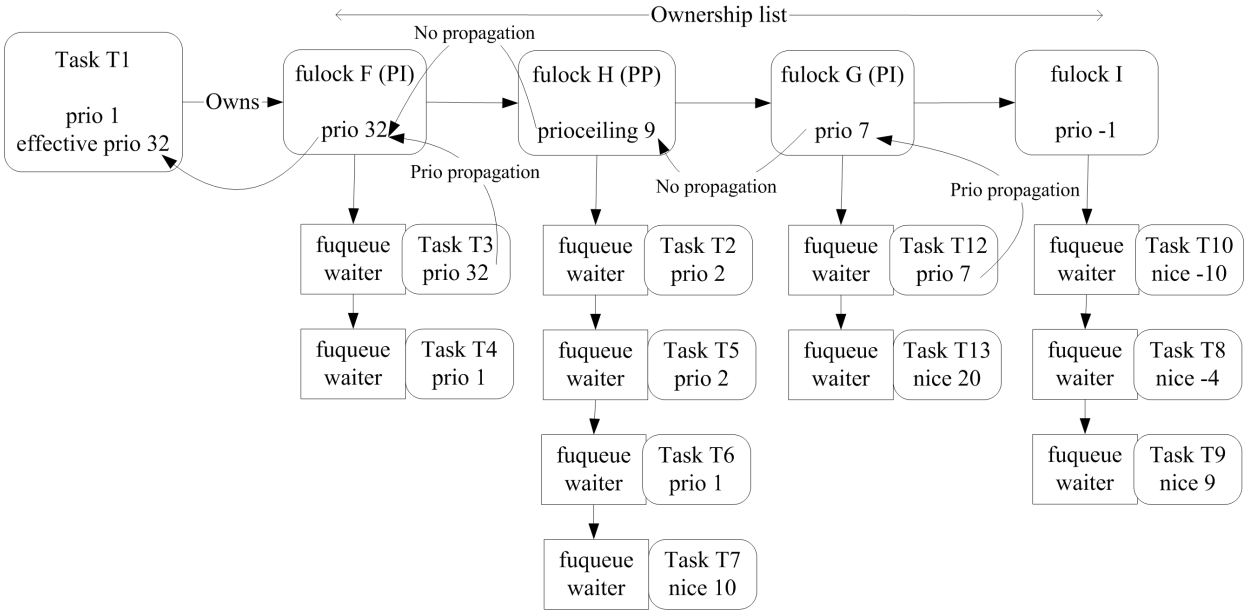


Figure 6: A task that owns four contested fulocks (two PI, one PP and one normal) showing the priority propagation flow. Note how fulock I, not being priority inheriting or protecting, has the minimal priority, -1 (which effectively disables all side effects).

ceiling of the fulock.

This way, each task has a list of the fulocks it owns sorted by priority. The ordering of the list means that the first fulock in the list has the minimum priority the task should have to meet the priority protection and/or priority inheritance criteria–and thus, the scheduler just has to select as effective task priority the highest between the task's final dynamic priority and that of the first fulock on its ownership list[11]. See Figure 6.

The process then becomes extremely simple: when a task queues waiting for a fulock (in `__fuqueue_waiter_queue()`), it might modify the plist priority because it sets a new *higher* priority–the function returns !0 in this case. This is propagated, with `__fulock_prio_update()` to the fulock's ownership list node, `fulock->olist_node`, that as we said above, is inserted in the ownership list of the fulock owner. The propagation could mean that a new maximum might be set in the ownership list, case in which the boost priority is updated for the scheduler to pick it up.

On top of that, the change might need to be propagated further on if the fulock owner is waiting for another fuqueue or fulock. `__fuqueue_waiter_chprio()` will take care of propagating that change until a task is reached that is higher priority or is not waiting for a priority-inheriting fulock.

Linking to user space

So far, the infrastructure presented is accessible only from kernel space. We have to allow user space programs to take advantage of these features, and for that, we copy the futex's method: associate a virtual address (word) to

[11] This is accomplished with a simple mechanism (improvement required to reduce invasiveness) that adds the concept of boost priority to the task struct (`boost_prio`), and modified through `__prio_boost()`.

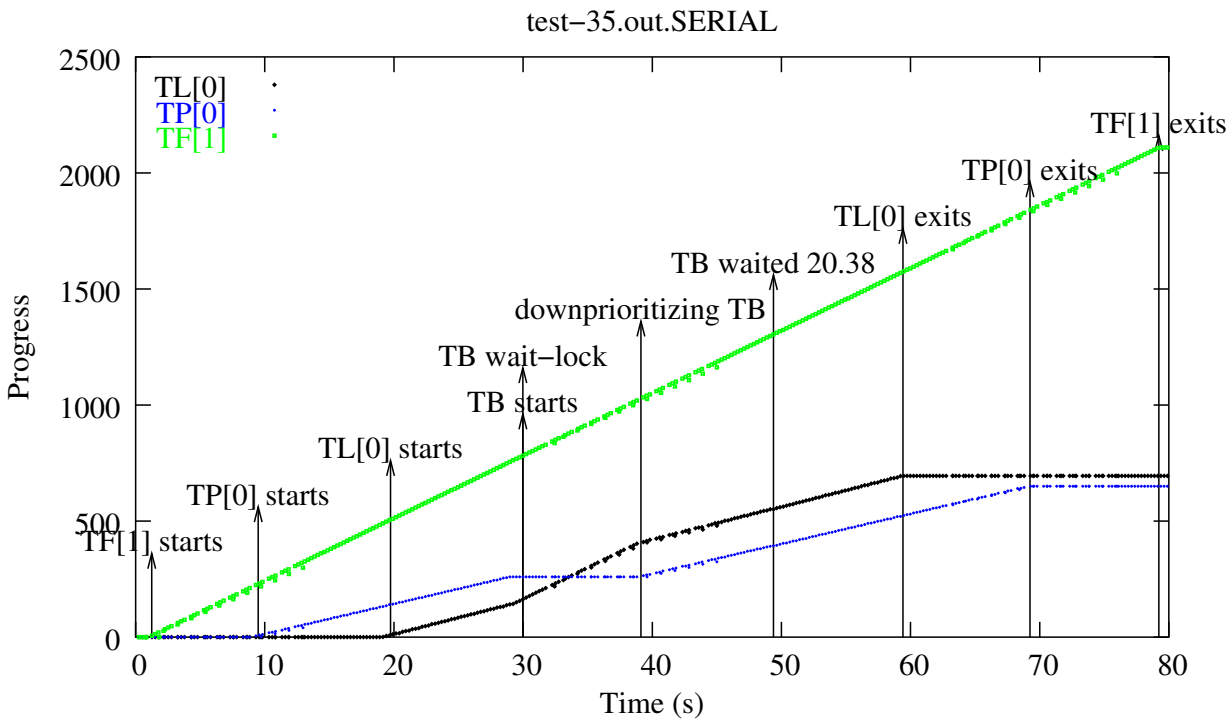


Figure 7: Testing priority inheritance: four threads of increasing priority (TL < TP < TB < TF) in an infinite loop counting up (progress); TF stays in CPU1 as a reference; TP sleeps from time to time in CPU0 to give TL a change; TL progresses what TP allows it. When TB starts (at $30s$), it claims a priority-inheriting fulock owned by TL and thus it gets boosted, TP doesn't progress any more. At almost $40s$, TB is down prioritized and that deboosts TL, allowing TP to progress again.

an object in memory (`struct vlocator`).

The API exposed in `linux/vlocator.h` provides a generic method for doing this by just embedding a vlocator; as well, this vlocator provides a reference-counting interface to simplify the object's life cycle management. And when it's use count is zero, it will be automatically disposed of[12].

This also improves scalability a little bit as the only global lock in the vlocator hash table is taken just to do the look up; once found, the vlocator is referenced before dropping the lock.

[12]Here is where the caching kicks in; the hash table is cleaned up of zero ref-counted items every certain amount of time, allowing for reuse.

ufuqueues and vfuqueues: imitating futexes

We need to create an interface equal to that of futexes for implementing conditional variables with real-time friendly functionality (for the wake up ordering).

We create a `struct ufuqueue` where we embed a vlocator and a fuqueue. A thin adaptation layer (`sys_ufuqueue_wait()` and `sys_ufuqueue_wake()`) will get the system call from user space, do the look up using the vlocator API, verify that the user space word (`vfuqueue`) hasn't changed and pass it down to the fuqueue layer.

The rest of the code in `kernel/ufuqueue.c` deals with creating the operation functions for

the vlocator structure.

Exposing the fulocks to user space

The same mechanism is used for exposing the fulocks to user space; in a similar fashion to fuqueues, we wrap together a vlocator and a fulock to create a `struct ufulock`.

However, more aspects have to be taken into consideration:

- If the fulock is not contested, the lock and unlock operations must happen entirely in user space (and thus the kernel will not know about it; this is the *fast-path*)
- When a lock has been locked through the fast path, the kernel has to be able to identify who locked it as well as its consistency status; this operation is called synchronization.
- When a lock becomes contested, the kernel has to update the user space word to indicate that future operations need to proceed in the kernel–as well, when it is eligible to be a fast-path only fulock again, the kernel must undo this, put the fulock structure in the cache tagged as requiring synchronization from user space and make sure the user space word has the consistency state of the fulock.
- The fulock structure in the kernel will be disposed if no task goes to the kernel querying about or operating on it for a while; as in the previous case, the information will be kept in user space word to enable proper synchronization.

For this we need some more information than the one used by the same futex mechanism for the fast path. A locker needs to identify itself in the user space word (that we call *vfulock*) by storing a cookie that can directly map to a task struct in the kernel space. The most obvious choice for the cookie would be the PID[13].

However, this operation must be atomic–this means that we need an atomic compare-and-exchange operation, and thus, the lock operation becomes the following: compare-and-exchange the cookie against 0 (meaning unlocked); if it succeeds, then the vfulock is locked, if not, dive into the kernel. The kernel will map the address to a fulock (possibly creating a new ufulock) get the value of the vfulock (`sys_ufulock_lock()` and `ufulock_lock()`) and map it to a task (in `__vfulock_sync()`. If the kernel is able to find the task, that task is made the owner and the caller is put to wait. As well, the vfulock is updated to a special value `VFULOCK_WP`, meaning waiters are present in the kernel.

If the kernel cannot find it, that will mean the task that fast-locked it in user space has died, the fulock will be declared *dead-owner* and the caller will get ownership. In this process, the vfulock will be set to another special value, `VFULOCK_DEAD` that indicates it as dead even across the kernel forgetting about its existence.

Unlocks are equally simple: atomically compare-and-exchange 0 (`VFULOCK_UNLOCKED`) against the cookie of the lock owner; if it succeeds, the job is done; else, the kernel does it. After mapping the vfulock to a ufulock, `ufulock_unlock()` is used to do the job and the vfulock is updated to reflect the new state: `VFULOCK_UNLOCKED` if unlocked, if there will be no waiters the new owner's cookie–enabling fast-path, `VFULOCK_WP` if waiters are still in the kernel, or `VFULOCK_DEAD` if the fulock is dead.

If parallelized unlocks are desired, the pro-

[13] This would break unique identification as PIDs are reused; a solution could be crypting the PID with the task creation date, but it needs to be tested.

cess is a little bit different. In the kernel, `__ufulock_unlock()` will unlock the vfulock and then wake up the first waiter, who then will contend (in the kernel) for the vfulock and possibly wait, as described above[14].

Note the two key moments in switching from fast-path enabled or not: the fulock becomes fast-path when it has no waiters in the kernel or when it is healed[15] without waiters. It looses the fast-path conditions as soon as a single waiter is queued. This means that to maintain proper semantics during the lifetime of a program that uses many locks, once a fulock has gone through the slow path, it needs to be destroyed in the kernel using the `sys_ufulock_ctl()` system call once it is not needed anymore. If not, there could be inconsistencies if a new lock is created in the same address where a previous one lived before.

KCO: When the fast-[un]lock path is not an option

The fast path, as we have seen, requires an atomic compare-and-exchange operation. Not all architectures provide this capability, so different strategies need to be considered here.

If robustness, and priority inversion protection[16] can be spared, the mutexes and conditional variables can be implemented as with futexes using fuqueues; the rest of the real-time featurettes are there (priority-based wake-ups and priority change semantics). If that can also be spared, futexes are still an option.

However, when that is not the case, the only possible choice is to use KCO mutexes, by ORing `FULOCK_FL_KCO` in the flags. That is an acronym for Kernel Controlled Ownership, or basically, the kernel takes care of everything. It needs to be called for locking and unlocking, there is no fast path (strictly speaking there is still a choice for fast path on some operations, as the vfulock is used to cache the consistency state of the fulock and any user space operation can check it before deciding if it should go to the kernel).

This feature also provides the highest level of protection for robustness. The per-thread cookie for the vfulock, be it the PID or any other, is not required, and the kernel deals directly with the task struct, so there is no possible collision conflict.

It has to be noted that priority-protected ufulocks always work in KCO mode. Even on uncontended acquisition or release the priority of the thread has to be changed to that of the prioceiling, and that task can only be done by the kernel.

4 Using it in the kernel

The fulock is a simple type like any other struct. To use it, we just need to do the following declarations:

```
...
#include <linux/fulock.h>
...

struct mystruct {
  struct fulock lock;
  ...
  my shared data;
};
```

It needs to be properly initialized before use, and of course, after releasing it (or more properly, telling all waiters to bail out) it shall not

[14] Not going back to user space to retry the operation has advantages: speed and maintaining the conditions for robustness.

[15] Moved from *dead-owner* consistency state back to normal (or healthy)

[16] Lock stealing avoidance, priority inheritance and priority protection.

be used. Note some flag combinations are not allowed (for example, querying for priority inheritance and protection at the same time is illegal) and will trigger a `BUG()`[17].

In this example we ask for a robust fulock with priority inheritance. It must be noted that fulocks are always robust–but clearly telling the kernel that we handle robust situations will suppress a kernel warning if the owner dies and it goes into *dead-owner* mode.

```
my_driver_probe(...)
{
 struct mystruct *my;
 ...
 my = kmalloc (...);
 if (my == NULL)
  goto err_alloc;
 fulock_init (&my->lock,
              FULOCK_FL_ROBUST
              | FULOCK_FL_PI);
 ...
};
```

As we see in the following snippet, the basic usage is the same as for every lock. However, in this case we add some recovery code for the case when some owner died[18]. Note also that the only fulock operation that is guaranteed to be safe in an atomic context is `fulock_unlock()`.

```
void my_something(
 struct mystruct *my) {
 ...
 result = fulock_lock(&my->lock,
                      0);
 if (result == -EOWNERDEAD
     && my_try_recover (my))
   goto notrecoverable;
 ...
 /* do our thing */
 ...
 fulock_unlock (&my->lock,
                FULOCK_FL_AUTO);
 ...
 return 0;

notrecoverable:
 /* Put it out of its misery,
  * release waiters, clean up,
  * user has to reload the
  * driver. */
 fulock_ctl (&my->lock,
             FULOCK_CTL_NR);
 my_put (my);
 return -ENOTRECOVERABLE;
};

int my_try_recover (struct
    *mystruct my) {
 int result, mode;
 ... try to recover *my ...
 if (successful) {
  result = 0;
  mode = FULOCK_CTL_HEAL;
 }
 else {
  result = !0;
  mode = FULOCK_CTL_NR);
 }
 fulock_ctl (&my->lock, mode);
 return result;
}
```

Finally, when we are done, we release all resources associated to the fulock to clean up. As indicated above, this merely makes sure that any waiter queued is woken up with an error condition and nobody can acquire it or queue again.

```
void my_cleanup (
 struct mystruct *my)
{
 ...
```

[17] For user space code, they will simply fail with `-EINVAL`.

[18] This is kind of an useless exercise, correct kernel code doesn't crash.

```
 fulock_release (&my->fulock);
 ...
}
```

The benefits that a fulock gives over a semaphore are the real-time characteristics, priority inheritance and protection and deadlock detection. The decision to use one or the other depends on the user needs, as it has to be taken into account that fulocks are somehow more heavyweight than semaphores.

5 Usage from user space

The main intention of the user space code is to do as little as possible in the fast path and delegate the rest to the slow path that will, in most cases, end up in the kernel.

Note these code snippets have been slightly simplified; for the authoritative reference, see the file `src/include/kernel-lock.h` in the test package `fusyn-package` available from the web site.

Locking

As mentioned, the fast lock operation needs an atomic compare and swap operation; for example, on i386:

```
unsigned acas (
 volatile unsigned *value,
 unsigned old_value,
 unsigned new_value)
{
 unsigned result;
 asm __volatile__ (
  "lock cmpxchg %3, %1"
  : "=a"(result),"+m"((*value))
  : "a"(old_value),"r"(new_value)
  : "memory");
 return result == old_value;
}
```

To simplify the code, this function returns true if it was successful in performing the swap operation. With this, we can create a generic, fast-path, user space lock operation:

```
int vfulock_timedlock (
 volatile unsigned *vfulock,
 unsigned flags, int pid,
 struct timespec *rel)
{
 if (acas(vfulock,
          VFULOCK_UNLOCKED,pid))
  return 0;
 return SYSCALL (ufulock_lock,
                 vfulock, flags,
                 rel);
}
```

We are using the thread's PID as the cookie for the vfulock, the user space memory word associated to the lock. Note the special syntax for timeouts understood by the kernel:

- Passing `NULL` means we don't want to wait, and this operation effectively becomes a trylock in the kernel.
- A `(void *)-1` timeout means block forever–no timeout.
- Any other specifies a pointer to a valid timeout structure.

From user space we have to always pass the same flags to the kernel for an specific vfulock, as it will check we are consistent during the lifetime of the fulock–when it dissapears from the cache, it is up to us to use still the same flags to maintain consistency in our program.

With a few additions, we can have a lock function that also works in KCO mode and that imitates the behavior of non-robust mutexes when owners die (*ie:* block forever):

```
int vfulock_timedlock (
 *vfulock, flags, pid, *rel)
{
 int result;
 if (!(flags & FULOCK_FL_KCO) &&
     acas(vfulock,
          VFULOCK_UNLOCKED,pid))
  return 0;
 result = SYSCALL (ufulock_lock,
                   vfulock, flags,
                   rel);
 if (!(flags & FULOCK_FL_RM) &&
     (result == -EOWNERDEAD
     |result == -ENOTRECOVERABLE))
  waiting_on_dead_fulock(vfulock);
 return result;
}
```

There are only two simple differences. First is to avoid the fast-path if we want to use KCO mode (and thus dive directly into the kernel). The second one takes care of non-robust mutexes returning in *dead-owner* state; in that case we block in `waiting_on_dead_fulock()`, a dummy function that blocks forever whose only purpose is to show up in program traces to indicate us the reason of a thread blocking.

Unlocking

The unlock operation is somehow more hairy. Although we could just make it simpler calling the kernel and letting it do all of the operations for us (as if it were in KCO mode), we want to have the fast-unlock path available:

```
int __vfulock_unlock (
  *vfulock, flags, unlock_type)
{
 unsigned old_value = *vfulock;

 if (flags & FULOCK_FL_KCO)
   goto straight;
retry:
 if (old_value < VFULOCK_WP) {
  if (acas (vfulock, old_value,
            VFULOCK_UNLOCKED))
   return 0;
  old_value = *vfulock;
  goto retry;
 }
straight:
 return old_value == VFULOCK_NR?
  -ENOTRECOVERABLE
  : SYSCALL (ufulock_unlock,
             vfulock, flags,
             unlock_type);
}
```

As with the lock() operation, we first check if the fulock is KCO; if so jump straight into the kernel (except if it is marked *not-recoverable*, in which case we fail).

In the case of the fast-path, we read the value of the vfulock; if it is looks like a cookie[19] then we try the fast-unlock, returning if successful. If it failed we retry from the beginning. When the value of the vfulock doesn't look like a cookie, we dive into the kernel, as it means that it is either dead or there are waiters (and thus the kernel handles it).

Note this unlock operation allows any thread to unlock the fulock, it doesn't need to be the owner.

Other operations

A `trylock()` operation is implemented in similar terms (please refer to the sample library code in the `fusyn-test` package, file `src/include/kernel-lock.h`; this package is available for download from the project's website).

Operations for manipulating or querying the

[19] The three values `VFULOCK_WP`, `VFULOCK_DEAD` and `VFULOCK_NR` are purposely chosen to be the last three values of the `unsigned` domain.

state of the fulock are implemented by calling the `ufulock_ctl()` system call directly, providing the vfulock and flags.

6 Integration with NPTL

The patches for integration with NPTL (that we call RTNPTL for short) allow any POSIX program to use these features, via a certain set of standard calls and ways to customize the operation mode of the fulock under the mutex's hood with other non-POSIX extensions.

RTNPTL uses the same or very similar user mode integration code than the one explained above, sitting down at the `lll_` layer in glibc. This code provides all the intended functionality only to the POSIX mutexes and conditional variables. Locks used internally by the library still need work (see the *future directions* section).

By default, RTNPTL provides non-robust fast-path enabled mutexes that unlock in automatic mode[20], without any priority inheritance and protection. However, by modifying the mutex attributes with the `pthread_mutexattr_set*()` calls, different parameters can be set:

- Manipulating the priority inversion protections:

  `pthread_mutexattr_setprotocol()` takes a mutex attribute and a protection protocol, `PTHREAD_PRIO_INHERIT` or `PTHREAD_PRIO_PROTECT`.

  `pthread_mutex_setprioceiling()` can be used to query and change the priority ceiling of a mutex.

  `pthread_mutexattr_setserial_np()` and `pthread_mutex_setserial_np()` allows setting the unlock method to use for lock-stealing avoidance out of `PTHREAD_MUTEX_SERIAL_NP`, `PTHREAD_MUTEX_PARALLEL_NP`, or `PTHREAD_MUTEX_AUTO_NP` (this one can be switched during the lifetime of the mutex).

- `pthread_mutexattr_setrobust_np()` enables robustness in the mutex to be. `pthread_mutex_setconsistency_np()` is used to heal or make *not-recoverable* a *dead-owner* mutex. The consistency state can be queried with `pthread_mutex_getconsistency_np()`.

- `pthread_mutexattr_setfast_np()` is used to select the use of a KCO fulock or not, effectively enabling/disabling fast-path operation.

The non-standard interfaces are still subject to some unlikely flux.

7 Current status and future direction

At the time of writing, the project has met most of the requirements that were set as targets, reaching stability and meeting performance goals of sub-millisecond latencies. The added overhead does not seem to affect too much compared to NPTL, being generally slightly slower.

Compatibility

We routinely test RTNPTL+fusyn by running:

[20] serialized or parallelized depending on the policy priority of the first waiter

- Miscellaneous multi-threaded applications (e.g.: Mozilla)
- SUN jdk-1.42_03 with SPECjbb2000[21].
- MySQL 2.23.58 with `super-smack` and `sql-bench`.

This has helped us to catch some bugs (with some pending for certain combinations) and to test the compatibility of our approach. Performance wise, no obvious differences have been found with plain NPTL running on futexes.

This set of macro benchmarks is incomplete and will be expanded in the future, time and resource availability permitting.

Latency

The current code performs fairly well latency wise (given the extra overhead). In an unloaded system[22], the latency of the serialized ownership change operation[23] is in the range of $60 \pm 10\mu s$. Adding some network load (ten simultaneous downloads of 40 MiB files) bumps it up to $110 \pm 10\mu s$. Simultaneous reading of 1 GiB from `/dev/hda` to `/dev/null` raises it up to $130 \pm 10\mu s$.

The code exposes a strange behavior when testing the ownership change latency in an unloaded system while increasing the number of waiters. The average latency stays stable for the first ten-to-fifteen waiters (threads of a single program) at around $18 \pm 10\mu s$ (see Figure 8).

However, when the number of queued waiters goes up to 2000 threads, the latency climbs up to $50 \pm 10\mu s$, stabilizing from there on, as seen on Figure 9. Of course this is an extremely unrealistic scenario, but it helps to test the scalability of the code, and nevertheless, we are trying to proof the root cause, being cache issues the most likely ones.

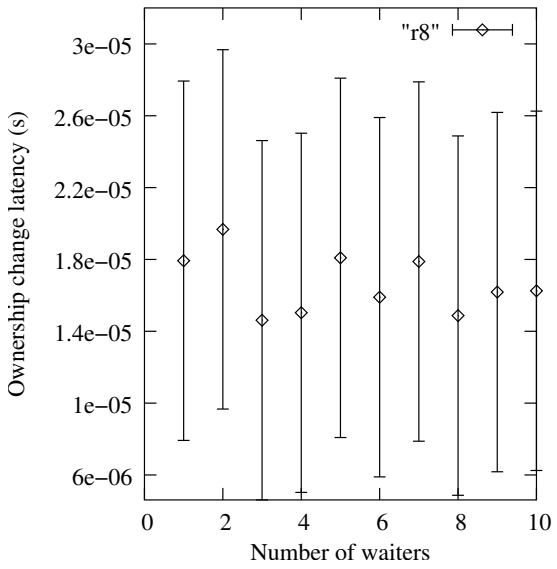


Figure 8: Scalability of the ownership-change latency *vs.* the number of waiters stays stable up until ten waiting threads.

.

Note: these numbers have been produced with a home-grown swiss-knife test program (to be published on the web site) called `ownership_change_latency`. Most of our timing efforts have concentrated in this particular case, although we have some other micro benchmarks planned.

Jitter

At this point, we haven't done yet any formal jitter studies.

Informally speaking, using the ownership change latency benchmark in unloaded systems, we have seen jitter increases over NPTL of about $1\mu s$, $0.3\mu s$ on a system fairly loaded with IDE and network traffic. However, bear in

[21] SPEC Java Business Benchmark 2000.

[22] as measured in a 2xP3 850 MHz 2.5 GiB RAM running version 2.3 of the code

[23] time since a serialized unlock is done until the first waiter gets the lock and executes.

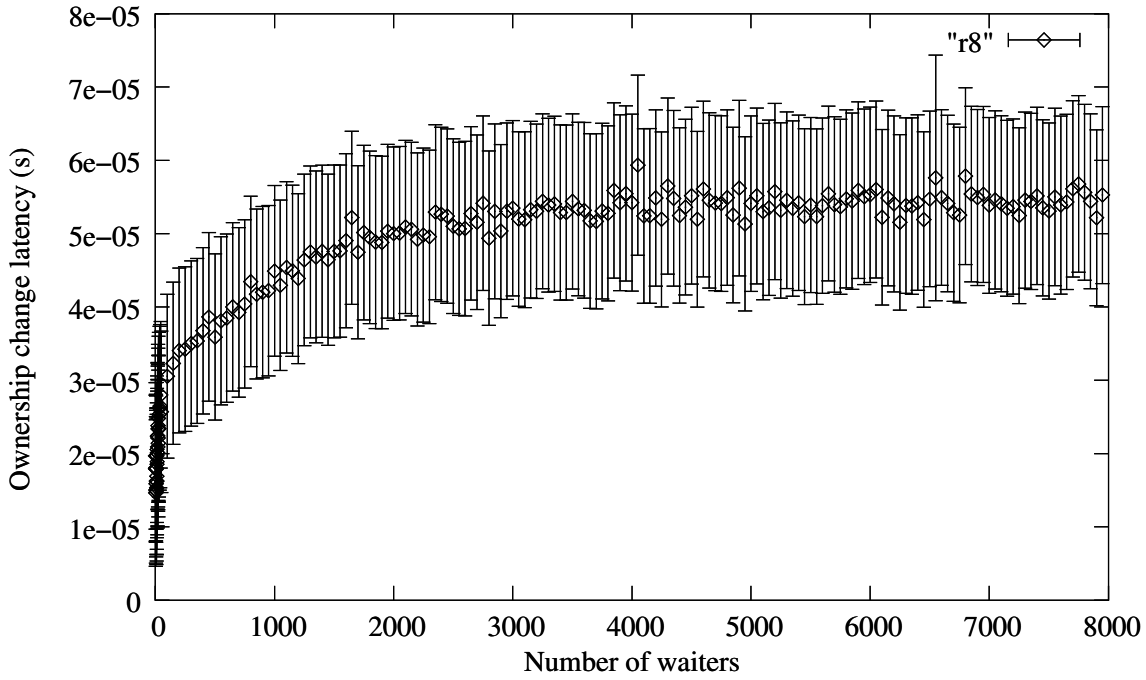


Figure 9: Scalability of the ownership-change latency *vs.* the number of waiters only stabilizes after two thousand waiters.

.

mind that these numbers are completely meaningless because the finest dependable clock resolution we can get (using the High Resolution Timers patch) is well higher, $10\mu s$. We can use them only to provide a hint.

Future direction

The project has reached an important milestone of maturity with the 2.2 release during the spring of 2004–nonetheless there is still much work to do. These some of the areas where we plan to target our future efforts:

- Some parties have asked for all these concepts (real-time, robustness, priority-protection) applied to read-write mutexes, much more complex than simple mutexes. We are still evaluation how worth is this.
- Some elusive bugs are still present.
- Accessing user space memory from the kernel by `kmapping` it poses some issues on architectures with *strange* cache consistency designs, such as some ARM and PA-RISC 8000. It is still not clear how to proceed for them and we would welcome any help.
- The kernel hash table for location of objects is a potential bottleneck in a system populated with many active user-space fusyn objects. We want to implement a proof of concept where a cookie identifying the object is placed in user space along the vfulock/vfuqueue. This cookie would consist of a two pointers crypted with two different keys by the kernel. In order to map a vfulock/vfuqueue to it's corresponding fusyn object, the kernel just has to decrypt the pointers. Having two crypted with different keys is used to enforce validity against garbage being writ-

ten by user space by mistake or to compromise the system.

- Providing a robust mutex infrastructure is OK, as long as it is used. Internally, glibc uses locks to protect many of its data structures—in order to be able to provide true robustness, we need to add robustness to those internal locks, as well as recovery strategies.

- Extend the coverage of our macro and micro benchmarks.

8 Downloading

The project maintains a website at:

`http://developer.osdl.org/dev/robustmutexes/`

from where all the current and older snapshots of the code can be obtained. As well, it offers pointers to the mailing list, bugzilla and CVS repositories.

We want to thank the Open Source Development Lab for making these resources available to us.

9 Conclusion

We have presented an infrastructure for providing real-time and robust synchronization services in the Linux kernel. We have been able to accomplish this with a minimum overhead impact over the current futex-based infrastructure and expect that it will be sufficient to satisfy the needs of multi-threaded, fault-proof and/or soft-real time designs.

10 Trademarks and acknowledgements

During development we kept discovering roadblocks, situations, and side effects we failed to spot or details we missed in the POSIX specifications. A lot of hair pulling that was counteracted by the thrill of the challenge, producing a love-and-hate relationship with the topic (and hence the title of this paper). We want to thank all of those who helped out by pointing out issues, contributing, reviewing, criticizing, and testing ideas and code.

References

[1] Mike Blasgen, Jim Gray, Mike Miltoma, and Tom Price. 1979. “The convoy phenomenon,” *ACM SIGOPS Operating Systems Review*, Volume 13, Issue 2 (April 1979). `http://portal.acm.org/citation.cfm?id=850659&jmp=cit&dl=GUIDE&dk=ACM`

[2] Arnd C. Heursch, Dirk Grambow, Dirk Roedel, and Helmut Rzehak. “Time-critical tasks in Linux 2.6,” `http://www.informatik.unibw-muenchen.de/inst3/index_de.php` Pages 6 and 7.

[3] Victor Yodaiken. 2001. “The dangers of priority inheritance,”

http://citeseer.ist.psdu.edu/yodaiken01dangers.html

[4] Hubertus Frankel, Rusty Russell, and Matthew Kirkwood. 2002. “Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux.” *Proceedings of the 2002 Ottawa Linux Symposium*, http://www.linux.org.uk/~ajh/ols2002_proceedings.pdf.gz Pages 479-495

Workload Dependent Performance Evaluation of the Linux 2.6 I/O Schedulers

Steven L. Pratt
IBM
slpratt@us.ibm.com

Dominique A. Heger
IBM
dheger@us.ibm.com

Abstract

The 2.6 release introduced the option to select a particular I/O scheduler at boot time. The 2.4 Linus elevator was retired, incorporated are now the anticipatory (AS), the deadline, the noop, as well as the completely fair queuing (CFQ) I/O schedulers. Each scheduler has its strengths and weaknesses. The question is under what workload scenarios does a particular I/O scheduler excel, as well as what is the performance gain that is possible by utilizing the available tuning options.

This study quantifies the performance of the 4 I/O schedulers under various workload scenarios (such as mail, web, and file server based conditions). The hardware is being varied from a single-CPU single-disk setup to machines with many CPUs that are utilizing large RAID arrays. In addition to characterizing the performance behavior and making actual recommendations on which scheduler to utilize under certain workload scenarios, the study looks into ways to actually improve the performance through either the existing tuning options or any potential code changes/enhancements.

Introduction

This study was initiated to quantify I/O performance in a Linux 2.6 environment. The I/O stack in general has become considerably more complex over the last few years. Contemporary I/O solutions include hardware, firmware, as well as software support for features such as request coalescing, adaptive prefetching, automated invocation of direct I/O, or asynchronous write-behind polices. From a hardware perspective, incorporating large cache subsystems on a memory, RAID controller, and physical disk layer allows for a very aggressive utilization of these I/O optimization techniques. The interaction of the different optimization methods that are incorporated in the different layers of the I/O stack is neither well understood nor been quantified to an extent necessary to make a rational statement on I/O performance. A rather interesting feature of the Linux operating system is the I/O scheduler [6]. Unlike the CPU scheduler, an I/O scheduler is not a necessary component of any operating system per se, and therefore is not an actual building block in some of the commercial UNIX® systems. This study elaborates how the I/O scheduler is embedded into the Linux I/O framework, and discusses the 4 (rather distinct) implementations and performance behaviors of the I/O schedulers that are available in Linux 2.6. Section 1 introduces the BIO layer, whereas Section 2 elaborates on the anticipatory (AS), the deadline, the noop, as well as the completely fair queuing (CFQ) I/O schedulers. Section 2 further highlights some of the performance issues that may surface based on which I/O scheduler is being utilized. Section 3 discusses some additional

hardware and software components that impact I/O performance. Section 4 introduces the workload generator used in this study and outlines the methodology that was utilized to conduct the analysis. Section 5 discusses the results of the project. Section 6 provides some additional recommendations and discusses future work items.

1 I/O Scheduling and the BIO Layer

The I/O scheduler in Linux forms the interface between the generic block layer and the low-level device drivers [2],[7]. The block layer provides functions that are utilized by the file systems 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. The device drivers consume the transformed requests and forward them (by using device specific protocols) to the actual device controllers that perform the I/O operations. As prioritized resource management seeks to regulate the use of a disk subsystem by an application, the I/O scheduler is considered an imperative kernel component in the Linux I/O path. It is further possible to regulate the disk usage in the kernel layers above and below the I/O scheduler. Adjusting the I/O pattern generated by the file system or the virtual memory manager (VMM) is considered as an option. Another option is to adjust the way specific device drivers or device controllers consume and manipulate the I/O requests.

The various Linux 2.6 I/O schedulers can be abstracted into a rather generic I/O model. The I/O requests are generated by the block layer on behalf of threads that are accessing various file systems, threads that are performing raw I/O, or are generated by virtual memory management (VMM) components of the kernel such as the kswapd or the pdflush threads. The producers of I/O requests initiate a call to `__make_request()`, which invokes various I/O scheduler functions such as `elevator_merge_fn()`. The enqueue functions in the I/O framework intend to merge the newly submitted block I/O unit (a bio in 2.6 or a `buffer_head` in the older 2.4 kernel) with previously submitted requests, and to sort (or sometimes just insert) the request into one or more internal I/O queues. As a unit, the internal queues form a single logical queue that is associated with each block device. At a later stage, the low-level device driver calls the generic kernel function `elv_next_request()` to obtain the next request from the logical queue. The `elv_next_request()` call interacts with the I/O scheduler's dequeue function `elevator_next_req_fn()`, and the latter has an opportunity to select the appropriate request from one of the internal queues. The device driver processes the request by converting the I/O submission into (potential) scatter-gather lists and protocol-specific commands that are submitted to the device controller. From an I/O scheduler perspective, the block layer is considered as the producer of I/O requests and the device drivers are labeled as the actual consumers.

From a generic perspective, every read or write request launched by an application results in either utilizing the respective I/O system calls or in memory mapping (mmap) the file into a process's address space [14]. I/O operations normally result in allocating PAGE_SIZE units of physical memory. These pages are being indexed, as this enables the system to later on locate the page in the buffer cache [10]. A cache subsystem only improves performance if the data in the cache is being reused. Further, the read cache abstraction allows the system to implement (file system dependent) read-ahead functionalities, as well as to construct large contiguous (SCSI) I/O commands that

can be served via a single direct memory access (DMA) operation. In circumstances where the cache represents pure (memory bus) overhead, I/O features such as direct I/O should be explored (especially in situations where the system is CPU bound).

In a general write scenario, the system is not necessarily concerned with the previous content of a file, as a `write()` operation normally results in overwriting the contents in the first place. Therefore, the write cache emphasizes other aspects such as asynchronous updates, as well as the possibility of omitting some write requests in the case where multiple `write()` operations into the cache subsystem result in a single I/O operation to a physical disk. Such a scenario may occur in an environment where updates to the same (or a similar) inode offset are being processed within a rather short time-span. The block layer in Linux 2.4 is organized around the `buffer_head` data structure [7]. The culprit of that implementation was that it is a daunting task to create a truly effective block I/O subsystem if the underlying `buffer_head` structures force each I/O request to be decomposed into 4KB chunks. The new representation of the block I/O layer in Linux 2.6 encourages large I/O operations. The block I/O layer now tracks data buffers by using struct page pointers. Linux 2.4 systems were prone to loose sight of the logical form of the writeback cache when flushing the cache subsystem. Linux 2.6 utilizes logical pages attached to inodes to flush dirty data, which allows multiple pages that belong to the same inode to be coalesced into a single bio that can be submitted to the I/O layer [2]. This approach represents a process that works well if the file is not fragmented on disk.

2 The 2.6 Deadline I/O Scheduler

The deadline I/O scheduler incorporates a per-request expiration-based approach and operates on 5 I/O queues [4]. The basic idea behind the implementation is to aggressively reorder requests to improve I/O performance while simultaneously ensuring that no I/O request is being starved. More specifically, the scheduler introduces the notion of a per-request deadline, which is used to assign a higher preference to read than write requests. The scheduler maintains 5 I/O queues. During the enqueue phase, each I/O request gets associated with a deadline, and is being inserted in I/O queues that are either organized by the starting logical block number (a sorted list) or by the deadline factor (a FIFO list). The scheduler incorporates separate sort and FIFO lists for read and write requests, respectively. The 5th I/O queue contains the requests that are to be handed off to the device driver. During a dequeue operation, in the case where the dispatch queue is empty, requests are moved from one of the 4 (sort or FIFO) I/O lists in batches. The next step consists of passing the head request on the dispatch queue to the device driver (this scenario also holds true in the case that the dispatch-queue is not empty). The logic behind moving the I/O requests from either the sort or the FIFO lists is based on the scheduler's goal to ensure that each read request is processed by its effective deadline, without starving the queued-up write requests. In this design, the goal of economizing the disk seek time is accomplished by moving a larger batch of requests from the sort list (logical block number sorted), and balancing it with a controlled number of requests from the FIFO list. Hence, the ramification is that the deadline I/O scheduler effectively emphasizes average read request response time over disk utilization and total average I/O request response time.

To reiterate, the basic idea behind the deadline

scheduler is that all read requests are satisfied within a specified time period. On the other hand, write requests do not have any specific deadlines associated with them. As the block device driver is ready to launch another disk I/O request, the core algorithm of the deadline scheduler is invoked. In a simplified form, the fist action being taken is to identify if there are I/O requests waiting in the dispatch queue, and if yes, there is no additional decision to be made what to execute next. Otherwise it is necessary to move a new set of I/O requests to the dispatch queue. The scheduler searches for work in the following places, BUT will only migrate requests from the first source that results in a hit. (1) If there are pending write I/O requests, and the scheduler has not selected any write requests for a certain amount of time, a set of write requests is selected (see tunables in Appendix A). (2) If there are expired read requests in the `read_fifo` list, the system will move a set of these requests to the dispatch queue. (3) If there are pending read requests in the sort list, the system will migrate some of these requests to the dispatch queue. (4) As a last resource, if there are any pending write I/O operations, the dispatch queue is being populated with requests from the sorted write list. In general, the definition of a certain amount of time for write request starvation is normally 2 iterations of the scheduler algorithm (see Appendix A). After two sets of read requests have been moved to the dispatch queue, the scheduler will migrate some write requests to the dispatch queue. A set or batch of requests can be (as an example) 64 contiguous requests, but a request that requires a disk seek operation counts the same as 16 contiguous requests.

2.1 The 2.6 Anticipatory I/O scheduler

The anticipatory (AS) I/O scheduler's design attempts to reduce the per thread read response time. It introduces a controlled delay component into the dispatching equation [5],[9],[11]. The delay is being invoked on any new read request to the device driver, thereby allowing a thread that just finished its read I/O request to submit a new read request, basically enhancing the chances (based on locality) that this scheduling behavior will result in smaller seek operations. The tradeoff between reduced seeks and decreased disk utilization (due to the additional delay factor in dispatching a request) is managed by utilizing an actual cost-benefit analysis [9].

The next few paragraphs discuss the general design of an anticipatory I/O scheduler, outlining the different components that comprise the I/O framework. Basically, as a read I/O request completes, the I/O framework stalls for a brief amount of time, awaiting additional requests to arrive, before dispatching a new request to the disk subsystem. The focus of this design is on applications threads that rapidly generate another I/O request that could potentially be serviced before the scheduler chooses another task, and by doing so, deceptive idleness may be avoided [9]. Deceptive idleness is defined as a condition that forces the scheduler into making a decision too early, basically by assuming that the thread issuing the last request has momentarily no further disk request lined up, and hence the scheduler selects an I/O request from another task. The design discussed here argues that the fact that the disk remains idle during the short stall period is not necessarily detrimental to I/O performance. The question of whether (and for how long) to wait at any given decision point is key to the effectiveness and performance of the implementation. In practice, the framework waits for the shortest possible period of time for which the scheduler expects (with a high probability) the benefits of actively waiting to outweigh the costs of keeping the disk subsystem in an idle state. An assessment of the costs and benefits is only pos-

sible relative to a particular scheduling policy [11]. To elaborate, a seek reducing scheduler may wish to wait for contiguous or proximal requests, whereas a proportional-share scheduler may prefer weighted fairness as one of its primary criteria. To allow for such a high degree of flexibility, while trying to minimize the burden on the development efforts for any particular disk scheduler, the anticipatory scheduling framework consists of 3 components [9]. (1) The original disk scheduler, which implements the scheduling policy and is unaware of any anticipatory scheduling techniques. (2) An actual scheduler independent anticipation core. (3) An adaptive scheduler-specific anticipation heuristic for seek reducing (such as SPTF or C-SCAN) as well as any potential proportional-share (CFQ or YFQ) scheduler. The anticipation core implements the generic logic and timing mechanisms for waiting, and relies on the anticipation heuristic to decide if and for how long to wait. The actual heuristic is implemented separately for each disk scheduler, and has access to the internal state of the scheduler. To apply anticipatory scheduling to a new scheduling policy, it is merely necessary to implement an appropriate anticipation heuristic.

Any traditional work-conserving I/O scheduler operates in two states (known as idle and busy). Applications may issue I/O requests at any time, and these requests are normally being placed into the scheduler's pool of requests. If the disk subsystem is idle at this point, or whenever another request completes, a new request is being scheduled, the scheduler's select function is called, whereupon a request is chosen from the pool and dispatched to the disk device driver. The anticipation core forms a wrapper around this traditional scheduler scheme. Whenever the disk becomes idle, it invokes the scheduler to select a candidate request (still basically following the same philosophy as always). However, instead of dequeuing and dispatching a request immediately, the framework first passes the request to the anticipation heuristic for evaluation. A return value (result) of zero indicates that the heuristic has deemed it pointless to wait and the core therefore proceeds to dispatch the candidate request. However, a positive integer as a return value represents the waiting period in microseconds that the heuristic deems suitable. The core initiates a timeout for that particular time period, and basically enters a new wait state. Though the disk is inactive, this state is considered different from idling (while having pending requests and an active timeout). If the timeout expires before the arrival of any new request, the previously chosen request is dispatched without any further delay. However, new requests may arrive during the wait period and these requests are added to the pool of I/O requests. The anticipation core then immediately requests the scheduler to select a new candidate request from the pool, and initiates communication with the heuristic to evaluate this new candidate. This scenario may lead to an immediate dispatch of the new candidate request, or it may cause the core to remain in the wait state, depending on the scheduler's selection and the anticipation heuristic's evaluation. In the latter case, the original timeout remains in effect, thus preventing unbounded waiting situations by repeatedly re-triggering the timeout.

As the heuristic being used is disk scheduler dependent, the discussion here only generalizes on the actual implementation techniques that may be utilized. Therefore, the next few paragraphs discuss a shortest positioning time first (SPTF) based implementation, where the disk scheduler determines the positioning time for each available request based on the current head position, and basically chooses the request that results into the shortest seek distance. In general, the heuristic has to evaluate the candidate request that was chosen by the scheduling policy. The intuition is that if

the candidate I/O request is located close to the current head position, there is no need to wait on any other requests. Assuming synchronous I/O requests initiated by a single thread, the task that issued the last request is likely to submit the next request soon, and if this request is expected to be close to the current request, the heuristic decides to wait for this request [11]. The waiting period is chosen as the expected YZ percentile (normally around 95%) think-time, within which there is a XZ probability (again normally 95%) that a request will arrive. This simple approach is transformed and generalized into a succinct cost-benefit equation that is intended to cover the entire range of values for the head positioning, as well as the think-times. To simplify the discussion, the adaptive component of the heuristic consists of collecting online statistics on all the disk requests to estimate the different time variables that are being used in the decision making process. The expected positioning time for each process represents a weighted-average over the time of the positing time for requests from that process (as measured upon request completion). Expected median and percentile think-times are estimated by maintaining a decayed frequency table of request think-times for each process.

The Linux 2.6 implementation of the anticipatory I/O scheduler follows the basic idea that if the disk drive just operated on a read request, the assumption can be made that there is another read request in the pipeline, and hence it is worth while to wait [5]. As discussed, the I/O scheduler starts a timer, and at this point there are no more I/O requests passed down to the device driver. If a (close) read request arrives during the wait time, it is serviced immediately and in the process, the actual distance that the kernel considers as close grows as time passes (the adaptive part of the heuristic). Eventually the close requests will dry out and the scheduler will decide to submit some of the write requests (see Appendix A).

2.2 The 2.6 CFQ Scheduler

The Completely Fair Queuing (CFQ) I/O scheduler can be considered to represent an extension to the better known Stochastic Fair Queuing (SFQ) implementation [12]. The focus of both implementations is on the concept of fair allocation of I/O bandwidth among all the initiators of I/O requests. An SFQ-based scheduler design was initially proposed (and ultimately being implemented) for some network scheduling related subsystems. The goal to accomplish is to distribute the available I/O bandwidth as equally as possible among the I/O requests. The implementation utilizes n (normally 64) internal I/O queues, as well as a single I/O dispatch queue. During an enqueue operation, the PID of the currently running process (the actual I/O request producer) is utilized to select one of the internal queues (normally hash based) and hence, the request is basically inserted into one of the queues (in FIFO order). During dequeue, the SFQ design calls for a round robin based scan through the non-empty I/O queues, and basically selects requests from the head of the queues. To avoid encountering too many seek operations, an entire round of requests is collected, sorted, and ultimately merged into the dispatch queue. In a next step, the head request in the dispatch queue is passed to the device driver. Conceptually, a CFQ implementation does not utilize a hash function. Therefore, each I/O process gets an internal queue assigned (which implies that the number of I/O processes determines the number of internal queues). In Linux 2.6.5, the CFQ I/O scheduler utilizes a hash function (and a certain amount of request queues) and therefore resembles an SFQ implementation. The CFQ, as well as the SFQ implementations strives to manage per-process I/O bandwidth, and provide fairness at the level of pro-

cess granularity.

2.3 The 2.6 noop I/O scheduler

The Linux 2.6 noop I/O scheduler can be considered as a rather minimal overhead I/O scheduler that performs and provides basic merging and sorting functionalities. The main usage of the noop scheduler revolves around non disk-based block devices (such as memory devices), as well as specialized software or hardware environments that incorporate their own I/O scheduling and (large) caching functionality, and therefore require only minimal assistance from the kernel. Therefore, in large I/O subsystems that incorporate RAID controllers and a vast number of contemporary physical disk drives (TCQ drives), the noop scheduler has the potential to outperform the other 3 I/O schedulers as the workload increases.

2.4 I/O Scheduler—Performance Implications

The next few paragraphs augment on the I/O scheduler discussion, and introduce some additional performance issues that have to be taken into consideration while conducting an I/O performance analysis. The current AS implementation consists of several different heuristics and policies that basically determine when and how I/O requests are dispatched to the I/O controller(s). The elevator algorithm that is being utilized in AS is similar to the one used for the deadline scheduler. The main difference is that the AS implementation allows limited backward movements (in other words supports backward seek operations) [1]. A backward seek operation may occur while choosing between two I/O requests, where one request is located behind the elevator's current head position while the other request is ahead of the elevator's current position.

The AS scheduler utilizes the lowest logical block information as the yardstick for sorting, as well as determining the seek distance. In the case that the seek distance to the request behind the elevator is less than half the seek distance to the request in front of the elevator, the request behind the elevator is chosen. The backward seek operations are limited to a maximum of MAXBACK (1024 * 1024) blocks. This approach favors the forward movement progress of the elevator, while still allowing short backward seek operations. The expiration time for the requests held on the FIFO lists is tuneable via the parameter's `read_expire` and `write_expire` (see Appendix A). When a read or a write operation expires, the AS I/O scheduler will interrupt either the current elevator sweep or the read anticipation process to service the expired request(s).

2.5 Read and Write Request Batches

An actual I/O batch is described as a set of read or write requests. The AS scheduler alternates between dispatching either read or write batches to the device driver. In a read scenario, the scheduler submits read requests to the device driver, as long as there are read requests to be submitted, and the read batch time limit (`read_batch_expire`) has not been exceeded. The clock on `read_batch_expire` only starts in the case that there are write requests pending. In a write scenario, the scheduler submits write requests to the device driver as long as there are pending write requests, and the write batch time limit `write_batch_expire` has not been exceeded. The heuristic used insures that the length of the write batches will gradually be shortened if there are read batches that frequently exceed their time limit.

When switching between read and write requests, the scheduler waits until all the requests from the previous batch are completed before scheduling any new requests. The read

and write FIFO expiration time is only being checked when scheduling I/O for a batch of the corresponding (read or write) operation. To illustrate, the read FIFO timeout values are only analyzed while operating on read batches. Along the same lines, the write FIFO timeout values are only consulted while operating on write batches. Based on the used heuristics and policies, it is generally not recommended to set the read batch time to a higher value than the write expiration time, or to set the write batch time to a greater value than the read expiration time. As the IO scheduler switches from a read to a write batch, the I/O framework launches the elevator with the head request on the write expired FIFO list. Likewise, when switching from a write to a read batch, the I/O scheduler starts the elevator with the first entry on the read expired FIFO list.

2.6 Read Anticipation Heuristic

The process of read anticipation solely occurs when scheduling a batch of read requests. The AS implementation only allows one read request at a time to be dispatched to the controller. This has to be compared to either the many write request scenario or the many read request case if read anticipation is deactivated. In the case that read anticipation is enabled (`antic_expire = 0`), read requests are dispatched to the (disk or RAID) controller one at a time. At the end of each read request, the I/O scheduler examines the next read request from the sorted read list (an actual rb-tree) [1]. If the next read request belongs to the same process as the request that just completed, or if the next request in the queue is close (data block wise) to the just completed request, the request is being dispatched immediately. Otherwise, the statistics (average think-time and seek distance) available for the process that just completed are being examined (cost-benefit analysis). The statistics are associated with each process, but these statistics are not associated with a specific I/O device *per se*. To illustrate, the approach works more efficiently if there is a one-to-one correlation between a process and a disk. In the case that a process is actively working I/O requests on separate devices, the actual statistics reflect a combination of the I/O behavior across all the devices, skewing the statistics and therefore distorting the facts. If the AS scheduler guesses right, very expensive seek operations can be omitted, and hence the overall I/O throughput will benefit tremendously. In the case that the AS scheduler guesses wrong, the `antic_expire` time is wasted. In an environment that consists of larger (HW striped) RAID systems and tag command queuing (TCQ) capable disk drives, it is more beneficial to dispatch an entire batch of read requests and let the controllers and disk do their magic.

From a physical disk perspective, to locate specific data, the disk drive's logic requires the cylinder, the head, and the sector information [17]. The cylinder specifies the track on which the data resides. Based on the layering technique used, the tracks underneath each other form a cylinder. The head information identifies the specific read/write head (and therefore the exact platter). The search is now narrowed down to a single track on a single platter. Ultimately, the sector value reflects the sector on the track, and the search is completed. Contemporary disk subsystems do not communicate in terms of cylinders, heads and sectors. Instead, modern disk drives map a unique block number over each cylinder/head/sector construct. Therefore, that (unique) reference number identifies a specific cylinder/head/sector combination. Operating systems address the disk drives by utilizing these block numbers (logical block addressing), and hence the disk drive is responsible for translating the block number into the appropriate cylinder/head/sector value. The culprit is

that it is not guaranteed that the physical mapping is actually sequential. But the statement can be made that there is a rather high probability that a logical block *n* is physically adjacent to a logical block *n+1*. The existence of the discussed sequential layout is paramount to the I/O scheduler performing as advertised. Based on how the read anticipatory heuristic is implemented in AS, I/O environments that consist of RAID systems (operating in a hardware stripe setup) may experience a rather erratic performance behavior. This is due to the current AS implementation that is based on the notion that an I/O device has only one physical (seek) head, ignoring the fact that in a RAID environment, each physical disk has its own physical seek head construct. As this is not recognized by the AS scheduler, the data being used for the statistics analysis is skewed. Further, disk drives that support TCQ perform best when being able to operate on *n* (and not 1) pending I/O requests. The read anticipatory heuristic basically disables TCQ. Therefore, environments that support TCQ and/or consist of RAID systems may benefit from either choosing an alternate I/O scheduler or from setting the `antic_expire` parameter to 0. The tuning allows the AS scheduler to behave similarly to the deadline I/O scheduler (the emphasis is on behave and not performance).

3 I/O Components that Affect Performance

In any computer system, between the disk drives and the actual memory subsystem is a hierarchy of additional controllers, host adapters, bus converters, and data paths that all impact I/O performance in one way or another [17]. Linux file systems submit I/O requests by utilizing `submit_bio()`. This function submits requests by utilizing the request function as specified during queue creation. Technically, device drivers do not have to use the I/O scheduler, however all SCSI devices in Linux utilize the scheduler by virtue of the SCSI mid-layer [1]. The `scsi_alloc_queue()` function calls `blk_init_queue()`, which sets the request function to `scsi_request_fn()`. The `scsi_request_fn()` function takes requests from the I/O scheduler (on dequeue), and passes them down to the device driver.

3.1 SCSI Operations

In the case of a simple SCSI disk access, the request has to be processed by the server, the SCSI host adapter, the embedded disk controller, and ultimately by the disk mechanism itself. As the OS receives the I/O request, it converts the request into a SCSI command packet. In the case of a synchronous request, the calling thread surrenders the CPU and transitions into a sleep state until the I/O operation is completed. In a next step, the SCSI command is transferred across the server's I/O bus to the SCSI host adapter. The host adapter is responsible for interacting with the target controller and the respective devices. In a first step, the host adapter selects the target by asserting its control line onto the SCSI-bus (as the bus becomes available). This phase is known as the SCSI selection period. As soon as the target responds to the selection process, the host adapter transfers the SCSI command to the target. This section of the I/O process is labeled as the command phase. If the target is capable of processing the command immediately, it either returns the requested data or the status information.

In most circumstances, the request can only be processed immediately if the data is available in the target controller's cache. In the case of a `read()` request, the data is normally not available. This results into the target disconnecting from the SCSI bus to allow other SCSI operations to be processed. If the I/O opera-

tion consists of a `write()` request, the data phase is followed immediately by a command phase on the bus, as the data is transferred into the target's cache. At that stage, the target disconnects from the bus. After disconnecting from the bus, the target resumes its own processing while the bus can be utilized by other SCSI requests. After the physical I/O operation is completed on the target disk, the target controller competes again for the bus, and reconnects as soon as the bus is available. The reconnect phase is followed by a data phase (in the case of `read()` operation) where the data is actually being moved. The data phase is followed by another status phase to describe the results of the I/O operation. As soon as the SCSI host adapter receives the status update, it verifies the proper completion of the request and notifies the OS to interrupt the requesting worker thread. Overall, the simple SCSI I/O request causes 7 phase changes consisting of a select, a command, a disconnect, a reconnect, a data, a status, and a disconnect operation. Each phase consumes time and contributes to the overall I/O processing latency on the system.

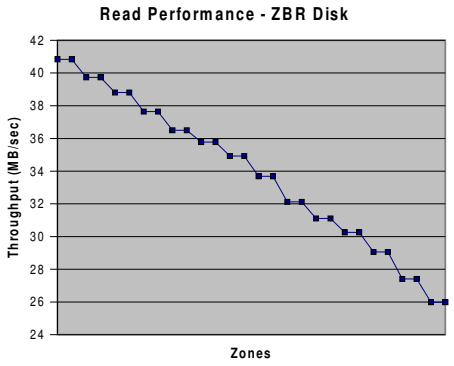
3.2 SCSI Disk Fence

When discussing SCSI disks, it is imperative to understand the performance impact of a relatively obscure disk control parameter that is labeled as the fence. When a SCSI disk recognizes a significant delay (such as a seek operation) in a `read()` request, the disk will surrender the bus. At the point where the disk is ready to transfer the data, the drive will again contend for the bus so that the `read()` request can be completed. The fence parameter determines the time at which the disk will begin to contend for the SCSI bus. If the fence is set to 0 (the minimum), the disk will contend for the SCSI bus after the first sector has been transferred into the disk controller's memory. In the case where the fence is set to 255 (the maximum), the disk will wait until almost all the requested data has been accumulated in the controller's memory before contending for the bus.

The performance implication of setting the fence to a low value is a reduced response time, but results in a data transfer that happens basically at disk speed. On the other hand, a high fence value will delay the start of the data transfer, but results in a data transfer that occurs at near burst speed. Therefore, in systems with multiple disks per adapter, a high fence value potentially increases overall throughput for I/O intensive workloads. A study by Shriver [15] observed fairness in servicing sufficiently large I/O requests (in the 16KB to 128KB range), despite the fact that the SCSI disks have different priorities when contending for the bus. Although each process attempts to progress through its requests without any coordination with other processes, a convoy behavior among all the processes was observed. Namely, all disk drives received a request and transmitted the data back to the host adapter before any disk received another request from the adapter (a behavior labeled as rounds). The study revealed that the host adapter does not arbitrate for the bus, despite having the highest priority, as long as any disk is arbitrating.

3.3 Zone Bit Recording (ZBR)

Contemporary disk drives utilize a technology called Zone Bit Recording to increase capacity [17]. Incorporating the technology, cylinders are grouped into zones, based on their distance from the center of the disk. Each zone is assigned a number of sectors per track. The outer zones contain more sectors per track compared to the inner zones that are located closer to the spindle. With ZBR disks, the actual data transfer rate varies depending on the physical sector location.



Note: Figure 1 depicts the average throughput per zone, the benchmark revealed 14 distinct performance steps.

Figure 1: ZBR Throughput Performance

Given the fact that a disk drive spins at a constant rate, the outer zones that contain more sectors will transfer data at a higher rate than the inner zones that contain fewer sectors. In this study, evaluating I/O performance on an 18.4 GB Seagate ST318417W disk drive outlined the throughput degradation for sequential read() operations based on physical sector location. The ZCAV program used in this experiment is part of the Bonnie++ benchmark suite. Figure 1 outlines the average zone read() throughput performance. It has to be pointed out that the performance degradation is not gradual, as the benchmark results revealed 14 clear distinct performance steps along the throughput curve. Another observation derived from the experiment was that for this particular ZBR disk, the outer zones revealed to be wider than the inner zones. The Seagate specifications for this particular disk cite an internal transfer rate of 28.1 to 50.7 MB/second. The measured minimum and maximum throughput `read()` values of 25.99 MB/second and 40.84 MB/second, respectively are approximately 8.1% and 19.5% (13.8% on average) lower, and represent actual throughput rates. Benchmarks conducted on 4 other ZBR drives revealed a similar picture. On average, the actual system throughput rates were 13% to 15% lower than what was cited in the vendor specifications. Based on the conducted research, this text proposes a first-order ZBR approximation nominal disk transfer rate model (for a particular request size *req* and a disk capacity *cap*) that is defined in Equation 1 as:

$$ntr_{zbr}(req) = 0.85 \cdot \left(tr_{max}\right) - \frac{req \cdot \left(tr_{max} - tr_{min}\right)}{cap} \quad (1)$$

tr_{max} = maximum disk specific internal transfer speed

tr_{min} = minimum disk specific internal transfer speed

The suggested throughput regulation factor of 0.85 was derived from the earlier observation that throughput rates adjusted for factors such as sector overhead, error correction, or track and cylinder skewing issues resulted in a drop of approximately 15% compared to the manufacturer reported transfer rates. This study argues that the manufacturer reported transfer rates could be more accurately defined as instantaneous bit rates at the read-write heads. It has to be emphasized that the calculated throughput rates derived from the presented model will have to be adjusted onto the target system's ability to sustain the I/O rate.

The theories of progressive chaos imply that anything that evolves out of a perfect order will over time become disordered due to outside forces. The progressive chaos concept can certainly be applied to I/O performance. The dynamic allocation (as well as de-allocation) of file system resources contributes to the progressive chaos scenario encountered in virtually any file system designs. Form a device driver and physical disk drive perspective, the results of disk access optimization strategies

are first, that the number of transactions per second is maximized and second, that the order in which the requests are being received is not necessarily the order the requests are getting processed. Thus, the response time of any particular request can not be guaranteed. A request queue may increase spatial locality by selecting requests in an order to minimize the physical arm movement (a workload transformation), but may also increase the perceived response time because of queuing delays (a behavior transformation). The argument made in this study is that the interrelationship of some the discussed I/O components has to be taken into consideration while evaluating and quantifying performance

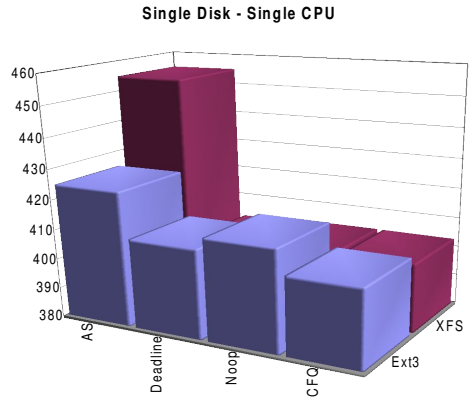
4 I/O Schedulers and Performance

The main goal of this study was to quantify I/O performance (focusing on the Linux 2.6 I/O schedulers) under varying workload scenarios and hardware configurations. Therefore, the benchmarks were conducted on a single-CPU single-disk system, a midrange 8-way NUMA RAID-5 system, and a 16-way SMP system that utilized a 28-disk RAID-0 configuration. The reader is referred to Appendix B for a more detailed description of the different benchmark environments. As a workload generator, the study utilized the flexible file system benchmark (FFSB) infrastructure [8]. FFSB represents a benchmarking environment that allows analyzing I/O performance by simulating basically any I/O pattern imaginable. The benchmarks can be executed on multiple individual file systems, utilizing an adjustable number of worker threads, where each thread may either operate out of a combined or a thread-based I/O profile. Aging the file systems, as well as collecting systems utilization and throughput statistics is part of the benchmarking framework. Next to the more traditional sequential read and sequential write benchmarks, the study used a filer server, a web server, a mail server, as well as a metadata intensive I/O profile (see Appendix B). The file, as well as the mail server workloads (the actual transaction mix) was based on Intel's Iometer benchmark [18], whereas the mail server transaction mix was loosely derived from the SPECmail2001 I/O profile [19]. The I/O analysis in this study was composed of two distinct focal points. One emphasis of the study was on aggregate I/O performance achieved across the 4 benchmarked workload profiles, whereas a second emphasis was on the sequential read and write performance behavior. The emphasis on aggregate performance across the 4 distinct workload profiles is based on the claim made that an I/O scheduler has to provide adequate performance in a variety of workload scenarios and hardware configurations, respectively. All the conducted benchmarks were executed with the default tuning values (if not specified otherwise) in an ext3 as well as an xfs file system environment. In this paper, the term response time represents the total run time of the actual FFSB benchmark, incorporating all the I/O operations that are executed by the worker threads.

5 Single-CPU Single-Disk Setup

The normalized results across the 4 workload profiles revealed that the deadline, the noop, as well as the CFQ schedulers performed within 2% and 1% percent on ext3 and xfs (see Figure 2). On ext3, the CFQ scheduler had a slight advantage, whereas on xfs the deadline scheduler provided the best aggregate (normalized) response time. On both file systems, the AS scheduler represented the least efficient solution, trailing the other I/O schedulers by 4.6% and 13% on ext3 and xfs, respectively. Not surprisingly, among the 4 workloads benchmarked in a single disk system, AS trailed the other 3 I/O schedulers by a rather significant

margin in the Web Server scenario (which reflects 100% random read operations). On sequential read operations, the AS scheduler outperformed the other 3 implementations by an average of 130% and 127% on ext3 and xfs. The sequential read results clearly support the discussion in this paper on where the design focus for AS was directed. In the case of sequential write operations, AS revealed the most efficient solution on ext3, whereas the noop scheduler provided the best throughput on xfs. The performance delta (for the sequential write scenarios) among the I/O schedulers was 8% on ext3 and 2% on xfs (see Appendix C).

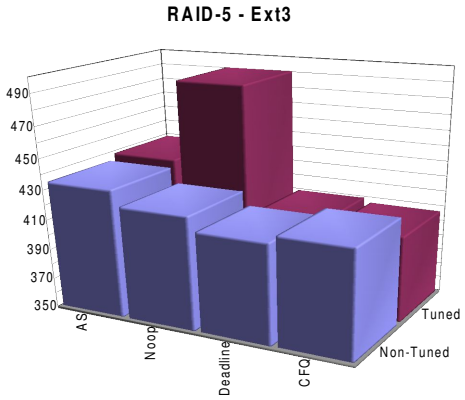


Note: In Figure 2, the x-axis depicts the I/O schedulers. The front row reflects the ext3 setup, whereas the back row shows xfs. The y-axis discloses the aggregate (normalized) response time over the 4 benchmarked profiles per I/O scheduler.

Figure 2: Aggregate Response Time (Normalized)

5.1 8-Way RAID-5 Setup

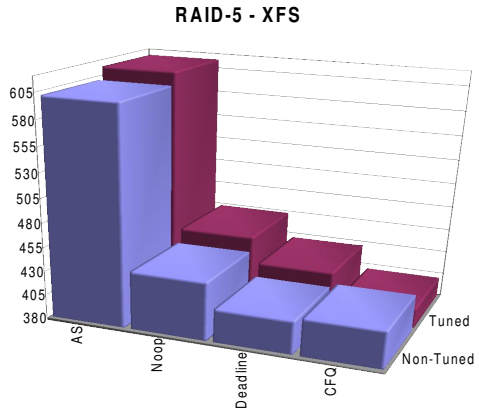
In the RAID-5 environment, the normalized response time values (across the 4 profiles) disclosed that the deadline scheduler provided the most efficient solution on ext3 as well as xfs (see Figure 3 and Figure 4). While executing in an ext3 environment, all 4 I/O schedulers were within 4.5%, with the AS I/O scheduler trailing noop and CFQ by approximately 2.5%. On xfs, the study clearly disclosed a profound AS I/O inefficiency while executing the metadata benchmark. The delta among the schedulers on xfs was much larger than on ext3, as the CFQ, noop, and AS implementations trailed the deadline scheduler by 1%, 6%, and 145%, respectively (see Appendix C). As in the single disk setup, the AS scheduler provided the most efficient sequential read performance. The gap between AS and the other 3 implementations shrunk though rather significantly compared to the single disk scenarios. The average sequential read throughput (for the other 3 schedulers) was approximately 20% less on both ext3 and xfs, respectively. The sequential write performance was dominated by the CFQ scheduler's response time that outperformed the other 3 solutions. The delta between the most (CFQ) and the least efficient implementation was 22% (AS) and 15% (noop) on ext3 and xfs, respectively (see Appendix C).



Note: In Figure 3, the x-axis depicts the I/O schedulers. The front-row reflects the non-tuned, and the back-row the tuned environments. The y-axis discloses the normalized response time (over the 4 profiles) per I/O scheduler.

Figure 3: EXT3 Aggregate Response Time (Normalized)

In a second phase, all the I/O scheduler setups were tuned by adjusting the (per block device)



Note: In Figure 4, the x-axis depicts the I/O schedulers. The front-row reflects the non-tuned, and the back-row the tuned environments. The y-axis discloses the normalized response time (over the 4 profiles) per I/O scheduler.

Figure 4: XFS Aggregate Response Time (Normalized)

tunable `nr_requests` (I/O operations in fly) from its default value of 128 to 2,560. The results revealed that the CFQ scheduler reacted in a rather positive way to the adjustment, and ergo was capable to provide on ext3 as well as on xfs the most efficient solution. The tuning resulted into decreasing the response time for CFQ in all the conducted (workload profile based) benchmarks on both file systems (see Appendix C). While CFQ benefited from the tuning, the results for the other 3 implementations were inconclusive. Based on the profile, the tuning either resulted in a gain or a loss in performance. As CFQ is designed to operate on larger sets of I/O requests, the results basically reflect the design goals of the scheduler [1]. This is in contrast to the AS implementation, where by design, any read intensive workload can not directly benefit from the change. On the other hand, in the case sequential write operations are being executed, AS was capable of taking advantage of the tuning as the response time decreased by 7% and 8% on ext3 and xfs, respectively. The conducted benchmarks revealed another significant inefficiency behavior in the I/O subsystem, as the write performance (for all the schedulers) on ext3 was significantly lower (by a factor of approximately 2.1) than on xfs. The culprit here is the ext3 reservation code. Ext3 patches to resolve the issue are available from kernel.org.

5.2 16-Way RAID-0 Setup

Utilizing the 28 disk RAID-0 configuration as the benchmark environment revealed that across the 4 workload profiles, the deadline implementation was able to outperform the other 3 schedulers (see Appendix C). It has to be pointed out though that the CFQ, as well as the noop scheduler, slightly outperformed the deadline implementation in 3 out of the 4 benchmarks. Overall, the deadline scheduler gained a substantial lead processing the Web server profile (100% random read requests), outperforming the other 3 implementations by up to 62%. On ext3, the noop scheduler reflected the most efficient solution while operating on sequential read and write requests, whereas on xfs, CFQ and deadline dominated the sequential read and write benchmarks. The performance delta among the schedulers (for the 4 profiles) was much more noticeable on xfs (38%) than on ext3 (6%), which reflects a similar behavior as encountered on the RAID-5 setup. Increasing nr_requests to 2,560 on the RAID-0 system led to inconclusive results (for all the I/O schedulers) on ext3 as well as xfs. The erratic behavior encountered in the tuned, large RAID-0 environment is currently being investigated.

5.3 AS Sequential Read Performance

To further illustrate and basically back up the claim made in Section 2 that the AS scheduler

design views the I/O subsystem based on a notion that an I/O device has only one physical (seek) head, this study analyzed the sequential read performance in different hardware setups. The results were being compared to the CFQ scheduler. In the single disk setup, the AS implementation is capable of approaching the capacity of the hardware, and therefore provides optimal throughput performance. Under the same workload conditions, the CFQ scheduler substantially hampers throughput performance, and does not allow the system to fully utilize the capacity of the I/O subsystem. The described behavior holds true for the ext3 as well as the xfs file system. Hence, the statement can be made that in the case of sequential read operations and CFQ, the I/O scheduler (and not the file system per se) reflects the actual I/O bottleneck. This picture is being reversed as the capacity of the I/O subsystem is being increased.

HW Setup	AS	CFQ
1 Disk	52 MB/sec	23 MB/sec
RAID-5	46 MB/sec	39 MB/sec
RAID-0	31 MB/sec	158 MB/sec

Table 1: AS vs. CFQ Sequential Read Performance

As depicted in Table 1, the CFQ scheduler approaches first, the throughput of the AS implementation in the benchmarked RAID-5 environment and second, is capable of approaching the capacity of the hardware in the large RAID-0 setup. In the RAID-0 environment, the AS scheduler only approaches approximately 17% of the hardware capacity (180 MB/sec). To reiterate, the discussed I/O behavior is reflected in the ext3 as well as the xfs benchmark results. From any file system perspective, performance should not degrade if the size of the file system, the number of files stored in the file system, or the size of the individual files stored in the file system increases. Further, the performance of a file system is supposed to approach the capacity of the hardware (workload dependent of course). This study clearly outlines that in the discussed workload scenario, the 2 benchmarked file systems are capable of achieving these goals, but only in the case the I/O schedulers are exchanged depending on the physical hardware setup. The fact that the read-ahead code in Linux 2.6 has to operate as efficiently as possible (in conjunction with the I/O scheduler and the file system) has to be considered here as well.

5.4 AS verses deadline Performance

Based on the benchmarked profiles and hardware setups, the AS scheduler provided in most circumstances the least efficient I/O solution. As the AS framework represents an extension to the deadline implementation, this study explored the possibility of tuning AS to approach deadline behavior. The tuning consisted of setting `nr_requests` to 2,560, `antic_expire` to 0, `read_batch_expire` to 1,000, `read_expire` to 500, `write_batch_expire` to 250, and `write_expire` to 5,000. Setting the `antic_expire` value to 0 (by design) basically disables the anticipatory portion of the scheduler. The benchmarks were executed utilizing the RAID-5 environment, and the results were compared to the deadline performance results reported this study. On ext3, the non-tuned AS version trailed the non-tuned deadline setup by approximately 4.5% (across the 4 profiles). Tuning the AS scheduler resulted into a substantial performance boost, as the benchmark results revealed that the tuned AS implementation outperformed the default deadline setup by approximately 6.5% (see Appendix C). The performance advantage was squandered though while comparing the tuned AS solution against the deadline environment with `nr_requests` set to 2,560. Across

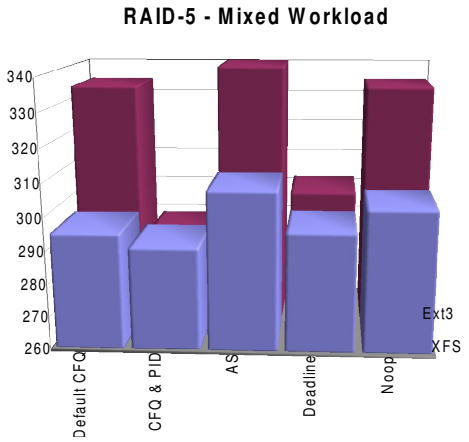
the 4 workload profiles, deadline again outperformed the AS implementation by approximately 17%. As anticipated, setting `antic_expire` to 0 resulted into lower sequential read performance, stabilizing the response time at deadline performance (see Appendix C). On xfs, the results were (based on the rather erratic metadata performance behavior of AS) inconclusive. One of the conclusions is that based on the current implementation of the AS code that collects the statistical data, the implemented heuristic is not flexible enough to detect any prolonged random I/O behavior, a scenario where it would be necessary to deactivate the active wait behavior. Further, setting `antic_expire` to 0 should force the scheduler into deadline behavior, a claim that is not backed up by the empirical data collected for this study. One explanation for the discrepancy is that the short backward seek operations supported in AS are not part of the deadline framework. Therefore, depending on the actual physical disk scheduling policy, the AS backward seek operations may be counterproductive from a performance perspective.

5.5 CFQ Performance

The benchmarks conducted revealed that the tuned CFQ setup provided the most efficient solution for the RAID-5 environment (see Section 5.1). Therefore, the study further explored varies ways to improve the performance of the CFQ framework. The CFQ I/O scheduler in Linux 2.6.5 resembles a SFQ implementation, which operates on a certain number or internal I/O queues and hashes on a per process granularity to determine where to place an I/O request. More specifically, the CFQ scheduler in 2.6.5 hashes on the thread group id (tgid), which represents the process PID as in POSIX.1 [1]. The approach chosen was to alter the CFQ code to hash on the Linux PID. This code change introduces fairness on a per thread (instead of per process) granularity, and therefore alters the distribution of the I/O requests in the internal queues. In addition, the `cfq_quantum` and `cfq_queued` parameters of the CFQ framework were exported into user space.

In a first step, the default tgid based CFQ version with `cfq_quantum` set to 32 (default equals to 8) was compared to the PID based implementation that used the same tuning configuration. Across the 4 profiles, the PID based implementation reflected the more efficient solution, processing the I/O workloads approximately 4.5% and 2% faster on ext3 and xfs, respectively. To further quantify the performance impact of the different hash methods (tgid verses PID based), in a second step, the study compared the default Linux 2.6.5 CFQ setup to the PID based code that was configured with `cfq_quantum` adjusted to 32 (see Appendix C). Across the 4 profiles benchmarked on ext3, the new CFQ scheduler that hashed on a PID granularity outperformed the status quo by approximately 10%. With the new method, the sequential read and write performance improved by 3% and 4%, respectively. On xfs (across the 4 profiles), the tgid based CFQ implementation proved to be the more efficient solution, outperforming the PID based setup by approximately 9%. On the other hand, the PID based solution was slightly more efficient while operating on the sequential read (2%) and write (1%) profiles. The ramification is that based on the conducted benchmarks and file system configurations, certain workload scenarios can be processed more efficiently in a tuned, PID hash based configuration setup.

To further substantiate the potential of the proposed PID based hashing approach, a mixed I/O workload (consisting of 32 concurrent threads) was benchmarked. The environment used reflected the RAID-5 setup. The I/O profile was decomposed in 4 subsets of 8 worker



Note: In Figure 5, the x-axis depicts the I/O schedulers. The front row reflects the xfs, whereas the back row depicts the ext3 based environment. The y-axis discloses the actual response time for the mixed workload profile.

Figure 5: Mixed Workload Behavior

threads, each subset executing either 64KB sequential read, 4KB random read, 4KB random write, or 256KB sequential write operations (see Figure 5). The benchmark results revealed that in this mixed I/O scenario, the PID based CFQ solution (tuned with `cfq_quantum = 32`) outperformed the other I/O schedulers by at least 5% and 2% on ext3 and xfs, respectively (see Figure 5 and Appendix C). The performance delta among the schedulers was greater on ext3 (15%) than on xfs (6%).

6 Conclusions and Future Work

The benchmarks conducted on varying hardware configurations revealed a strong (setup based) correlation among the I/O scheduler, the workload profile, the file system, and ultimately I/O performance. The empirical data disclosed that most tuning efforts resulted in reshuffling the scheduler performance ranking. The ramification is that the choice of an I/O scheduler has to be based on the workload pattern, the hardware setup, as well as the file system used. To reemphasize the importance of the discussed approach, an additional benchmark was conducted utilizing a Linux 2.6 SMP system, the jfs file system, and a large RAID-0 configuration, consisting of 84 RAID-0 systems (5 disks each). The SPECsfs [20] benchmark was used as the workload generator. The focus was on determining the highest throughput achievable in the RAID-0 setup by only substituting the I/O scheduler between SPECsfs runs. The results revealed that the noop scheduler was able to outperform the CFQ, as well as the AS scheduler. The result reverses the order, and basically contradicts the ranking established for the RAID-5 and RAID-0 environments benchmarked in this study. On the smaller RAID systems, the noop scheduler was not able to outperform the CFQ implementation in any random I/O test. In the large RAID-0 environment, the 84 rb-tree data structures that have to be maintained (from a memory as well as a CPU perspective) in CFQ represent a substantial, noticeable overhead factor.

The ramification is that there is no silver bullet (a.k.a. I/O scheduler) that consistently provides the best possible I/O performance. While the AS scheduler excels on small configurations in a sequential read scenario, the non-tuned deadline solution provides acceptable performance on smaller RAID systems. The CFQ scheduler revealed the most potential from a tuning perspective on smaller RAID-5 systems, as increasing the `nr_requests` parameter provided the lowest response time. As the noop scheduler represents a rather light-way solution, large RAID systems that consist of many individual logical devices may benefit from the reduced memory, as well as CPU overhead encountered by this solution. On large RAID systems that consist of many logical devices, the other 3 implementations have to maintain (by design) rather complex data structures as part of the operating framework. Further, the study

revealed that the proposed PID based and tunable CFQ implementation reflects a valuable alternative to the standard CFQ implementation. The empirical data collected on a RAID-5 system supports that claim, as true fairness on a per thread basis is being introduced.

Future work items include analyzing the rather erratic performance behavior encountered by the AS scheduler on xfs while processing a metadata intensive workload profile. Another focal point is an in-depth analysis of the inconsistent `nr_requests` behavior observed on large RAID-0 systems. Different hardware setups will be used to aid this study. The anticipatory heuristics of the AS code used in Linux 2.6.5 is the target of another study, aiming at enhancing the adaptiveness of the (status quo) implementation based on certain workload conditions. Additional research in the area of the proposed PID based CFQ implementation, as well as branching the I/O performance study out into even larger I/O subsystems represent other work items that will be addressed in the near future.

Legal Statement

References

1. The Linux Source Code

2. Arcangeli, A., “Evolution of Linux Towards Clustering,” EFD R&D Clamart, 2003.

3. Axboe, J., “Deadline I/O Scheduler Tunables,” SuSE, EDF R&D, 2003

4. Corbet, J., “A new deadline I/O scheduler.” `http://lwn.net/Articles/10874`.

5. Corbet, J., “Anticipatory I/O scheduling.” `http://lwn.net/Articles/21274`.

6. Corbet, J., “The Continuing Development of I/O Scheduling.” `http://lwn.net/Articles/21274`.

7. Corbet, J., “Porting drivers to the 2.5 kernel,” Linux Symposium, Ottawa, Canada, 2003.

8. Heger, D., Jacobs, J., McCloskey, B., Stultz, J., “Evaluating Systems Performance in the Context of Performance Paths,” IBM Technical White Paper, Austin, 2000.

9. Iyer, S., Drushel, P., “Anticipatory Scheduling – A disk scheduling framework to overcome deceptive idleness in synchronous I/O,” SOSP 2001

10. Lee Irwin III, W., “A 2.5 Page Clustering Implementation,” Linux Symposium, Ottawa, 2003

11. Nagar, S., Franke, H., Choi, J., Seetharaman, C., Kaplan, S., Singhvi, N., Kashyap, V., Kravetz, M., “Class-Based Prioritized Resource Control in Linux,” 2003 Linux Symposium.

12. McKenney, P., “Stochastic Fairness

Queueing," INFOCOM, 1990

13. Molnar, I., "Goals, Design and Implementation of the new ultra-scalable O(1) scheduler." (`sched-design.txt`).

14. Mosberger, D., Eranian, S., "IA-64 Linux Kernel, Design and Implementation," Prentice Hall, NJ, 2002.

15. Shriver, E., Merchant, A., Wilkes, J., "An Analytic Behavior Model with Readahead Caches and Request Reordering," Bell Labs, 1998.

16. Wienand, I., "An analysis of Next Generation Threads on IA64," HP, 2003.

17. Zimmermann, R., Ghandeharizadeh, S., "Continuous Display Using Heterogeneous Disk-Subsystems," ACM Multimedia, 1997.

18. `http://www.iometer.org/`

19. `http://www.specbench.org/osg/mail2001`

20. `http://www.specbench.org/sfs97r1/docs/chapter1.html`

Appendix A: Scheduler Tunables

Deadline Tunables

The `read_expire` parameter (which is specified in milliseconds) is part of the actual deadline equation. As already discussed, the goal of the scheduler is to insure (basically guarantee) a start service time for a given I/O request. As the design focuses manly on read requests, each actual read I/O that enters the scheduler is assigned a deadline factor that consists of the current time plus the `read_expire` value (in milliseconds).

The `fifo_batch` parameter governs the number of request that are being moved to the dispatch queue. In this design, as a read request expires, it becomes necessary to move some I/O requests from the sorted I/O scheduler list into the block device's actual dispatch queue. Hence the `fifo_batch` parameter controls the batch size based on the cost of each I/O request. A request is qualified by the scheduler as either a seek or a stream request. For additional information, please see the discussion on the `seek_cost` as well as the `stream_unit` parameters.

The `seek_cost` parameter quantifies the cost of a seek operation compared to a `stream_unit` (expressed in Kbytes). The `stream_unit` parameter dictates how man Kbytes are used to describe a single stream unit. A stream unit has an associated cost of 1, hence if a request consists of XY Kbytes, the actual cost can be determined as *cost = (XY + stream_unit - 1)/ stream_unit*. To reemphasize, the combination of the `stream_unit`, `seek_cost`, and `fifo_batch` parameters, respectively, determine how many requests are potentially being moved as an I/O request expires.

The `write_starved` parameter (expressed in number of dispatches) indicates how many times the I/O scheduler assigns preference to read over write requests. As already discussed, when the I/O scheduler has to move requests to the dispatch queue, the preference scheme in the design favors read over write requests. However, the write requests can not be staved indefinitely, hence after the read requests were favored for `write_starved` number of times, write requests are being dispatched.

The `front_merges` parameter controls the request merge technique used by the scheduler. In some circumstances, a request may enter the scheduler that is contiguous to a request that is already in the I/O queue. It is feasible to assume that the new request may have a correla-

tion to either the front or the back of the already queued request. Hence, the new request is labeled as either a front or a back merge candidate. Based on the way files are laid out, back merge operations are more common than front merges. For some workloads, it is unnecessary to even consider front merge operations, ergo setting the `front_merges` flag to 0 disables that functionality. It has to be pointed out that despite setting the flag to 0, front merges may still happen due to the cached `merge_last` hint component. But as this feature represents an almost 0 cost factor, this is not considered as an I/O performance issue.

AS Tunables

The parameter `read_expire` governs the timeframe until a read request is labeled as expired. The parameter further controls to a certain extent the interval in-between expired requests are serviced. This approach basically equates to determining the timeslice a single reader request is allowed to use in the general presence of other I/O requests. The approximation `100 * ((seek time / read_expire) + 1)` describes the percentile of streaming read efficiency a physical disk should receive in a environment that consists of multiple concurrent read requests.

The parameter `read_batch_expire` governs the time assigned to a batch (or set) of read requests prior to serving any (potentially) pending write requests. Obviously, a higher value increases the priority allotted to read requests. Setting the value to less than `read_expire` would reverse the scenario, as at this point the write requests would be favored over the read requests. The literature suggests setting the parameter to a multiple of the `read_expire` value. The parameters `write_expire` and `write_batch_expire`, respectively, describe and govern the above-discussed behavior for any (potential) write requests.

The `antic_expire` parameter controls the maximum amount of time the AS scheduler will idle before moving on to another request. The literature suggests initializing the parameter slightly higher for large seek time devices.

Appendix B: Benchmark Environment

The benchmarking was performed in a Linux 2.6.4 environment. For this study, the CFQ I/O scheduler was back-ported from Linux 2.6.5 to 2.6.4.

1.16-way 1.7Ghz Power4+™ IBM p690 SMP system configured with 4GB memory. 28 15,000-RPM SCSI disk drives configured in a single RAID-0 setup that used Emulex LP9802-2G Fiber controllers (1 in use for the actual testing). System was configured with the Linux 2.6.4 operating system.

2.8-way NUMA system. IBM x440 with Pentium™ IV Xeon 2.0GHz processors and 512KB L2 cache subsystem. Configured with 4 qla2300 fiber-cards (only one was used in this study). The I/O subsystem consisted of 2 FAStT700 I/O controllers and utilized 15,000-RPM SCSI 18GB disk drives. The system was configured with 1GB of memory, setup as a RAID-5 (5 disks) configuration, and used the Linux 2.6.4 operating system.

3.Single CPU system. IBM x440 (8-way, only one CPU was used in this study) with Pentium™ IV Xeon 1.5GHz processor, and 512k L2 cache subsystem. The system was configured with a Adaptec aic7899 Ultra160 SCSI adapter and a single 10,000 RPM 18GB disk. The system used the Linux 2.6.4 operating system and was configured with 1GB of memory.

Workload Profiles

1. Web Server Benchmark. The benchmark utilized 4 worker threads per available CPU. In a first phase, the benchmark created several hundred thousand files ranging from 4KB to 64KB. The files were distributed across 100 directories, The goal of the create phase was to exceed the size of the memory subsystem by creating more files than what can be cached by the system in RAM. Each worker thread executed 1,000 random read operations on randomly chosen files. The workload distribution in this benchmark was derived from Intel's Iometer benchmark.

2. File Server Benchmark. The benchmark utilized 4 worker threads per available CPU. In a first phase, the benchmark created several hundred thousand files ranging from 4KB to 64KB. The files were distributed across 100 directories. The goal of the create phase was to exceed the size of the memory subsystem by creating more files than what can be cached by the system in RAM. Each worker thread executed 1,000 random read or write operations on randomly chosen files. The ratio of read to write operations on a per thread basis was specified as 80% to 20%, respectively. The workload distribution in this benchmark was derived from Intel's Iometer benchmark.

3. Mail Server Benchmark. The benchmark utilized 4 worker threads per available CPU. In a first phase, the benchmark created several hundred thousand files ranging from 4KB to 64KB. The files were distributed across 100 directories. The goal of the create phase was to exceed the size of the memory subsystem by creating more files than what can be cached by the system in RAM. Each worker thread executed 1,000 random read, create, or delete operations on randomly chosen files. The ratio of read to create to delete operations on a per thread basis was specified as 40% to 40% to 20%, respectively. The workload distribution in this benchmark was (loosely) derived from the SPECmail2001 benchmark.

4. MetaData Benchmark. The benchmark utilized 4 worker threads per available CPU. In a first phase, the benchmark created several hundred thousand files ranging from 4KB to 64KB. The files were distributed across 100 directories. The goal of the create phase was to exceed the size of the memory subsystem by creating more files than what can be cached by the system in RAM. Each worker thread executed 1,000 random create, write (append), or delete operations on randomly chosen files. The ratio of create to write to delete operations on a per thread basis was specified as 40% to 40% to 20%.

(i) Sequential Read Benchmark. The benchmark utilized 4 worker threads per available CPU. In a first phase, the benchmark created several hundred 50MB files in a single directory structure. The goal of the create phase was to exceed the size of the memory subsystem by creating more files than what can be cached by the system in RAM. Each worker thread executed 64KB sequential read operations, starting at offset 0 reading the entire file up to offset 5GB. This process was repeated on a per worker thread basis 20 times on randomly chosen files.

(ii) Sequential Write (Create) Benchmark. The benchmark utilized 4 worker threads per available CPU. Each worker thread executed 64KB sequential write operations up to a target file size of 50MB. This process was repeated on a per worker-thread basis 20 times on newly created files.

Appendix C: Raw Data Sheets (Mean Response Time in Seconds over 3 Test Runs)

	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
File Server	610.9	574.6	567.7	579.1	613.5	572.9	571.3	569.9
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
MetaData	621	634.1	623.6	597.5	883.8	781.8	773.3	771.7
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
Web Server	531.4	502.1	498.3	486.8	559	462.7	461.6	462.9
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
Mail Server	508.9	485.3	522.5	505.5	709.3	633	648.5	650.4
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
Seq. Read	405	953.2	939.4	945.4	385.2	872.8	881.3	872.4
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
Seq. Write	261.3	276.5	269.1	282.6	225.7	222.6	220.9	222.4

Table 2: Single Disk Single CPU – Mean Response Time in Seconds

	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
File Server	77.2	81.2	86.5	82.7	83.8	90.3	96.6	90.7
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
MetaData	147.8	148.4	133	145.3	205.8	90.8	101.6	100.8
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
Web Server	70.2	58.4	66.2	59.2	82.1	81.3	78.8	75.2
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
Mail Server	119.2	114.8	115.3	119.3	153.9	92.1	100.7	92.2
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
Seq. Read	517.5	631.1	654.1	583.5	515.8	624.4	628.7	604.5
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
Seq. Write	1033.2	843.7	969.5	840.5	426.6	422.3	462.6	400.4

Table 3: RAID-5 8-Way Setup – Mean Response Time in Seconds

	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
File Server	78.3	72.1	87.1	70.7	94.1	75	89.2	76
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
MetaData	127.1	133	137.3	124.9	189.1	101.1	104.6	99.3
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
Web Server	62.4	58.8	75.3	57.5	79.4	72.83	80.6	71.7
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
Mail Server	110.2	92.9	118.8	99.6	152.5	100.2	95.1	81
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
Seq. Read	523.8	586.2	585.3	618.7	518.5	594.8	580.7	594.4
	AS - ext3	DL- ext3	NO - ext3	CFQ - ext3	AS - xfs	DL - xfs	NO - xfs	CFQ - xfs
Seq. Write	968.2	782.9	1757.8	813.2	394.3	395.6	549.9	436.4

Table 4: RAID-5 8-Way Setup – `nr_requests = 2,560` – Mean Response Time in Seconds

	AS - ext3	DL - ext3	AS Tuned - ext3	AS - xfs	DL - xfs	AS Tuned - xfs
File Server	77.2	81.2	72.1	83.8	90.3	84.5
	AS Default	DL Default	AS Tuned	AS Default	DL Default	AS Tuned
MetaData	147.8	148.4	133.7	205.8	90.8	187.4
	AS Default	DL Default	AS Tuned	AS Default	DL Default	AS Tuned
Web Server	70.2	58.4	62	82.1	81.3	75.9
	AS Default	DL Default	AS Tuned	AS Default	DL Default	AS Tuned
Mail Server	119.2	114.8	103.5	153.9	92.1	140.2
	AS Default	DL Default	AS Tuned	AS Default	DL Default	AS Tuned
Seq. Read	517.5	631.1	634.5	515.8	624.4	614.1
	AS Default	DL Default	AS Tuned	AS Default	DL Default	AS Tuned
Seq. Write	1033.2	843.7	923.4	426.6	422.3	389.1

Table 5: RAID-5 8-Way - Default AS, Default deadline, and Tuned AS Comparison - Mean Response Time in Seconds

	CFQ-ext3	PID-Tuned-ext3	CFQ Tuned-ext3	CFQ-xfs	PID-Tuned-xfs	CFQ Tuned-xfs
File Server	70.7	71.1	70.6	76	75.9	74.3
	CFQ	PID - Tuned	CFQ Tuned	CFQ	PID - Tuned	CFQ Tuned
MetaData	124.9	122	125.1	99.3	92.9	97.4
	CFQ	PID - Tuned	CFQ Tuned	CFQ	PID - Tuned	CFQ Tuned
Web Server	57.5	55.8	58	71.7	73	72.5
	CFQ	PID - Tuned	CFQ Tuned	CFQ	PID - Tuned	CFQ Tuned
Mail Server	99.6	94.5	93.3	81	93.6	93.3
	CFQ	PID - Tuned	CFQ Tuned	CFQ	PID - Tuned	CFQ Tuned
Seq. Read	618.7	599.5	595.4	594.4	583.7	604.1
	CFQ	PID - Tuned	CFQ Tuned	CFQ	PID - Tuned	CFQ Tuned
Seq. Write	813.2	781.1	758.4	436.4	432.1	414.6

Table 6: RAID-5 8-Way- Default CFQ, PID Hashed CFQ & `cfq_quantum=32`, Default CFQ & `cfq_quantum=32` – Mean Response Time in Seconds

File Server	AS - ext3 44.5	DL- ext3 40	NO - ext3 41.9	CFQ - ext3 40.8	AS - xfs 42.5	DL - xfs 43	NO - xfs 45.9	CFQ - xfs 42.5
MetaData	AS - ext3 66.7	DL- ext3 64.6	NO - ext3 66.2	CFQ - ext3 64	AS - xfs 101.8	DL - xfs 71.7	NO - xfs 72.4	CFQ - xfs 66.7
Web Server	AS - ext3 43.4	DL- ext3 38.2	NO - ext3 37.9	CFQ - ext3 42.9	AS - xfs 68.3	DL - xfs 42.8	NO - xfs 69.3	CFQ - xfs 64.5
Mail Server	AS - ext3 60.3	DL- ext3 58.5	NO - ext3 58.7	CFQ - ext3 58.1	AS - xfs 100.3	DL - xfs 66.2	NO - xfs 65.8	CFQ - xfs 65.1
Seq. Read	AS - ext3 2582.1	DL- ext3 470.4	NO - ext3 460.2	CFQ - ext3 510.9	AS - xfs 2601.2	DL - xfs 541	NO - xfs 576.1	CFQ - xfs 511.2
Seq. Write	AS - ext3 1313.8	DL- ext3 1439.3	NO - ext3 1171.1	CFQ - ext3 1433.5	AS - xfs 508.5	DL - xfs 506.2	NO - xfs 508.5	CFQ - xfs 509.8

Table 7: RAID-0 16 – Default I/O Schedulers, No Tuning, Mean Response Time in Seconds

Mixed ext3	CFQ 334.1	CFQ-T 288.1	AS 371.2	DL 301.2	NO 333.5
Mixed xfs	CFQ 295	CFQ-T 291	AS 308.4	DL 296	NO 302.8

Table 8: RAID-5 8-Way Mixed Workload Behavior, Mean Response Time in Seconds

Creating Cross-Compile Friendly Software

Sam Robb
TimeSys
sam.robb@timesys.com

Abstract

Typical OSS packages make assumptions about their build environment that are not necessarily true when attempting to cross compile the software. There are two significant contributors to cross compile problems: platform specific code, and build/host confusion. Several examples of problems existing in current OSS packages are presented for each of these root causes, along with explanations of how they can be identified, how they can have been avoided, and how they can be resolved.

1 Why Cross Compile?

Cross compiling is the process of building software on a particular platform (architecture and operating system), with the intent of producing executables that will run on an entirely different platform. Generally, the platform the software is built on is referred to as the "build" system, while the platform the executables are run on is referred to as the "host" system.[1]

The process of cross compiling software is somewhat related to, but distinct from, the process of porting software to run on a different platform. The critical distinction is in the difference between the build and host system characteristics. Often times, software that can be built natively on different platforms will exhibit problems when cross compiling. These problems arise because the software fails to distinguish between the build system and the host system during one or more of the four distinct stages in the process of cross compiling software: configuration, compilation, installation, and verification.

Cross compiling is an absolute necessity for a very small number of software packages. In the OSS world, there are several software packages that are specifically designed with cross compiling in mind (binutils, gcc, busybox, the Linux kernel itself, etc.) These packages are often used to bootstrap a new system, providing a high-quality, low-cost way of obtaining a minimal working system with a small amount of effort. Once a minimal OS and related utilities are present on a system, a developer can then build additional software for the system as required.

As Linux becomes more prevalent in the embedded market space, there is an increased desire among embedded systems developers for more cross compile friendly software packages. While modern embedded systems are often resource rich in terms of processing power, I/O capabilities, memory, and disk space when compared to embedded systems of only a few years ago, compiling software natively on such a system still poses problems for an embedded developer. In extreme cases, compiling a moderately complex software package on an em-

[1] Unfortunately, not everyone chooses the same terminology. For example, the Scratchbox documentation (http://www.scratchbox.org/) uses the terms "host" and "target" where this paper uses "build" and "host" to refer to the same concepts.

bedded system natively may take hours instead of minutes.

Embedded developers therefore prefer cross compiling. Most significantly, it gives the embedded developer the advantage of working in a more comfortable, resource-rich environment—typically on a high-end workstation or desktop system—where they can take advantage of superior hardware to reduce their compile/link/debug cycles. Also importantly, cross compiling makes it easier to set up a system by which an entire system can easily be built from scratch in a reproducible manner.

2 Terminology and Assumptions

Cross compiling is a specialized subset of the software development world, and as such, employs its own terminology in an attempt unambiguously identify certain concepts. The following terms are definitions based on those provided by the GNU autoconf documentation [2], and used commonly in OSS projects such as binutils, gcc, etc.

platform - an architecture and OS combination

build system - the platform that a software package will be *configured* and *compiled* on

host system - the platform that a software package will *run* on

target system - the platform that the software package will *produce* output for

toolchain - the collection of tools (compiler, linker, etc.) along with the headers, libraries, etc. needed to build software for a platform

cross compiler - a **toolchain** that runs on a **host system**, but produces output for a **target system**

Typically, the target system is really only of interest to those working on compilers and related tools, where that extra degree of precision is needed in order to specify the final binary format those tools are intended to produce. In the OSS world, aside from binutils, gcc, and similar software packages, one can usually ignore the additional possibilities and complications introduced by variations in the target system.

The remainder of this paper will assume the existence of a cross compiler[3] that runs on an unspecified build system, and is capable of producing executables that will run on a different unspecified host system. The paper ignores the process of porting software to run on a new platform, in order to concentrate solely on issues that arise from the process of cross compiling the software.

3 Configuration Issues

All but the most simple software packages generally require some means of configuration. This is a process by which the software determines how it should be built—which libraries it should reference, which headers it may include, any particular quirks or workarounds in system calls it needs to deal with, etc.

Configuration is an area ripe for introducing cross compile problems. It provides software packages with the unique opportunity to completely confuse a build by assuming that the build system and the host system are one and the same. All cross compile configuration

[2] Available at `http://www.gnu.org/manual/`

[3] Those interested in building their own cross compiler may wish to consult the 'Resources' section at the end of this paper.

problems are some reflection of this confusion between the identity of the build and host systems.

3.1 Avoid using the wrong tools

This particular problem is caused by misidentifying which tools are to be used as part of the build process. Some software packages expect to be able to build and execute utility programs as part of their build process; a good example of this is the Linux kernel configuration utility. While the final output of the software package will need to run on the host system, these utility programs will need to be run on the build system.

Figure 1 shows an example of this problem. In this case, `CC_FOR_BUILD` is set to the same value as `CC`, which would be appropriate if it wasn't for the fact that earlier in the configuration process, `CC` was explicitly set to reference the cross compiler being used for the build.

```
 # compilers to use to create programs
 # which must be run in the build environment.
-CC_FOR_BUILD = $(CC)
-CXX_FOR_BUILD = $(CXX)
+CC_FOR_BUILD = gcc
+CXX_FOR_BUILD = g++

 SUBDIRS = "this is set via configure, \
           don't edit this"
 OTHERS =
```

Figure 1: Using the wrong tools

In this particular instance, there are several solutions. The most correct, and most expensive, is to update the makefile templates to use the proper variables (`CC_FOR_BUILD` and `CC`) in their proper context. Another possible solution is to override the definition of `CC_FOR_BUILD` and `CC` prior to invoking the makefile. The solution presented in Figure 1 is a simple, straightforward, get-it-working approach where `CC_FOR_BUILD` is simply set to an appropriate value for the majority of build systems.

3.2 Be cautious when executing code on the build system

As part of the configuration process, many software packages—particularly those built on top of `autoconf`—will try to compile, link, or even execute code on the host system.

For `autoconf` based projects, most of the standard `autoconf` macros (`AC_CHECK_LIB`, `AC_CHECK_HEADER`, etc.) do a good job of dealing with cross compile issues. In some instances, though, these standard macros fail when trying to test for the presence of an uncommon header file or library. Developers typically deal with these case by writing custom `autoconf` macros.

If the developer is not cautious, s/he may produce a custom macro that ends up performing a more extensive check than what is really needed. Often times, a developer will create a custom macro that makes use of the `autoconf AC_TRY_RUN` macro. This macro attempts to compile, link, and execute an arbitrary code fragment. The problem here is that the conditions being tested for may not actually require that the resulting binary be executed.

When cross compiling a package that uses custom macros, this leads to a situation where test code will compile and link properly (thanks to the cross compiler), but will then fail to run, or will run and produce incorrect output. In either case, it is highly unlikely that the configure script will reach the proper conclusion about whether or not the header file or library is actually available.

A simple solution to this problem is to check and see if the output from the test program is ever actually used. If not, then the call to `AC_TRY_RUN` in the test macro can be re-

placed with a call to AC_TRY_COMPILE or AC_TRY_LINK, as shown in Figure 2. These two macros implement checks for the ability to compile and link the provided code fragment, respectively.

```
      SKEY_MSG="yes"

      AC_MSG_CHECKING([for s/key support])
-     AC_TRY_RUN(
+     AC_TRY_LINK(
              [
 #include <stdio.h>
 #include <skey.h>
```

Figure 2: Avoiding execution when linking will suffice

3.3 Allow the user to override a 'detected' configuration value

In some cases, use of AC_TRY_RUN is absolutely essential; the automatic configuration process may need to be able to compile, link, and execute code in order to determine the characteristics of the host system. This is a definite stumbling block when trying to configure a software package for cross compiling.

A good configuration script allows the user to explicitly identify or override what would otherwise be an automatically detected value. For autoconf based projects, this typically means adding AC_ARG_ENABLE macros to your configure.in file that allow the user to explicitly set the value of questionable autoconf variables.

In the case of existing software packages, there may not be an explicit method for setting a questionable variable. In this case, it may be possible to set the appropriate variable by hand before configuring the software package, in order to force the desired outcome. This may still fail under some circumstances; for example, some configuration scripts do not bother to check to see if the a configuration variable has been set before attempting to automatically deduce its value.

In those cases, the configuration script may be modified[4] to guard the detection code by checking to see if the variable has already been assigned a value. If a value has already been assigned, the configuration script can use the specified value, and skip executing the detection code. In other cases, it may be more appropriate to fix the detection code itself so that it sets the variable to the proper value.

4 Compilation Issues

For the majority of portable software packages, attempting to cross compile will generally not uncover any issues with the code itself.[5] Even though individual source files may compile when pushed through the cross compiler, though, the overall way in which the software is built can still exhibit problems.

4.1 Avoid hard-coded tool names

Figure 4 shows a makefile fragment that originally made an explicit call to ar. In a package that is otherwise cross compile friendly, this is a particularly annoying occurrence. Depending on the specifics of the cross compiler, the call to ar may succeed, but produce an unusable static library.

Correcting this kind of problem is straightforward—replace the hard-coded tool name with a reference to a make variable

[4] For autoconf based software packages, keep in mind that the configure script is generated by processing configure.in. Editing the configure script direclty can be helpful for testing fixes, but changes will have to be made to configure.in as well to ensure they persist if the configure script is regenerated.

[5] Provided, of course, that the software has already been ported to the host platform.

that names the appropriate tool for the system the binary is intended to run on.

4.2 Avoid decorated tool names

Occaisionally, project makefiles will avoid hardcoded tool names by defining a variable, but then attempt to eliminate the an "unneeded" variable by combining a tool reference with the default flags that should be passed along to the tool, as shown in Figure 3.

While the intent was noble, this type of definition makes it difficult for a user to supply a different definition for a tool. Instead of simply setting the value of of the tool when invoking the makefile (ex, `make AR=ppc7xx-linux-ar`), a user now has to know to define AR in a way that includes the default arguments (ex, `make AR='ppc7xx-linux-ar cr'`).

Again, correcting this type of problem is straightforward—split the definition of the tool reference into a reference to the simple tool name and a variable that indicates the default flags that should be passed to the tool.

```
-AR = @AR@ cq
+AR = @AR@
+ARFLAGS = cq

 all: $(OBJS)
        -rm -f libsupport.a
-       $(AR) libsupport.a $(OBJS)
+       $(AR) $(ARFLAGS) libsupport.a $(OBJS)
        @RANLIB@ libsupport.a
```

Figure 3: Avoiding execution when linking will suffice

4.3 Avoid hard-coded paths

It is very easy for an otherwise cross compile friendly software package to mistakenly set up an absolute include path that looks reasonable. In many situations, the added include path may in fact be harmless, particularly if the build system and host system have roughly the same OS version, library versions, etc. However, even slight differences in structure definitions, enumerated constants, etc. between build system and host system headers can very easily result in either compilation errors, or in the cross compiler producing an unusable binary.

Figures 5 and 6 shows a simple and straightforward solution—remove the hard-coded include path. If the include path is required, then you will need to alter it so that it can be specified relative to the location of the include files appropriate for the host system.

4.4 Avoid assumptions about the build system

While this is nominally a porting issue, sometimes a software package will make what seems to be a reasonable assumption about the build system. In particular, software packages that are intended to run only on a particular class of operating systems (Linux, POSIX complaint systems, etc.) may assume that even if they are cross compiled, they will at least be cross compiled on a build system that has characteristics similar to the host system.

Figure 7 illustrates this problem. This makefile fragment assumes that the build system will have a case-sensitive file system, and that the file patterns `'*.os'` and `'*.oS'` will therefore refer to a distinct set of files—in this case, files for inclusion in a static library and files for inclusion in a shared library, respectively.

This particular assumption breaks down when compiling on a case-insensitive file system like VFAT, NTFS, or HPFS.[6] When encountering this type of problem, there is no easy workaround—the build logic for the software

[6] While these file systems are case-insensitive, they are case preserving, which sometimes helps mask potential case-sensitivity issues.

will need to be altered in order to adjust to the conditions of the unexpected build system.

In this case, the solution was to replace '`*.oS`' with '`*.on`', a file pattern that is distinct from '`*.os`' on either a case-insensitive or a case-sensitive file system.

5 Installation Issues

Software installation is sometimes seen as a simple problem. After all, how hard can it be to just copy files around and make sure they all end up in the right place? As with configuration and compilation, though, cross compiling software introduces additional complexities when installing software.

5.1 Avoid `install -s`

Figure 8 shows a makefile fragment that at first glance looks reasonable; as originally written, it attempted to install a binary using the detected version of the `install` program available on the build system.

The problem here is that the original `install` command specified the `-s` option, which instructs `install` to strip the binary after installing it. Because the command uses the build system's version of `install`, this means that the stripping will be accomplished using the build system's version of `strip`. Depending on the version of `strip` installed on the build system, this command may appear to succeed, yet result in a useless binary being installed.

The solution here is to avoid the use of `install -s`, and instead explicitly strip the binary after installation using the version of `strip` provided with the cross compile toolchain that built the binary.

5.2 Avoid hard-coded installation paths

When cross compiling software, it is often convenient to treat a directory on the build system as the logical root of the host system's file system.[7] This allows a developer to "install" the software into this logical root file system (RFS); often times, the RFS is made available to the host system via NFS.

Autoconf packages typically use variables to specify the prefix for installation paths, which makes installing them into an RFS a simple matter. As Figure 9 shows, non-`autoconf` makefiles may need to be modified to make the same sort of adjustments to installation paths.

Even if the software package already makes use of `prefix` or a similar variable, it may overload the meaning of that variable. This can happen in any type of software package, `autoconf` based or not. For example, a package may use the `prefix` variable to both control the installation path, and also generate `#define` statements that specify paths to configuration files or other important data. In this case, it may still be necessary to modify the makefile to introduce the idea of an installation prefix, as shown in Figure 10.

5.3 Create the required directory structure

Often times, software packages assume that they are being installed on an existing, full-featured system—which implies the existence of a certain directory structure. A cross compiled software package may be installed on the build system into a location that is lacking part or all of a normal directory structure. In this case, the install steps of the software package must be pessimistic, and assume that it will always be necessary to create whatever directory

[7] See the Scratchbox website (`http://www.scratchbox.org`) for more information on the hows and whys of build sandboxing.

structure it requires for the installation to succeed.

Figure 11 shows a patch for a makefile fragment that originally assumed the pre-existence of a particular directory structure. Appropriate calls to `mkdir -p` are enough to ensure that the existing directory structure is in place prior to the install.

6 Verification Issues

There are a number of OSS packages that very conveniently provide self-test capabilities. Along with the usual targets in their makefiles, they include targets that allow the user to build and run a test suite against the software after it is built, but before it is installed.

The main problem here is that these test targets generally run each individual test in the suite using a "compile, execute, analyze" cycle. Even if the compilation and result analysis steps succeed on the build system, test execution will most likely fail if the package has been cross compiled, since the tests were built with the host system in mind. If you are fortunate, these tests will simply fail; otherwise, you will not be able to gauge the accuracy of the tests, as they may be picking up information or artifacts from the build system.

A simple solution is to rewrite test targets to separate test compilation from test execution and result analysis. Providing a distinct install or packaging target for the test suite so that it can be easily moved over to a host system for execution is an added bonus.

Don't assume that you can execute self-tests as part of the normal build cycle (see Figure 12). If you do include a test target as part of your default target dependencies, at least make sure that it is only enabled or run if it knows that it can execute the tests on the build system.

7 Conclusions

By now, it should be apparent that while there are any number of subtle ways that cross compiling software can fail, they are for the most part simple problems with simple solutions.

Developers interested in supporting cross compiling of software packages they maintain can use these problems as a guideline of potential problem areas in their own projects. Detecting potential cross compile issues is often a simple matter of examining project source code and identifying the potential for confusing the meaning of build and host systems.

Finally—the best possible way to examine a software package to see if (or how well) it supports cross compiling is to actually try and cross compile it. While the truly adventurous may wish to try and build their own cross compiler, there are any number of locations on the web where an interested developer can obtain a pre-built toolchain for this purpose. Those working primarily on an x86 Linux host may wish to consider using one of the available pre-built cross compilers that can be found through the rpmfind (`http://www.rpmfind.net`) service. For those interested in building their own cross compiler, or in researching other cross compile issues, are a number of resources (see Table 8) on the net that deal specifically with cross compile issues. The emphasis of these resources is generally on embedded system development, though much of the information available is still applicable when discussing cross compiling in general.

8 Appendix—Code Examples

The following figures are referred to in the paper, and are collected here (instead of presented inline) for the sake of providing clarity in the text. Each figure represents a patch (or a partial patch) for a common OSS package that was used at TimeSys to work around cross compile problems. These selections were chosen to illustrate, in a compact fashion, both the problems described in the text and some possible solutions.

```
       decompress.o \
       bzlib.o

-all: libbz2.a bzip2 bzip2recover test
+all: libbz2.a bzip2 bzip2recover #test

 bzip2: libbz2.so bzip2.c
        $(CC) $(CFLAGS) -o bzip2 $\^
```

Figure 12: Avoid making tests part of the default build target

The CrossGCC Mailing List	`http://sources.redhat.com/ml/crossgcc/` A list for discussing embedded ('cross') programming using the GNU tools.
The CrossGCC FAQ	`http://www.sthoward.com/CrossGCC/`
crosstool	`http://www.kegel.com/crosstool/` A set of scripts to build gcc and glibc for most architectures supported by glibc.
Linux from Scratch	`http://www.linuxfromscratch.org/` A project that provides you with the steps necessary to build your own custom Linux system.
Scratchbox	`http://www.scratchbox.org/` A cross-compile toolkit for embedded Linux application development.
Embedded Gentoo	`http://www.gentoo.org/proj/en/base/embedded/index.xml` Gentoo project concerned with cross compiling and embedded systems.
The GNU configure and build system	`http://www.airs.com/ian/configure/` Document describing the GNU configure and build systems. A bit out of date (circa 1998), but still very useful.
GNU Autoconf, Automake, and Libtool	`http://sources.redhat.com/autobook/` Online version of the classic book covering GNU autotools.

Table 1: Selected internet resources on cross compiling

```
 libbz2.a: $(OBJS)
        rm -f libbz2.a
-       ar cq libbz2.a $(OBJS)
-       @if ( test -f /usr/bin/ranlib -o -f /bin/ranlib -o \
-               -f /usr/ccs/bin/ranlib ) ; then \
-               echo ranlib libbz2.a ; \
-               ranlib libbz2.a ; \
-       fi
+       $(AR) cq libbz2.a $(OBJS)
+       $(RANLIB) libbz2.a
+       #@if ( test -f /usr/bin/ranlib -o -f /bin/ranlib -o \
+       #       -f /usr/ccs/bin/ranlib ) ; then \
+       #       echo ranlib libbz2.a ; \
+       #       ranlib libbz2.a ; \
+       #fi

 libbz2.so: libbz2.so.$(somajor)
```

Figure 4: Avoiding hard-coded tool references

```
 export GCC_WARN   =  -Wall -W -Wstrict-prototypes -Wshadow $(ANAL_WARN)
-export INCDIRS    =  -I/usr/include/ncurses
-export CC         =  gcc
+#export INCDIRS    =  -I/usr/include/ncurses
+#export CC         =  gcc
 export OPT        =  -O2
 export CFLAGS     =  -D_GNU_SOURCE $(OPT) $(GCC_WARN) -I$(shell pwd) $(INCDIRS)
```

Figure 5: Avoiding hard-coded include paths

```
 INSTALL        = install -o $(BIN_OWNER) -g $(BIN_GROUP)

 # Additional libs for Gnu Libc
-ifneq ($(wildcard /usr/lib/libcrypt.a),)
 LCRYPT         = -lcrypt
-endif

 all:           $(PROGS)
```

Figure 6: Avoiding tests for hard-coded path names

```
 # Bounded pointer thunks are only built for *.ob
 elide-bp-thunks = $(addprefix $(bppfx),$(bp-thunks))

-elide-routines.oS += $(filter-out $(static-only-routines),\
+elide-routines.on += $(filter-out $(static-only-routines),\
    $(routines) $(aux) $(sysdep_routines)) \
       $(elide-bp-thunks)
 elide-routines.os += $(static-only-routines) $(elide-bp-thunks)
```

Figure 7: Avoiding assumptions about the build system

```
-       $(INSTALL) -m 0755 -s ssh $(DESTDIR)$(bindir)/ssh
+       $(INSTALL) -m 0755 ssh $(DESTDIR)$(bindir)/ssh
+       $(STRIP) $(DESTDIR)$(bindir)/ssh
```

Figure 8: Replacing install -s with an explicit call to strip

```
 NAME       =  proc

 # INSTALLATION OPTIONS
-TOPDIR     = /usr
+TOPDIR     = $(DESTDIR)/usr
 HDRDIR     = $(TOPDIR)/include/$(NAME)#        where to put .h files
 LIBDIR     = $(TOPDIR)/lib#       where to put library files
-SHLIBDIR   = /lib#                where to put shared library files
+SHLIBDIR   = $(DESTDIR)/lib#      where to put shared library files
 HDROWN     = $(OWNERGROUP) #      owner of header files
 LIBOWN     = $(OWNERGROUP) #      owner of library files
 INSTALL    = install
```

Figure 9: Avoiding hard-coded install paths

```
 # Where is include and dir located?
 prefix=/
+installdir=/

 .c.o:
        $(CC) $(CFLAGS) -c $<
@@ -47,28 +48,32 @@
        -if [ ! -d pic ]; then mkdir pic; fi

 install: lib install-dirs install-data
-       -if [ -f $(prefix)/lib/$(SHARED_LIB) ]; then \
-          mkdir -p $(prefix)/lib/backup; \
-          mv $(prefix)/lib/$(SHARED_LIB) \
-               $(prefix)/lib/backup/$(SHARED_LIB).$$$$; \
+       -if [ -f $(installdir)/$(prefix)/lib/$(SHARED_LIB) ]; then \
+          mkdir -p $(installdir)/$(prefix)/lib/backup; \
+          mv $(installdir)/$(prefix)/lib/$(SHARED_LIB) \
+               $(installdir)/$(prefix)/lib/backup/$(SHARED_LIB).$$$$; \
        fi
-       cp $(SHARED_LIB) $(prefix)/lib
-       chown $(OWNER) $(prefix)/lib/$(SHARED_LIB)
+       cp $(SHARED_LIB) $(installdir)/$(prefix)/lib
+       chown $(OWNER) $(installdir)/$(prefix)/lib/$(SHARED_LIB)
        if [ -x /sbin/ldconfig -o -x /etc/ldconfig ]; then \
          ldconfig; \
```

Figure 10: Working around the use of an overloaded prefix variable

```
 install-only:
   n=‘echo gdbserver | sed ’$(program_transform_name)’‘; \
   if [ x$$n = x ]; then n=gdbserver; else true; fi; \
+  mkdir -p $(bindir); \
+  mkdir -p $(man1dir); \
   $(INSTALL_PROGRAM) gdbserver $(bindir)/$$n; \
   $(INSTALL_DATA) $(srcdir)/gdbserver.1 $(man1dir)/$$n.1
```

Figure 11: Creating required directories at install time

Page-Flip Technology for use within the Linux Networking Stack

John A. Ronciak
Intel Corporation
john.ronciak@intel.com

Jesse Brandeburg
Intel Corporation
jesse.brandeburg@intel.com

Ganesh Venkatesan
Intel Corporation
ganesh.venkatesan@intel.com

Abstract

Today's received network data is copied from kernel-space to user-space once the protocol headers have been processed. What is needed is to provide a *hardware (NIC) to user-space* zero-copy path. This paper discusses a page-flip technique where a page is *flipped* from kernel memory into user-space via page-table manipulation. Gigabit Ethernet was used to produce this zero-copy receive path within the Linux stack which can then be extrapolated to 10 Gigabit Ethernet environments where the need is more critical. Prior experience in the industry with page-flip methodologies is cited.

The performance of the stack and the overall system is presented along with the testing methodology and tools used to generate the performance data. All data was collected using a modified TCP/IP stack in a 2.6.x kernel. The stack modifications are described in detail. Also discussed is what hardware and software features are required to achieve page-flipping.

The issues involving page-flipping are described in detail. Also discussed are problems related to this technology concerning the Virtual Memory Manager (VMM) and processor cache. Another issue that is discussed is what would be needed in an API or code changes to enable user-space applications.

The consequences and possible benefits of this technology are called out within the conclusions of this study. Also described are the possible next steps needed to make this technology viable for general use. As faster networks like 10 Gigabit Ethernet become more commonplace for servers and desktops, understanding and developing zero-copy receive mechanisms within the Linux kernel and networking stack is becoming more critical.

Introduction

Data arriving at a network port undergoes two copy operations (a) from the device memory to kernel memory as a DMA by the device into host memory and (b) from kernel memory to application memory, copied by the processor. Techniques that avoid the second copy are designated zero-copy; no additional copy operations are involved once the data is copied into host memory. Avoiding the second copy can potentially improve throughput and reduce CPU utilization. This has been demonstrated in [Hurd] [Duke] and [Gallatin]. Several techniques have been discussed in the literature for

avoiding the second copy namely page flipping, direct data placement (DDP) and remote DMA (RDMA).

Significant performance benefits were demonstrated with the zero copy implementation in the transmit path. We investigate the effectiveness of the page flipping on newer platforms (faster processor(s) and faster memory). Additional motivation for this experiment and paper came from a discussion on the netdev (and linux-kernel) mailing list where David Miller mentioned his idea of

> On receive side, clever RX buffer flipping tricks are the way to go and require no protocol changes and nothing gross like TOE or weird buffer ownership protocols like RDMA requires.[1]

Approaches

Our initial approach consisted of attempting to modify the 2.4 kernel to support direct modification of PTE's in user and kernel space. This method was based on the assumption that any PTE could represent any location in memory which we later found out not to be true. Our findings indicated that we needed to rely more upon the OS abstraction layers to complete our page-flip implementation. This had the side benefit of making our changes less x86 specific as well. Eventually we settled upon a 2.6 based kernel and effectively implemented our original idea but instead just install a new page into the application space in much the same way as the swapper does. The biggest hurdles came from understanding how the Linux memory manager and its various kernel structures work and relate to each other.

[1] `http://marc.theaimsgroup.com/?l=linux-netdev&w=2&r=1&s=TCP+offloading+interface&q=b`

For our final experiments we used 2.6.4 or newer kernels with what eventually amounted to small changes to the kernel to support page flipped PAGE_SIZE data.

The kernel code consisted of these changes (see patch at the end of this document):

1. Driver modifications to support header and data portions of a packet in separate buffers, where the data buffer is always aligned to a PAGE_SIZE boundary.
2. Add a flag to the skb structure to indicate to the stack that the hardware and driver prepared a zero copy capable receive structure.
3. Modifications to the `skb_copy_datagram_iovec()` function to support calling the new `flip_page_mapping()` function when zero copy capable skbs are received.
4. A new `flip_page_mapping()` function that executes the installation of the driver page into the user's receive data space. This routine handles fixing up permissions.
5. A modification was made to the skb free routines to handle a frags[i] where the .page member was zero after that page had changed ownership to user space.

Experiment

Our test platform consisted of a pre-release system with a dual 2.4 GHz Intel® Pentium® 4 processor supporting Hyper-Threading Technology, and 512 megabytes of RAM. This machine had a network card that supported splitting the header and data portions of a packet into different buffers, and validating the IP, TCP and Ethernet checksums.

Assumptions

For this experiment we made some assumptions to simplify and to work with the hardware that we had available.

- Our application had to allocate a receive data area in multiples of 4K bytes, and that memory had to be PAGE_SIZE aligned.
- We modified the freely available nttcp-1.47 to use valloc instead of malloc, resulting in PAGE_SIZE aligned memory starting addresses.
- Our network used Maximum Transmission Units (MTU) to allow for 4KB or 8KB of data to be packaged in every packet.
- Upon splitting of the packet into header and data portions, this resulted in an aligned data block
- The 2.6.4 kernel was configured for standard 4KB PAGE_SIZE and debugging options were turned off.

Methodologies

After making the required code changes and debugging, we measured the performance of the new "page flip" code against the "copy once" method of receiving data.

These measurements consisted of two major test runs, one where the application never touched the data (notouch) being received, and the other where the application did a comparison of the data to an expected result (touch), effectively forcing the data into the cache and also validating that data was not corrupted in any way through this process.

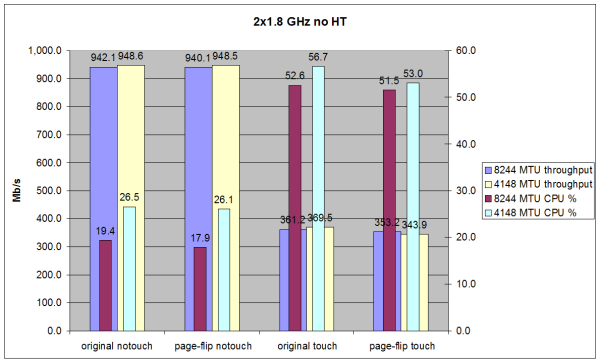


Figure 1: 1.8 GHz comparison

For every instance of the test, three runs were done and the results were averaged for each data point.

Oprofile was used to record the hot-spots for each run.

CPU utilization and network utilization were measured with sar from the sysstat package. NOTE: Our initial results were skewed by a version of sar that incorrectly measured CPU and network utilization (showing more than 1Gb/s transferred in a single direction), be aware that some versions of sar that shipped with your distribution may need to be updated.

Results of Performance Analysis

It is apparent from the touch graphs in Figure 1 that the page flip slightly reduces CPU on slower processors. However, the touch throughput decreases as well, with a decrease in efficiency (Mbits/CPU = eff) for the 4148 MTU from 6.52 (original) to 6.48 (page-flip). The decrease in efficiency is even smaller for 8244 MTUs, where the efficiency went from 6.86 to 6.85. The difference in CPU from the 8244 to the 4148 MTU case is most likely due to header processing as the data throughput is very similar.

The difference between Figure 1 and Figure 2 is simply the processor's speed being adjusted

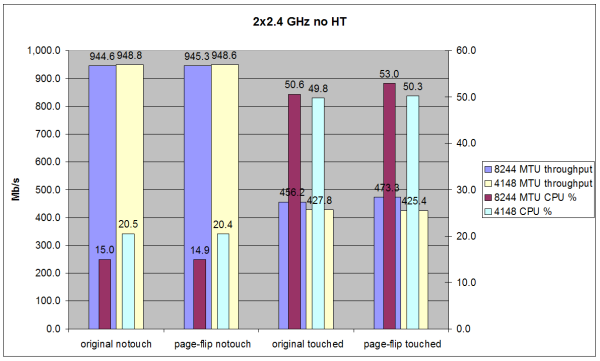


Figure 2: 2.4 GHz comparison

in the bios using a multiplier change. The results from Figure 2 show that the faster processor is more efficient overall, but that even if there is a slight increase in throughput for the page-flip case, the efficiency is still less than if the copy was being done. The efficiency for the 4148 MTU touch data case went from 8.59 to 8.45. For the 8244 byte MTU the efficiency goes from 9.02 to 8.93, even though the throughput goes up.

Surprises and Unexpected Results

We expected that the copy may actually have some beneficial side effects, and our data shows that it does. Especially as processor clock rate increases, the copy becomes less costly in CPU-utilization, while the page flip maintains a constant load which is heavier than the copy was initially.

Oprofile analysis indicated that the locks associated with the page-flip code cause the majority of the stalls in this code path.

Oprofile also showed that the stall associated with the TLB (translation look-aside buffer) flush was very painful.

Conclusions

We had several surprises along the way, but feel confident that at least with our current code base, we can conclude that using a page-flip methodology to receive network data is less efficient than simply doing a copy. The major contributors to this counterintuitive result seem to be cache issues (especially obvious in the "touched data" tests), and a heavier cost associated with the work necessary to prepare and complete the page-flip.

There may be environments such as embedded systems and slower processors where page-flipping will help significantly in decreasing CPU utilization or increasing performance.

Our feeling is that page flipping will not scale in CPU utilization as well as a plain copy does.

There is much room however for optimization of the page-flip code path, which will be followed up with the community. Our expectation is that this optimization will be fighting an uphill battle just to achieve parity with a copy, and then will mostly likely not be able to keep up with speed advances in the processor.

Also, we had to remind ourselves that the cache warming cost must be paid somewhere along all receive paths. Using page-flip methods only moves the cost of the cache miss to the application instead of taking the cost of the miss in the kernel. If the application is waiting impatiently for data, its likely that the cache will be seeded with the data and the application will get all of its data out of cache and have very fast access at that point.

Current issues

The current patch has several outstanding issues that we worked around.

1. There isn't much (if any) commercially available hardware that supports header split receives.
2. Ideally hardware (as mentioned by David Miller) would be able to have flow identification and fill `PAGE_SIZE` buckets with data. This would eliminate the requirement for specific MTU sizes.
3. The current code has a bug when a network data consumer causes a `clone_skb()` to occur. If a page-flipped page pointer `nr_frags[].page` is referenced in the skb being cloned, then a zero pointer is read and the system faults. This is due to the ownership of the page changing from kernel space to user space before the clone is completed. It is not immediately clear if this is an easily surmountable problem, but is easy to work around for our tests.
4. The assumptions we made to enable testing this new code path, like specifying MTU, recompiling the application, etc, create such strict requirements that the usefulness of this code outside of an academic environment is severely limited.

Future directions possible

It is likely that on a system with lots of context switching going on (high load) that the page-flip would be more beneficial. Testing in these environments would provide useful results.

If tested on other architectures besides x86, such as x86-64, IA64 and PPC this code may yield significantly different results.

We did create a driver patch (Appendix B) for the currently available e1000 driver and hardware that prepares packets (using a copy) for processing through the page-flip modified network stack to the user application. We saw that the copy necessary in the driver to do this made the differences between "driver with copy followed by a flip in the stack" and a "driver with a copy followed by another copy in the stack" almost nonexistent. We believe this is because of the cache warming done by the Appendix B driver as it prepares the flip capable structure. Making this code behave more like the flip capable hardware (possibly with a cache flush) would be very useful to increase the amount of experimentation that could be done with the non-hardware specific kernel patches.

References

[Hurd] Dana Hurd Zero-copy interfacing to TCP/IP Dr. Dobbs Journal Sep 1995

[Duke] Trapeze Project: `http://www.cs.duke.edu/ari/trapeze/slides/freenix/sld001.htm`

[Gallatin] Drew Gallatin `http://people.freebsd.org/~ken/zero_copy/`

Appendix A Kernel Patch

This patch will be available at `http://www.aracnet.com/~micro/flip/flip_2_6_4.patch.bz2`

Appendix B mock zero copy e1000 patch

This patch will be available at `http://www.aracnet.com/~micro/flip/e1000_flip.patch.bz2`

Linux Kernel Hotplug CPU Support

Zwane Mwaikambo
FSMLabs
zwane@fsmlabs.com

Ashok Raj
Intel
ashok.raj@intel.com

Rusty Russell
IBM
rusty@rustcorp.com.au

Joel Schopp
IBM
jschopp@austin.ibm.com

Srivatsa Vaddagiri
IBM
vatsa@in.ibm.com

Abstract

During the 2.5 development series, many people collaborated on the infrastructure to add (easy) and remove (hard) CPUs under Linux. This paper will cover the approaches we used, tracing back to the initial PowerPC hack with Anton Blanchard in February 2001, through the multiple rewrites to inclusion in 2.6.5.

After the brief history lesson, we will describe the approach we now use, and then the authors of the various platform-specific code will describe their implementations in detail: Zwane Mwaikambo (i386) Srivatsa Vaddagiri (i386, ppc64), Joel Schopp (ppc64), Ashok Raj (ia64). We expect an audience of kernel programmers and people interested in dynamic cpu configuration in other architectures.

1 The Need for CPU Hotplug

Linux is growing steadily in the mission critical data-center type installations. Such installations requires Reliability, Availability and Serviceability (RAS) features. Modern processor architectures are providing advanced error correction and detection techiniques. CPU hotplug provides a way to realize these features in mission critical applications. CPU hotplug feature adds the following ability to Linux to compete in the high end applications.

- **Dynamic Partitioning**

  Within a single system multiple Linux partitions can be running. As workloads change CPUs can be moved between partitions without rebooting and without interrupting the workloads.

- **Capacity Upgrade on Demand**

  Machines can be purchaced with extra CPUs, without paying for those CPUs until they are needed. Customers can at a later date purchase activiation codes that enable these extra CPUs to match increases in demand, without interrupting service. These activiation codes can either be for temporary activation or permanant activation depending on customer needs.

- **Preventive CPU Isolation**

  Advanced features such as CPU Guard in PPC64 architectures, and Machine Check Abort (MCA) features in Itanium® Product Family (IPF) permit the hardware to catch recoverable failures that are symptomatic of a failing CPU and remove that CPU before an unrecoverable failure occurs. An unused CPU can later be brought online to replace the failed CPU.

2 The Initial Implementation

In February 2001, Anton Blanchard and Rusty Russell spent a weekend modifying the ppc32 kernel to switch CPUs on and off. Stress tests on a 4-way PPC crash box showed it to be reasonably stable. The resulting 60k patch to 2.4.1 was posted to the linux-kernel on February the 4th: `http://www.uwsg.iu.edu/hypermail/linux/kernel/0102.0/0751.html`.

Now we know that the problem could be solved, we got distracted by other things. Upon joining IBM, Rusty had an employer who actually had a use for hotplugging CPUs, and in 2002 the development started up again.

The 2.4 kernels used `cpu_number_map()` to map from the CPU number given by `smp_processor_id()` (between 0 and `NUM_CPUS`) to a unique number between 0 and `smp_num_cpus`. This allows simple iteration between 0 and `smp_num_cpus` to cover all the CPUs, but this cannot be maintained easily in the case where CPU are coming and going. Given my experience that `cpu_number_map()` and `cpu_logical_map()` (which are noops on x86) are a frequent source of errors, Rusty chose to eliminate them, and introduce a `cpu_online()` function which would indicate if the CPU was actually online. Much of the original patch consisted of removing the number remapping, and rewriting loops appropriately.

This change went into Linus' tree in 2.5.24, June 2002, which made the rest of the work much less intrusive.

In the next month, as we were trying to get the `cpu_up()` function used for booting, Linus insisted that we also change the boot order so that we boot as if we were uni-processor, and then bring the CPUs up. Unfortunately, this patch broke Linus' machine, and he partially reverted it, leaving us with the current situation where a little initialization is done before secondary CPUs come online, and normal `__initcall` functions are done with all CPUs enabled. This change also introduced the `cpu_possible()` macro, which can be used to detect whether a CPU could ever be online in the future.

The old boot sequence for architectures was:

1. `smp_boot_cpus()` was called to initialize the CPUs, then
2. `smp_commence()` was called to bring them online.

In addition, each arch optionally implemented a "maxcpus" boot argument. This was made into an arch-independent boot argument, and the boot sequence became:

1. `smp_prepare_cpus(maxcpus)` was called to probe for cpus and set up `cpu_present(cpu)`[1], then

[1] On arch's that dont fill in `cpu_present(cpu)` the function `fixup_cpu_present_map` just uses what `cpu_possible_map` was set during probe. See the section in IA64 for more details.

2. `__cpu_up(cpu)` was called for each CPU where `cpu_present(cpu)` was true, then

3. `smp_cpus_done(maxcpus)` was called after every CPU has been brought up.

At this stage, the CPU notifier chain and the `cpu_up()` function existed, but CPU removal was not in the mainstream kernel. Indeed, significant scheduler changes occurred, preemption went into the kernel, and Rusty was distracted by the module reworking. The result: hotplug CPU development floundered outside the main tree for over a year.

3 The Problem of CPU Removal

The initial CPU removal patch was very simple: the process scheduled on the dying CPU, moved interrupts away, set `cpu_online()` to false, and then scheduled on every other CPU to ensure that noone was looking at the old CPU values. The scheduler's `can_schedule()` macro was changed to return false if the CPU was offline, so the CPU would always run the idle task during this time. Finally, the arch-specific `cpu_die()` function actually killed the CPU.

Three things made this approach harder as the 2.5 kernel developed:

1. Ingo Molnar's O(1) scheduler was included. Rather than checking if the CPU was offline every time we ran `schedule()`, we wanted to avoid touching the highly-optimized code paths.

2. The kernel became preemptible. This means that scheduling on every CPU is not sufficient to ensure that noone is using the old online CPU information.

3. Workqueue and other infrastructure was introduced which used per-cpu threads, which had to be cleanly added and removed.

4. More per-CPU statistics were used in the kernel, which sometimes need to be merged when a CPU went offline (or each sum must be for every possible CPU, not just currently online ones)

5. Sysfs was included, meaning that the interface should be there, instead of in proc, along with structure for other CPU features

Various approaches were discussed and tried: some architectures (like i386) merely simulate CPUs going away, by looping in the idle thread. This is useful for testing. Others (like PPC64 and IA64) actually need to re-start CPUs.

The following were the major design points which were tested and debated, and the resolution of each:

- How should we handle userspace tasks bound to a single CPU?

  Our original code sent a SIGPWR to tasks which were bound such that we couldn't move them to another CPU. This has the default behaviour of killing the task, which is unfortunate if the task merely inherited the binding from its parent. The ideal would be a new signal which would also be delivered on other reconfiguration events (like addition of CPUs, memory), but the Linux ABI does not allow the addition of new signals.

  The final result was to rely on the hotplug scripts to handle this information, and rely on userspace to ensure that removing a CPU was OK before telling the kernel to switch it off.

- How should we handle kernel threads bound to a single CPU?

  Unlike userspace, kernel threads often have a correctness requirement that they run on a particular CPU. Our original approach used a notifier between marking the CPU offline, and actually taking it down; these threads would then shut themselves down. This two-stage approach caused other complications, and the legendary Ingo Molnar recommended a single-stage takedown, and that the kernel threads could be cleaned up later. While that simplified things in general, it involved some new considerations for such kernel threads.

- Issues Creating And Shutting Down Kernel Threads

  In general, the amount of code required to stop kernel threads proved to be significant: barriers and completions at the very least. The other issue is that most kernel threads assume they are started at boot: they don't expect to be started from whatever random process which brought up the CPU.

  This lead Rusty to develop the "kthread" infrastructure, which encapsulated the logic of starting and stopping threads in one place. In particular, it uses keventd (which is always started at boot) to create the new thread, ensuring that there is no contamination by forking the userspace process. The `daemonize()` function attempts to do this, but it's more certain to start from a clean slate than to try to fix a existing one.

- Issues Using keventd for CPU Hotplug

  keventd is used as a general purpose kernel thread for performing some deferred work in a thread context. The "kthread" infrastructure uses this framework to start and stop threads. In addition when various kernel code attempts to call user-space scripts and agents use `call_usermode_helper()`. This function used the keventd thread to spawn the user space program. This approach caused a dead lock situation when the `call_usermode_helper()` is called as part of the `_cpu_disable()`, since keventd threads are per-CPU threads. This results in queueing work to keventd thread via `schedule_work()`, then waiting for completion. This results in blocking the keventd thread. Unless the work queued gets to run, this keventd thread would never be woken again. To avoid this scenario, Rusty introduced the `create_singlethread_workqueue` which now provides a separate thread that is not bound to any particular CPU.

- How to Avoid Having To Lock Around Every Access to Online Map

  Naturally, we wanted to avoid locking around every access to `cpu_online_map` (via `cpu_online()` for example). The method was one Rusty invented for the module code: the so-called "bogolock". To make a change, we schedule a thread on every CPU and have them all simultaneously disabled interrupts, then make the change. This code was generalized from the module code, and called `stop_machine_run()`. This means that we only need to disable preemption to access `cpu_online_map` reliably. If you need to sleep, the `cpu_control` semaphore also protects the CPU hotplug code, so there is a slow-path alternative.

- How to Avoid Doing Too Much Work With the Machine Stopped

  While all CPUs are not taking interrupts,

we don't want to take too long. The initial code walked the task list while the machine was frozen, moving any tasks away from the dying CPU. Nick Piggin came up with an improvement which only migrated the tasks on the CPU's runqueue, and then ensured no other tasks were migrated to the CPU, which reduced the hold time by an order of magnitude. Finally Srivatsa Vaddagiri went one better: by simply raising the priority of the idle task with a special `sched_idle_next()` function, we ensure that nothing else runs on the dying CPU.

The process by which the CPU actually goes offline is as follows:

1. Take `cpu_control` semaphore,
2. Check more than one CPU is online (a bug Anton discovered in the first implementation!),
3. Check that the CPU which they are taking down is actually online,
4. Take the target CPU out of the CPU mask of this process. When the other steps are finished, they will wake us up, and we must not migrate back onto the dead CPU!
5. Use `stop_machine_run()` to freeze the machine and run the following steps on the target CPU
6. Take the CPU out of `cpu_online_map` (easier for arch code to do this first).
7. Call the arch-specific `__cpu_disable()` which must ensure that no more hardware interrupts are received by this CPU (by reprogramming interrupt controllers, or whatever),
8. If that call fails, we restore the `cpu_online_map`. Otherwise we call `sched_idle_next()` to ensure that when we exit the CPU will be idle.
9. At this point, back in the caller, we wait for the CPU to become idle, then call the arch-specific `__cpu_die()` which actually kills the offline CPU, by setting a flag which the idle task polls for, or using an IPI, or some other method.
10. Finally, the `CPU_DEAD` notifier is called, which the scheduler uses to migrate tasks off the dead CPU, the workqueues use to remove the unneeded thread, etc.

The implementation specifics of each architecture can be found in the following sections.

4 Remaining Issues

The main remaining issue is the interaction of the NUMA topology and addition of new CPUs. An architecture can choose a static NUMA topology which covers all the possible CPUs, but for logical partitioning this might not be possible (we might not know in advance).

- Per-CPU variables are allocated using `__alloc_bootmem_node()` at boot, for performance reasons. Unknown CPUs are usually assumed to be in the boot node, which will impact performance.
- sysfs node topology entries need to be updated when a CPU comes online, if the node association is not known at boot.
- The NUMA topology itself should be updated if it is only known when a CPU comes online. This is now possible, using the `stop_machine_run()` function,

but no architectures, other than PPC64, currently do this.

- There are likely some tools in use today that would require minor changes as well. One such tool identified is the top(1) utility, which has trouble dealing with the fact that CPU's available in the system are not logically contiguous. For e.g in a 4-way system, if logical cpu2 was offlined, when cpu0, cpu1, cpu3 were still functional, top would display some error information. Also the tool does not update the CPU information and not able to dynamically update them when new CPU's are added, or removed from the system.

5 i386 Implementation

Commercial i386 hardware available today offer very limited support for CPU Hotplug. Hence the i386 implementation, as it exists, is more of a toy for fun and experimentation. Nevertheless, it was used intensively during development for exercising various code paths and, needless to say, it exposed numerous bugs. Most of these bugs were in arch-independent code.

Since the hardware does not support physical hotplugging of CPUs, only logical removal of a CPU is possible. Once removed from the system, a dead CPU does not participate in any OS activity. Instead, it keeps spinning, waiting for a online command, in the context of its idle thread. Once it gets the online command, it breaks out of the spin loop, puts itself in `cpu_online_map`, flushes TLB and comes alive!

Some important i386 specific issues faced during development are described below:

- **Boot processor**
  There are a few interrupt controller configurations, which necessitate that we not offline the boot processor. Systems may be running with the I/O APIC disabled in which case all interrupts are being serviced by the boot processor via the i8259A, which cannot be programmed to direct interrupts to other processors. Another being interrupts which may be configured to go via the boot processor's LVT (Local Vector Table) such as various timer interrupt setups.

- **smp_call_function**
  smp_call_function is one tricky function which haunted us a long time. Since it deals with sending IPIs to online CPUs and waiting for acknowledgement, number of races was found in this function wrt CPUs coming and going while this function runs on some CPU. Fortunately, when CPU offline was made atomic, most of these race conditions went away. CPU online operation, being still non-atomic, exposes a race wherein an IPI can be sent to a CPU coming online and the sender will not wait for it to acknowledge the IPI. The race was fixed by taking a spinlock (`call_lock`) before putting CPU in the online_map.

- **Interrupt redirection**
  If I/O APIC is enabled, then its redirection table entries (RTEs) need to be reprogrammed every time a CPU comes and goes. This is so that interrupts are delivered to only online CPUs.

  According to Ashok Raj, a safe time to reprogram I/O APIC RTE for any interrupt is when that interrupt is pending, or when the interrupt is masked in RTE.

  Going by the first option, we would have to wait for each interrupt to become pending before reprogramming its RTE. Waiting like this for all interrupts to become

pending may not be a viable solution during CPU Hotplug. Hence the method followed currently is to reprogram RTEs from the dying CPU and wait for a small period (20 microseconds) with interrupts enabled to flush out any pending interrupts. This, in practice, has been enough to avoid lost interrupts.

The right alternative however would be to mask the interrupt in RTE before reprogramming it, but also accounting for the case where the interrupt might have been lost during the interval the entry was left masked. A detailed description of this method is provided in IA64 implementation section.

- **Disabling Local Timer Ticks**
  Local timer ticks are local to each CPU and are not affected by I/O APIC reprogramming. Hence when a CPU is brought down, we have to stop local timer ticks from hitting the dying CPU. This feature is not implemented in the current code. As a consequence, local timer ticks keep hitting and are discarded in software by a `cpu_is_offline` check in its interrupt handler. There are a few solutions under consideration in order to avoid adding a conditional in the timer interrupt path. One method was setting up an offline processor IDT (Interrupt Descriptor Table) which would be loaded when the processor was in the final offline state. The offline IDT would be populated with an entry stub which simply returns from the interrupt. This method would mean that any interrupts hitting the offline processor would be blindly discarded, something which may cause problems if an ACK was required. So what may be safer and sufficient is simply masking the timer LVT for that specific cpu and unmasking it again on the way out of the offline loop.

6 IA64 Implementation

6.1 What is Required to Support CPU Hotplug in IA64?

IA64 CPU hotplug code was developed once Rusty had the base infrastructure support ready. Some of the work that was done to bring the code to stable state include:

- Remove section identifiers marked with `__init` that are required after completing SMP boot. for e.g `cpu_init()`, `do_boot_cpu()` used to wakeup a CPU from SAL_BOOT_RENDEZ mode, `fork_by_hand()` used to fork idle threads for newly added CPUs on the fly.
- Perform a safe interrupt migration from the CPU being removed to another CPU without loss of interrupts.
- Handing off the CPU being removed to SAL_BOOT_RENDEZ mode back to SAL.
- Handling platform level dependencies that trigger physical CPU hotplug in a platform capable of performing it.

6.2 Handling IA64 CPU removal

The arch-specific call `_cpu_disable()` implements the necessary functionality to offline a CPU. The different steps taken are:

1. Check if the platform has any restrictions on this CPU being removed. Returning an error from `_cpu_disable()` ensures that this CPU is still part of the `cpu_online_map`.
2. Turn of local timer interrupt. In IA64 there is a timer interrupt per CPU and not

an external interrupt as in i386 case. It is required that the `timer_interrupt` does not happen any further. It is possible there is one pending, hence check if this interrupt is from an this is an offline CPU, and ignore the interrupt, but just return IRQ_HANDLED, so that the local SAPIC can honour other interrupt vectors now.

3. Ensure that all IRQs bound to this CPU are now targeted to a different CPU by programming the RTEs for a new CPU destination. On return from this step, there must be no more interrupts sent to this CPU being removed from any IOSAPIC.

4. Now the idle thread gets scheduled last, and waits until the CPU state indicates that this CPU must be taken down. Then it hands the CPU to SAL.

6.3 Managing IA64 Interrupts

6.3.1 When Is It Safe to Reprogram an IOSAPIC?

IOSAPIC RTE entries should not be programmed when its actively receiving interrupt signals. The recommended method is to mask the RTE, reprogram for new destination, and then re-enable the RTE. The `/proc/irq` write handlers were calling the set affinity handlers immediately which can cause loss of interrupts, including IOAPIC lockups. In `i386` the introduction of IRQ_BALANCE did this the right way, which is to perform the reprograming operation when an interrupt is pending by storing the intend to change interrupt destinations in a deferred array `pending_irq_balance`.

The same concept was extended to `ia64` as well for the proc write handlers. With the CPU hotplug patches, the write to `/proc/irq` entries are stored in an array and performed when the interrupt is serviced, rather than calling it potentially when an interrupt can also be fired. Due to the delayed nature of these updates, with CPU hotplug, the new destination CPU may be offlined before an interrupt fired and the RTE can be re-programmed. Hence before setting IRQ destination CPU for an RTE, the code should check if the new destination processor is in the `cpu_online_map`.

6.3.2 Why Turn Off Interrupt Redirection Hint With CPU Hotplug?

Interrupt destination in any IOSAPIC RTE must be re-programmed to a different CPU if the CPU being removed is a possible interrupt destination. Since we cannot wait for the interrupt to fire to do the reprogramming, we must force the interrupt destination in safe way. IA64 interrupt architecture permits a platform chipset to perform redirection based on lowest priority based on a hint in the interrupt vector (bit 31) provided by the operating system. If platform interrupt redirection is enabled, it would imply that we need to reprogram all the interrupt destinations, because hotplug code in OS cannot be sure which CPU the chipset is going to direct this interrupt to. Hence if CONFIG_HOTPLUG_CPU is enabled, then we disable platform redirection hint at boot time.

6.3.3 Safely Migrating Interrupt Destinations

The function `fixup_irqs()` performs all the necessary tasks for safely migrating interrupts, and reprogramming interrupt destinations for which this CPU being removed was a destination. The handling of IRQ is managed in 3 distinct phases.

- `migrate_irqs()` performs the job of identifying all IRQs with this CPU as the interrupt destination. This iteration also keeps track of IRQs identified in `vectors_in_migration[]` for later processing to cover cases of missed interrupts, since we mask RTEs during reprogramming, if the device asserted an interrupt during that time, they get lost.

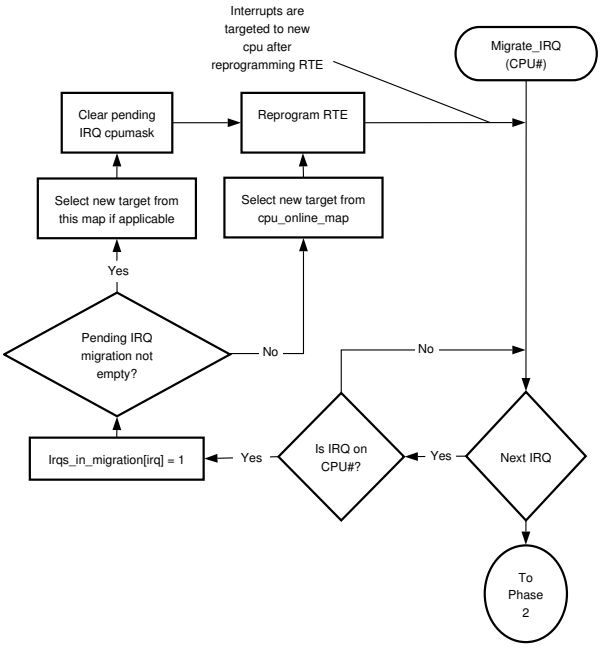


Figure 1: Phase1: Migrate IRQ

- `ia64_process_pending_intr()` Does normal interrupt style processing. During this phase, we look at the local APIC interrupt vector register `ivr` and process all pending interrupts on this CPU. For each processed interrupt, we also clear the bits set in `vectors_in_migration[]`.

- Phase 3 accounts for cases where a device possibly attempted to assert an interrupt, but got lost during the window the RTE was also being re-programmed. This phase looks at entries not accounted for in phase 2, and issues interrupt handler callbacks as if an interrupt happened. It is likely there were no interrupts asserted. We rely on the fact that most device drivers can tolerate calls even if there was no work to perform due to the fact that IRQs may be shared.

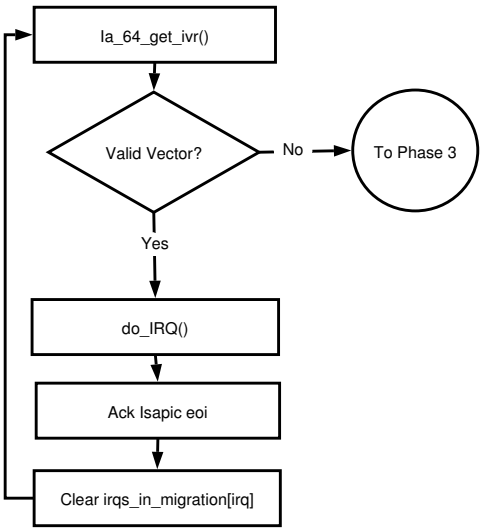


Figure 2: Phase2: Processing Pending intr

6.3.4 Managing Platform Interrupt Sources

IA64 architecture specifies platform interrupt sources to report corrected platform errors to the OS. ACPI specifies these sources via the Platform Interrupt Source Structures. These are communicated to the OS with data such as the following.

- Interrupt Type, indicating if the interrupt is Platform Management Interrupt (PMI), INIT, or CPEI.
- IOSAPIC vector the OS should program.
- The processor that should receive this interrupt, by specifying the APIC id.

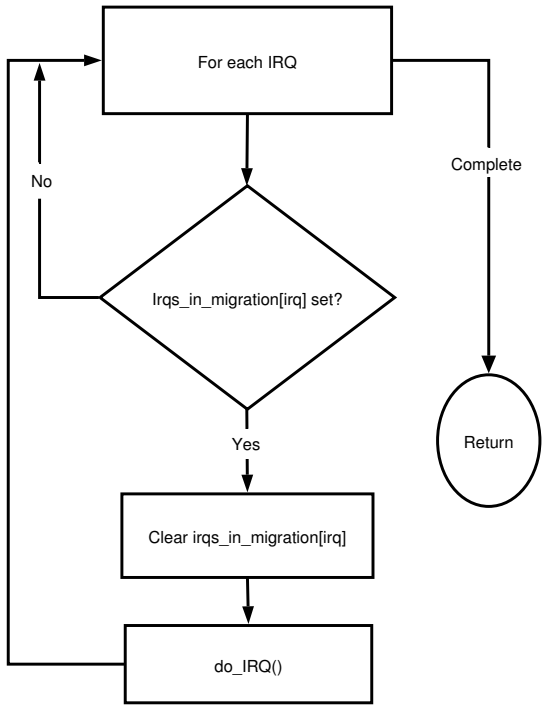


Figure 3: Phase3: Account for Lost Interrupts

- The interrupt line used to signal the interrupts by specifying the global system interrupt.

Some platforms do not support an interrupt model for retrieving platform errors via CPEI. Such platforms provide support via specifying polling tables that list all processors that can poll for Correctable Platform Errors by using the Correctable Platform Error Polling (CPEP) tables.

The issue with both above schemes is that CPEI specifies just one entry for a destination processor. This automatically restricts the target CPU that handles CPEI not removable. On the other hand with CPEP polling tables, although the scheme permits specifying more than one processor, the tables are static and cannot be expanded dynamically as new processors capable of handling polling to be updated.

The motivation for restricting certain processors was that for some platforms that are asymmetric, not all CPUs can retrieve the platform error registers. Hence it is required that only certain processors are permitted. Most platforms that support interruptible model are symmetric in nature. Hence any CPU is capable of accepting the interrupt for CPEI.

We are working with the ACPI specification team to try and address this capability to support platforms supporting CPU hotplug. In the interim before a specification change permits either specifying any CPU as a target, or a method to dynamically update the processors before a CPU gets removed, the code would fail removal of a CPU that is a target of CPEI. In the case of polling, the last processor in the list would be made non-removable.

6.4 Why Should the CPU be handed off to SAL?

The Itanium® processor architecture provides a machine check abort mechanism for reporting and recovering from a variety of errors that can be detected by the processor or chipset. In the event of global MCA, it is required that the slave processors perform checkin with the monarch processor, before which the master could call the recovery to resume execution. SAL would exclude processors in SAL_BOOT_RENDEZ mode. Hence it is important that we return the offlined processors to SAL to avoid processing MCA events on the offlined processor, as the OS would not have it in the active map of CPUs.

6.5 Handling Boot CPU Removal

IA64 architecture does not have any direct dependency that would preclude the boot CPU being removable. There may be some platform level issues such as the boot CPU is usually the target of CPEI or some such dependency that

would make the boot CPU from being removable. In the existing IA64 code base, there is one dependency, that the boot CPU (CPU0) is the master time keeper. This dependency can be easily removed by electing a new CPU as the master timekeeper.

6.6 Recovering the Idle Thread After CPU Removal

Idle threads are created on demand when a new CPU is added to the OS image. These threads are special, since when we return the processor back to SAL, this is done from the context of the idle thread. These calls don't return, and don't have a natural exit path as other threads. The simplest thing to do would be to keep these free idle threads, and just reuse them the next time we need to create a new idle thread for a new CPU.

6.7 Why Was `cpu_present_map` Introduced?

There are several pieces of kernel code that size resources upfront. Before the advent of CPU hotplug, the variable `cpu_possible_map` also indicated the CPUs physically available in the system and would eventually be booted via `smp_init()`. It is very intrusive to make all these callers behave dynamically to CPU hotplug code. There are some issues around this is use of `boot_mem_allocator`. In order to simplify these issues the map `cpu_possible_map` was set to all bits indicating `NR_CPUS`. In order to start only CPUs that are physically present in the system, the new map `cpu_present_map` was added. On platforms capable of supporting CPU hotplug, this map would dynamically change depending on a new CPU being added or removed from the system. In order to accommodate systems that don't directly populate `cpu_present_map` the function `fixup_cpu_present_map` was introduced to just copy the bits from `cpu_possible_map` to `cpu_present_map`.

6.8 ACPI and Platform Issues With CPU hotplug

Any platform capable of supporting hotpluggable CPUs must provide a mechanism to initiate hotplug. Platforms supporting ACPI aware OSs could use ACPI mechanisms to initiate hotplug activity which I would call Physical CPU hotplug. The CONFIG_HOTPLUG_CPU provides the kernel capability and could still be useful if a CPU can be taken offline based on say, the number of correctable error rate.

A typical sequence of operations on a platform supporting a physical CPU is described below. Each specific platform may have additional steps, the following is only a possible sequence and applies to the ACPI based implementations as well.

1. Insert the CPU or the module that contains the CPU into the platform.
2. Platform BIOS does some preparation, and notifies the OS. The kernel platform component such as ACPI that registered to receive the notification, processes this event.
3. Platform dependent OS component prepares necessary information required to bring this CPU to the OS image. For example, in IA64, the code would initialise the following data structures before calling the `cpu_up()`:
   - `ia64_cpu_to_sapicid[]`, in the case of NUMA also populate `node_to_cpu_mask` and `cpu_to_node_map` necessary for NUMA kernels.

- Populate `cpu_present_map` so that kernel now knows about this new CPU is present in the system.

4. Create the necessary entries such as `/sys/devices/system/cpu/cpu#`.

5. Launch the `/sbin/hotplug` script that will now invoke the CPU hotplug agent, which in turn would use the sysfs entry just created to bring up the new CPU.

7 PPC64 Implementation

7.1 What PPC64 Specific Tasks Occur During a CPU Removal?

The architecure specific kernel pieces of a CPU removal focus on three functions mentioned previously: `__cpu_disable()`, `cpu_down()`, and `__cpu_die()`.

In `__cpu_disable()` all interrupts are disabled and migrated, with the exception of interprocessor interrupts (IPIs).

1. The process of disabling interrupts starts off by writing 0 into the processor's current processor priority register (CPPR) to reject any possible queued interrupts.

2. With the CPPR set to 0 it is safe to remove ourselves from the global interrupt queue server, which is done via a Run-Time Abstraction Service (RTAS) set-indicator call that is provided by the firmware. This has the effect of refusing new interrupts from being added to the processor.

3. After new interrupts are refused the next step is to set the CPPR back to default priority, which allows us to recieve IPIs again.

4. All interrupts are iterated through, checking via an RTAS "get-xive" call if any of the interrupts are specific to the target processor.

5. If an interrupt is specific to the target processor it is migrated via an RTAS "set-xive" call.

6. With the processor removed from the global interrupt queue server and all interrupts migrated it would be safe to remove the target processor without affecting the delivery of interrupts. Success is returned.

During `__stopmachine_run()` the online attribute of a CPU is set to to 0. On PPC64 we stop the CPU at this point by calling `cpu_die()` (not to be confused with `__cpu_die()`)

1. Depending on the machine model and kernel configuration, the idle function will be `default_idle()`, `dedicated_idle()`, or `shared_idle()`. All three idle functions check `cpu_is_offline()` and if it is true call `cpu_die()`.

2. `cpu_die` first disables IRQs.

3. After disabling IRQs it clears the CPPR.

4. Finally `rtas_stop_self()` is called, stopping the processor.

Most architectures use `__cpu_die()` to stop the processor. Because on PPC64 we poll for offline CPUs we only need to wait and confirm the CPU has been stopped while in this function.

1. We confirm the CPU has been stopped by using the RTAS `query-cpu-stopped-state` call.

2. Because this call can return busy, and because the CPU may not yet be stopped we loop and schedule timeouts.

3. After confirming the CPU is stopped we do a little extra cleanup by clearing the corresponding entry in the `cpu_callin_map` and `xProcStart` in the PACA.

7.2 What About Adding CPUs?

The initial structure of PPC64 CPU bringup required a lot of modification to be able to add CPUs after the system was already running. Most of the changes are trivial and straightforward, but one bears mentioning.

PPC64 used to number CPUs based on their physical id. With CPU hotplug it would have been necessary to reserve a CPU entry and corresponding structures for each possible physical CPU. It was quite possible that the machine could have more CPUs than the kernel was compiled to work with, as many CPUs would be assigned to other partitions. Furthermore, the number of CPUs in the machine was not necessarily a static number. Also, from a usability point of view there were going to be far too many entries in `/sys/devices/system/cpu/` compared to how many CPUs were actually online.

The CPU numbering was logically abstracted so that for kernel use there was a logical number, and when interfacing to the hardware there was a corresponding physical number. The kernel is able to read at boot time the maximum number of CPUs the partition is configured to be able to grow to. Thus it reserves less space in structures that must be allocated at boot time, allows reuse of logical CPUs for different physical CPUs, and presents a cleaner directory structure.

7.3 Other Software

While outside the scope of this paper it is worth mentioning that there is other software running on PPC64 platforms to enable customers halfway around the world from the machines they administer to use their mouse and move CPUs. This software is downloadable from IBM, and should be available on the bonus CD shipped with new machines.

Issues with Selected Scalability Features of the 2.6 Kernel

Dipankar Sarma
Linux Technology Center
IBM India Software Lab
dipankar@in.ibm.com

Paul E. McKenney
Linux Technology Center and Storage Software Architecture
IBM Beaverton
paulmck@us.ibm.com

Abstract

The 2.6 Linux™ kernel has a number of features that improve performance on high-end SMP and NUMA systems. Finer-grain locking is used in the scheduler, the block I/O layer, hardware and software interrupts, memory management, and the VFS layer. In addition, 2.6 brings new primitives such as RCU and per-cpu data, lock-free algorithms for route cache and directory entry cache as well as scalable user-level APIs like `sys_epoll()` and futexes. With the widespread testing of these features of the 2.6 kernel, a number of new issues have come to light that needs careful analysis. Some of these issues encountered thus far are: overhead of multilple lock acquisitions and atomic operations in critical paths, possibility of denial-of-service attack on subsystems that use RCU-based deferred free algorithms and degradation of realtime response due to increased softirq load.

In this paper, we analyse a select set of these issues, present the results, workaround patches and future courses of action. We also discuss applicability of some these issues in new features being planned for 2.7 kernel.

1 Introduction

Support for symmetric multi-processing (SMP) in the Linux kernel was first introduced in 2.0 kernel. The 2.0 kernel had a single `kernel_flag` lock AKA Big Kernel Lock (BKL) which essentially single threaded almost all of the kernel [Love04a]. The 2.2 kernel saw the introduction of finer-grain locking in several areas including signal handling, interrupts and part of I/O subsystem. This trend continued in 2.4 kernel.

A number of significant changes were introduced in during the development of the 2.6 kernel that helped boost performance of many workloads. Some of the key components of the kernel were changed to have finer-grain locking. For example, the global `runqueue_lock` lock was replaced by the locks on the new per-cpu runqueues. Gone was `io_request_lock` with the introduction of the new scalable `bio`-based block I/O subsystem. BKL was peeled off from ad-

ditional commonly used paths. Use of data locking instead of code locking became more widespread. In addition, Read-Copy Update(RCU) [McK98a, McK01a] allowed further optimization of critical sections by avoiding locking while reading data structures which are updated less often. RCU enabled lock-free lookup of the directory-entry cache and route cache, which provided considerable performance benefits [Linder02a, Blanchard02a, McK02a]. While these improvements targeted high-end SMP and NUMA systems, the vast majority of the Linux-based systems in the computing world are small uniprocessor or low-end SMP systems that remain the main focus of the Linux kernel. Therefore, scalability enhancements must not cause any performance regressions in these smaller systems, and appropriate regression testing is required [Sarma02a]. This effort continues and has since thrown light on interesting issues which we discuss here.

Also, since the release of the 2.6 kernel, its adoption in many different types of systems has called attention to some interesting issues. Section 2 describes the 2.6 kernel's use of fine-grained locking and identifies opportunities in this area for the 2.7 kernel development effort. Section 3 discusses one such important issue that surfaced during Robert Olsson's router DoS testing. Section 4 discusses another issue important for real-time systems or systems that run interactive applications. Section 5 explores the impact of such issues and their workarounds on new experiments planned during the development of 2.7 kernel.

2 Use of Fine-Grain Locking

Since the support for SMP was introduced in the 2.0 Linux kernel, granularity of locking has gradually changed toward finer critical sections. In 2.4 and subsequently 2.6 kernel, many of the global locks were broken up to improve scalability of the kernel. Another scalability improvement was the use of reference counting in protecting kernel objects. This allowed us to avoid long critical sections. While these features benefit large SMP and NUMA systems, on smaller systems, benefit due to reduction of lock contention is minimal. There, the cost of locking due to atomic operations involved needs to be carefully evaluated. Table 1 shows cost of atomic operations on a 700MHz Pentium™ III Xeon™ processor. The cost of atomic increment is more than 4 times the cost of an L2 hit. In this section, we discuss some side effects of such finer-grain locking and possible remedies.

Operation	Cost (ns)
Instruction	0.7
Clock Cycle	1.4
L2 Cache Hit	12.9
Atomic Increment	58.2
cmpxchg Atomic Increment	107.3
Atomic Incr. Cache Transfer	113.2
Main Memory	162.4
CPU-Local Lock	163.7
cmpxchg Blind Cache Transfer	170.4
cmpxchg Cache Transfer and Invalidate	360.9

Table 1: 700 MHz P-III Operation Costs

2.1 Multiple Lock Acquisitions in Hot Path

Since many layers in the kernel use their own locks to protect their data structures, we did a simple instrumentation (Figure 1) to see how many locks we acquire on common paths. This counted locks in all variations of spinlock and rwlock. We used a running counter which we can read using a system call `get_lcount()`. This counts only locks acquired by the task in non-interrupt context.

With this instrumented kernel, we measured writing 4096 byte buffers to a file on ext3 filesystem. Figure 2 shows the test code

```
+static inline void _count_lock(void)
+{
+   if ((preempt_count() & 0x00ffff00) == 0) {
+           current_thread_info()->lcount++;
+   }
+}

....

 #define spin_lock(lock)         \
 do { \
+   _count_lock(); \
    preempt_disable(); \
    _raw_spin_lock(lock); \
 } while(0)
```

Figure 1: Lock Counting Code

```
if (get_lcount(&lcount1) != 0) {
        perror("get_lcount 1 failed\n");
        exit(-1);
}
write(fd, buf, 4096);
if (get_lcount(&lcount2) != 0) {
        perror("get_lcount 2 failed\n");
        exit(-1);
}
```

Figure 2: Lock Counting Test Code

that reads the lock count before and after the `write()` system call.

4K Buffer	Locks Acquired
0	19
1	11
2	10
3	11
4	10
5	10
6	10
7	10
8	16
9	10
Average	11.7

Table 2: Locks acquired during 4K writes

Table 2 shows the number of locks acquired during each 4K write measured on a 2-way Pentinum IV HT system running 2.6.0 kernel. The first write has a lock acquisition count of 19 and an average of 11.7 lock round-trips per 4K write. This does not count locks associated with I/O completion handling which is done from interrupt context. While this indicates scalability of the code, we still need to analyze this to see which locks are acquired in such hot path and check if very small adjacent critical sections can be collapsed into one. The modular nature of some the kernel layers may however make that impossible without affecting readability of code.

```
1 struct file * fget(unsigned int fd)
2 {
3       struct file * file;
4       struct files_struct *files =
5                       current->files;
6
7       read_lock(&files->file_lock);
8       file = fcheck(fd);
9       if (file)
10              get_file(file);
11      read_unlock(&files->file_lock);
12      return file;
13 }
```

Figure 3: fget() Implementation

2.2 Refcounting in Hot Path

As described in Section 2.1, atomic operations can be costly. In this section, we discuss such an issue that was addressed during the development of the 2.6 kernel. Andrew Morton [Morton03a] pointed out that in 2.5.65-mm4 kernel, CPU cost of writing a large amount of small chunks of data to an ext2 file is quite high on uniprocessor systems and takes nearly twice again as long on SMP. It also showed that a large amount of overheads there were coming from `fget()` and `fput()` routines. A further look at Figure 3 shows how `fget()` was implemented in 2.5.65 kernel.

Both `read_lock()` and `read_unlock()` involve expensive atomic operations. So, even if there is no contention for `->file_lock`, the atomic operations hurt performance [McKenney03a]. Since most programs do not share their file-descriptor tables, the reader-writer lock is usually not really necessary. The lock need only be acquired when the reference count of the `file` structure indicates sharing. We optimized this as shown in Fig-

```
1  struct file *fget_light(unsigned int fd,
2          int *fput_needed)
3  {
4      struct file *file;
5      struct files_struct *files = current->files;
6
7      *fput_needed = 0;
8      if (likely((atomic_read(&files->count)
9                          == 1))) {
10         file = fcheck(fd);
11     } else {
12         read_lock(&files->file_lock);
13         file = fcheck(fd);
14         if (file) {
15             get_file(file);
16             *fput_needed = 1;
17         }
18         read_unlock(&files->file_lock);
19     }
20     return file;
21 }
```

Figure 4: fget_light() Implementation

ure 4.

By optimizing the fast path to avoid atomic operation, we reduced the system time use by 11.2% in a UP kernel while running Andrew Morton's micro-benchmark with the command `dd if=/dev/zero of=foo bs=1 count=1M`. The complete results measured in a 4-CPU 700MHz Pentium III Xeon system with 1MB L2 cache and 512MB RAM is shown in Table 3

Kernel	sys time	Std Dev
2.5.66 UP	2.104	0.028
2.5.66-file UP	1.867	0.023
2.5.66 SMP	2.976	0.019
2.5.66-file SMP	2.719	0.026

Table 3: fget_light() results

However, the reader-writer lock must still be acquired in `fget_light()` fast path when the file descriptor table is shared. This is now being further optimized using RCU to make the file descriptor lookup fast path completely lock-free. Optimizing file descriptor look-up in shared file descriptor table will improve performance of multi-threaded applications that do a lot of I/Os. Techniques such as this are extremely useful for improving performance in both low-end and high-end SMP systems.

```
1  static __inline__ void rt_free(
2          struct rtable *rt)
3  {
4    call_rcu(&rt->u.dst.rcu_head,
5      (void (*)(void *))dst_free,
6      &rt->u.dst);
7  }
8
9  static __inline__ void rt_drop(
10           struct rtable *rt)
11 {
12   ip_rt_put(rt);
13   call_rcu(&rt->u.dst.rcu_head,
14       (void (*)(void *))dst_free,
15       &rt->u.dst);
16 }
```

Figure 5: `dst_free()` Modifications

3 Denial-of-Service Attacks on Deferred Freeing

[McK02a] describes how RCU is used in the IPV4 route cache to void acquiring the per-bucket reader-writer lock during lookup and the corresponding speed-up of route cache lookup. This was included in the 2.5.53 kernel. Later, Robert Olsson subjected a 2.5 kernel based router to DoS stress tests using `pktgen` and discovered problems including starvation of user-space execution and out-of-memory conditions. In this section, we describe our analysis of those problems and potential remedies that were experimented with.

3.1 Potential Out-of-Memory Situation

Starting with the 2.5.53 kernel, the IPv4 route cache uses RCU to enable lock-free lookup of the route hash table.

The code in Figure 5 shows how route cache entries are freed. Because each route cache entry's freeing is deferred by `call_rcu()`, it is not returned to its slab immediately. However

```
CLONE_SKB="clone_skb 1"
PKT_SIZE="pkt_size 60"
COUNT="count 10000000"
IPG="ipg 0"
PGDEV=/proc/net/pktgen/eth0
echo "Configuring $PGDEV"
pgset "$COUNT"
pgset "$CLONE_SKB"
pgset "$PKT_SIZE"
pgset "$IPG"
pgset "flag IPDST_RND"
pgset "dst_min 5.0.0.0"
pgset "dst_max 5.255.255.255"
pgset "flows 32768"
pgset "flowlen 10"
```

Figure 6: pktgen parameters

the route cache imposes a limit of total number of in-flight entries at `ip_rt_max_size`. If this limit is exceeded, subsequent allocation of route cache entries are failed. We reproduced Robert's experiment in a setup where we send 100,000 packets/sec to a 2.4GHz Pentium IV Xeon 2-CPU HT system with 256MB RAM running 2.6.0 kernel set up as a router. Figure 6 shows the parameters used in `pktgen` testing. This script sends 10000000 packets to the router with random destination addresses in the range 5.0.0.0 to 5.255.255.255. The router has an outgoing route set up to sink these packets. This results in a very large number of route cache entries along with pruning of the cache due to aging and garbage collection.

We then instrumented RCU infrastructure to collect lengths of RCU callback batches invoked after grace periods and corresponding grace period lengths. As indicated by the graph plotted based on this instrumentation (Figure 7), it is evident that every spike in RCU batch length as an associated spike in RCU grace period. This indicates that prolonged grace periods are resulting in very large numbers of pending callbacks.

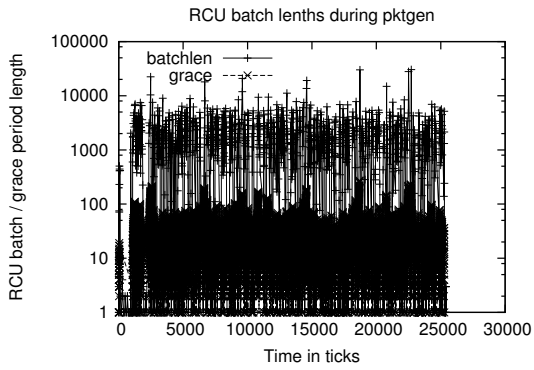


Figure 7: Effect of pktgen testing on RCU

Next we used the same instrumentation to understand what causes long grace periods. We measured total number of softirqs received by each cpu during consecutive periods of 4 `jiffies` (approximately 4 milliseconds) and plotted it along with the corresponding maximum RCU grace period length seen during that period. Figure 8 shows this relationship. It clearly shows that all peaks in RCU grace period had corresponding peaks in number of softirqs received during that period. This conclusively proves that large floods of softirqs holds up progress in RCU. An RCU grace period of 300 milliseconds during a 100,000 packets/sec DoS flood means that we may have up to 30,000 route cache entries pending in RCU subsystem waiting to be freed. This causes us to quickly reach the route cache size limits and overflow.

In order to avoid reaching the route cache entry limits, we needed to reduce the length of RCU grace periods. We then introduced a new mechanism named *rcu-softirq* [Sarma04a] that considers completion of a softirq handler a *quiescent* state. It introduces a new interface `call_rcu_bh()`, which is to be used when the RCU protected data is mostly used from softirq handlers. The update function will be invoked as soon as all CPUs have performed a context switch or been seen in the

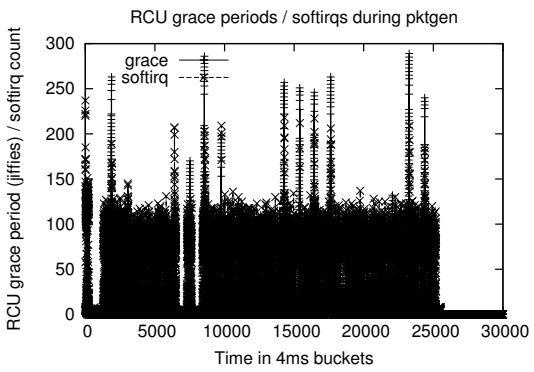


Figure 8: Softirqs during pktgen testing

idle loop or in a user process or or has exited a softirq handler that it may have been executing. The reader side of critical section that use call_rcu_bh() for updating must be protected by `rcu_read_lock_bh()` and `rcu_read_unlock_bh()`. The IPv4 route cache code was then modified to use these interfaces instead. With this in place, we were able to avoid route cache overflows at the rate of 100,000 packets/second. At higher packet rates, route cache overflows have been reported. Further analysis is being done to determine if at higher packet rates, current softirq implementation doesn't allow route cache updates to keep up with new route entries getting created. If this is the case, it may be necessary to limit softirq execution in order to permit user-mode execution to continue even in face of DoS attacks.

3.2 CPU Starvation Due to softirq Load

During the `pktgen` testing, there was another issue that came to light. At high softirq load, user-space programs get starved of CPU. Figure 9 is a simple piece of code that can be used to test this under severe `pktgen` stress. In our test router, it indicated user-space starvation for periods longer that 5 seconds. Application of the *rcu-softirq* patch reduced it by a few seconds. In other words, introduction of quicker

```
gettimeofday(&prev_tv, NULL);

for (;;) {
    gettimeofday(&tv, NULL);
    diff = (tv.tv_sec - prev_tv.tv_sec)*
        1000000 +
        (tv.tv_usec - prev_tv.tv_usec);
        if (diff > 1000000)
            printf("%d\n", diff);
    prev_tv = tv;
}
```

Figure 9: user-space starvation test

RCU grace periods helped by reducing size of pending RCU batches. But the overall softirq rate remained high enough to starve user-space programs.

4 Realtime Response

Linux has been use in realtime and embedded applications for many years. These applications have either directly used Linux for soft realtime use, or have used special environments to provide hard realtime, while running the soft-realtime or non-realtime portions of the application under Linux.

4.1 Hard and Soft Realtime

Realtime applications require latency guarantees. For example, such an application might require that a realtime task start running within one millisecond of its becoming runnable. Andrew Morton's `amlat` program may be used to measure an operating system's ability to meet this requirement. Other applications might require that a realtime task start running within 500 microseconds of an interrupt being asserted.

Soft realtime applications require that these guarantees be met *almost* all the time. For example, a building control application might require that lights be turned on within 250 milliseconds of motion being detected within a

given room. However, if this application occasionally responds only within 500 milliseconds, no harm is likely to be done. Such an application might require that the 250 millisecond deadline be met 99.9% of the time.

In contrast, hard realtime applications require that guarantees *always* be met. Such applications may be found in avionics and other situations where lives are at stake. For example, Stealth aircraft are aerodynamically unstable in all three axes, and require frequent computer-controlled attitude adjustments. If the aircraft fails to receive such adjustments over a period of two seconds, it will spin out of control and crash [Rich94]. These sorts of applications have traditionally run on "bare metal" or on a specialized realtime OS (RTOS).

Therefore, while one can validate a soft-realtime OS by testing it, a hard-realtime OS must be validated by inspection and testing of *all* non-preemptible code paths. Any non-preemptible code path, no matter how obscure, can destroy an OS's hard-realtime capabilities.

4.2 Realtime Design Principles

This section will discuss how preemption, locking, RCU, and system size affect realtime response.

4.2.1 Preemption

In theory, neither hard nor soft realtime require preemption. In fact, the realtime systems that one of the authors (McKenney) worked on in the 1980s were all non-preemptible. However, in practice, preemption can greatly reduce the amount of work required to design and validate a hard realtime system, because while one must validate *all* code paths in a non-preemptible system, one need only validate all *non-preemptible* code paths in a preemptible system.

4.2.2 Locking

The benefits of preemption are diluted by locking, since preemption must be suppressed across any code path that holds a spinlock, even in UP kernels. Since most long-running operations are carried out under the protection of at least one spinlock, the ability of preemption to reduce the Linux kernel's hard realtime response is limited.

That said, the fact that spinlock critical sections degrade realtime response means that the needs of the hard realtime Linux community are aligned with those of the SMP-scalability Linux community.

Traditionally, hard-realtime systems have run on uniprocessor hardware. The advent of hyperthreading and multicore dies have provided cheap SMP, which is likely to start finding its way into realtime and embedded systems. It is therefore reasonable to look at SMP locking's effects on realtime response.

Obviously, a system suffering from heavy lock contention need not apply for the job of a realtime OS. However, if lock contention is sufficiently low, SMP locking need not preclude hard-realtime response. This is shown in Figure 10, where the maximum "train wreck" lock spin time is limited to:

$$S_{max} = (N_{CPU} - 1)C_{max} \quad (1)$$

where N_{CPU} is the number of CPUs on the system and C_{max} is the maximum critical section length for the lock in question. This maximum lock spin time holds as long as each CPU spends at least S_{max} time outside of the critical section.

It is not yet clear whether Linux's lock contention can be reduced sufficiently to make this

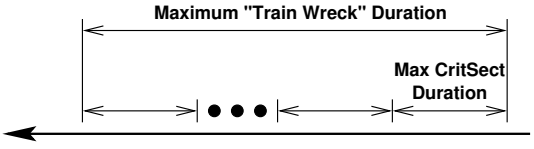


Figure 10: SMP Locking and Realtime Response

level of hard realtime guarantee, however, this is another example of a case where improved realtime response benefits SMP scalability and vice versa.

4.2.3 RCU

Towards the end of 2003, Robert Love and Andrew Morton noted that the Linux 2.6 kernel's RCU implementation could degrade realtime response. This degradation is due to the fact that, when under heavy load, literally thousands of RCU callbacks will be invoked at the end of a grace period, as shown in Figure 11.

The following three approaches can each eliminate this RCU-induced degradation:

1. If the batch of RCU callbacks is too large, hand the excess callbacks to a preemptible per-CPU kernel daemon for invocation. The fact that these daemons are preemptible eliminates the degradation.

2. On uniprocessors, in cases where pointers to RCU-protected elements are not held across calls to functions that remove those elements, directly invoke the RCU callback from within the `call_rcu_rt()` primitive, which is identical to the `call_rcu()` primtive in SMP kernels. The separate `call_rcu_rt()` primitive is necessary because direct invocation is not safe in all cases.

3. Throttling RCU callback invocation so that only a limited number are invoked at a given time, with the remainder being invoked later, after there has been an opportunity for realtime tasks to run.

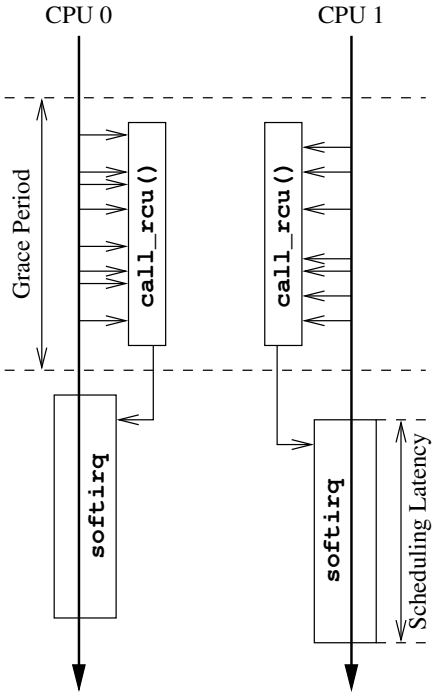


Figure 11: RCU and Realtime Response

The throttling approach seems most attractive currently, but additional testing will be needed after other realtime degradations are resolved. The implementation of each approach and performance results are presented elsewhere [Sarma04b].

4.2.4 System Size

The realtime response of the Linux 2.6 kernel depends on the hardware and software configuration. For example, the current VGA driver degrades realtime response, with multimillisecond scheduling delays due to screen blanking.

In addition, if there are any non-O(1) oper-

ations in the kernel, then increased configuration sizes will result in increased realtime scheduling degradations. For example, in SMP systems, the duration of the worst-case locking "train wreck" increases with the number of CPUs. Once this train-wreck duration exceeds the minimum time between release and later acquisition of the lock in question, the worst-case scheduling delay becomes unbounded. Other examples include the number of tasks and the duration of the tasklist walk resulting from `ls /proc`, the number of processes mapping a given file and the time required to truncate that file, and so on.

In the near term, it seems likely that realtime-scheduling guarantees would only apply to a restricted configuration of the Linux kernel, running a restricted workload.

4.3 Linux Realtime Options

The Linux community can choose from the following options when charting its course through the world of realtime computing:

1. "Just say no" to realtime. It may well be advisable for Linux to limit how much realtime support will be provided, but given recent measurements showing soft-realtime scheduling latencies of a few hundred *micro*seconds, it seems clear that Linux has a bright future in the world of realtime computing.

2. Realtime applications run only on UP kernels. In the past, realtime systems have overwhelmingly been single-CPU systems, it is much easier to provide realtime scheduling guarantees on UP systems. However, the advent of cheap SMP hardware in the form of hyperthreading and multi-CPU cores makes it quite likely that the realtime community will choose to support SMP sooner rather than later. One possibility would be to provide tighter guarantees on UP systems, and, should Linux provide hard realtime support, to provide this support only on UP systems. Another possibility would be to dedicate a single CPU of an SMP system to hard realtime.

3. Realtime applications run only on small hardware configurations with small numbers of tasks, mappings, open files, and so on. This seems to be an eminently reasonable position, especially given that dirt-cheap communications hardware is available, allowing a small system (perhaps on a PCI card) to handle the realtime processing, with a large system doing non-realtime tasks requiring larger configurations.

4. Realtime applications use only those devices whose drivers are set up to provide realtime response. This also seems to be an eminently reasonable restriction, as open-source drivers can be rewritten to offer realtime response, if desired.

5. Realtime applications use only those services able to provide the needed response-time guarantees. For example, an application that needs to respond in 500 microseconds is not going to be doing any disk I/O, since disks cannot respond this quickly. Any data needed by such an application must be obtained from much faster devices or must be preloaded into main memory.

It is not clear that Linux will be able to address each and every realtime requirement, nor is it clear that this would even be desirable. However, it was not all that long ago that common wisdom held that it was not feasible to address both desktop and high-end server requirements with a single kernel source base. Linux is well

on its way to proving this common wisdom to be quite wrong.

It will therefore be quite interesting to see what realtime common wisdom can be overturned in the next few years.

5 Future Plans

With the 2.6 kernel behind us, a number of new scalability issues are currently being investigated. In this section, we outline a few of them and the implications they might have.

5.1 Parallel Directory Entry Cache Updates

In the 2.4 kernel, the directory entry cache was protected by a single global lock `dcache_lock`. In the 2.6 kernel, the look-ups into the cache were made lock-free by using RCU [Linder02a]. We also showed in [Linder02a] that for several benchmarks, only 25% of acquisitions of `dcache_lock` is for updating the cache. This allowed us to achieve significant performance improve by avoiding the lock during look-up while keeping the updates serialized using `dcache_lock`. However recent benchmarking on large SMP systems have shown that `dcache_lock` acquisitions are proving to be costly. Profile for a mutli-user benchmark on a 16-CPU Pentium IV Xeon with HT indicates this:

Function	Profile Counts
`.text.lock.dec_and_lock`	34375.8333
`atomic_dec_and_lock`	1543.3333
`.text.lock.libfs`	800.7429
`.text.lock.dcache`	611.7809
`__down`	138.4956
`__d_lookup`	93.2842
`dcache_readdir`	70.0990
`do_page_fault`	45.0411
`link_path_walk`	9.4866

On further investigation, it is clear that `.text.lock.dec_and_lock` cost is due to frequent `dput()` which uses `atomic_dec_and_test()` to acquire `dcache_lock`. With the multi-user benchmark creating and destroying large number of files in `/proc` filesystem, the cost of corresponding updates to the directory entry cache is hurting us. During the 2.7 kernel development, we need to look at allowing parallel updates to the directory entry cache. We attempted this [Linder02a], but it was far too complex and too late in the 2.5 kernel development effort to permit such a high-risk change.

5.2 Lock-free TCP/UDP Hash Tables

In the Linux kernel, INET family sockets use hash tables to maintain the corresponding `struct sock`s. When an incoming packet arrives, this allows efficient lookup of these per-bucket locks. On a large SMP system with tens of thousands on tcp and ip header information. TCP uses `tcp_ehash` for connected sockets, `tcp_listening_hash` for listening sockets and `tcp_bhash` for bound sockets. On a webserver serving large number of simultaneous connections, lookups into `tcp_ehash` table are very frequent. Currently we use a per-bucket reader-writer lock to protect the hash tables and `tcp_ehash` lookups are protected by acquiring the reader side of these per-bucket locks. The hash table makes CPU-CPU collisions on hash chains unlikely and prevents the reader-writer lock from providing any possible performance benefit. Also, on a large SMP system with tens of thousands of simultaneous connection, the cost of atomic operation during `read_lock()` and `read_unlock()` as well as the bouncing of cache line containing the lock becomes a factor. By using RCU to protect the hash tables, the lookups can be done without acquiring the per-bucket lock. This will benefit bot low-end and high-end SMP systems. That said, issues similar to the ones discussed in Section 3 will need to be addressed. RCU can be stressed us-

ing a DoS flood that opens and closes a lot of connections. If the DoS flood prevents user-mode execution, it can also prevent RCU grace periods from happening frequently, in which case, a large number of `sock` structures can be pending in RCU waiting to be free leading to potential out-of-memory situations. The *rcu-softirq* patch discussed in Section 3 will be helpful in this case too.

5.3 Balancing Interrupt and Non-Interrupt Loads

In Section 3.2, we discussed user programs getting starved of CPU time under very high network load. In 2001, Ingo Molnar attempted limiting hardware interrupts based on number of such interrupts serviced during one jiffy [Molnar01a]. Around the same time, Jamal Hadi et al. demonstrated the usefulness of limiting interrupts through *NAPI* infrastructure [Jamal01a]. *NAPI* is now a part of 2.6 kernel and it is supported by a number of network drivers. While *NAPI* limits hardware interrupts, it continues to raise softirqs for processing of incoming packets while polling. So, under high network load, we see user processes starved of CPU. This has been seen with *NAPI* (Robert Olsson's lab) as well as without *NAPI* (in our lab). With extremely high network load like DoS stress, softirqs completely starve user processes. Under such situation, a system administrator may find it difficult to take log into a router and take necessary steps to counter the DoS attack. Another potential problem is that network I/O intensive benchmarks like SPECWeb99™ can have user processes stalled due to high softirq load. We need to look for a new framework that allows us to balance CPU usage between softirqs and process context too. One potential idea being considered is to measure softirq processing time and mitigate it for later if it exceeds its tunable quota. Variations of this need to be evaluated during the development of 2.7 kernel.

5.4 Miscellaneous

1. Lock-free dcache Path Walk: Given a file name, the Linux kernel uses a path walking algorithm to look-up the `dentry` corresponding to each component of the file name and traverse down the `dentry` tree to eventually arrive at the `dentry` of the specified file name. In 2.6 kernel, we implemented a mechanism to look-up each path component in dcache without holding the global `dcache_lock` [Linder02a]. However this requires acquiring a per-`dentry` lock when we have a successful look-up in dcache. The common case of paths starting at the root directory results in contention on the root `dentry` on large SMP systems. Also, the per-`dentry` lock acquisition happens in the fast path (`__d_lookup()`) and avoiding this will likely provide nice performance benefits.

2. Lock-free Tasklist Walk: The system-wide list of tasks in the Linux kernel is protected by a reader-writer lock `tasklist_lock`. There are a number of occasions when the list of tasks need to be traversed while holding the reader side of `tasklist_lock`. In systems with very large number of tasks, the readers traversing the task list can starve out writers. One approach to solving this is to use RCU to allow lock-free walking of the task list under limited circumstances. [McKenney03a] describes one such experiment.

3. Cache Thrashing Measurements and Minimization: As we run Linux on larger SMP and NUMA systems, the effect of cache thrashing becomes more prominent. It would prudent to analyze cache behavior of performance critical code in the Linux kernel using various performance

monitoring tools. Once we identify code showing non-optimal cache behavior, redesigning some of it would help improve performance.

4. Real-time Work—Fix Excessively Long Code Paths: With Linux increasingly becoming preferred OS for many soft-realtime systems, we can further improve its usefulness by identifying excessively long code paths and fixing them.

6 Conclusions

In 2.6 kernel, we have solved a number of scalability problems without significantly sacrificing performance in small systems. A single code base supporting so many different workloads and architectures is an important advantages of the Linux kernel has over many other operating systems. Through this analysis, we have continued the process of evaluating scalability enhancements from many possible angles. This will allow us to run Linux better on many different types of system—large SMP to small TCP/IP routers.

We are continuing to work on some of the core issues discussed in the paper including locking overheads, RCU DoS attack prevention and softirq balancing. We expect to do some of this work in the 2.7 kernel timeframe.

7 Acknowledgments

We owe thanks to Andrew Morton for drawing attention to locking issues. Robert Olsson was the first to show us the impact of denial-of-service attacks on route cache and without his endless testing with our patches, we would have gone nowhere. Ravikiran Thirumalai helped a lot with our lab setup, revived some of the lock-free patches and did some of the measurements. We are thankful to him for this. We would also like to thank a number of Linux kernel hackers, including Dave Miller, Andrea Arcangeli, Andi Kleen and Robert Love, all of whom advised us in many different situations. Martin Bligh constantly egged us on and often complained with large SMP benchmarking results, and for this, he deserves our thanks. We are indebted to Tom Hanrahan, Vijay Sukthankar, Dan Frye and Jai Menon for being so supportive of this effort.

References

[Blanchard02a] A. Blanchard *some RCU dcache and ratcache results*, Linux-Kernel Mailing List, March 2002. `http://marc.theaimsgroup.com/?l=linux-kernel&m=101637107412972&w=2`,

[Jamal01a] Jamal Hadi Salim, R. Olsson and A. Kuznetsov *Beyond Softnet*, Proceedings of USENIX 5th Annual Linux Showcase, pp. 165–172, November 2001.

[Linder02a] H. Linder, D. Sarma, and Maneesh Soni. *Scalability of the Directory Entry Cache*, Ottawa Linux Symposium, June 2002.

[Love04a] Robert Love *Kernel Korner: Kernel Locking Techniques*, *Linux Journal*, Issue 100, August 2002. `http://www.linuxjournal.com/article.php?sid=5833`,

[McK98a] P. E. McKenney and J. D. Slingwine. *Read-copy update: using execution history to solve concurrency problems*, Parallel and Distributed Computing and Systems, October 1998. (revised version available at `http://www.rdrop.com/users/paulmck/rclockpdcsproof.pdf`),

[McK01a] P. E. McKenney and D. Sarma. *Read-Copy Update Mutual Exclusion in Linux*, http://lse.sourceforge.net/locking/rcu/rcupdate_doc.html, February 2001.

[McK01b] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, M. Soni. *Read-Copy Update*, Ottawa Linux Symposium, July 2001. (revised version available at http://www.rdrop.com/users/paulmck/rclock/rclock_OLS.2001.05.01c.sc.pdf),

[McK02a] P. E. McKenney, D. Sarma, A. Arcangeli, A. Kleen, O. Krieger *Read-Copy Update*, Ottawa Linux Symposium, June 2002. http://www.linux.org.uk/~ajh/ols2002_proceedings.pdf.gz

[McKenney03a] MCKENNEY, P.E. Using RCU in the Linux 2.5 kernel. *Linux Journal 1*, 114 (October 2003), 18–26.

[Molnar01a] Ingo Molnar *Subject: [announce] [patch] limiting IRQ load, irq-rewrite-2.4.11-B5*, Linux Kernel Mailing List, Oct 2001. http://www.uwsg.iu.edu/hypermail/linux/kernel/0110.0/0169.html,

[Morton03a] Andrew Morton *Subject: smp overhead, and rwlocks considered harmful*, Linux Kernel Mailing List, March 2003. http://www.uwsg.iu.edu/hypermail/linux/kernel/0303.2/1883.html

[Rich94] B. Rich *Skunk Works*, Back Bay Books, Boston, 1994.

[Sarma02a] D. Sarma *Subject: Some dcache_rcu benchmark numbers*, Linux-Kernel Mailing List, October 2002. http://marc.theaimsgroup.com/?l=linux-kernel&m=103462075416638&w=2,

[Sarma04a] D. Sarma *Subject: Re: route cache DoS testing and softirqs*, Linux-Kernel Mailing List, April 2004. http://marc.theaimsgroup.com/?l=linux-kernel&m=108077057910562&w=2,

[Sarma04b] D. Sarma and P. McKenney *Making RCU Safe for Deep Sub-Millisecond Response Realtime Applications*, USENIX'04 Annual Technical Conference (UseLinux Track), Boston, June 2004.

8 Legal Statement

Achieving CAPP/EAL3+ Security Certification for Linux

Kittur (Doc) S. Shankar
IBM Linux Technology Center
dshankar@us.ibm.com

Olaf Kirch
SUSE Linux AG
okir@suse.de

Emily Ratliff
IBM Linux Technology Center
emilyr@us.ibm.com

Abstract

As far as we know, no Open Source program has been certified for security—until now. Although some people believed that it was not possible for an Open Source program to receive a security certification, we have proven otherwise by obtaining a Common Criteria security certification for SuSE SLES 8 SP3. With the increasing use of Open Source in general and Linux in particular within government and commercial environments, security of Open Source products is of increasing importance and as a result the demand for the security evaluation of Linux is evident. It is also generally believed that security certifications are time consuming and can take years to accomplish. We were able to obtain the Common Criteria certification of Linux in a few months. The presentation will cover our experience and the technical challenges associated with this Linux evaluation. In particular, we will discuss the enhancements we made to SLES 8 SP3 including the Linux kernel to support CAPP audit requirements. In addition the business advantages of the evaluation for Open Source software will be covered.

1 Introduction

In promoting Linux to IBM's enterprise and government customers, the requirement for Common Criteria certification emerged as a barrier to entry. All of Linux's commercial competitors have the required level of certification. As Linux continues to be adopted by the enterprise market, many customers, especially those from the government sector, have raised concerns regarding Linux security and questioned whether Linux was capable of achieving certification. These customers view security certification as table stakes for proving a minimal level of operating system security. In order to increase Linux adoption by these customers, certification is required. The expense of achieving certification makes certification unobtainable by community projects without corporate or government sponsorship. For these reasons, and after a careful analysis, IBM decided to sponsor a Common Criteria (CC) security certification for Linux. SUSE agreed to partner with IBM to evaluate SUSE LINUX Enterprise Server 8 (SLES 8).

In this paper, we will begin with a brief overview of the Common Criteria standard. We will then describe our approach and expe-

rience during this certification effort. We will describe in detail the additional functionality that was needed in kernel and user space to fulfill the requirements of the certification. We will also describe the level of documentation and test needed to obtain the certification.

Throughout the paper, we use the pronoun 'we.' By 'we,' we mean individuals or subteams from the large team of people who contributed time and effort in achieving this evaluation, including:

- IBM, the evaluation sponsor
- SUSE, the developer
- atsec information security GmbH, the evaluator
- BSI, the German agency for information security, the evaluation body

2 Common Criteria Overview

Common Criteria (CC) is documented in the ISO standard 15408 for the security analysis of IT products. The governments of 18 nations have officially adopted the Common Criteria, including the United States, Canada, Germany, France, and the UK. The U.S. government has required a Common Criteria evaluation for all IT-products used for the processing of security-critical data since July 1, 2002.[1]

Common Criteria splits the requirements into two sets: functional and assurance. Functional requirements describe the security attributes of the product under evaluation. Assurance requirements describe the activities that must take place to increase the evaluator's confidence that the security attributes are present, effective, and are designed and implemented correctly. Examples of assurance activities include documentation of the developer's search for vulnerabilities and testing.

[1]The requirement is codified by NSTISSP No. 11.

2.1 Functional Requirements

The functional requirements desired by the customer are described in the Protection Profile (PP). Protection Profiles are targeted at specific types of systems. For example, there are unique protection profiles for operating systems, firewalls, databases and other complex or security sensitive products. Protection Profiles are often created by the product developer, standards bodies, or government agencies, rather than by the customer. To be officially recognized, the Protection Profile must itself be evaluated. Protection Profiles are intended to be reusable and thus typically define standard sets of security attributes that can be used to compare different implementations of a product type. The name of the Protection Profile is therefore often used as shorthand to describe the functional level of the evaluation.

The product being evaluated is known as the Target of Evaluation (TOE). The security policy used by the TOE is known as the TOE Security Policy (TSP) and the functionality that enforces the TSP is known as the TOE Security Functions (TSF). The TSP may be enforced by software, hardware or firmware, but no matter what the enforcement mechanism is, the enforcement functionality is included in the TSF. The TOE does not exist in a vacuum; external forces that act on the TOE are known as the TOE (security) environment. The TOE environment may consist of elements such as non-privileged processes running in an operating system and the network to which a system is attached. The main purpose of an evaluation is to determine whether or not the TSP is correctly enforced.

2.2 Assurance Requirements

Evaluated Assurance Levels (EALs) are defined on a scale of increasingly rigorous development methodologies. The Common Criteria defines multiple classes of assurance components with multiple levels of difficulty for each component. The assurance levels are then composed from these components. These components include items such as level of documentation and testing. The assurance components used for this evaluation are described in more detail in the EAL3 Overview Section. Each higher assurance level requires more proof that security was a fundamental element of the development process; therefore, each higher level is more difficult to achieve than the previous level. There are seven ordered EALs[2]:

- EAL1 – Functionally Tested
- EAL2 – Structurally Tested
- EAL3 – Methodically tested and checked
- EAL4 – Methodically designed, tested and reviewed
- EAL5 – Semiformally designed and tested
- EAL6 – Semiformally verified, designed and tested
- EAL7 – Formally verified, designed and tested

EAL1 is the entry level assurance level. EAL4 is the highest assurance level that any product is expected to be able to achieve without significant expense and rework if it had not been specifically developed with Common Criteria evaluation in mind.

[2]Common Criteria Part 3 available from `http://csrc.nist.gov/cc/Documents/CC%20v2.1%20-%20HTML/CCCOVER.HTM`

2.3 Evaluation Approach

When the developer has decided on a Target of Evaluation and a Protection Profile, the first step towards evaluation is writing a Security Target (ST) which describes the security objectives of the TOE and how they meet the security requirements defined in the chosen PP. It is possible for an ST to claim conformance to multiple PPs or no PP at all. The claims in the security target determine the scope of the evaluation. Every facet of the evaluation is directly impacted by what is claimed in the Security Target. After the evaluation is completed, the Security Target is always made available for customer scrutiny so that the customer can understand exactly what was evaluated.

3 Description of the Evaluated TOE

Our target of evaluation (TOE) was the SUSE LINUX Enterprise Server 8 operating system with Service Pack 3 and the certification-sles-eal3.rpm package.

The SLES evaluation covers a distributed, but isolated, network of IBM® xSeries®, pSeries®, iSeries®, and zSeries® servers running the evaluated version of SLES. The hardware platforms selected for the evaluation consisted of commercially available machines from across the IBM product line.

The TOE Security Functions (TSF) consist of Linux kernel functions plus some trusted processes. These functions enforce the security policy as defined in the Security Target. The TOE includes standard networking applications, such as ftp, ssh, ssl, and xinetd. System administration tools include standard admin commands. Yast2 and several yast2 modules were also included in the package list that formed the TOE. The X Window System was

not included in the evaluated configuration.

The hardware and the system firmware are not considered to be part of the TOE but rather are a part of the TOE environment. The TOE environment also includes applications that are not evaluated, but are used as unprivileged tools to access public system services. For example, an HTTP server using a port above 1024 (e.g., on port 8080) can be used as a normal application running without root privileges on top of the TOE. The Security Guide provides guidance on how to set up a http server on the TOE without violating the evaluated configuration.

4 ST Description

The Security Target specifies that the evaluation covers the Controlled Access Protection Profile (CAPP) functionality at the EAL3 augmented assurance level.[3] The primary security features and assurance documentation are described below, along with how the requirements were satisfied. The key features are supported by domain separation and reference mediation, which ensure that the features are always invoked and cannot be bypassed. Most of the security and assurance features are included in the vanilla kernel (e.g., object reuse) or are standard to most Linux distributions (e.g., PAM, OpenSSH, OpenSSL) and were thus already present in SLES 8. A few, most notably audit, had to be added for the evaluation.

4.1 EAL3 Overview

EAL3 provides assurance by an analysis of the security functions, using its functional and interface specifications, guidance documentation, and the high-level design of the TOE to understand the security behavior. The EAL3 assurance requirements fall into the following seven categories:

- Configuration Management
- Delivery and Operations
- Development
- Guidance Documents
- Life Cycle Support
- Security Testing
- Vulnerability Assessment.

Many of the documents created to support the assurance requirements can be reviewed at `http://oss.software.ibm.com/linux/pubs/?topic_id=5`

4.1.1 Configuration Management

The Configuration Management assurance class specifies the means for establishing that the integrity of the TOE is preserved during development. The Configuration Management process must provide a mechanism for tracking changes and ensuring that all the changes are authorized.

Configuration management procedures within SUSE are highly automated using a process supported by the AutoBuild tool. Source code, generated binaries, documentation, test plan, test cases and test results are maintained under configuration management. Because of this, SUSE already exceeded the requirements for this evaluation, so we just had to document existing procedures to fulfill this requirement.

This assurance requirement is the one that was commonly expected to be the source of difficulty in achieving certification of code developed via the open source methodology. The

[3] The augmentation is the flaw remediation procedure.

key to meeting this assurance requirement is that every line of new code that comes into the SUSE AutoBuild environment is assigned to an owner within SUSE who becomes responsible for its integrity.

4.1.2 Delivery and Operations

The Delivery and Operations class provides requirements for the assurance that the TOE is not corrupted between the time the developer releases it and the customer fires it up.

SLES is delivered on CD/DVD in shrink-wrapped package to the customer. SUSE verifies the integrity of the production CDs and DVDs by checking a production sample. Service Pack 3, the certification-sles-eal3.rpm package, as well as other packages that contain fixes must be downloaded from the SUSE maintenance Web site. Because those packages are digitally signed, the user is both able to and required to verify the integrity and authenticity of those packages. Guidance for installation and system configuration is provided in the Security Guide.

Again, existing SUSE processes met the requirements for the EAL3 assurance level, so documenting existing procedures was sufficient for this evaluation.

4.1.3 Development

The Development class encompasses requirements for documenting the TSF at various levels of abstraction, from the functional interface to the implementation representation. For EAL3, we needed a functional specification and a high-level design. In addition, the correspondence between the security functionality, the functional specification, and the high level design had to be documented.

The functional specification for SLES consists of the man pages that describe the system calls, the trusted commands, and a description of the security-relevant configuration files. A spreadsheet tracks all system calls, trusted commands, and security-relevant configuration files with a mapping (correspondence) to their description in the high-level design and man page(s). The high-level design of the security functions of SLES provides an overview of the implementation of the security functions within the subsystems of SLES, and points to other existing documents for further details where appropriate.

To fulfill this requirement, the functional specification spreadsheet, correspondence, and high-level design were written. Additionally, several new man pages were created for undocumented system calls, PAM modules and utilities, and many man pages required minor corrections.

4.1.4 Guidance Documents

The Guidance Documents class provides the requirements for user and administrator guidance documentation. A security guide is also necessary to fulfill the requirements of this class at EAL3.

SLES 8 already shipped with User and Administrator Guides. The Security Guide and a special README file were created that contain the specifics for the secure administration and usage of the evaluated configuration. The Security Guide explicitly documents setting up and maintaining the system in an evaluated configuration.

4.1.5 Life Cycle Support

The Life Cycle Support assurance requirement includes requirements for processes that deal with vulnerabilities found after release of the product, as well as the physical security of the developer's lab.

The SUSE security procedures are defined and described in documents in the SUSE intranet. The defect handling procedure SUSE has in place for the development of SLES requires the description of the defect with its effects, security implications, fixes and required verification steps.

Again, existing (and previously planned updates to) SUSE procedures met the requirements for this class and were merely required to be documented to fulfill the assurance requirements.

4.1.6 Security Testing

The emphasis of the Security Testing class is on the confirmation that the TSF operates according to its specification. This testing provides assurance that the TOE satisfies the security functionality requirements. Coverage (completeness) and depth (level of detail) are separated for flexibility.

A detailed test plan was produced to test the functions of SLES on each evaluated platform. The test plan includes an analysis of the test coverage, an analysis of the functional interfaces tested, and an analysis of the testing against the high level design. Test coverage of internal interfaces was defined and described in the test plan documents and the test case descriptions. The tests were executed on every platform. The test results are documented so that the tests can be repeated and the results independently confirmed.

Although, SUSE has an excellent test infrastructure for regression testing already in place, additional tests were required to test new functionality, such as audit, and ensure coverage of security relevant events. The Linux Test Project provided an excellent base for the test suite needed for EAL3. It already contained almost all of the necessary test cases for every system call. In some cases, we had to add tests of expected failure cases to ensure that the security was being correctly enforced. We added some test cases for security-relevant programs, such as su, cron, at, and ssh. We created tests to ensure that the system was configured in the evaluated manner. We also created many tests for correct ACL behavior. Many of the system call and security-relevant program test cases were created during the course of the EAL2 evaluation and then reused during the EAL3 evaluation. The largest class of new test cases for EAL3 was tests of the new audit system. Testing the audit subsystem required showing that all security-relevant system calls are logged correctly, all trusted programs (including PAM) correctly logged security-relevant events, the audit userspace tools contained correct functionality, and that audit exhibits Controlled Access Protection File (CAPP)-compliant behavior during threshold and failure events (for example, low disk space). Gcov was used to show test coverage of the kernel internal interfaces. Writing, documenting, and running these test cases on all of the evaluated platforms was a significant portion of the evaluation effort.

4.1.7 Vulnerability Assessment

The Vulnerability Assessment class defines requirements for evidence that the developer looked for vulnerabilities that might arise during development and use of the TOE.

Our search for vulnerabilities was documented

in the Vulnerability Assessment document. This assessment included TOE misuse analysis and a password strength of function analysis. The analysis also describes the approach used to identify vulnerabilities of SLES and the results of the findings.

The Vulnerability Assessment was performed and written as part of this evaluation.

4.2 CAPP Overview

The Controlled Access Protection Profile (CAPP) is based on the C2 class of the "Department of Defense Trusted Computer Systems Evaluation Criteria" (DoD 5200.28 – STD) colloquially known as the "Orange Book." CAPP requires that the operating system implement the Discretionary Access Control (DAC) security policy. DAC allows the information owner to control who is allowed to access the information.

The CAPP functional requirements fall in the following five broad categories:

- Identification and Authentication
- User Data Protection
- Security Management
- Protection of the TSF
- Security Audit.

4.2.1 Identification and Authentication

Identification and Authentication include the functionality required to uniquely identify the user.

SLES provides identification and authentication using pluggable authentication modules (PAM) based upon user passwords. Other authentication methods (e.g., Kerberos authentication, token based authentication) that are supported by SLES as pluggable authentication modules are not part of the evaluated configuration. PAM was configured to ensure medium password strength, to ensure password quality to limit the use of the su command, and to restrict root login to specific terminals.

Meeting the CAPP requirements for Identification and Authentication involved changing the default PAM configuration for SLES 8. The new configuration is documented by the Security Guide.

4.2.2 User Data Protection

User Data Protection specifies the functionality that protects data from unauthorized access and modification—the enforcement of the Discretionary Access Control policy. In addition, deleted information must not be accessible and newly created objects must not contain residual information.

The Discretionary Access Control policy restricts access to file system objects based on Access Control Lists (ACLs) that include the standard UNIX® permissions for user, group, and others. Access control mechanisms also protect IPC objects from unauthorized access.

The evaluated configuration used the ACL support in the ext3 file system. The vanilla kernel already clears file system, memory and IPC objects before they can be reused by a process belonging to a different user. Thus, the User Data Protection functionality requirements were already being met by SLES 8.

4.2.3 Security Management

The Security Management class specifies how security attributes, security data and security functions are managed by the TOE. Security Management includes management of groups and roles, separation of capability, and management of audit data.

Management of the security critical parameters of the TOE is performed by administrative users. Commands that require root privileges, such as useradd and groupdel, are used for system management. Security parameters are stored in specific files that are protected by the access control mechanisms of the TOE against unauthorized access by non-administrative users.

Other than the audit data management commands (which are described in the Security Audit section below) all security management functionality was provided by standard functionality already included in SLES 8.

4.2.4 TSF Protection

Protection of the TSF specifies the requirements for maintaining the integrity of the TSF and its data, particularly the protection of configuration data. The TSF will need to perform the appropriate testing to demonstrate the security assumptions about the underlying abstract machine upon which the TSF relies. In addition, the TSF must be demonstrated to be complete and tamperproof.

While in operation, the kernel software and data are protected by the hardware memory protection mechanisms. The memory and process management components of the kernel ensure that user processes cannot access kernel storage or storage belonging to other processes. Non-kernel TSF software and data are protected by DAC and process isolation mechanisms. In the evaluated configuration, the root user owns the directories and files that define the TSF configuration. Files and directories containing internal TSF data (e.g., configuration files, batch job queues) are also protected by DAC permissions.

The TOE and the hardware and firmware components are required to be physically protected from unauthorized access. The system kernel mediates all access to the hardware mechanisms themselves, other than program visible CPU instruction functions.

4.2.5 Abstract Machine Test Utility (AMTU)

To completely fulfill the TSF Protection requirement, we had to produce a tool to test the underlying abstract machine: "The TSF shall run a suite of tests [selection: during initial start-up, periodically during normal operation, or at the request of an authorized administrator] to demonstrate the correct operation of the security assumptions provided by the abstract machine that underlies the TSF."[4] This requirement is sometimes fulfilled by Power-On Self Test (POST) procedures, but given the diversity of platforms that were included in the certification, we decided that a userspace administrative tool, AMTU, would be the simpler approach. AMTU can be run by an administrator at any time and ensures that the hardware enforced security protection is still in effect. To this end, the tool runs a simple check for memory errors, checks for enforcement of memory separation, checks the correct operation of network and disk I/O controllers, and verifies

[4] Controlled Access Protection Profile available from `http://www.radium.csc.mil/tpep/library/protection_profiles/CAPP-1.d.pdf`

that privileged instructions cannot be executed when the hardware is in user mode.

The source code for AMTU is available at `http://www-124.ibm.com/developerworks/projects/amtu`.

4.2.6 Security Audit

Auditing systems collect information about events related to security-relevant activities. Security-relevant activities are defined as those events that are governed by the security policy. The resulting audit records can be examined to determine which security-relevant activity took place and which user is responsible for them. No fully CAPP-compliant audit subsystem was available for Linux, so we implemented this feature to achieve the certification. The audit subsystem developed for the evaluation is called Linux Audit System or LAuS.[5]

LAuS Conceptual Overview The Linux Audit System (LAuS) consists of three primary components: a kernel module responsible for intercepting system calls and recording relevant events, an audit daemon (auditd) that retrieves the records generated by the kernel and writes them to disk, and a number of command line utilities for displaying, querying and archiving the audit trail. See Figure 1.

The interface between kernel and user space uses a character device named /dev/audit. The audit daemon uses I/O Control operations (ioctls) on this device to configure the audit module, and it retrieves audit records from it using the read() system call.

To improve performance, filtering of audit events is performed at the kernel level. Unlike some existing implementations, the audit daemon does not perform any filtering itself. This eliminates a serious performance bottleneck.

The set of filter primitives provided by LAuS is fairly rich, and primitives can be combined using boolean operations. For instance, it is possible to audit open(2) calls made by a setuid application, while ignoring all other open(2) calls, or to restrict auditing to certain files. The eal3-certification RPM contains the evaluated audit configuration files.

At startup, auditd reads its configuration and the set of filter expressions from one or more files, loads the filters to the kernel, and starts auditing.

Auditd then proceeds to listen for audit events generated by the kernel. It retrieves and writes all records directly to disk. Because of the CAPP requirement that audit records must never be lost, this process is more complex than it might seem. auditd constantly monitors disk usage and can be configured to respond in different ways if free disk space drops below certain thresholds. Possible reactions to low disk space include notifying the administrator, suspending all audited processes, or shutting down the system immediately. Both the thresholds and auditd's reactions can be configured by the administrator.

LAuS supports different output modes to provide a flexible way to configure data collection. The simplest approach simply writes the audit trail to a single file in append mode, similar to the way syslogd works.

In "bin mode," audit writes data to a number of fixed sized files (bins), switches to the next file when the current one fills up, and invokes an external command to archive the full bin. Finally, there is a so-called "stream mode" that lets you pipe the audit trail directly into an ex-

[5] The LAuS Design Document is available at `ftp://ftp.suse.com/pub/projects/security/laus/doc/LAuS-Design.pdf`

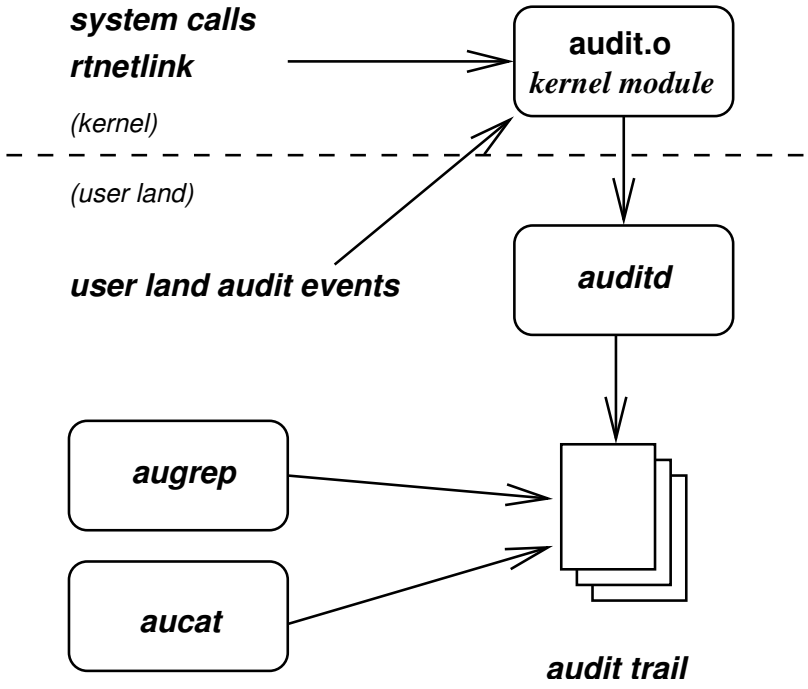


Figure 1: LAuS Conceptual Overview

ternal command; this can be useful if you want to forward the trail to a central storage server.

Auditing can be enabled globally or on a per-process basis; in the latter case, all the child processes are audited as well. The only processes always exempt from auditing are init and the audit daemon itself.

User land utilities were created to parse and read the audit log files. aucat 'cats' the file, transforming all of the audit records to a human readable format. augrep 'greps' the audit records and allows the administrator to selectively review the records. augrep allows the administrator to select audit records based on type, time (range), user, syscall, program (by name or PID) that generated the event, or any combination of these attributes.

Even though the user land utilities are far from trivial, the kernel portion of LAuS proved far more complex; in fact, the kernel portion of the LAuS is a lot more complex than we had initially anticipated. The rest of this section deals with the questions surrounding the audit kernel module.

Additional Design Constraints In addition to making our audit implementation compliant with the CAPP requirements, we had to deal with several constraints which are worth noting.

One was to minimize performance overhead. In the case where auditing was compiled into the kernel, but not configured by the administrator, we wanted it to have zero performance impact if possible. Our kernel developers spent quite a lot of time on additional kernel tuning, making sure the kernel performed and scaled well. Breaking this was not an option.

We also wanted to have a performance overhead as small as possible for the audited case, even though this wasn't as high on our agenda. This definitely took second place to correctness

and CAPP compliance.

A third objective was entirely non-technical, but played a crucial role in choosing an approach to intercepting system calls. We wanted our modifications to the core kernel as small as possible; most of the code should be inside a loadable module.

The rationale behind this was to minimize the probability of introducing bugs (except, of course, bugs in the audit code itself), and to ease maintenance.

The latter point was a fairly important item in the context of the SLES 8 kernel, which includes well above 1,500 additional patches applied on top of the mainline kernel. Updating SLES 8 to a new mainline kernel version was a bit of an adventure, so we wanted to avoid adding audit patches to the kernel that changed lots of files all over the place.

Where to intercept system calls There are basically three ways to intercept system calls on a 2.4 Linux kernel.

The first approach is to create wrappers for those system calls you wish to track, and replace the original function pointers in the system call table with those of the new wrapper functions. This sounds simple enough, and would also satisfy our requirements for zero performance impact in the non-audit case, and a minimally intrusive kernel patch. Unfortunately, this approach doesn't work on all architectures.

The next approach is to add hooks to all kernel functions that must be audited. The major drawback to this approach is that the kernel patch would touch lots of files in the kernel, which we wanted to avoid.

The third approach, which we chose, was to hook into the code path that intercepts system calls for ptrace. This intercept happens very early in the platform-specific assembler code, before the system-call function itself is invoked. The assembly code retrieves a set of flags associated with the calling process, and checks the `PT_TRACESYSbit`. If that bit is set, it jumps to a separate code branch dealing with ptracing. The same test is performed when returning from a system call.

In our audit implementation, we simply defined an additional task flag named PT_AUDITED, and extended the bit test in the system-call entry and exit code to test for both bits at the same time. This gave us system-call intercept with zero performance overhead in the normal, non-audited code path.

See Figure 2 for a picture showing the flow of control when auditing a system call.

Defining which system calls to audit By far, the most important part of auditing concerns system calls. As mentioned above, CAPP requires auditing all security-relevant system calls. We needed to determine which system calls are security relevant and which aren't.

The obvious ones are those that change the state of a process, the file system, or other system resources. These includes calls such as setuid, open, close, and setting the system's host name or clock. An audit implementation also needs to cover less obvious operations, such as binding a socket to a port, attaching shared memory segments, and performing ioctls.

Most system calls are fairly straightforward to handle, and much of the information on system calls and the arguments they take can be encoded statically in tables. Some calls, such as msgrcv, which comes in two versions on the i386 platform for historical reasons, were difficult to handle. 64-bit platforms usually require an additional table as they support a 32-bit sys-

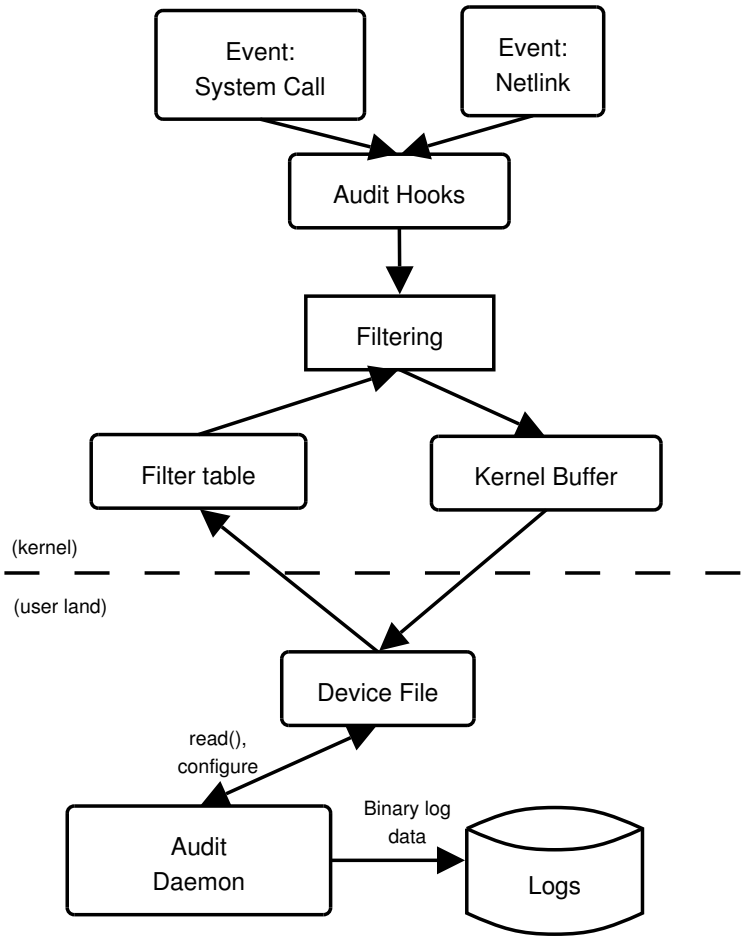


Figure 2: Auditing a System Call

tem call interface in addition to their native 64-bit interface.

However, some of the operations we wanted to audit proved a little more elusive; these were the ioctl system call and network configuration changes.

Auditing ioctls The ioctl system call is the dirty little back alley of UNIX-like operating systems. If a driver for a piece of hardware, a network protocol or a file system needs to expose some driver-specific mechanism or tunable parameters to user-space applications, the most common method for doing so is to define one or more ioctls.

The ioctl system call takes an open file descriptor, which must refer to something controlled by the driver (for example, a terminal, a device file, or a socket), an integer number specifying the request, and an opaque pointer to some chunk of memory. Exactly what to do with this piece of memory depends on the driver that is being talked to, and the integer passed as the request ID.

Unfortunately for us, the Linux kernel supports well over a thousand ioctls, and while many of them are rather obscure, they do change the system's state and are thus subject to auditing. It is obvious that compiling and maintaining a list of 1000 ioctls and their arguments was not an option.

Most ioctl numbers nowadays encode sufficient information on whether the operation passes data into the kernel, retrieves data, or both, and the size of the argument. Therefore, writing audit records for these is straightforward. However, there is still a fair number of ioctls that do not follow this convention.

What is far worse is that ioctl numbers are not unique—frequent users of strace will probably know that the TCGETS ioctl uses the same number as some obscure sound card operation. But this is not the only conflict.

However, the most difficult aspect of auditing ioctls is that it isn't sufficient to simply generate audit records for these calls; you must also be able to display the information of each audit record to the user.

The way we solved this problem was entirely non-technical. Our target of evaluation clearly stated that the super user account remains special. The super user can do everything, from loading unsupported modules not covered by the certification, to disabling the audit subsystem altogether.

Instead of trying to handle each and every ioctl in the audit module, we went through all ioctls available in our to-be-certified configuration and categorized them. The ones we needed to audit were those that were security relevant in some way, but did not require administrative privilege; the list we came up with this way was much smaller than the original list and more manageable.

Auditing network configuration Another aspect that proved to be a challenge was tracking network configuration changes, because only a fraction of those are done through ioctl calls. Most network configuration changes are performed by passing data to a netlink socket. These changes can be audited by simply recording all sendmsg and recvmsg calls on netlink sockets, but that is far from optimal. On the one hand, a send or receive operation on a netlink socket can include more than one request. On the other hand, the outcome of a netlink call is not returned through the system-call return value, but in a separate netlink message generated by the kernel and queued to that socket. Simply logging the raw netlink data sent and received would require quite a bit of built-in intelligence on the part of the user land applications that are supposed to display this data.

So instead, we decided to tap into the netlink layer directly, where a data blob sent to a netlink socket is broken up into separate requests, and each request is processed in return. This allowed us to record each netlink request separately, and place the outcome of the operation into the same audit record as the original request.

The Login User ID An aspect of auditing that is worth mentioning is how to deal with the CAPP requirement that each record identifies the user performing the operation.

The obvious solution (which would be to use the real user ID associated with the calling process) is not sufficient, as setuid applications can change these IDs at will. Tracking all uid changes, and thereby allowing the audit utilities to piece together the original user ID from this mosaic, is not practical either. It is not uncommon for some processes on a dedicated server to run for hundreds of days, so the amount of data to look at would be prohibitive.

The only viable solution in this case is to attach a "login uid" to each process. The login uid remained constant across all other changes of real, effective, and saved user IDs, and was inherited by all child processes. Of course, this required changes to PAM so that this uid would

be set on login.

Nightmare on Audit Street There is one major problem with the approach to system call intercept that we chose and which in hindsight made it a less than optimal choice. The problem is that our approach requires data to be copied twice. To understand why this is a problem, let's look at the open(2) system call, which takes a path name as an argument. This path name is passed into the kernel function as a pointer to a string (essentially a chunk of memory) in the address space of the user process. In order to operate on this string, the kernel must copy it to a buffer in the kernel address space, possibly paging in memory as it goes.

When entering the kernel, the audit module retrieves the path name from user space, and decides whether to create an audit record for this call. If it does decide to create an audit record, it sets up an audit record containing the system call number and a copy of all arguments, including the path.

The system call proceeds as normal, and the kernel function `sys_open` retrieves the path name from user space a second time, and carries out the requested operation based on this data.

The problem is that the memory in user space may have changed in the meantime, so that the record written by the audit module does not correspond to what was actually performed by the operating system.

There are several ways this can happen. Of course, the calling process itself cannot modify this memory, as it is currently executing the system call. However, memory can be shared between processes in a variety of ways. Threads can share the entire address space; processes can attach to the same shared memory segment; memory can be mapped from a file, which can be mapped by other processes as well.

Such an attack on the audit module is not really practical, because proper timing is probably quite hard, and any attempt to perform this attack would most likely leave a trail in the audit file. But even the theoretical possibility of circumventing the audit subsystem is unacceptable in terms of CAPP compliance.

The cases described above can be detected and dealt with by the audit module. Dealing with these problems, however, incurs additional complexity and performance loss (especially in the case of multithreaded applications). Needless to say, the added complexity engendered a considerable number of bugs. For this reason, these additional checks can be turned off by the administrator. These checks are turned off in the evaluated configuration of audit and the associated risk, considered minimal, is documented in the Vulnerability Assessment.

SUSE Linux Server 9 SUSE Linux Server 9 will include an updated version of the LAuS kernel patches. In many respects, the updated LAuS module will work in the same way as the SLES 8 version did, with the major exception being the way system calls are intercepted.

When planning audit for SUSE Linux Server 9, we considered two options.

The first option was a solution we had already looked at for SLES 8 and abandoned, namely adding hooks to all system call functions relevant for CAPP. This is the approach we chose for SUSE Linux Server 9, mainly in order to avoid having to jump through all those extra hoops in an attempt to prevent the race conditions described in the previous section. One pleasant side effect of this approach is that it also eliminated a lot of platform-specific code.

The second option we considered was to add audit support as an LSM module, or extending an existing LSM module such as SELinux. The security framework in the 2.6 kernel goes a long way toward intercepting all security-relevant operations. Adding audit hooks in this place is appealing, because it would mean no additional performance cost (the security hooks do come with a certain performance penalty already) and no additional maintenance problems (because the audit patch would not have to touch multiple kernel files).

The main reason why we did not choose this approach was that the security hooks provide a more abstracted view than we had chosen for LAuS in SLES 8. Security hooks do not correspond directly to system calls, but rather represent the security check necessary to validate whether an operation is permitted. There is a fine distinction between "user X attempted to perform operation Y, and the outcome was Z" and "user X attempted an operation on object A that caused us to perform security check B, and the outcome of this check was C." In particular, we are neither aware of the operation that triggered the security check, nor of its final outcome, because the operation can still fail even if security clearance is given.

Moreover, a single system call may require several security checks, such as renaming a file, where we need permission to remove the file from the source directory and permission to add it to the destination directory.

Changing LAuS to use the security hooks would have meant rewriting much more code than we wanted to, including the filtering code and much of the user-land applications. We also would have had to modify considerable parts of the documentation required for recertification.

Future Directions This is not to say that it is not possible to write an audit implementation leveraging some features of the LSM framework. In fact, we hope to have a common audit implementation in the mainstream kernel one day. It would greatly help acceptance by the kernel community if that solution did not add another set of hooks into many performance-critical functions.

5 Evaluation Roadmap

Performing a security evaluation should never be a one-time accomplishment. To maintain the security level achieved, the security certificate must be maintained. In the case of Linux, the intent is to go a step further: to increase, step-by-step, the assurance level and the security functionality until Linux achieves the highest assurance level of any commercial operating system product, while offering the richest set of security functions. The first step was accomplished in July 2003, when we obtained an EAL2+ evaluation for SUSE SLES 8 as-is. This paper documented the results of the second step, where we obtained a CAPP/EAL3+ certification for SLES 8 SP3 in January 2004. Linux, like its commercial competitors, has now been successfully evaluated for compliance with the requirements of the US government-defined CAPP. As a further step, Linux is currently in evaluation for compliance with the requirements of the EAL4 level. This includes the development of a low-level design of the Linux kernel (the evaluation will be based on the 2.6 version of the kernel) as well as a more sophisticated vulnerability analysis being performed. The experience gathered in the EAL2 and the EAL3 evaluations have given us the confidence that compliance with EAL4 can be achieved in fairly short order.

6 Value of Certification

The value of certification can be considered from two perspectives: business and technical.

In order for Linux to be adopted by the commercial and government markets, it faces stiff competition from entrenched incumbents. All of the incumbent products have been evaluated using the Common Criteria. In addition, the U.S government instituted a national security community policy against procuring unevaluated products (NSTISSP No. 11). There is a high probability that other governments and commercial entities will do the same.

While there is much skepticism surrounding the technical value of certification, certification is very much in line with the "many eyes" philosophy. For commercial products, certification is often the only time the code is reviewed by people outside of the development team. The assurance requirements of Common Criteria add to the number of trained eyes looking at the design and source of a project using defined and rigorous procedures. During the course of the EAL3 evaluation, we found and fixed several bugs, created lots of documentation, and shipped an integrated CAPP-compliant audit system. We noticed an anomaly on the iSeries platform while testing the Abstract Machine Testing Utility. Analysis of this anomaly by the ppc64 development team led to the discovery of a memory separation bug on the iSeries platform.[6] Many PAM module bugs were identified and fixed in SLES 8, including a double free bug in `pam_pwcheck`.[7] Man pages were created for several undocumented system calls, PAM modules and admin utilities, including `io_setup`, `readahead`, `set_thread_area`, `pam_wheel`, `pam_securetty`, and others.

[6] Paul Mackerras fix to "Make kernel RAM user-inaccessible on iSeries" `http://www.kernel.org/diff/diffview.cgi?file=/pub/linux/kernel/v2.4/patch-2.4.23.bz2;z=290`

[7] `http://www.atsec.com/01/index.php?id=03-0002-01&news=28` Patches are available from klaus@atsec.com

7 Conclusion

Achieving the EAL2 and EAL3/CAPP certifications was significant because it proved that Linux is indeed certifiable. The certification opened the market up to include U.S. government agencies and commercial entities that require certification. Future evaluations of Linux distributions can be made easier by Linux adoption of a CAPP-compliant audit subsystem.

8 Legal Statement

Improving Linux resource control using CKRM

Shailabh Nagar
IBM T.J. Watson Research Center
nagar@watson.ibm.com

Rik van Riel
Red Hat, Inc.
riel@redhat.com

Hubertus Franke
IBM T.J. Watson Research Center
frankeh@watson.ibm.com

Chandra Seetharaman
IBM Linux Technology Center
chandra.sekharan@us.ibm.com

Vivek Kashyap
IBM Linux Technology Center
vivk@us.ibm.com

Haoqiang Zheng
Columbia University
hzheng@cs.columbia.edu

Abstract

One of the next challenges faced in Linux kernel development is providing support for workload management. Workloads with diverse and dynamically changing resource demands are being consolidated on larger symmetric multiprocessors. At the same time, it is desirable to reduce the complexity and manual involvement in workload management. We argue that the goal-oriented workload managers that can satisfy these conflicting objectives require the Linux kernel to provide class-based differentiated service for all the resources that it manages. We discuss an extensible framework for class-based kernel resource management (CKRM) that provides policy-driven classification and differentiated service of CPU, memory, I/O and network bandwidth. The paper describes the design and implementation of the framework in the Linux 2.6 kernel. It shows how CKRM is useful in various scenarios including the desktop. It also presents preliminary performance evaluation results that demonstrate the viability of the approach.

1 Introduction

Workload management is an increasingly important requirement of modern enterprise computing systems. There are two trends driving the development of enterprise workload management middleware. One is the consolidation of multiple workloads onto large symmetric multiprocessors (SMPs) and mainframes. Their diverse and dynamic resource demands require workload managers (WLMs) to provide efficient differentiated service at finer time scales to maintain high utilization of expensive hardware. The second trend is the move towards specification of workload performance in terms of the business importance of the workload rather than in terms of low-level system resource usage. This has led to the increasing use of goal-oriented workload managers, described shortly, which are more tightly integrated into the business processes of an enterprise.

Traditional system administration tools have been built with two layers. The lower, OS specific layer deals with modifying and monitoring operating system parameters. The upper

layer(s) provide an OS independent API, generally through a graphical user interface, allowing a multi-tier or clustered system to be managed through a unified API despite containing heterogenous operating systems. While such tools provide a convenient administrative interface to heterogeneous operating systems they do little to address the complexity of managing workloads that span multiple tiers. The burden of translating business goals into workload resource requirements and the latter into OS specific tuning parameters remains on the system administrators. Increasing workload consolidation only adds more complexity to an already onerous problem.

As described in [7], the first stage in improving workload management are *entitlement-based* workload managers (WLMs) such as [9, 5, 11] which enforce entitlements or shares on resources consumed by groups of processes, users, etc. This allows the more important groupings to see improved response times and higher bandwidth due to preferential access to the server hardware. As importantly, it allows expensive SMP servers to have higher utilizations since system administrators can afford to load them more without fear of penalizing the important groupings.

However, the complexity of determining the right entitlements (henceforth called shares) for a grouping remains on the human system administrator. Not only does s/he need to map the importance of a workload to its entitlements, s/he also needs to adjust these shares dynamically when the demand and/or importance of *any* workload changes. Such dynamic share changes have become increasingly difficult to compute in a timely manner when manual involvement is part of the adaptive feedback loop.

To address the complexity of share specifications, *goal-oriented workload managers* have been developed [1, 10] which allow a system to be more self-managed. Such WLMs allow the human system administrator to specify high level performance objectives in the form of policies, closely aligned with the business importance of the workload. The WLM middleware then uses adaptive feedback control over OS tuning parameters to realize the given objectives.

In mainstream operating systems, including Linux, the control of key resources such as memory, CPU time, disk I/O bandwidth and network bandwidth is typically strongly tied to processes, tasks and address spaces and are highly tuned to maximize system utilization. This introduces additional complexity to the WLM which needs to translate the QoS requirements into these low level per task requirements, tough typically QoS is enforced at work class level. Hence, in order to isolate the autonomic goal oriented layers of the system management from the intricacies of the operating system, we introduce the class concept into the operating system kernel and require the OS to provide differentiated service for all major resources at a class granularity defined by the WLM.

In this paper, we discuss a framework called class-based kernel resource management (CKRM) that implements this support under Linux. In CKRM, a class is defined as a dynamic grouping of OS objects of a particular type (classtype) and defined through policies provided by the WLM. Each class has an associated share of each of its resources. For instance, CKRM tasks classes provides resource management for four principal physical resources managed by the kernel namely CPU time, physical memory pages, disk I/O and bandwidth. Sockets classes provide inbound network bandwidth resource control. The Linux resource schedulers are modified to provide differentiated service at a class granu-

larity based on the assigned shares. The WLM can dynamically modify the composition of a class and its share in order to meet higher level business goals. We evaluate the performance of the CKRM using simple benchmarks that demonstrate the efficacy of its approach.

This work makes several contributions that distinguish it from previous related work such as resource containers [2] and cluster reserves [4]. First, it describes the design of a flexible kernel framework for class-based management that can be used to manage both physical and virtual resources (such as number of open files). The framework allows the various resource schedulers and classification engine to be developed and deployed independent of each other. Second, it shows how incremental modifications to existing Linux resource schedulers can make them provide differentiated service effectively at a class granularity. To our knowledge, this is the first open-source resource management package that attempts to provide control over all the major physical resources—i.e., CPU, memory, I/O, and network. Third, it provides a policy-driven classification engine that eases the development of new higher level WLMs and enables better coordination between multiple WLMs through policy exchange. Thirdly, through the resource class filesystem the WLM goals can be manipulated by normal users, making it useful on the desktop. Finally, it develops a tagging mechanism that allows server applications to participate in their resource management in conjunction with the WLM.

The rest of the paper is organized as follows. Section 2 gives an overview of CKRM and its core bits. Sections 3 briefly describes the classification engine. Section 4 presents the facilities provided by CKRM for monitoring. The inbound network controller, the first major controller ported to CKRM's new interface, is described in Section 5. Section 6 describes the filesystem interface which replaces the system call interface used in CKRM's earlier design presented in OLS 2003 [13]. Section 7 describes how CKRM might be used, both on a desktop system and on some server workloads. Section 8 concludes with directions for future work in the project.

2 Framework

A typical WLM defines a workload to be any system work with a distinct business goal. From a Linux operating system's viewpoint, a workload is a set of kernel tasks executing over some duration. Some of these tasks are dedicated to this workload. Other tasks, running server applications such as database or web servers, perform work for multiple workloads. Such tasks can be viewed as executing in phases with each phase dedicated to one workload. Server tasks can explicitly inform the WLM of its phase by setting an application tag. A WLM can also infer the phase by monitoring significant system events such as forks, execs, setuid, etc. and classifying the server task as best as possible.

In this scenario, a WLM translates a high level business goal of a workload (say response time) into system goals for the set of tasks executing the workload. The system goals are a set of delays seen by the workload in waiting for individual resources such as CPU ticks, memory pages, etc. The WLM monitors the business goals, possibly using application assistance, and the system usage of its resources. If the business goal is not being met, it identifies the system resource(s) which form a performance bottleneck for the workload and adjusts the workload's share of the resource appropriately. The CKRM framework enables a WLM to regulate workloads through a number of components, as shown in Fig. 1:

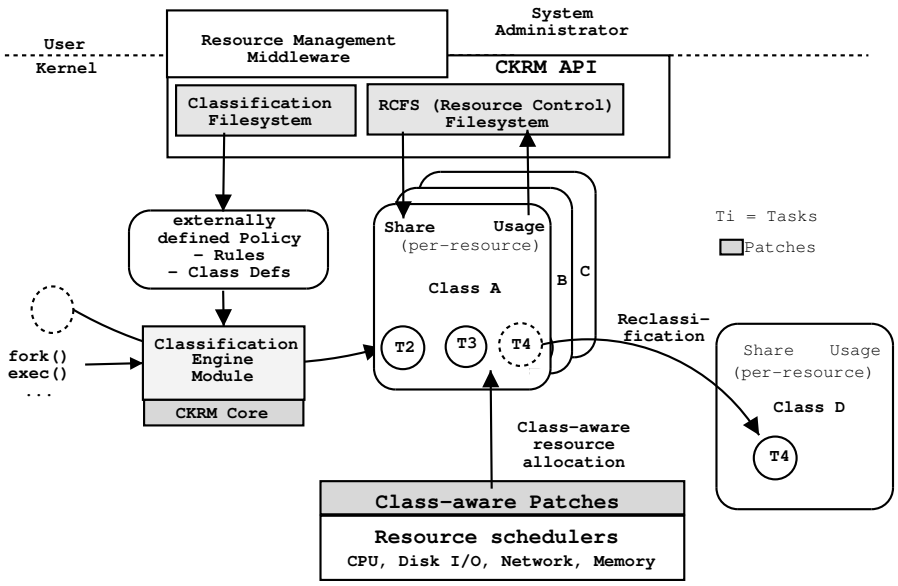


Figure 1: CKRM lifecycle

Core: The core defines the basic entities used by CKRM and serves as the link between all the other components. A class is a group of kernel objects with an associated set of constraints for resource controllers operating on those kernel objects—e.g., a class could consist of a group of tasks which have a joint share of cpu time and resident page frames. Each class has an associated classtype which identifies the kernel object being grouped. CKRM currently defines two classtypes called `task_class` and `socket_class` for grouping tasks and sockets. For brevity, the term taskclass and socketclass will be used to denote a class of classytpe `task_class` and `socket_class` respectively. Classtypes can be enabled selectively and independent of each other. A user not interested in network regulation could choose to disable `socket_classes`. Classes in CKRM are hierarchical. Children classes can be defined to subdivide the resources allocated to the parent.

Classification engine (CE): This optional components assists in the association of kernel objects to classes of its associated classtype. Each kernel object managed by CKRM is always associated with some class. If no classes are defined by the user, all objects belong to the default class for the classtype. At significant kernel events such as fork, exec, setuid, listen, when the attributes of a kernel object are changed, the Core queries the CE, if one is present, to get the class into which the object should be placed. CE's are free to use any logic to return the classification. CKRM provides a rule-based classification engine (RBCE) which allows privileged users to define rules which use attribute matching to return the class. RBCE is expected to meet the needs of most users though they can define their own CE's or choose not to have any and rely upon manual classification of each kernel object through CKRM's rcfs user interface (described later).

Resource Controllers: Each classtype has a set of associated resource controllers, typically one for each resource associated with the classtype—e.g., taskclasses have cpu, memory, and I/O controllers to regulate the cpu ticks, resident page frames and per-disk I/O bandwidth consumed by it while socketclasses have an accept queue controller to regulate the num-

ber of TCP connections accepted by member sockets. Resource requests by a kernel object in a class are regulated by the corresponding resource controller, if one exists and is enabled. The resource controllers are deployed independent of each other so a user interested only in controlling CPU time for taskclasses could choose to disable the memory and I/O controllers (as well as the socketclass classtype and all its resource controllers).

Resource Control File System (RCFS): It forms the main user-kernel interface for CKRM. Once RCFS is mounted, it provides a hierarchy of directories and files which can be manipulated using well-known file operations such as open, close, read, write, mkdir, rmdir and unlink. Directories of rcfs correspond to classes. User-kernel communication of commands and responses is done through reads/writes to virtual files in the directories. Writes to the virtual files trigger CKRM Core functions and responses are available through reads of the same virtual file.

The CKRM architecture outlined above achieves three major objectives:

- Efficient, class-based differentiation of resource allocation and monitoring for dynamic workloads: Regulate and monitor kernel resource allocation by classes which are defined by the privileged user and not only in terms of tasks. The differentiation should work in the face of relatively rapid changes in class membership and over roughly the same time intervals at which process-centric regulation currently works.

- Low overhead for non-users: Users disinterested in CKRM's functionality should see minimum overhead even if CKRM support is compiled into the kernel. Signs of user disinterest include omitting to mount rcfs or not defining any classes. Even for users, CKRM tries to keep overheads proportional to the features used.

- Flexibility and extensibility through minimization of cross-component dependencies: Classification engines should be independent of classtypes and optional, classtypes should be independent of each other and so should resource controllers, even within the same classtype. This goal is achieved through object-oriented interfaces between components. Minimizing dependencies allows kernel developers to selectively include components based on their perception of its utility, performance and stability. It also permits alternative versions of the components to be used depending on the target environment—e.g., embedded Linux distributions could have a different set of taskclass resource controllers (or even classtypes) than server-oriented distributions.

3 Classification

The Classification Engine (CE) is an optional component that enables CKRM to automatically classify kernel objects within the context of its classtype. Since the CE is optional and since we want to main flexibility in its implementation, functionality and deployment, it is supplied as a dynamically loadable module. The CE interacts with CKRM core as follows. The CKRM core defines a set of ckrm events that constitute a point during execution where a kernel object could potentially change its class. A classtype can register a callback at any of these events. As an example, the task class hooks the fork, exec, exit, setuid, setgid calls where as the socket class hooks the listen and accept calls. In these callbacks the classtypes typically invoke the optional CE to obtain a new class. If no CE is registered or the

CE does not determine a class, the object remains in its current class, otherwise the object is moved to the new class and the corresponding resource managers of that class's type are informed about the switch.

For every classtype the CE wants to provide automatic classification for, it registers a classification callback with the classtype and the set of events to which the callback is limited to. The task of CE is then to provide a target class for the kernel objects passed in the context of the classtype. For instance, task classes pass only the task, while socket classes pass the socket kernel object as well as the task object. Though the implementation of the classification engine is completely independent of CKRM, the CKRM project provides a default classification, called RBCE, that is based on classification rules. Rules consist of a set of rule terms and a target class. A rule term specifies one particular kernel object attribute, a comparision operator (=,<,>,!) and a value expression. To speed up the classification process we maintain state with tasks about which rules and rule terms have been examined for a particular task and only reexamine those terms that are indicated by the event. RBCE provides rules based on task parameters ((pid, gid, uid, executable) and socket information (IP info). The rules in conjunction with the defined classes constitute a site policy for workload managment and is dynamically changable (See user interface section) into the RBCE. Hence, this approach ensures the separation of policy and enforcement.

To facilitate the interaction with WLMs to provide event monitoring and tracing, the CE can also register a notification callback with any classtype, that is called when a kernel object is assigned to a new class. Similar so the classification callback, the notification callback can be limited to a set of ckrm events. This facility is utilized in resource monitoring, described next.

4 Monitoring

We now describe the monitoring infrastructure. Strictly speaking, the per-class monitoring components are part of CKRM while the per-process components are not. However, we shall describe them together as they both can be utilized by goal-based WLMs. Furthermore, they are bundled with the classification engine and utilize the CE's notification callback to obtain classification events. The monitoring infrastructure illustrated in Fig. 2 is based on the following design principles:

1. Event-driven: Every significant event in the kernel that affectsthe state of a task is recorded and reported back to the state-agent. The events of importance are aperiodic such as process fork, exit and reclassification as well as periodic events such as sampling. Commands sent by the state-agent are also treated as events by the kernel module.

2. Communication Channel: A single logical communication channel is maintained between the state-agent and the kernel module and is used for transferring all commands and data. Most of the data flow is from the kernel to user space in the form of records resulting from events.

3. Minimal Kernel State: The design minimizes the additional per-process state that needs to be maintained within the kernel. Most of the state needed for high level control purposes is kept within the state agent and updated through the records sent by the kernel.

The state-agent, which can also be integrated within a WLM, maintains state on each existing and exited task in the system and provides it to the WLM. Since the operating system

does not retain the state of exited processes, the stateagent must maintain it for future consumption by the WLM. The state-agent communicates with a kernel module through a single bidirectional communication chan-nel, receiving updates to the process state in the form of records and occasionally sending com-mands. Events in the kernel such as process fork, exit, reclassify (resulting from change in any process attribute such as gid, pid) cause records to be generated through functions provided by the kernel module.

Server tasks can assist the WLM by informing it about the phase in which they are operating (each phase corresponds to a workload). Such tasks invoke CKRM to set a tag associated with their `task_struct` in the kernel. CKRM uses this event to reclassify the task and also records the event (to be transmitted to the WLM through the state-agent). Other kernel events that might cause a task to be reclassified (such as the exec and setuid system calls, etc.) are also noted by CKRM and passed to the WLM through the state-agent. In addition, CKRM performs periodic sampling of each task's state in the kernel to determine the resource it is waiting on (if any), its resource consumption so far and the class to which it belongs. The sample information is transmitted to the state-agent. The WLM can correlate the information with the tag setting to statistically determine the resource consumption and delays of both server and dedicated processes executing a workload. Sampling is done through a kernel module function that is invoked by a selfrestarting kernel timer. Commands sent by the state-agent cause appropriate functions in the kernel module to execute and also return data in the form of records. The kernel components are kept simple and only minimal additional state has to be maintained in the kernel. In particular, the kernel does not have to maintain extra state about exited processes which introduces problems with PID reusage, memory management to name a few. Instead, relevant task information is replicated in user space, is by definition received in the correct time order (see below) and can be kept around until the WLM has consumed the information. Furthermore, the semantics of a reclassification in the kernel, which identifies a new phase in a server process, does not have to be introduced into the kernel space.

The following small changes are required to the linux kernel to track system delays. The `struct delay_info` is added to the `task_struct`. Delay_info contains 32-bit variables to store cpu delay, cpu using, io delay and memory io delay. The counters provide micro second accuracy. The current cpu scheduler records timestamps whenever i) a task becomes runnable and is entered into a runqueue and ii) when a context switch occurs from one task to another. We use these same timestamps to get per-task cpu wait and cpu using times recorded respectively. I/O delays are measured by the difference of timestamps taken when a task blocks waiting for I/O to complete and when it returns. All I/O is normally attributed to the blocking task. Page-fault delays, however, are treated as special I/O delays. On entrance to and exit from the page fault handler the task is marked or unmarked as being in a memory path using flags in `task_struct`. If during the I/O delay, this flag is set, the I/O delay is counted as a memory delay instead of as a pure I/O delay. The per-task delay information is accessible through the file `/proc/<pid>/delay`. Similarly, each class contains a `delay_info` structure.

In contrast to the precise accounting of delays, sampling examines the state of tasks at fixed interval. In particular, we sample at fixed intervals (~1sec) the entire set of tasks in the system and increment per task counters that are integrated into the task private structure attached

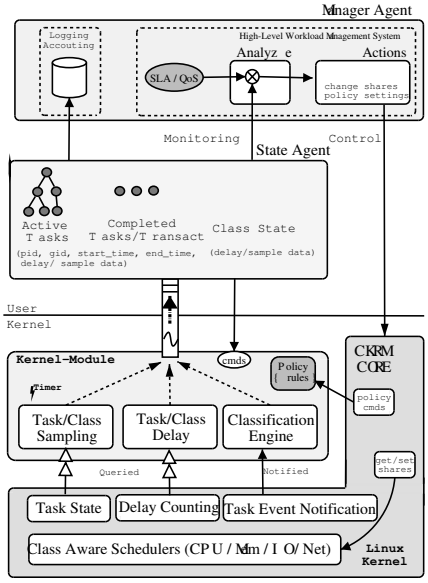


Figure 2: CKRM lifecycle

by the classification engine that builds the core of the kernel module. We increment counters if a task is running, waiting to run, performing I/O or handles a pagefault I/O. Task data (sampled and/or precise) is requested by and sent to the state-agent in coarser intervals. We can send data in continuous aggregate mode or in delta mode, i.e. only if task data has changed do we send a new data record and then reset the local counters. The task transition events are sent at the time they occur. We distinguish the fork, exit, and reclassification events as records. At each reclassification (which could potentially be the end of a phase) we transmit the sample and delay data and reset them locally.

As a communication channel we utilize the linux relayfs pseudo filesystem, a highly efficient mechanism to share data between kernel and user space. The user accesses the shared buffers, called channels, as files, while the kernel writes to them using buffer reservations and memory read/write operations. The content and structure of the buffer is determined by the kernel and user client. Currently the communication channel is self pacing. The underlying relayfs channel buffer will dynamically resize upto a maximum size. If for any reason the relayfs buffer overflows, record sending will automatically stop, an indication is sent and the state-agent will have to drain the channel and request a full state dump from the kernel.

We have measured the data rate during a standard kernel build, which creates a significant amount of task events (fork,exec,exits). For a 2-CPU system with 2 seconds sample collection we observed a data rate of 8KB/second and a total of 190 records/sec, well within a limit that can be processed without creating significant overhead in the system.

5 Inbound Network

Various OS implementations offer well established QoS infrastructure for outbound bandwidth management, policy-based routing and Diffserv [3]. Linux in particular, has an elaborate infrastructure for traffic control [8] that consists of queuing disciplines(qdisc) and filters. A qdisc consists of one or more queues and a packet scheduler. It makes traffic con-

form 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. The traffic control mechanims can be used invarious places where bandwidth is the primary resource to control.

Due to the above features, Linux is widely used for routers, gateways, edge servers; in other words, in situtations where network bandwidth is the primary resource to differentiate among classes. When it comes to endservers networking, QoS has not received as much attention since QoS is primarily governed by the systems resources such as memory, CPU and I/O and less by network bandwidth. When we consider end-to-end service quality, we should require networking QoS in the end servers as exemplified in the fair share admission control mechanism proposed in this section.

We present a simple change to the existing TCP accept mechanism to provide differentiated service across priority classes. Recent work in this area has introduced the concept of prioritized accept queues [6] and accept queue schedulers using adaptive proportional shares to self-managed web [14]. In a typical TCP connection, the client initiates a request to connect to a server. This connection request is queued in a global accept queue belonging to the socket associated with the server's port. The server process picks up the next queued connection request and services it. In effect, the incoming connections to a particular TCP socket are serialized and handled in FIFO order. When the incoming connection request load is higher than the level that can be handled by the server requests have to wait in the accept queue until the next can be picked up.

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. The accept queue implements a weighted fair scheduler such that the rate of acceptance from a particular accept queue is proportional to the weight of the queue. In the first version of the priority accept queue design initially proposed by the CKRM project [13], starvation of certain priority classes was a possibility as the accepting process picked up connection requests in the order of descending priority.

The efficacy of the proportional accept queue mechanism is demonstrated by an experiment. We used Netfilter [12] to MARK options to characterize traffic into two priority classes with respective weights of 3:1. The server process utilises a configurable number of threads to service the requests. The results are shown in Figure 3. When the load is low and there are service threads available no differentiation takes place and all requests are processed as they arrive. Under higher load, requests are queued in the accept queue with class 1 receiving a proportionally higher service rate than class 2. The expriment was repeated, maintaining a constant inbound connection request rate. The proportions of the two classes were then switched to see the service rate for the two classes reverse as seen in Figure 4

6 Resource Control Filesystem

In the Linux kernel development community, filesystems have become very popular as user

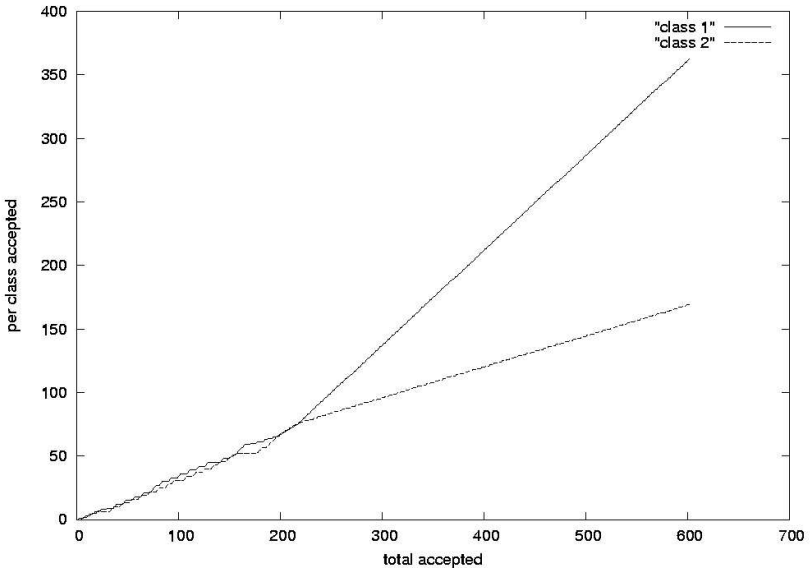


Figure 3: Proportional Accept Queue: Results

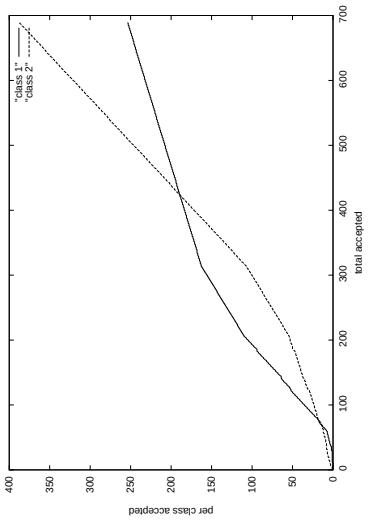


Figure 4: Proportional Accept Queue: Results under change

interfaces to kernel functionality, going well beyond the traditional use for disk-based persistent storage. The Linux kernel's object-oriented Virtual File System (VFS) makes it easy to implement a custom filesystem. Common file operations like open, close, read and write map naturally to initalization, shutdown, kernel-to-user and user-to-kernel communication. For CKRM, the tree structured namespace of a filesystem offers the additional benefit of an intuitive representation of the class hierarchy. Hence CKRM uses the Resource Control Filesystem (RCFS) as its user interface.

The first-level directories in RCFS contain the roots of subtrees associated with classtypes build or loaded into the kernel (`socket_class` and taskclass currently) and the clas-

sification engine (ce). Within the classtype subtrees, directories represent classes. Users can create new classes by creating a directory as long as they have the proper access rights. Within the `task_class` directory, each directory represents a task class. `/rcfs/taskclass`, the root of the `task_class` classtype, represents the default taskclass which is always present when CKRM is enabled in the kernel. Each `task_class` directory contains a set of virtual files that are created automatically when the directory is created. Each virtual file has a specific function as follows:

1. members: Reading it gives the names of the tasks in the taskclass.
2. config: To get/set any configuration parameters specific to the taskclass.
3. target: Writing a task's pid to this file causes the task to be moved to the taskclass, overriding any automatic classification that may have been done by a classification engine.
4. shares: Writing to this file sets new lower and upper bounds of the resource shares for the taskclass for each resource controller. Reading the file returns the current shares. The controller name is specified on a write which makes it possible to set the values for controllers independent of each other.
5. stats: Reading the file returns the statistics maintained for the taskclass by each resource controller in the system. Writing to the file (specifying the controller) resets the stats for that controller.

The `socket_class` directory is somewhat similar. Directories under `/rcfs/socket_class/` represent listen classes and have the same magic files as task_classes. Whereas task_classes use the pid to identify the class member, socket_classes, which group listening sockets, use ip address + port name to identify their members. Within each listen class, there are automatically created directories, one for each accept queue class. The accept queue directories, numbered 1 through 7, have their own shares and stats virtual files similar to those for `task_classes`.

The `/rcfs/ce` directory is the user interface to the optional classification engine. It contains the following virtual files and directory:

1. reclassify: writing a pid or ipadress+port to the file causes the corresponding task or listen socket to be put back under the control of the classification engine. On subsequent significant kernel events, the ce will attempt to reclassify the task/socket to a new taskclass/socketclass if the task/sockets attributes have changed.
2. state: to set/get the state (active or inactive) of the classification engine. To allow a new policy to be loaded atomically, CE's can be set to inactive before loading a set of rules and activated thereafter.
3. Rules: The directory allows privileged users to create files with each file representing one rule. Reading the files, permitted for all, gives the classiication policy which is currently active. The ordering of rules in a policy is determined either by creation time of the corresponding file or by an explicitly specified order number within the file. The rule files contain rule terms consisting of attribute-value pairs and a target class. E.g., the rule *gid=10, cmd = bash, target = /rcfs/taskclass/A* indicates that tasks with gid 10 and running the bash program (shell) should get reclassified to task_class A.

7 Example uses

In this section we will describe a number of uses for CKRM, ranging from the traditional large server workload consolidation, to a university shell server, to the desktop—a novel use of workload management systems, made possible through the resource class filesystem.

7.1 Workload Consolidation

The classical use of a workload management system is workload consolidation, whether it's multiple departmental database servers on one large server, or one small server balancing resources between apache, ftpd, postfix and the interactive users. In either scenario the main objective is to make sure that none of the workloads can, through excessive resource use, cause the machine to become unusable for any of the others.

The simple solution is to start each of the services up in their own resource class and guaranteeing a certain amount of resources (say, 10% of the CPU and 20% of memory) for each of the services. Simultaneously the services can also have resource limits (say, 50% of memory). This combination of guarantees and limits gives the system a certain amount of freedom to balance the actual amount of resources each workload gets, while still putting effective guarantees and limits in place.

7.2 Shell Server

A shell server at a university faces a number of challenges. For example, the staff and postdocs should be protected from the load the students put on the machine and the students should be protected from each other. Similarly, batch jobs will usually have larger resource use limits (e.g. max cpu time used, max memory allocated), but a lower resource priority, as compared to any of the interactive programs. These problems can be solved by starting each class of process in the right process class.

On the other hand, if a staff member sends email to a student, the resources used by the student's mail filter should be accounted against that student's limits. This problem cannot be solved by having programs start out in a certain resource class, since the MTA process needs to transition between resource classes automatically. This can be solved by setting up a classification engine to automatically transfer a process to the *email* resource class when it execs `/usr/sbin/sendmail`. Similarly, when `/usr/bin/procmail` is being execed with a certain UID, the classification engine can move the process to the resource class where that user's interactive processes would normally run.

7.3 Desktop

With the right file and directory ownerships in the resource class filesystem, CKRM can be used in an area where traditional resource management systems tend to be cumbersome: on the desktop. A typical desktop configuration would have as its main goals that the system remains responsive to the user, no matter the background load, and would look something like the following.

The X server would get a good resource guarantee, e.g. 20% of CPU time and 20% of RAM. This makes sure that no matter what other processes run on the system, X can run smoothly and react to the console user with acceptably low latency.

At login time a PAM module would make sure that the rest of the user's processes get a good resource guarantee, too. An acceptable guarantee would be 50% of CPU time and 50% of RAM. This leaves enough resources free so that other things in the system can run (e.g. dis-

tro updates, updatedb, mail delivery), yet keeps most of the system dedicated to the user. The resource class created for the console user, e.g. `/rcfs/taskclass/console`, is set up to be writable for the console user. This way the user's processes can set resource guarantees and limits to certain classes of applications.

The user's GUI menu would take care of this subdividing of the resources guaranteed to the user. For example, the web browser could be restricted to 40% of RAM, so as to not put much pressure on the user's other processes. Multimedia processes could get part of the user's resource guarantees, e.g. 30% of the CPU and 10% of RAM guaranteed for the multimedia applications. This way the playback of multimedia should remain smooth, regardless of what the user's web browser and office suite are doing.

No superuser privileges are needed to configure these resource classes, or to move the user's processes between them. Any GUI framework or individual application will be able to determine the resources allocated to it, leading to more flexibility than possible with resource management systems that can only be configured by the super user. Note that since the user cannot raise the resource limits or guaranteed allocated to his main class, there should be no security risks involved with letting the user processes manipulate their own resource guarantees and limits.

8 Conclusion and Future Work

The consolidation of increasingly dynamic workloads on large server platforms has considerably increased the complexity of systems management. To address this, goal-oriented workload managers are being proposed which seek to automate low-level system administration requiring human intervention only for defining high level policies that reflect business goals.

In an earlier paper [13], we had argued that goal-oriented WLMs require support from the operating system kernel for class-based differentiated service where a class is a dynamic policy-driven grouping of OS processes. We had introduced a framework, called class-based kernel resource management, 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.

In this paper, we have described more details of the evolving design. In particular, CKRM has become more generic and supports groups of any kernel object involved in resource management, not just tasks. It has a new filesystem-based user API. Finally, the design introduces hierarchies into classes which permits greater flexibility for resource managers but also introduces challenges for CKRM controllers. A working prototype which includes an inbound network controller has been developed and made available through [15].

Future work in the project will involve redeveloping controllers for CPU, memory and I/O that are not only class-aware but can handle hierarchies of classes while keeping overheads low. Another important direction is the interactions of the resource schedulers and the impact of these interactions on the shares specified.

References

[1] J. Aman, C.K. Eilert, D. Emmes, P. Yocom, and D. Dillenberger. Adaptive algorithms for Managing a Distributed Data Processing Workload. In *IBM Systems Journal*, volume 36(2), 1997.

[2] G. Banga, P. Druschel, and J.C. Mogul. Resource containers: A new facility for resource management in server systems. In *Operating Systems Design and Implementation*, pages 45–58, 1999.

[3] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. RFC 2475, Dec 1998.

[4] J. Blanquer, J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. Resource management for qos in eclipse/bsd. In *Proc. FreeBSD 1999 Conference*, Oct 1999.

[5] IBM Corp. AIX 5L Workload Manager. http://www.redbooks.ibm.com/redbooks/SG245977.html.

[6] IBM DeveloperWorks. Inbound connection control home page. http://www-124.ibm.com/pub/qos/paq_index.html.

[7] Inc. D.H.Brown Associates. HP Raises the Bar for UNIX Workload Management. http://h30081.www3.hp.com/products/wlm/docs/hp.raises.bar.wkld.mgmt.pdf.

[8] Bert Hubert. Linux Advanced Routing & Traffic Control. http://www.lartc.org.

[9] Hewlett Packard Inc. HP Process Resource Manager. http://h30081.www3.hp.com/products/prm/.

[10] Hewlett Packard Inc. HP-UX Workload Manager. http://h30081.www3.hp.com/products/wlm/.

[11] Sun Microsystems Inc. Solaris Resource Manager. http://www.sun.com/software/resourcemgr/wp-srm/.

[12] J. Kadlecsik, H. Welte, J. Morris, M. Boucher, and R. Russel. Netfilter: Firewalling, NAT, and packet mangling for Linux 2.4. http://www.netfilter.org.

[13] S. Nagar, H. Franke, J. Choi, M. Kravetz, C. Seetharaman, V. Kashyap, and N. Singhvi. Class-based prioritized resource control in Linux. In *Proc. 2003 Ottawa Linux Symposium, Ottawa*, July 2003. http://ckrm.sf.net/documentation/ckrm-ols03-paper.pdf.

[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] CKRM Open Source Project. Class-based kernel resource management. http://ckrm.sf.net/.

Linux on a Digital Camera

Porting 2.4 Linux kernel to an existing digital camera

Alain Volmat
Ricoh Company Ltd.
avolmat@src.ricoh.co.jp

Shigeki Ouchi
Ricoh Company Ltd
shigeki@src.ricoh.co.jp

Abstract

The RDC-i700 is one of high specs digital camera of Ricoh. Its relatively big size, large amount of different interfaces, input methods (buttons, or touch panel), have made it a good candidate for prototyping the world first Linux embedded digital camera. This paper presents our experiences of porting the 2.4 linux kernel to an existing digital camera. (the RDC-i700 is originally build on top of VxWorks). Eventhough embedded systems running on Linux are getting more and more popular, the digital camera field remains to be unexplored. The paper introduces how digital cameras differ from any other PC-like devices (PDA, HDD recorder...) and what problems, such as timing or software design issues, have to be (have been) solved in order to get the world first linux digital camera running on the linux 2.4 kernel.

1 The hardware

Ricoh's RDC-i700 [1] is a relatively old digital camera (released late 2000 in japan) running on VxWorks, a famous Real-Time OS (RTOS). Some might be asking the reason why we decided to port Linux OS to the camera. The reason is to make it become a *programmable camera*. Once it becomes a programmable device, many VARs or individual programmers may write a lot of useful software for it. Then it will be a good platform for business imaging use.

Figure 1: The RDC-i700 digital camera

The RDC-i700 is one of high specs digital camera of Ricoh. It integrates all peripherals traditional digital camera has, but also several different interfaces, allowing wide range of application to run on it. The VxWorks version allows user to perform various tasks such as taking picture or movie, recording voice memo, browse the Internet, send email or upload picture to a remote server. Its relatively big size, large amount of different interfaces, input methods (buttons, or touch panel), makes it a good candidate for prototyping the world first Linux embedded digital camera.

The RDC-i700 is a 3.2 million pixels digital camera equipped with a Hitachi SH3

[1] http://www.ricohzone.com/product_rdci700.html

(SH7709A) CPU. The SH7709A is a 32 bit RISC CPU which include MMU and several other peripherals such as serial communication interfaces (SCIs), D/A - A/D converters. Around this CPU, traditional digital camera peripherals (CCD, LCD, buttons, Image Processor) but also 1 PCMCIA and 1 CF socket, touch panel, audio input/output interface, USB device controller and a serial port are available. Figure 2 shows a block diagram of the RDC-i700.

2 Digital camera is not "PDA combined with camera function"

Nowadays, embedded Linux has become a very hot topic in the Linux community. More and more Linux gadget are becoming available and the share of embedded related paper published has literally exploded in the last 3 years. Linux seems to be everywhere, lots of devices that were running on RTOS in the past are now running on Linux. However, one field seems to be still unexplored: digital camera. Some might say that a digital camera is just a PDA combined with a CCD (this kind of combination is actually already available, for example the Zaurus CF Digital Camera option), but this is not that simple.

The quality point of view: PDA combined with a digital camera option can take pictures or even movies and in that sense can be compared to a digital camera. But digital cameras still have some advantages that make them irreplaceable. Indeed optical zoom, but also autofocus or strobe are all precious elements that are currently not available on Linux PDA. For example, the Zaurus camera has a focus but this one is manual. Auto-exposure is also another very important part when taking picture; for that, Zaurus PDA has some-kind of gain control but this cannot have same quality as a traditional digital camera auto-exposure system.

The technical point of view: We will see that having digital camera specific peripherals is a very good plus in term of quality, but it also creates lots of problem that traditional Linux PDA doesn't face. Keeping the Zaurus PDA as an example, only few parameters are configurable and the CPU doesn't actually have to perform much work in order to get an image. On the contrary, in case of a fully configurable digital camera, the OS must orchestrate all devices in order to get a picture.

2.1 Zoom and Focus

Several motors are used inside the camera. Two of them are used for zoom and focus in order to adjust the lens position. Due to high precision requirements, those two motors are stepping motors. As the name says, this kind of motors are controlled step by step (at the difference with traditional motors which only have start/stop command). The CPU has to set ports of the motor at a quite fast frequency in order to make the motor turn. In that case the period between two steps is only few milli-seconds.

2.2 Strobe

In case of strobe, the problem is not doing thing at very high speed, but making perfect synchronization between the moment the strobe is going to flash and the moment the CCD sensor will acquire the picture.(see figure 3) For that purpose, we will need precision of only very few milli-seconds.

2.3 Auto-Exposure / White Balance

So called Auto-Exposure is the algorithm in charge of adjusting the exposure time (that is to say the time period while the CCD is exposed to light) in order to have a good image

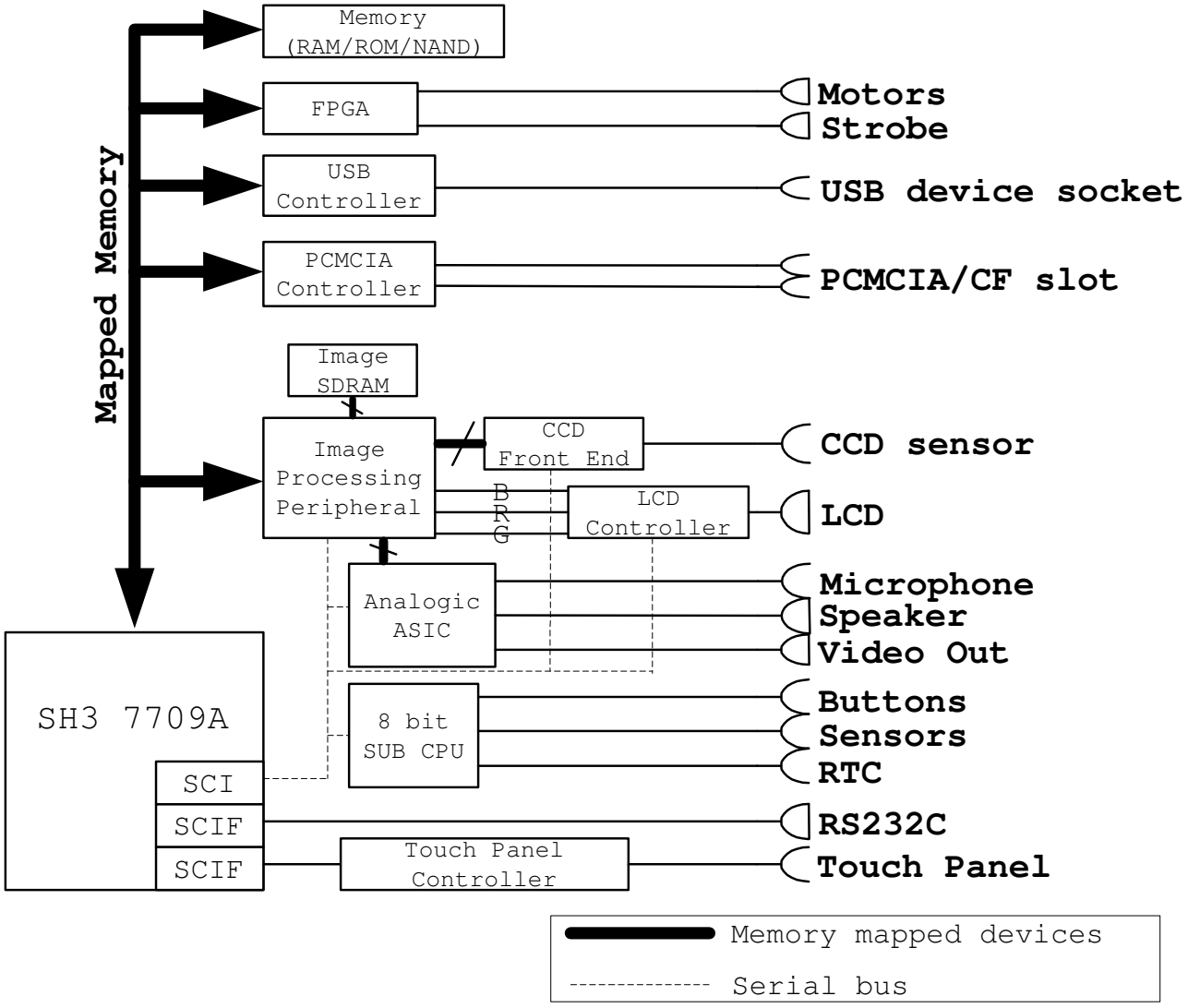


Figure 2: The RDC-i700 peripherals diagram

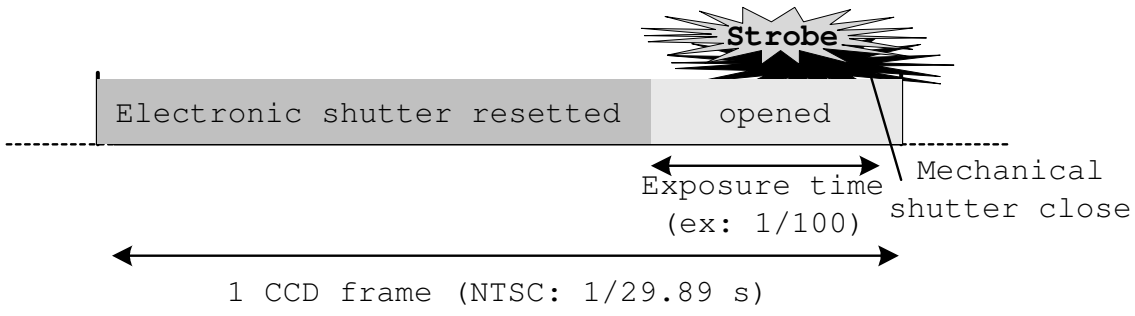


Figure 3: CCD Frame sequence

in both dark and bright conditions. White balance algorithm needs to analyze data coming from the CCD sensor and adjusts the Image Processing Peripheral (IPP) settings in order to have good color matching between the generated image and the reality. Several implementations of those two algorithms can exist (some needs very heavy calculations while others can be very simple), but the main issue is that those algorithms have to be performed very often (in the worst case, every frame of the CCD, that is to say every 33 milli-seconds).

This second part introduced particularity of digital cameras. The next part will discuss how those functions have been implemented into the Linux RDC-i700.

3 Current Support

As the name "Linux on a digital camera" suggests, the RDC-i700 can now run using Linux OS. Although some work remains in some areas, kernel support now exists for most of the hardware and features of the camera. This part explains current status for important features (digital camera related) of the kernel.

3.1 SH-Linux

The RDC-i700 linux kernel is originally based on the work of the SH-Linux [1] team. First tested with the kernel version 2.4.2, the camera is now using the version 2.4.19 of the kernel. SH-Linux kernel already had support for almost all parts of the SH3 7709, but since the RDC-i700 is using the CPU in big endian mode, some modifications were necessary in that field. Source code necessary to run the kernel on this new platform has also been added into `/arch/sh/kernel`.

3.2 RDC-i700 device drivers

RDC-i700 drivers can be separated into two kinds (or two layers). (See figure 4) The lower layer contains so-called **Low level drivers**, or drivers providing control to a specific device (such as focus sensor, IPP ...). All those drivers doesn't have any algorithm included and only provide basic access to the device capabilities. For example in case of the driver controlling motors (MECH driver), only functions provided are to set or get the position of the motor (motors have some predefined positions). RDC-i700 currently has 5 device drivers controlling imaging related devices (we will avoid non-imaging specific drivers here):

- **The CCD F/E (Front End)** which permits to control the CCD parameters (such as exposure time, gain ...)
- **The IPP (Image Processing Peripheral)** which is actually the heart of the camera (almost everything goes through the IPP)
- **The Strobe** driver which allows to charge or flash the strobe
- **The Focus Sensor** which permits to evaluate the distance between the camera and the target
- **Mech** driver which controls all mechanical parts of the camera, that is to say, iris, shutter, zoom and focus.

All those drivers are very system dependent and might change from one camera to another.

On the top of those 5 drivers is what we could called the "Algorithms" layer. This layer contains "intelligent" drivers such as auto focus driver, or auto exposure driver. One more driver, simply called CCD driver, is actually the driver which performs actions such as taking a picture or switching to monitoring mode.

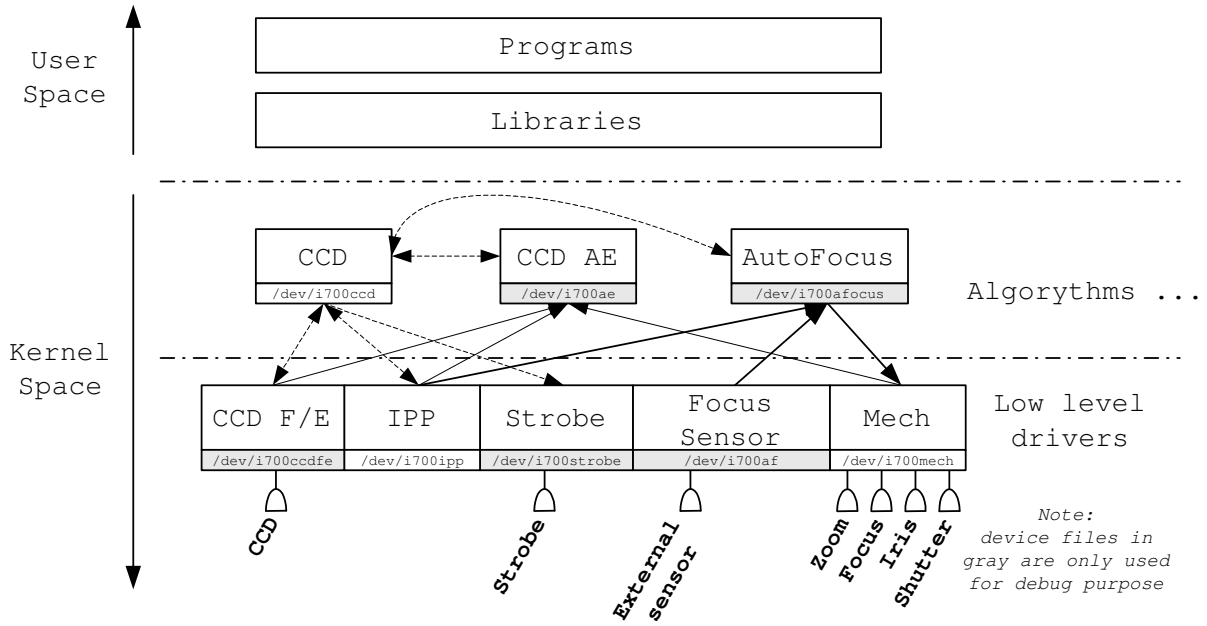


Figure 4: RDC-i700 device drivers

This driver has to access both low level drivers and algorithms drivers. Drivers of the upper layer are "virtually" platform independent. However since the current system lacks of a well defined abstraction layer, upper layer drivers are currently directly accessing lower layer driver which make them unable to work with any other lower driver without having to slightly change the source code. (see Future Work section). Currently device drivers communicate with each others by accessing EXPORTED functions.

All 8 drivers are registered to the kernel as characters drivers and can be accessed from user-level using each device file. Some of those drivers don't actually need to be accessed from user space and in that case device file is only used for debugging purpose. In user space, libraries provide easy access to camera functionalities, avoiding an intensive usage of IOCTL commands.

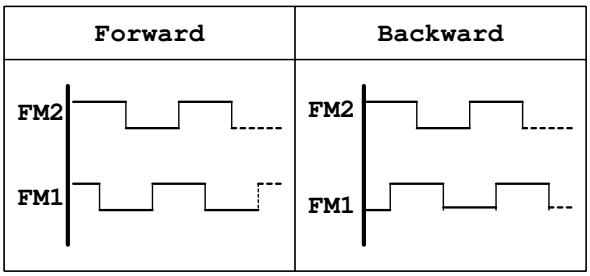


Figure 5: Motors step sequence

3.3 Motors

As the name says, stepping motors are going step by step; the CPU sets 2 I/O ports in order to specify the position of the rotor. Figure 5 shows motor ports state sequence when going forward and backward. Rotation speed is determined by the time between 2 states. Each motor has already predetermined positions, 19 for zoom and 18 for focus; however, if positions for the zoom are fixed, focus position varies depending on the current zoom position. All

those things are handled by the MECH driver and are accessible via IOCTL command such as "Get/Set position." The MECH driver is timer based, that is to say, the delay between 2 steps is performed using a timer (2.8 ms in case of focus, and 1.4 ms in case of zoom); we will see in part 4 the current problems when using this implementation. The driver provides both SYNC and NOSYNC mode; that is to say, in the first one, the ioctl command will hold until the command finish, but in NOSYNC mode, the ioctl will immediately return, allowing to call another ioctl command, even if the motor is still running. This is useful in the case of user's adjustment of the zoom. Since motor needs time to start and stop, it would be inefficient to request each time 1 position change. Instead of this, when the user uses the zoom lever, the first IOCTL command request the motor to go to max position and then when the user release the button, the ioctl STOP command will be requested. In other cases, such as when controlling the FOCUS motor from the auto focus driver, the SYNC mode should be used.

3.4 Auto Exposure

In order to control exposure, the auto exposure driver is accessing 3 different device drivers (IPP, MECH and the CCD front end). The IPP has the ability to divide a CCD image into several block and inform about each block luminance. By looking at those luminance values, the auto exposure algorithm decide how parameters should be modified; it can decide to change iris diameter (MECH driver), or use the strobe (STROBE driver), and in most of the case change parameters of the CCD front end (electronic shutter speed...). The digital camera can work in 2 different modes: the monitoring mode which permits to see in real time what the CCD sensor is targeting, and the still image mode which is used when the user pushes the shutter button to take a still image. The behaviour of the auto exposure module depends on the camera mode:

- **monitoring mode**

in that mode, we adjust CCD F/E parameters every 2 CCD frames. The auto exposure driver starts a kernel thread which needs to be synchronized with the CCD frame (an hardware interrupt is generated by the CCD F/E at every start of frame). Synchronization is achieved by using wait queues. [2] The function which needs to get synchronized creates a wait queue (as follows):

```
struct task_struct *tsk = current;
DECLARE_WAITQUEUE(wait, tsk);
add_wait_queue(&ccd_vd_wq,&wait);
set_current_state(TASK_INTERRUPTIBLE);
schedule();
set_current_state(TASK_RUNNING);
remove_wait_queue(&ccd_vd_wq,&wait);
```

and the wait queue is woken up by the interrupt handler (as follow):

```
wake_up(&ccd_vd_wq);
```

The CCD F/E is controlled using the SCI port of the SH3 which use is shared with some other devices. In some case, it might be necessary to wait for the SCI port availability and, for that reason, the kernel thread implementation has been preferred to some other solutions such as bottom halves (it is not possible to schedule from a bottom halves while kernel thread allows that).

- **still image mode**

in that mode, exposure parameters are only adjusted once before taking the picture. The CCD

[2] This synchronization method is also heavily used by the CCD driver

driver requests information to the auto exposure module which will calculate parameters used to take the picture. After that, the CCD driver will directly control lower driver to set those parameters. Synchronization between all devices is very important and for that reason it is easier to perform everything sequentially from a unique driver. No thread is used and the CCD driver get synchronized with the CCD frame using the same wait queue method as in monitoring mode.

Currently only a very simple algorithm is available for both monitoring and still mode. This algorithm doesn't make use of neither iris nor strobe and the exposure is only controlled using CCD FE's parameters and the mechanical shutter.

3.5 Auto Focus

Compared to the auto exposure, the auto focus driver is quite an easy one. The IPP driver has the ability to determine the "focus level" (the more the focus is correct, the more the value returned by the IPP will be high). In normal mode, the auto-focus driver should get an approximation of the distance to the target by using the focus sensor, then first adjust the focus to this approximation. This permits to perform the "fine focus" (using the IPP capability) to a smaller range. However, the current implementation doesn't use the focus sensor approximation which means that the "fine focus" is performed to the full range of the focus (this is actually the mode which is used in case of MACRO mode). The consequence is that the auto-focus process is much slower than in normal mode. Currently the driver performs the following things:

- check zoom status to calculate focus positions
- retrieve focus level for all focus positions
- go back to the position with the highest focus level

3.6 CCD - IPP

The IPP driver is some kind of library which, except performing initialization of the device, mainly provides a lot of functions, accessible from other drivers and permitting to control the hardware. The driver is quite big since the IPP performs very various things such as

- JPEG compression/decompression
- YUV-RGB conversion
- video output (for the LCD and TV)
- image scaler

The CCD driver is considered as the main driver since almost everything starts from it. It is in charge of coordination between all other drivers. The driver can be controlled using a user land library permitting to control the monitoring mode or to take still image. The driver mainly uses other drivers functions (CCD F/E, IPP, MECH) and performs synchronization using the waitqueue method introduced previously.

3.7 LCD

The RDC-i700 LCD has a fixed resolution of 640x480 pixels. What we could call video card is actually a part of the IPP chip and can control 4 layers of display (1 layer for image/video data, and 3 On Screen Display or OSDs). In the current design the first layer is controlled by the IPP driver and doesn't have direct interface to the user land. Even if 3 OSDs are available, only one is currently used as a framebuffer device. The OSD uses a 8 bit YUV palette (maintained by the IPP device) which

means that `_setcolreg` and `_getcolreg` entry points are used to perform conversion between RGB and YUV color space. This solution allows to use the camera LCD as any traditional Linux console and run any software that usually works on the top of a Linux Framebuffer. One reason why only 1 OSD level is supported is because all OSDs share the same palette which means that it cannot be simply designed as 3 different framebuffer devices. However there is also currently no real need for 3 OSDs so this is not actually a big issue.

3.8 Filesystem

The RDC-i700 has 8 MB of NAND Flash internal memory. The Linux kernel now provides support for this kind of memory by using the Memory Technology Devices (MTD) [2] support. Only a very small layer needs to be written in order to get the camera's NAND work. [3] JFFS2 [4] is usually used on the top of a NAND device, however we will see that in case of digital camera, it might not be the best solution. In our case, internal flash memory is usually exclusively used for storage of compressed data such as JPEG or MPEG. In that case, using JFFS2, which is a compressed filesystem, makes the CPU spend lots of time compressing data which anyway will almost not get compressed more than they are. In such case, YAFFS [5] should be preferred to JFFS2 since it is not a compressed filesystem.(see table 1 for details of tests performed on the RDC-i700)

Time consumption for 1 transaction (secs)		
	YAFFS	JFFS2
JPG (80k)	0.37	0.54
JPG (193k)	0.79	1.32
JPG (547k)	2.21	3.63
MJPG (1463k)	5.6	9.51
*1 transaction = NAND to NAND file copy		

Table 1: YAFFS / JFFS2 tests on RDC-i700

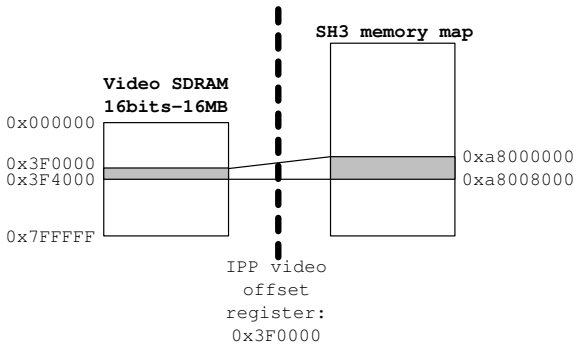


Figure 6: Accessing the video SDRAM

4 Issues

Several problems have been encountered while developing the Linux RDC-i700. Some have been solved but some are still under progress.

4.1 Framebuffer with non-linear memory

The purpose of a framebuffer device is to provide a standard way to access **linearly** video memory from the user space. This is the case of almost (or probably all) video card running on linux. Usually the kernel CPU can directly access any part of the video memory, linearly. However, in case of the RDC-i700, this is not the case. As figure 6 shows, the video SDRAM is not directly accessible from the CPU but is seen through the IPP chip. The IPP provides a 8KB window of memory directly accessible from the SH CPU. By setting a register of the IPP it becomes possible to define which "page" of the SDRAM becomes visible to the CPU. This makes problem with the framebuffer since the video memory is supposed to be linearly accessible, so no method is provided for such kind of system.

Inside a framebuffer device driver, two memory access methods exist:

linux console access mode is performed via function call. 6 functions (read and write for byte, word and long type) are provided. In case of the RDC-i700, the trick is just to overwrite those function in order to set the IPP register to the page needed to be accessed.

mmap access mode is necessary to allow user land application to access the framebuffer memory. The problem in case of mmap access is that the driver doesn't know which part of the memory is being accessed and so it becomes impossible to correctly set the page selector.

To solve this problem, the framebuffer uses a NOPAGE memory handler, combined with the `remap_page_range` function. By always leaving only 1 page mapped at a time, we ensure that the NOPAGE handler will be called everytime the application is trying to access a page which is not mapped. The handler will then unmap the previous page and map the page corresponding to the address to be accessed. One problem remains, which is, if `remap_page_range` allows to map page, it seems there is no function to "unmap" a previously mapped page. The function `zap_page_range` seems to do similar thing and by implementing the nopage handler as follows, the trick seems to work.

```
nopage_handler(...)
{
  calculate pointed addr in SDRAM;
  calculate physical addr;
  if(already_mapped){
    zap_page_range(...);
    flush_tlb_range(...);
  }
  remap_page_range(...);
  already_mapped=1;
}
```

However, some errors occurs time to time when using mmap on the framebuffer and those errors might come from this implementation.

4.2 Timers

We have seen in previous section that several timers are used in various drivers. Some must be very short and for that reason, timer is a very hot topic in our case. Several implementations are possible to perform delay.

busy wait: this solution should be avoided since it would for example almost stop the camera everytime the zoom is adjusted.

kernel timers: kernel timers should be used in order to avoid problems introduced by the busy wait solution. However, in order to achieve such implementation we need first to solve one big problem. While we need about 1ms or less resolution timer, a vanilla 2.4.19 kernel only permits to use timers with resolution of 10ms. In case of stepping motors, this doesn't really make big problem except that motors will just run about 10 times slower than their nominal speed. However, the low accuracy of kernel timers makes problem when used to control other devices such as mechanical shutter, strobe or iris since it can result in low quality image or, even worse, wrong operation performed (narrow instead of large for the iris). In order to solve this problem, several possibilities exist:

- **High Resolution Timer: [7]** is a project hosted on sourceforge in order to add high resolution (nano seconds) capable timer to the Linux kernel. However currently only the i386 architecture is supported, which means that some work is needed in order to get it work on the SH architecture.
- **Hardware Timer:** if such short delay cannot be achieved using software timer,

it still remains the possibility to use hardware timers of the SH3. However, this would make drivers very architecture dependent which should be avoided if possible. Moreover, even if the hardware timer can generate very short period, we need to ensure that the time between the hardware interrupt is generated and the interrupt handler get called is not too long otherwise using hardware timer wouldn't have any meaning. In order to achieve such requirement, it might be necessary to use preemptive kernel.

- **Vanilla kernel with HZ=1000:** changing tick period from 10ms to 1ms allows to use 1ms timer. However, if this solution works fine in some case, we need further experimentation in order to check the accuracy under heavy load condition.

5 Future works

5.1 Design of device driver architecture and user access

Kernel driver layering: the goal is to create a proper abstraction layer permitting to have upper kernel drivers (algorithms) totally independent from the hardware. Currently EXPORTED functions are used to allow function calls between drivers, however it means that upper drivers must understand the behavior of hardware drivers. The abstraction layer needs to define both function prototypes and structures used to access lower driver functionalities. Such interface would permit to easily customize any upper drivers, for example auto exposure algorithm or auto white balance.

Exporting functionalities to user space: currently small libraries are available, permitting to control camera functionalities. However, it doesn't seem reasonable to write new libraries specifically for the camera. Modifying device drivers to make them compatible with some existing standard should be the solution to take advantage of the large amount of existing software. In the camera field, Video For Linux would probably be a good candidate, and especially the second release which is currently under development. The idea would be to provide access to the CCD as a standard video input interface, similar to any USB camera for example. Other functionalities, such as JPEG compression, decompression could be accessed as a CODEC.

5.2 Remaining tasks

Some features still need to be implemented on the camera such as:

Power Management: currently no power management is performed while running Linux on the RDC-i700. This makes the battery life as short as about 25 minutes when using PCMCIA cards. This part should be the next big issue for the linux RDC-i700.

USB controller: the RDC-i700 includes a PDIUSBD12 USB device (slave) controller[3]. The Linux-USB Gadget API [8] allows to easily implement USB device class on the top of controller drivers, however this device controller is currently not supported yet.

6 Conclusion

The linux RDC-i700 has now enough support in order to be used as a digital camera. Most of the constrains due to the architecture and specific hardware have been solved but we still need some more performance testing in order to ensure that everything can run well. But remaining issues are not only technical one. Since we are now preparing for distributing the

[3] http://www.semiconductors.philips.com/pip/PDIUSBD12.html

source codes, we still needs some more coordination in our company. We also have to think of how to make and support a developing community.

References

[1] SH-Linux
http://linuxsh.sourceforge.net

[2] Memory Technology Devices (MTD)
http://www.linux-mtd.infradead.org

[3] MTD's NAND Flash support
http://www.linux-mtd.infradead.org/tech/nand.html

[4] JFFS2 homepage
http://source.redhat.com/jffs2/

[5] YAFFS homepage
http://www.aleph1.co.uk/yaffs/

[6] Linux Framebuffer Driver Writing HOWTO
http://linux-fbdev.sourceforge.net/HOWTO/

[7] High Resolution Timer Project
http://high-res-timers.sourceforge.net/

[8] USB-Gadget API
http://www.linux-usb.org/gadget/

[9] *Linux Device Drivers*, 2nd Edition
Alessandro Rubini & Jonathan Corbet

`ct_sync`: state replication of `ip_conntrack`

Harald Welte
netfilter core team / Astaro AG / hmw-consulting.de
laforge@gnumonks.org

Abstract

With traditional, stateless firewalling (such as ipfwadm, ipchains) there is no need for special HA support in the firewalling subsystem. As long as all packet filtering rules and routing table entries are configured in exactly the same way, one can use any available tool for IP-Address takeover to accomplish the goal of failing over from one node to the other.

With Linux 2.4/2.6 netfilter/iptables, the Linux firewalling code moves beyond traditional packet filtering. Netfilter provides a modular connection tracking susbsystem which can be employed for stateful firewalling. The connection tracking subsystem gathers information about the state of all current network flows (connections). Packet filtering decisions and NAT information is associated with this state information.

In a high availability scenario, this connection tracking state needs to be replicated from the currently active firewall node to all standby slave firewall nodes. Only when all connection tracking state is replicated, the slave node will have all necessary state information at the time a failover event occurs.

Due to funding by Astaro AG, the netfilter/iptables project now offers a `ct_sync` kernel module for replicating connection tracking state accross multiple nodes. The presentation will cover the architectural design and implementation of the connection tracking failover sytem.

1 Failover of stateless firewalls

There are no special precautions when installing a highly available stateless packet filter. Since there is no state kept, all information needed for filtering is the ruleset and the individual, separate packets.

Building a set of highly available stateless packet filters can thus be achieved by using any traditional means of IP-address takeover, such as Heartbeat or VRRPd.

The only remaining issue is to make sure the firewalling ruleset is exactly the same on both machines. This should be ensured by the firewall administrator every time he updates the ruleset and can be optionally managed by some scripts utilizing scp or rsync.

If this is not applicable, because a very dynamic ruleset is employed, one can build a very easy solution using iptables-supplied tools iptables-save and iptables-restore. The output of iptables-save can be piped over ssh to iptables-restore on a different host.

Limitations

- no state tracking
- not possible in combination with iptables stateful NAT

- no counter consistency of per-rule packet/byte counters

2 Failover of stateful firewalls

Modern firewalls implement state tracking (a.k.a. connection tracking) in order to keep some state about the currently active sessions. The amount of per-connection state kept at the firewall depends on the particular configuration and networking protocols used.

As soon as `any` state is kept at the packet filter, this state information needs to be replicated to the slave/backup nodes within the failover setup.

Since Linux 2.4.x, all relevant state is kept within the *connection tracking subsystem*. In order to understand how this state could possibly be replicated, we need to understand the architecture of this conntrack subsystem.

2.1 Architecture of the Linux Connection Tracking Subsystem

Connection tracking within Linux is implemented as a netfilter module, called `ip_conntrack.o` (`ip_conntrack.ko` in 2.6.x kernels).

Before describing the connection tracking subsystem, we need to describe a couple of definitions and primitives used throughout the conntrack code.

A connection is represented within the conntrack subsystem using `struct ip_conntrack`, also called *connection tracking entry*.

Connection tracking is utilizing *conntrack tuples*, which are tuples consisting of

- source IP address
- source port (or icmp type/code, gre key, ...)
- destination IP address
- destination port
- layer 4 protocol number

A connection is uniquely identified by two tuples: The tuple in the original direction (`IP_CT_DIR_ORIGINAL`) and the tuple for the reply direction (`IP_CT_DIR_REPLY`).

Connection tracking itself does not drop packets[1] or impose any policy. It just associates every packet with a connection tracking entry, which in turn has a particular state. All other kernel code can use this state information[2].

2.1.1 Integration of conntrack with netfilter

If the `ip_conntrack.[k]o` module is registered with netfilter, it attaches to the `NF_IP_PRE_ROUTING`, `NF_IP_POST_ROUTING`, `NF_IP_LOCAL_IN`, and `NF_IP_LOCAL_OUT` hooks.

Because forwarded packets are the most common case on firewalls, I will only describe how connection tracking works for forwarded packets. The two relevant hooks for forwarded packets are `NF_IP_PRE_ROUTING` and `NF_IP_POST_ROUTING`.

Every time a packet arrives at the `NF_IP_PRE_ROUTING` hook, connection tracking creates a conntrack tuple from the packet. It then compares this tuple to the original and re-

[1] well, in some rare cases in combination with NAT it needs to drop. But don't tell anyone, this is secret.

[2] State information is referenced via the `struct sk_buff.nfct` structure member of a packet.

ply tuples of all already-seen connections[3] to find out if this just-arrived packet belongs to any existing connection. If there is no match, a new conntrack table entry (`struct ip_conntrack`) is created.

Let's assume the case where we have already existing connections but are starting from scratch.

The first packet comes in, we derive the tuple from the packet headers, look up the conntrack hash table, don't find any matching entry. As a result, we create a new `struct ip_conntrack`. This `struct ip_conntrack` is filled with all necessarry data, like the original and reply tuple of the connection. How do we know the reply tuple? By inverting the source and destination parts of the original tuple.[4] Please note that this new `struct ip_conntrack` is **not** yet placed into the conntrack hash table.

The packet is now passed on to other callback functions which have registered with a lower priority at `NF_IP_PRE_ROUTING`. It then continues traversal of the network stack as usual, including all respective netfilter hooks.

If the packet survives (i.e., is not dropped by the routing code, network stack, firewall ruleset, ...), it re-appears at `NF_IP_POST_ROUTING`. In this case, we can now safely assume that this packet will be sent off on the outgoing interface, and thus put the connection tracking entry which we created at `NF_IP_PRE_ROUTING` into the conntrack hash table. This process is called *confirming the conntrack*.

The connection tracking code itself is not monolithic, but consists of a couple of separate modules[5]. Besides the conntrack core, there are two important kind of modules: Protocol helpers and application helpers.

Protocol helpers implement the layer-4-protocol specific parts. They currently exist for TCP, UDP, and ICMP (an experimental helper for GRE exists).

2.1.2 TCP connection tracking

As TCP is a connection oriented protocol, it is not very difficult to imagine how conntection tracking for this protocol could work. There are well-defined state transitions possible, and conntrack can decide which state transitions are valid within the TCP specification. In reality it's not all that easy, since we cannot assume that all packets that pass the packet filter actually arrive at the receiving end...

It is noteworthy that the standard connection tracking code does **not** do TCP sequence number and window tracking. A well-maintained patch to add this feature has existed for almost as long as connection tracking itself. It will be integrated with the 2.5.x kernel. The problem with window tracking is its bad interaction with connection pickup. The TCP conntrack code is able to pick up already existing connections, e.g. in case your firewall was rebooted. However, connection pickup is conflicting with TCP window tracking: The TCP window scaling option is only transferred at connection setup time, and we don't know about it in case of pickup...

[3] Of course this is not implemented as a linear search over all existing connections.

[4] So why do we need two tuples, if they can be derived from each other? Wait until we discuss NAT.

[5] They don't actually have to be separate kernel modules; e.g. TCP, UDP, and ICMP tracking modules are all part of the linux kernel module `ip_conntrack.o`.

2.1.3 ICMP tracking

ICMP is not really a connection oriented protocol. So how is it possible to do connection tracking for ICMP?

The ICMP protocol can be split in two groups of messages:

- ICMP error messages, which sort-of belong to a different connection ICMP error messages are associated *RELATED* to a different connection. (`ICMP_DEST_UNREACH`, `ICMP_SOURCE_QUENCH`, `ICMP_TIME_EXCEEDED`, `ICMP_PARAMETERPROB`, `ICMP_REDIRECT`).
- ICMP queries, which have a `request-reply` character. So what the conntrack code does, is let the request have a state of *NEW*, and the reply *ESTABLISHED*. The reply closes the connection immediately. (`ICMP_ECHO`, `ICMP_TIMESTAMP`, `ICMP_INFO_REQUEST`, `ICMP_ADDRESS`)

2.1.4 UDP connection tracking

UDP is designed as a connectionless datagram protocol. But most common protocols using UDP as layer 4 protocol have bi-directional UDP communication. Imagine a DNS query, where the client sends an UDP frame to port 53 of the nameserver, and the nameserver sends back a DNS reply packet from its UDP port 53 to the client.

Netfilter treats this as a connection. The first packet (the DNS request) is assigned a state of *NEW*, because the packet is expected to create a new 'connection.' The DNS server's reply packet is marked as *ESTABLISHED*.

2.1.5 conntrack application helpers

More complex application protocols involving multiple connections need special support by a so-called "conntrack application helper module." Modules in the stock kernel come for FTP, IRC (DCC), TFTP, and Amanda. Netfilter CVS currently contains patches for PPTP, H.323, Eggdrop botnet, mms, DirectX, RTSP, and talk/ntalk. We're still lacking a lot of protocols (e.g. SIP, SMB/CIFS)—but they are unlikely to appear until somebody really needs them and either develops them on his own or funds development.

2.1.6 Integration of connection tracking with iptables

As stated earlier, conntrack doesn't impose any policy on packets. It just determines the relation of a packet to already existing connections. To base packet filtering decision on this state information, the iptables *state* match can be used. Every packet is within one of the following categories:

- **NEW**: packet would create a new connection, if it survives
- **ESTABLISHED**: packet is part of an already established connection (either direction)
- **RELATED**: packet is in some way related to an already established connection, e.g. ICMP errors or FTP data sessions
- **INVALID**: conntrack is unable to derive conntrack information from this packet. Please note that all multicast or broadcast packets fall in this category.

2.2 Poor man's conntrack failover

When thinking about failover of stateful firewalls, one usually thinks about replication of state. This presumes that the state is gathered at one firewalling node (the currently active node), and replicated to several other passive standby nodes. There is, however, a very different approach to replication: concurrent state tracking on all firewalling nodes.

While this scheme has not been implemented within `ct_sync`, the author still thinks it is worth an explanation in this paper.

The basic assumption of this approach is: In a setup where all firewalling nodes receive exactly the same traffic, all nodes will deduct the same state information.

The implementability of this approach is totally dependent on fulfillment of this assumption.

- *All packets need to be seen by all nodes*. This is not always true, but can be achieved by using shared media like traditional ethernet (no switches!!) and promiscuous mode on all ethernet interfaces.
- *All nodes need to be able to process all packets*. This cannot be universally guaranteed. Even if the hardware (CPU, RAM, Chipset, NICs) and software (Linux kernel) are exactly the same, they might behave different, especially under high load. To avoid those effects, the hardware should be able to deal with way more traffic than seen during operation. Also, there should be no userspace processes (like proxies, etc.) running on the firewalling nodes at all. WARNING: Nobody guarantees this behaviour. However, the poor man is usually not interested in scientific proof but in usability in his particular practical setup.

However, even if those conditions are fulfilled, there are remaining issues:

- *No resynchronization after reboot*. If a node is rebooted (because of a hardware fault, software bug, software update, etc.) it will lose all state information until the event of the reboot. This means, the state information of this node after reboot will not contain any old state, gathered before the reboot. The effects depend on the traffic. Generally, it is only assured that state information about all connections initiated after the reboot will be present. If there are short-lived connections (like http), the state information on the just rebooted node will approximate the state information of an older node. Only after all sessions active at the time of reboot have terminated, state information is guaranteed to be resynchronized.
- *Only possible with shared medium*. The practical implication is that no switched ethernet (and thus no full duplex) can be used.

The major advantage of the poor man's approach is implementation simplicity. No state transfer mechanism needs to be developed. Only very little changes to the existing conntrack code would be needed in order to be able to do tracking based on packets received from promiscuous interfaces. The active node would have packet forwarding turned on, the passive nodes, off.

I'm not proposing this as a real solution to the failover problem. It's hackish, buggy, and likely to break very easily. But considering it can be implemented in very little programming

time, it could be an option for very small installations with low reliability criteria.

2.3 Conntrack state replication

The preferred solution to the failover problem is, without any doubt, replication of the connection tracking state.

The proposed conntrack state replication soltution consists of several parts:

- A connection tracking state replication protocol
- An event interface generating event messages as soon as state information changes on the active node
- An interface for explicit generation of connection tracking table entries on the standby slaves
- Some code (preferrably a kernel thread) running on the active node, receiving state updates by the event interface and generating conntrack state replication protocol messages
- Some code (preferrably a kernel thread) running on the slave node(s), receiving conntrack state replication protocol messages and updating the local conntrack table accordingly

Flow of events in chronological order:

- *on active node, inside the network RX softirq*
  - `ip_conntrack` analyzes a forwarded packet
  - `ip_conntrack` gathers some new state information
  - `ip_conntrack` updates conntrack hash table
  - `ip_conntrack` calls event API
  - function registered to event API builds and enqueues message to send ring
- *on active node, inside the conntrack-sync sender kernel thread*
  - `ct_sync_send` aggregates multiple messages into one packet
  - `ct_sync_send` dequeues packet from ring
  - `ct_sync_send` sends packet via in-kernel sockets API
- *on slave node(s), inside network RX softirq*
  - `ip_conntrack` ignores packets coming from the `ct_sync` interface via NOTRACK mechanism
  - UDP stack appends packet to socket receive queue of `ct_sync_recv` kernel thread
- *on slave node(s), inside conntrack-sync receive kernel thread*
  - `ct_sync_recv` thread receives state replication packet
  - `ct_sync_recv` thread parses packet into individual messages
  - `ct_sync_recv` thread creates/updates local `ip_conntrack` entry

2.3.1 Connection tracking state replication protocol

In order to be able to replicate the state between two or more firewalls, a state replication protocol is needed. This protocol is used

over a private network segment shared by all nodes for state replication. It is designed to work over IP unicast and IP multicast transport. IP unicast will be used for direct point-to-point communication between one active firewall and one standby firewall. IP multicast will be used when the state needs to be replicated to more than one standby firewall.

The principal design criteria of this protocol are:

- **reliable against data loss**, as the underlying UDP layer only provides checksumming against data corruption, but doesn't employ any means against data loss
- **lightweight**, since generating the state update messages is already a very expensive process for the sender, eating additional CPU, memory, and IO bandwith.
- **easy to parse**, to minimize overhead at the receiver(s)

The protocol does not employ any security mechanism like encryption, authentication, or reliability against spoofing attacks. It is assumed that the private conntrack sync network is a secure communications channel, not accessible to any malicious third party.

To achieve the reliability against data loss, an easy sequence numbering scheme is used. All protocol messages are prefixed by a sequence number, determined by the sender. If the slave detects packet loss by discontinuous sequence numbers, it can request the retransmission of the missing packets by stating the missing sequence number(s). Since there is no acknowledgement for sucessfully received packets, the sender has to keep a reasonably-sized[6] backlog of recently-sent packets in order to be able to fulfill retransmission requests.

[6] *reasonable size* must be large enough for the round-trip time between master and slowest slave.

The different state replication protocol packet types are:

- **`CT_SYNC_PKT_MASTER_ANNOUNCE`**: A new master announces itself. Any still existing master will downgrade itself to slave upon reception of this packet.
- **`CT_SYNC_PKT_SLAVE_INITSYNC`**: A slave requests initial synchronization from the master (after reboot or loss of sync).
- **`CT_SYNC_PKT_SYNC`**: A packet containing synchronization data from master to slaves
- **`CT_SYNC_PKT_NACK`**: A slave indicates packet loss of a particular sequence number

The messages within a `CT_SYNC_PKT_SYNC` packet always refer to a particular *resource* (currently `CT_SYNC_RES_CONNTRACK` and `CT_SYNC_RES_EXPECT`, although support for the latter has not been fully implemented yet).

For every resource, there are several message types. So far, only `CT_SYNC_MSG_UPDATE` and `CT_SYNC_MSG_DELETE` have been implemented. This means a new connection as well as state changes to an existing connection will always be encapsulated in a `CT_SYNC_MSG_UDPATE` message and therefore contain the full conntrack entry.

To uniquely identify (and later reference) a conntrack entry, the only unique criteria is used: `ip_conntrack_tuple`.

2.3.2 `ct_sync` sender thread

Maximum care needs to be taken for the implementation of the ctsyncd sender.

The normal workload of the active firewall node is likely to be already very high, so generating and sending the conntrack state replication messages needs to be highly efficient.

It was therefore decided to use a pre-allocated ringbuffer for outbound `ct_sync` packets. New messages are appended to individual buffers in this ring, and pointers into this ring are passed to the in-kernel sockets API to ensure a minimum number of copies and memory allocations.

2.3.3 `ct_sync` initsync sender thread

In order to facilitate ongoing state synchronization at the same time as responding to initial sync requests of an individual slave, the sender has a separate kernel thread for initial state synchronization (and `ct_sync_initsync`).

At the moment it iterates over the state table and transmits packets with a fixed rate of about 1000 packets per second, resulting in about 4000 connections per second, averaging to about 1.5 Mbps of bandwith consumed.

The speed of this initial sync should be configurable by the system administrator, especially since there is no flow control mechanism, and the slave node(s) will have to deal with the packets or otherwise lose sync again.

This is certainly an area of future improvement and development—but first we want to see practical problems with this primitive scheme.

2.3.4 `ct_sync` receiver thread

Implementation of the receiver is very straightforward.

For performance reasons, and to facilitate code-reuse, the receiver uses the same pre-allocated ring buffer structure as the sender. Incoming packets are written into ring members and then successively parsed into their individual messages.

Apart from dealing with lost packets, it just needs to call the respective conntrack add/modify/delete functions.

2.3.5 Necessary changes within netfilter conntrack core

To be able to achieve the described conntrack state replication mechanism, the following changes to the conntrack core were implemented:

- Ability to exclude certain packets from being tracked. This was a long-wanted feature on the TODO list of the netfilter project and is implemented by having a “raw” table in combination with a “NOTRACK” target.
- Ability to register callback functions to be called every time a new conntrack entry is created or an existing entry modified. This is part of the nfnetlink-ctnetlink patch, since the ctnetlink event interface also uses this API.
- Export an API to externally add, modify, and remove conntrack entries.

Since the number of changes is very low, their inclusion into the mainline kernel is not a problem and can happen during the 2.6.x stable kernel series.

2.3.6 Layer 2 dropping and `ct_sync`

In most cases, netfilter/iptables-based firewalls will not only function as packet filter but also

run local processes such as proxies, dns relays, smtp relays, etc.

In order to minimize failover time, it is helpful if the full startup and configuration of all network interfaces and all of those userspace processes can happen at system bootup time rather then in the instance of a failover.

l2drop provides a convenient way for this goal: It hooks into layer 2 netfilter hooks (immediately attached to `netif_rx()` and `dev_queue_xmit`) and blocks all incoming and outgoing network packets at this very low layer. Even kernel-generated messages such as ARP replies, IPv6 neighbour discovery, IGMP, ... are blocked this way.

Of course there has to be an exemption for the state synchronization messages themselves. In order to still facilitate remote administration via SSH and other communication between the cluster nodes, the whole network interface used for synchronization is subject to this exemption from l2drop.

As soon as a node is propagated to master state, l2drop is disabled and the system becomes visible to the network.

2.3.7 Configuration

All configuration happens via module parameters.

- `syncdev`: Name of the multicast-capable network device used for state synchronization among the nodes
- `state`: Initial state of the node (0=slave, 1=master)
- `id`: Unique Node ID (0..255)
- `l2drop`: Enable (1) or disable (0) the l2drop functionality

2.3.8 Interfacing with the cluster manager

As indicated in the beginning of this paper, `ct_sync` itself does not provide any mechanism to determine outage of the master node within a cluster. This job is left to a cluster manager software running in userspace.

Once an outage of the master is detected, the cluster manager needs to elect one of the remaining (slave) nodes to become new master. On this elected node, the cluster manager will write the ascii character `1` into the `/proc/net/ct_sync` file. Reading from this file will return the current state of the local node.

3 Acknowledgements

The author would like to thank his fellow netfilter developers for their help. Particularly important to `ct_sync` is Krisztian KOVACS `<hidden@balabit.hu>`, who did a proof-of-concept implementation based on my first paper on `ct_sync` at OLS2002.

Without the financial support of Astaro AG, I would not have been able to spend any time on `ct_sync` at all.

Increasing the Appeal of Open Source Projects

Experiences from the LSB Project

Mats Wichmann
Intel Corporation / LSB Project
mats.d.wichmann@intel.com

Abstract

It is often said that open source projects will "win" or "lose" based purely on technical merit. Experiences from the LSB Project's interface standardization efforts indicate there are some concrete steps an open-source project producing interface libraries for general use can take to make the project more usable for a wider audience, leading to greater chance of widespread acceptance. Such projects have a reasonable chance of becoming standards, whether de-facto or by inclusion in formal specifications such as the LSB.

The evidence is that projects ready for large-scale use typically meet most of a set of criteria that include: demand; stable, well-documented interfaces; comprehensive interface and regression tests; an easily-deployed (portable) working implementation; and an appropriate choice of license. With the exception of demand, most of these criteria can be consciously worked towards. The paper will present some case studies of libraries that have successfully been incorporated into the LSB specification. It will also discuss some tools the LSB has developed that may help in describing public interfaces and developing tests, and discuss some ways in which portability of the code base can be improved.

1 Introduction

The Free Software and Open Source Software models present some unique concepts which seem to work best when the software is widely used and there's an active feedback loop to debug and improve the software. In order for this to be possible, it's important that some core requirements that apply to all software are attended to in this space as well: consistency and compatibility, documentation, and ease of use. If the software is to hard to deploy or make use of, the user base will remain small and the synergy which is so important to these projects will be harder to achieve.

While ease of use is a concept that is hard to measure for the developer as it means different things to different users, for an individual user it's pretty easy to tell when an application or library is not easy enough to use—it's painful to install, get running, or program to, making it hard to use it to solve the problem at hand. Where money did not change hands to obtain the software, the likely response will be to give up and look for a different solution, while what we as developers would rather have is feedback about the problems and suggestions for improvement. Often lacking a "marketing department" to drive requirements (whatever one may think of such a situation), this feedback is crucial to the open source process.

The Linux Standard Base (LSB) project

(`http://www.linuxbase.org`) aims to drive the creation of a consistent runtime environment for applications. Drawing from the experiences of the LSB project, we will examine a pair of issues, one on either side of the "runtime environment" boundary: building better libraries, and making applications (and libraries) easier to deploy.

2 Building Better Libraries

Libraries are an effective mechanism to provide for code reuse.

In Linux, libraries are normally provided as shared objects, although they may also be provided as static archives. Some libraries are foundational in that they are expected to be used by a broad variety of applications, such as the GNU C library, which is used by all programs; or the GNOME glib, which is used (directly or indirectly) by all graphical applications written to GNOME. Other libraries may export a programming interface specific to one application family such as libMagick for ImageMagick.

If a project produces libraries which are to be usable by others there are some particular issues that apply.

2.1 Stable Interfaces

A library provides certain programming interfaces which are available to programs to use (external), and probably also contains interfaces which are not intended to be used outside the library (internal). The set of external interfaces provides the Application Programming Interface (API). As programmers become familiar with the library, they will want the API to provide some stability so that they don't always have to recode their programs when the library is revised.

When a program is linked with a shared library, it will contain references to library interfaces which are resolved at runtime by the dynamic linker. The runtime instantiation of the library interface set provides the Application Binary Interface (ABI), and programmers will want the ABI to remain stable as well, or their programs may work incorrectly run against a different version of the shared library then it was originally linked against.

The dilemma for the library developer is that it's hard to get it (completely) right the first time. Bugs will be found, often the design will be found to be limiting or even incorrect, or the library may simply need to evolve to meet new needs. It would be terribly limiting to never be able to evolve the library just because users and developers demand stability. Fortunately, there are some techniques that can be used to make life a little easier.

A useful step is to identify the intended API and make sure that is all the library exports to programmers. If the API is designed as an abstraction layer distinct from the internal implementation, considerable freedom will be available to modify the library "under the covers" while still keeping the ABI stable. It's worth taking the time to design the API in this manner. It is also very useful if programmers cannot reach the internal routines which may need to change—experience has shown that if an interface can be found, someone will find a way to use it. A linker script can be used to export the desired symbols, hiding the others:

```
{
    global:
        lsbfoo;
    local:
        foo*;
};
```

A linker script is used when build-

ing a shared library by including the `-version-script=`*scriptname* directive in the gcc link line.

It's quite possible, however, that some interface in the ABI will need to change in an incompatible way. To provide for this, the symbols making up the ABI can be assigned versions, leaving the possibility of changing the version. The following example shows the use of a linker script which exports two routines and assigns them version `LSBLIB_1.0`:

```
LSBLIB_1.0 {
    global:
        lsbfoo;
        lsbbar;
    local:
        foo*;
};
```

If the symbol version is changed, old binaries won't run against the new library as the symbol version in those binaries will not be found; while binaries compiled against the new library will pick up the new symbol version. It is also possible—and may be desirable—to provide both the old and the new version of the interface in the newer library, this way old binaries can continue to run, while new binaries will be linked against the newer version of the interface by default, but could also be explicitly linked against the old version. The following example shows creating a new symbol version set which is inclusive of the previous one, only the `lsbfoo` interface will get the version tag `LSBLIB_1.1`.

```
LSBLIB_1.0 {
    lsbfoo;
    lsbbar;
};
LSBLIB_1.1 {
    lsbfoo;
} LSBLIB_1.0;
```

To make this work, the GNU linker is needed, and some special directives (`__asm__(".symver realname, alias, version");` are needed in the code, so that the old routine can be bound to the old version and the new code to the new version. The GNU linker documentation has more details on this.

If a lot of interfaces need to change incompatibly, it is better to change the major version of the library. The library version will be bound into binaries compiled against it. With major changes, multiple versions of the library can be provided, giving compatibility for old and new code.

In the LSB project, symbol versioning is used for those libraries which are already normally built that way, essentially the GNU libc set. Adding symbol versioning is a nice way to avoid breaking compatibility if a small number of interfaces have to be changed in incompatible ways. The LSB specification calls out specific library versions which must be provided by a conforming runtime, and where the symbols are versioned, the specific symbol versions. As conforming runtimes may have evolved the interfaces in the manner described, a trick is used for linking LSB conforming applications: a set of stub libraries has been constructed which contains only the LSB interfaces, with the versions required by the spec, and these are used for link-time symbol resolution.

2.2 API Documentation

A factor in how useful a library is is the quality of api documentation The documentation must describe in detail the programming interfaces available, with function calling and return conventions, boundaries, and error con-

ditions. This is the kind of information traditionally captured in the "manpage." The best measure of the quality of API documentation seems to be whether assertion-based tests (see next section) can be developed completely from the documentation, or whether the source code must be referred to fill in the details.

It is especially useful to use a tool to automate a part of this process. There are a number of tools that understand how to produce documentation from commented source code, one example would be *doxygen* (`http://www.doxygen.org`) although documentation generators seem to be more commonly used with higher-level languages (e.g. Javadoc for Java, Pydoc for Python, etc.)

The advantages of a generator approach is that the interface descriptions in the documentation don't depend on human transcription to get them right in the first place, and then don't go out of skew if the interfaces in the code ever change. It's particularly galling to try to code to an interface that does not work as documented.

The LSB specification has to date included mostly libraries which are already standardized at the API level—for example, the GNU C library is designed to be compatible with POSIX specification, so the LSB specification for the C library is able to reference this existing specification for almost all of the functional descriptions. As the LSB seeks to expand the base to other important libraries found on Linux systems, the API documentation will have to be imported by copy or by reference into the specification, so the existence of such documentation has become an LSB selection criteria.

The LSB itself has a slightly different documentation problem, as it has to capture an ABI description to describe the binary interface programs will see. A single API prototype or structure definition has been captured the way it will be seen on each of the (currently seven) architectures the LSB supports, based on things like data model (sizes of integers and pointers, for example). The symbol versions matching the interfaces must also be captured. All of this information is represented in a MySQL database which is browsable on the web (`http://www.linuxbase.org/dbadmin`) but which is also used to generate LSB header files, the stub libraries mentioned in the previous section, and the portion of the LSB specification that contains library listings, interface listings, and data definitions.

The database is also used to generate test code. Of particular note, the LSB generates two test programs, one to test the presence of the libraries and interfaces on a runtime, and another to test that an application uses only the libraries and interfaces in the specification. The data for these two programs is generated directly out of the specification database.

The LSB database schema and tools to extract data and build code (essentially a set of Perl scripts) are freely available for use by other projects, although they are probably mostly applicable to projects that support a large number of libraries and want to build similar test tools. They can be browsed from the LSB CVS tree (`cvs.gforge.freestandards.org`).

The summary is that while there's no magic to producing good documentation, it's important in producing a stable library that can be widely used. It's worth the time to see if some level of automation can help with the tasks, particularly if there are several areas that need to be kept in sync.

2.3 Interface Tests

Another area for consideration is detailed interface testing. Good tests allow checking

that interfaces perform as intended. The POSIX testing standard calls for such tests to be assertion-based, which means a written description of an intended behavior is produced, this is then used to develop the test case. The following example of an assertion is taken from the Open POSIX Test Suite (`http://sourceforge.net/projects/posixtest`:

mmap assertion 9 When MAP_FIXED is set in the flags argument, the implementation is informed that the value of pa shall be addr, exactly. If MAP_FIXED is set, mmap( ) may return MAP_FAILED and set errno to [EINVAL]. If a MAP_FIXED request is successful, the mapping established by mmap( ) replaces any previous mappings for the process' pages in the range [pa,pa+len].

Tests intended to operate at the source code level can be built and executed as part of the product build and are an effective way to catch regressions introduced during regular maintenance and development activity.

Binary level tests operate against an already built library, and are a way to test that a particular library is compatible with a particular API definition. Such tests increase the confidence of developers in the stability of the library.

In the LSB project, interface testing is the most important way of measuring a runtime against the LSB specification. However, the process of writing assertions and developing tests is not easy. It depends on a quality interface specification, good choice of testing methodologies, etc. There is little doubt that the most effective place for this work to take place is within the project itself. The source code file describing an interface can contain the interface, documentation, test assertions, and test code. All can be developed together without the kind of extra overhead incurred if each of the four items is developed separately by separate persons. The author is not aware of an existing toolkit which could automatically generate all of the necessary pieces from a single source file so endowed, but this would certainly make an interesting open source project of its own!

2.4 License Choice

The choice of license under which to release a library makes a considerable difference in who can use the libraries and how. This paper does not attempt a license recommendation as only the developer can know their own targets, needs and desires, which will guide the choice of license.

A Free Software license along the lines of the well-known GPL effectively restricts usage to programs under the same or compatible licenses. Such code cannot be used in closed source programs, even through dynamic linking, and also cannot be used by code under certain open source licenses that are not considered compatible, perhaps because they place some restriction on the user (one example might be a license that restricts usage to academic or personal use and disallows commercial use). The related LGPL license allows the use of the library by code of any sort through dynamic linking, but makes no similar provision for static linking. There are a variety of other licenses which grant greater or lesser freedoms in the ways the code may be used.

Some applications release code under dual licenses, for example a GPL-like license for those who can use it, and a separate license with commercial terms for those who cannot. It is also possible to release a package consisting of program code and library code with separate licenses for each.

As noted above, some licenses have compatibility clauses relating to how to code may be mingled with code under certain other licenses.

Various potential users of the code may have their own selection criteria that includes license choice. For example, the Debian project has a particular definition of "free" and consigns code which does not meet these criteria to the "nonfree" area.

Continuing with the use of LSB project experiences to illustrate, the LSB is concerned with functional interface descriptions, not with specific implementations. So the license of an *implementation* is not crucial—unless it's effectively the only implementation available, in which case it becomes a determining factor in practical use of the interface set.

An example may help clarify: the popular Qt toolkit was for a while the subject of some controversy in the open source community over its license terms, and a project was started to create an open source reimplementation of the Qt interface specification. When Qt licensing was changed to a dual license (one GPL-like, with a separate license for commercial developers) the open source reimplementation project was dropped as the problem people had with the previous license was resolved. However, the LSB project favors a "no strings attached" selection policy which suggests *against* the inclusion of a library where the only implementation doesn't allow a certain class of developers to just make use of the library in their code without arranging a commercial license.

The upshot is that choice of license needs to be considered very carefully.

3 Software Packaging and Deployment

The other major consideration this paper will examine is improving the accessibility of the software through producing a package that is easy to put into use. This discussion applies to both libraries and to complete applications.

The most common way to install software on Linux must be to install a distribution-specific package that has already been prepared. This has many advantages, as it's configured, compiled, and tested for that distribution, and the package will be tagged with dependencies so the user can determine what else needs to be installed to make it work. It will normally have security update patches made available should such become necessary.

Of course, not every package can be chosen for distribution packaging, and it's quite possible that an interested user for your software may find that a package is not available at all, or just not available for her distribution of choice. This should pose no problem since by definition the source code is available, and the software can simply be built from source. Unfortunately, in many cases the *simply* is a misnomer since there may be dependencies on other software, toolchain versions, etc. that may prove to be impediments.

3.1 How Not to Install Software

Although probably everyone reading this paper has had some negative experiences of their own with software installation, by way of example here is a condensed version of a situation that befell the author, and indirectly provided the motivation for recording these thoughts here:

At one point, I became interested in doing some transpositions on a piece of music, and I thought there must be a piece of software that would help with this. There are certainly commercial PC-centric applications that do this very well but there must be something open source as well. Some searching turned up a promising application named *noteedit*. Surprisingly, **rpmfind** told me that the one distribution for which a current version was pack-

aged was Mandrake, luckily my distribution of choice. The package did indicate Cooker, which is Mandrake's early-access build tree, but since it was only a couple of weeks after that last release, I assumed the Cooker could not have migrated too far and it would probably work.

After obtaining and installing the package, plus an attendant library package as well as another library (libtse) also needed, I installed and tried to run the package. Alas, it had been linked against a different C++ library version and so had references to some symbols that were not in my C++ library and thus was not runnable.

My next effort was to download the noteedit and tse library tarballs and attempt to build them from source. This was not a great success either, as the configuration scripts kept reporting fatal problems due to missing build headers and libraries, of course I had to correlate these back to the packages they would be installed by and install those. After several cycles I abandoned this approach and went to the third try, going back to rpmfind and pulling down the source, rather than binary, rpms and trying to build from source that way. This ultimately yielded a runnable binary although not without some further pain which involved tweaking the rpm specfiles. And this success still came because some Mandrake user contributed a build to the Cooker, which although it was for the wrong version (from my point of view) could be adjusted at the source level to work. What if I were running something different?

3.2 Binary Software Distribution

A project can certainly make their software easier to check out if there's a binary package available. Even if packaged by some distributions (and for many projects even this does not happen, especially early on), there's still the question of reaching users of other distributions.

The difficulty with a project building binary packages is deciding what to build for: there are an endless number of combinations of distributions and versions, and only a small fraction could be targeted. Further, this potentially puts the project into a "distro support" mode, that is worrying about oddities on the particular distro/version they have chosen to build for. A better solution seems to be to build a portable (distro-neutral) version.

Producing a portable binary package as an example has many advantages for a project:

- One package works on multiple kinds of systems
- Users interested in the software can get it running quickly
- Bugreports don't have to worry about the user's build environment
- Bugreports will be against a known set of configure and build options

There's still plenty of use for users building from source as well, including trying out combinations the developers have not tried, but the opportunity to come up quickly should broaden the base of potential users since not everybody wants to go through building from source.

Of course a really good build procedure from source—which clearly identifies dependencies, is also very valuable. Configure scripts have the unfortunate habit of quitting on the first "fatal error," which means after you satisfy that build dependency you try again and occasionally run into another, and then another. In frustration, the author once coded a configure script which issues warnings (AC_MSG_WARN) instead of errors (AC_MSG_FAIL), setting a flag which

is used to signal a fatal error at the end of the script. The author is not sure this hack is a "really good build procedure" however!

3.3 Using the LSB to Build Binary Packages

If a portable binary package is a target, the LSB provides a good model. The LSB specification describes a runtime platform, and also describes some things about how the package is delivered.

To build a portable binary, a relatively short set of rules needs to be followed:

- Link with the LSB runtime linker
- Use only LSB-specified libraries with the correct version
- Use only LSB-specified interfaces and symbol versions from those libraries
- All other interfaces must be supplied with the application

The runtime linker has a distinct name for LSB programs. For example, on the IA32 architecture, `ld-lsb.so.1` is used instead of `ld-linux.so.2`. This allows an implementation to do something different for LSB programs, such as resolving against libraries in a different directory. This capability is rarely used: most runtimes simply make the LSB linker name a symbolic link to the regular linker.

An application may only count on LSB libraries to be present on a conforming runtime, thus the restriction to link only with those libraries. If other libraries are needed, they can be statically linked, or provided in an application-supplied shared library. It is also possible to depend on *another* LSB-conforming package which supplies a shared library. Any such libraries must be constructed LSB conforming, which in practice means they need to watch their own dependencies on other libraries.

Some libraries may have more public interfaces than are described in the LSB specification. The most notable example is GNU libc. Even though these interfaces are likely to be present on every conforming system's version of those libraries, this is not required by the specification, and thus a conforming runtime may not count on them. For libraries which are symbol versioned, the binary must be linked against the symbol versions described in the specification.

While these rules are not terribly complex, it would be painful to modify build trees with many makefiles to apply them, so the LSB project supplies a compiler wrapper program `lsbcc` (as well as `lsbc++` for C++ programs) which applies the rules by fiddling with the compiler line before handing it off to the regular compiler, usually `gcc`.

If we get lucky, an LSB build can be as simple as:

```
CC=lsbcc ./configure
make
```

Of course it's not always this easy, and usually the problem is the use of libraries which are not in the LSB. The wrapper will actually turn references to non-LSB libraries into static links (the tool can be told to warn about this behavior as it's often useful to know what's happening behind your back). Sometimes static linking is a reasonable solution, sometimes packaging up the missing library in LSB mode is workable, and sometimes nothing will help but to lobby the LSB project to add the library—which will undoubtedly result in a polite request for help! The LSB still has quite a bit of evolving to do

and it's hoped that exposing it here will help identify the features which need to be added to future versions.

The other helpful aspect the LSB covers has to do with delivery of the software. Again, there are several areas:

- Portable format for the package
- Rules for where the package may place files
- Rules about names of packages to avoid clashes
- Special features such an an installer for startup scripts

The package format called out in the LSB specification is that used by the rpm package manager. This is a relatively portable format in that tools such as `alien` can convert these packages into other formats which can be handled by a system's package manager. There's no requirement that a runtime be rpm-based itself, and the only thing a package needs to (or is allowed to) depend on are provides for LSB modules (currently `lsb-core` and `lsb-graphics`) or other LSB packages.

It's also possible to deliver a package in other formats; in this case the rule is that the installer must be an LSB-conforming binary or an LSB-required command. A combination of a shell script and a tarball actually meet this requirement as both commands are required by the LSB specification. The use of other than the LSB package format is discouraged, however, as it makes it hard for system administrators to keep a view of what has been installed as would be the case if all software used the same package manager.

The File Hierarchy Standard (FHS) is imported into the LSB by reference and describes where an application may place files. To state these rules imprecisely, the package name serves as a tag, and it may install files into `/opt/`*tag*, `/etc/opt/`*tag*, and `/var/opt/`*tag*. This avoids clashes with distribution-provided packages and locally added software.

The naming of the package is also described by the LSB; essentially the rule is to register either a single package name, or a provider name, with the Linux Assigned Names and Numbers Authority or LANANA (`http://www.lanana.org`).

Finally, there are some provisions for things which don't fit into the above picture. For example, startup ("init") scripts and cron entries have to go in specific places. The LSB describes a special installer which may be invoked to create the links in the `/etc/rcX.d` directories.

With the specified behavior and tools, the LSB makes possible the creation of portable binary packages.

4 Summary

There are many considerations towards making software projects more popular. This paper has concentrated on only a small portion of those.

We have examined some issues towards making shared libraries useful. The assertion is that as a library becomes more Standard, whether that be a self-published standard or one promoted by a larger group or even a standards organization, it becomes easier for a wider audience to depend on it, software that uses it can be free of compatibility fears, and the larger community will lead to more and better feedback to continue to improve. Some steps that could help move a project towards such a state include developing solid interface specifications; stabilizing the interfaces as seen by

software through versioning, which leaves the freedom to continue to innovate while provide backward compatibility; and through comprehensive interface tests. We also looked at how choice of license plays into the usability of a library.

Another consideration towards usable software is lending the ability for potential users to get "on the air" with the software quickly, so they can evaluate it and see if it suits their needs without going through a lot of trouble. To that end, we looked at some benefits of projects delivering binary package in addition to source packages. The components of the LSB project which help in producing portable binary packages were also covered, to show how a project might be able to build a single binary package which helps the software become more accessible.

5 Disclaimer

The opinions expressed in this paper are those of the author and do not necessarily represent the position of Intel Corporation.

Repository-based System Management Using Conary

Michael K. Johnson, Erik W. Troan, Matthew S. Wilson
Specifix, Inc.
ols2004@specifixinc.com

Abstract

There have been few advances in software packaging systems since the creation of dpkg and RPM. Conary is being developed to provide a fresh approach to Open Source Software management and provisioning, one that applies new ideas from distributed software version control tools such as GNU arch and Monotone. Rather than concentrating on package files, Conary provides an architecture built around distributed repositories and change sets, and includes features designed to make branching and tracking Linux distributions simple operations.

The rise of distributions such as Fedora and Gentoo has moved the development of Linux distributions from small, tightly-connected groups to widely-dispersed groups of informal collaborators. These changes have brought to light many shortcomings of the dominant packaging metaphor. By providing version trees distributed across Internet-based software repositories, Conary allows these casual groupings of contributors to work together much more effectively than they can today.

1 Packaging Limitations

Traditional package management systems (such as RPM and dpkg) provided a major improvement over the previous regime of installing from source or binary tar archives. However, they suffer from a few shortcomings, and some of these shortcomings are felt more acutely as the Internet and the Open Source communities have developed and expanded. The authors' experience with the shortcomings of current package management systems strongly motivated Conary's design.

1.1 Branching

Traditional package management systems use simple version numbers to allow the different package versions to be sorted into "older" and "newer" packages, adding concepts such as **epochs** to work around version numbers that do not follow the packaging system's ideas of how they are ordered. While the concepts of "newer" and "older" seem simple, they break down when multiple streams of development are maintained simultaneously using the package model. For example, a single version of a set of sources can yield different binary packages for different versions of a Linux distribution. A simple linear sorting of version numbers cannot represent this situation, as neither of those binary packages is newer than the other; the packages simply apply to different contexts.

1.2 Package Repository Limitations

Traditional package management systems provide no facilities for coordinating work between independent repositories.

- Repositories have version clashes; the same version-release string means different things in different repositories. Repositories can even have name clashes—the same name in two different repositories might not mean the same thing.
- There is no way to identify which distribution, let alone which version of the distribution, a package is intended and built for.

For example, is the `aalib-1.4.0-5.1fc2.fr` package newer than the `aalib-1.4.0-0-fdr.0.8.rc5.2` package? One is from the freshrpms repository, and the other is from the fedora.us repository. Which package should users apply to their systems? Does it depend on which version of which distribution they have? How are the two packages related? Are they related at all?

This is not really a problem in a disconnected world. However, when you install packages from multiple sources, it can be hard to tell how to update them—or even what it means to update a package. You have to rely on your memory of where you fetched a package from in order even to look in the right repository. Once you look there, it is not necessarily obvious which packages are intended for the particular version of the distribution you have installed. Automated tools for fetching packages from multiple repositories have increased the number of independent package repositories over the past few years, making the confusion more and more evident.

The automated tools helped exacerbate this problem (although they did not create it); they have not been able to solve it because the packages do not carry enough information to allow the automated tools to do so.

1.3 Source Disconnected from Binaries

Traditional package management does not closely associate source code with the packages created from it. The binary package may include a hint about a filename to search for to find the source code that was used to build the package, but there is no formal link contained in the packages to the actual code used to build the packages.

Many repositories carry only the most recent versions of packages. Therefore, even if you know which repository you got a package from, you may not be able to access the source for the binary packages you have downloaded because it may have been removed when the repository was upgraded to a new version. (Some tools help ameliorate this problem by offering to download the source code with binaries from repositories that carry the source code in a related directory, but this is only a convention and is limited.)

1.4 Namespace Arbitrary and Unmanaged

Traditional package management does not provide a globally unique mechanism for avoiding package name, version, and release number collisions; all collision-avoidance is done by convention and is generally successful only when the scope is sufficiently limited. Package dependencies (as opposed to file dependencies) suffer from this; they are generally valid only within the closed scope of a single distribution; they generally have no global validity.

It can also be difficult for users to find the right packages for their systems. Both SUSE and Fedora provide RPMs for version 1.2.8 of the iptables utility; if a user found release 101 from

SUSE and thought it was a good idea to apply it to Fedora Core 2, they would quite likely break their systems.

1.5 Build Configuration

Traditional packaging systems have a granular definition of architecture, not reflecting the true variety of architectures available. They try to reduce the possibilities to common cases (`i386`, `i486`, `i586`, `i686`, `x86_64`, etc.) when, in reality, there are many more variables. But to build packages for many combinations means storing a new version of the entire package for every combination built, and then requires the ability to differentiate between the packages and choose the right one. While some conventions have been loosely established in some user communities, most of the time customization has required individual users to rebuild from source code, whether they want to or not.

In addition, most packaging systems build their source code in an inflexible way; it is not easy to keep local modifications to the source code while still tracking changes made to the distribution (Gentoo is the most prominent exception to this rule).

1.6 Fragile Scripts

Traditional package management systems allow the packager to attach arbitrary shell scripts to packages as metadata. These scripts are run in response to package actions such as installation and removal. This approach creates several problems.

- Bugs in scripts are often catastrophic and require complicated workarounds in newer versions of packages. This can arbitrarily limit the ability to revert to old versions of packages.
- Most of the scripts are boilerplate that is copied from package to package. This increases the potential for error, both from faulty transcription (introducing new errors while copying) and from transcription of faults (preserving old errors while copying).
- Triggers (scripts contained in one package but run in response to an action done to a different package) introduce levels of complexity that defy reasonable QA efforts.
- Scripts cannot be customized to handle local system needs.
- Scripts embedded in traditional packages often fail when a package written for one distribution is installed on another distribution.

2 Introduction to Conary

Conary provides a fresh approach to open source software management and provisioning, one that applies new ideas from distributed configuration management tools such as GNU arch and monotone. Rather than concentrating on separate package files as RPM and dpkg do, Conary uses networked repositories containing a structured version hierarchy of all the files and organized sets of files in a distribution.

This new approach gives us exciting new features:

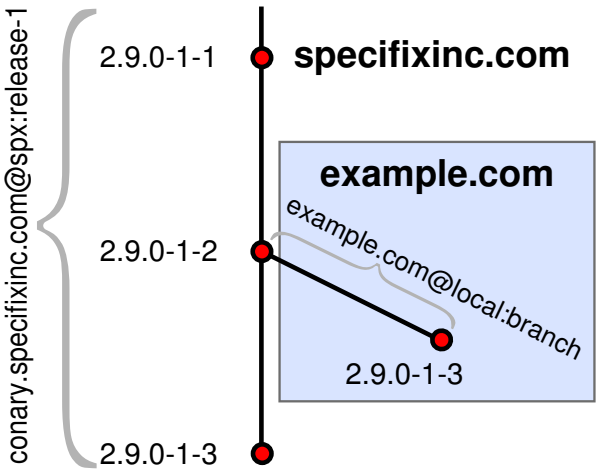
- Conary allows you to maintain and publish changes, both by allowing you to create new branches of development, and by helping track changes to existing branches of development while maintaining local changes.
- Conary intelligently preserves local changes on installed systems. An update

will not blindly obliterate changes that you have made on your local system.

- Conary can duplicate local changes made on one machine, installing those changes systematically on other machines, thereby easing provisioning of large sets of similar or identical systems.

3 Distributed Version Tree

Conary keeps track of versions in a tree structure, much like a source code control system. The difference between Conary and many source code control systems is that Conary does not need all the branches of a tree to be kept in a single place. For example, if Specifix maintains a kernel at `specifixinc.com`, and you, working for `example.com`, want to maintain a branch from that kernel, your branch could be stored on your machines, with the root of that branch connected to the tree stored on Specifix's machines.



3.1 Repository

Conary stores everything in a **distributed repository**, instead of in package files. The repository is a network-accessible database that contains files for multiple packages, and multiple versions of these packages, on multiple development branches. Nothing is ever removed from the repository once it has been added. In simple terms, Conary is like a source control system married to a package system.

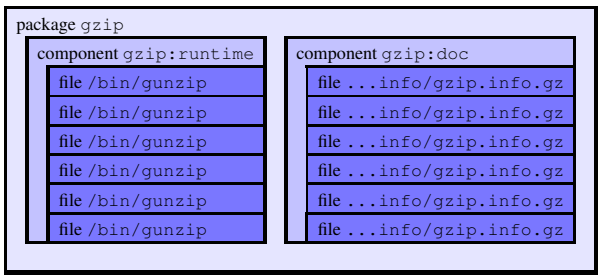
3.2 Files

When Conary stores a file in the repository, it tracks it by a unique file identifier rather than by name. Among other things, this allows Conary to track changes to file names—the file name is merely one piece of metadata associated with the file, just like the ownership, permission, timestamp, and contents. If you think of the repository as a filesystem, the file identifier is like an inode number.

3.3 Troves, Packages, and Components

When you build software with Conary, it collects the files into **components**, and then collects the components into one or more **packages**. Components and packages are both called **troves**. A trove is (generically) a collection of files or other troves.

A package does not directly contain files; a package references components, and the components reference files. Every component's name is constructed from the name of its container package, a `:` character, and a suffix describing the component. Conary has several standard component suffixes: `:source`, `:runtime`, `:devel`, `:docs`, and so forth. Conary automatically assigns files to components during the build process, but you can overrule its assignments and create arbitrary component suffixes as appropriate.



One component, with the suffix `:source`, holds all source files (archives, patches, and build instructions); the other components hold files to be installed. The `:source` component is not included in any package, since several different packages can be built from the same source component. For example, the `mozilla:source` component builds the packages `mozilla`, `mozilla-mail`, `mozilla-chat`, and so forth. The version structure in Conary's repositories always tells exactly which source component was used to build any other component.

3.4 Labels and Versions

Conary uses strongly descriptive strings to compose the version and branch structure. The amount of description makes them quite long, so Conary hides as much of the string as possible for normal use. Conary version strings act somewhat like domain names, in that for normal use you need only a short portion. For example, the version `/conary.specifixinc.com@spx:trunk/2.2.3-4-2` can usually be referred to and displayed as `2.2.3-4-2`. The entire version string uniquely identifies both the source of a package and its intended context. These longer names are globally unique, preventing any confusion.

Let's dissect the version string `/conary.specifixinc.com@spx:trunk/2.2.3-4-2`. The first part, `conary.specifixinc.com@spx:trunk`, is a **label**. It holds three pieces of information:

- **The repository host name:** `conary.specifixinc.com`
- **Namespace:** `spx` A high-level context specifier that allows branch names to be reused by independent groups. Specifix will maintain a registry of namespace identifiers to prevent conflicts. Use `local` for branches that will never need to be shared with other organizations.
- **Branch name:** `trunk` This is the only portion of the label that is essentially arbitrary; and will be defined by the owner of the namespace it is part of.

The next part, `2.2.3-4-2`, contains the more traditional version information.

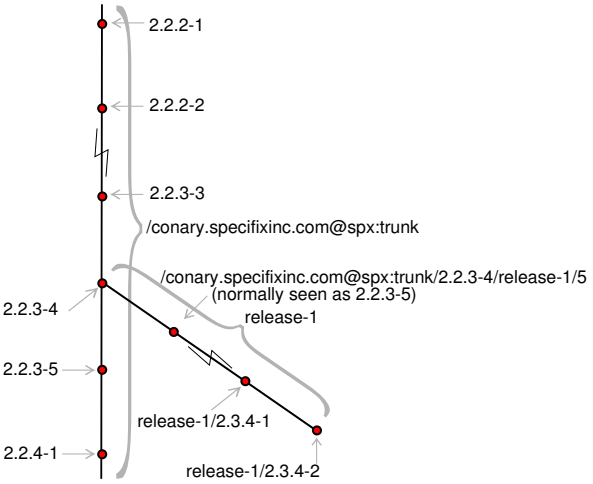
- **Upstream version string:** `2.2.3` This is the version number or string assigned by the upstream maintainer: Conary never interprets this string in any way; the only check it does is whether it is the same or different. It is there primarily to present useful information to the user. Conary never tries to determine whether one upstream version is "newer" or "older" than another. It makes these decisions based on the ordering specified by the repository's version tree.
- **Conary revision:** `4-2` This pair is composed from:
  - **Source build serial number:** `4` Incremented each time a version of the sources with the same upstream version string is checked in. It is similar to the release number used by traditional packaging systems.
  - **Binary build serial number:** `2` How many times this particular source package has been built. This number is not provided for source

packages, because it is meaningless in that context.

Conary describes branch structure by appending version strings, separated by a `/` character. The first step to make a release is to create a branch that specifies what is in the release. Let's create the `release-1` branch off the trunk: `/conary.specifixinc.com@spx:trunk/2.2.3-4/release-1` (note that because we are branching the source, there is no binary build number).

In this branch, `release-1` is a label. The label inherits the repository and namespace of the node it branches from; in this case, the full label is `conary.specifixinc.com@spx:release-1`

The first change that is committed to this branch can be specified in somewhat shortened form as `/conary.specifixinc.com@spx:trunk/2.2.3-4/release-1/5` Because the upstream version is the same as the node from which the branch descends, the upstream version may be omitted, and only the Conary version provided. Users will normally see this version expressed as `2.2.3-5`, so this string, still long even when it has been shortened by elision, will not degrade the user experience.



Labels also have an unusual property: a single label can reference *multiple* branches. To demonstrate why this is useful, let's look at the glib library. Like many other libraries, glib is designed to allow more than one version to be installed on the system at once. Older programs require glib 1.2; newer programs require glib 2. All new releases of glib 1.2 are compatible with programs written and compiled for older versions of glib 1.2; all new releases of glib 2 are compatible with programs written and compiled for older versions of glib 2. They are not, however, compatible with each other; a program compiled for glib 1.2 will certainly not run with glib 2. Therefore, a complete system requires that glib 1.2 and glib 2 both be installed.

Packaging systems often solve this problem by naming the packages differently, putting part of the version number into the name of the package (i.e. `glib` and `glib2`). This works, but it dilutes the revision history that the repository model provides.

By contrast, Conary solves this problem by allowing labels to apply to more than one branch. To see how, we will start by "going back in time" and looking at the version string for glib on the trunk with only glib 1.2 packaged: `/conary.specifixinc.com@spx:trunk/1.2.10-19-3`

Now, we want to add glib 2 to the repository. We want to have a branch for continuing maintenance of maintain glib 1.2, though, so let's create that first: `/conary.specifixinc.com@spx:trunk/1.2.10-19-3/glib1.2`

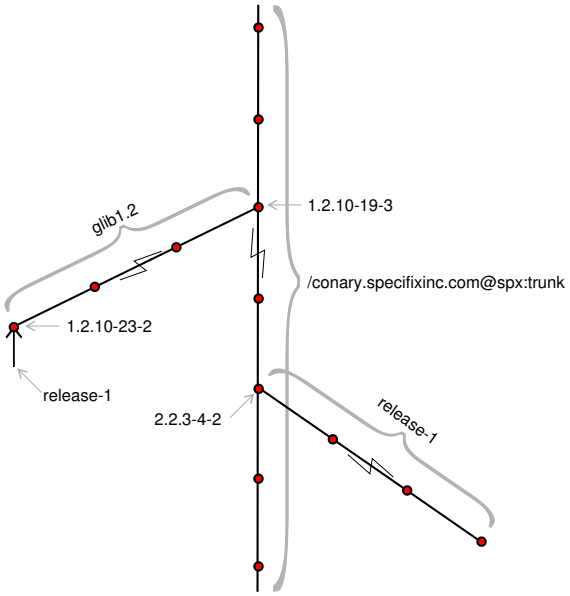
Now, we upgrade the trunk to glib 2: `/conary.specifixinc.com@spx:trunk/2.2.3-1-1`

Having maintained both glib 1.2 and glib 2 for a while, we decide that we want to make

our first release. We will label every package in the release, including two versions of glib: `/conary.specifixinc.com@spx:trunk/2.2.3-4-2/release-1/4-2` and `/conary.specifixinc.com@spx:trunk/1.2.10-19-3/glib1.2/23-2/release-1/23-2`

The label `conary.specifixinc.com@spx:release-1` now specifies *both* versions of glib. Therefore, if you install `glib conary.specifixinc.com@spx:release-1`, you will get both versions of glib.

Normally, the label to install will be set by installation scripts, and Conary will automatically install both versions of glib. Of course, updates will be applied only when there is a change; an update to glib 1.2 does not affect glib 2. In other words, it "just works" without you having to worry about it.



3.5 Shadows

The most powerful way to manage local changes is (of course) to build changes from source code. Conary makes this possible in two ways. One way is a simple branch, just like you would do with any source code control software. Unfortunately, this is not always the best solution.

Imagine a stock 2.6 Linux kernel packaged in Conary, being maintained on the `/linux26` branch (we have omitted the repository host name and namespace identifier from the label for brevity) of the `kernel:source` package, currently at version `2.6.5-1` (note that because it is a source package, there is no binary build number). You have one patch that you want to add relative to that version, and then you wish to track that maintenance branch, keeping your own change up to date with the maintenance branch, and building new versions as you go.

If you create a new branch from `/linux26/2.6.5-1`, say `/linux26/2.6.5-1/mybranch`, all the work you do is relative to that one version. Creating a new branch does not help you, because the new branch goes off in its own direction from one point in development, rather than tracking changes. Therefore, when the new version `/linux26/2.6.6-1` is committed to the repository, the only way to represent that version in your branch would be to manually compare the changes and apply them all, bring your patch up to date, and commit your changes to your branch. This is time-consuming, and the branch structure does not represent what is really happening in that case.

Conary introduces a new concept: a **shadow**. A shadow acts primarily as a repository for local changes to a tree. A shadow tracks changes relative to a particular upstream version string and source build serial number. Therefore, you cannot change the upstream version of the package—though you can apply any patch you like. (In order to change the upstream version of the package, you would need to create a branch rather than a shadow.) The

name of a shadow is the name of the branch with `//shadowname` appended; for example, `/branch//shadow`. The whole branch is shadowed, so if `/branch/1.2.3-3` and `/branch//shadow` exist, then so does `/branch//shadow/1.2.3-3`, regardless of whether `/branch/1.2.3-3` existed at the time the shadow was created. Similarly, if `/branch/1.2.3-3/rel1/1.2.3-3` exists, then so does `/branch//shadow/1.2.3-3/rel1/1.2.3-3`.

Both `/branch/1.2.3-3` and `/branch//shadow/1.2.3-3` refer to exactly the same contents. Changes are represented with a dotted source build serial number, so the first change to `/branch/1.2.3-3` that you check in on the `/branch//shadow` shadow will be called `/branch//shadow/1.2.3-3.1`.

So, to track changes to the `/linux26` branch of the `kernel:source` package, you create the `mypatch` shadow of the `/linux26` branch, `/linux26//mypatch`, and therefore `/linux26//mypatch/2.6.5-1` now exists. Commit a patch to the shadow, and `/linux26//mypatch/2.6.5-1.1` exists. Later, when the `linux26` branch is updated to version `2.6.6-1`, you merely need to update your shadow, modify the patch to apply to the new kernel source code if necessary, and commit the your new changes to the shadow, where they will be named `/linux26//mypatch/2.6.6-1.1`. You can use the shadow branch name `/linux26//mypatch` just like you can use the branch name `/linux26`; you can install that branch, and `conary update` will use the same rules to find the latest version on the shadow that it uses to find the latest version on the branch.

3.6 Flavors

Conary has a unified approach to handling multiple architectures and modified configurations. It has a very fine-grained view of architecture and configuration. Architectures are viewed as an instruction set, including settings for optional capabilities. Configuration is set with system-wide flags. Each separate architecture/configuration combination built is called a **flavor.**

Using flavors, the same source package can be built multiple times with different architecture and configuration settings. For example, it could be built once for `x86` with `i686` and `SSE2` enabled, and once for `x86` with `i686` enabled but `SSE2` disabled. Each of those architecture builds could be done twice, once with `PAM` enabled, and once with `PAM` disabled. All these versions, built from exactly the same sources, are stored together in the repository.

At install time, Conary picks the most appropriate flavor of a component to install for the local machine and configuration (unless you override Conary's choice, of course). Furthermore, if two flavors of a component do not have overlapping files, and both are compatible with the local machine and configuration, both can be installed. For example, library files for the `i386` family are kept in `/lib` and `/usr/lib`, but for `x86_64` they are kept in `/lib64` and `/usr/lib64`, so there is no reason that they should not both be installed, and since the AMD64 platform can run both, it is convenient to have them both installed.

4 Changesets

Just as source code control systems use patch files to describe the differences between two versions of a file, Conary uses **changesets** to

describe the differences between versions of troves and files. These changesets include information on how files have changed, as well as how the troves that reference those files have changed.

These changesets are often transient objects; they are created as part of an operation and disappear when that operation has completed. They can also be stored in files, however, which allows them to be distributed like the packages produced by a classical package management system.

Applying changesets rather than installing new versions of packages allows Conary to update only the parts of a package that have changed, rather than blindly reinstalling every file in the package.

Besides saving space and bandwidth, representing updates as changes has another advantage: it allows merging. Conary intelligently merges changes not only to file contents, but also to file metadata such as permissions.

This capability is very useful if you wish to maintain a branch or shadow of a package—for example, keeping current with vendor maintenance of a package, while adding a couple of patches to meet local needs.

Conary also keeps track of local changes in essentially the same way, preserving them. When, for example, you add a few lines to a configuration file on an installed system, and then a new version of a package is released with changes to that configuration file, Conary can merge the two unless there is a direct conflict (unusual but possible). If you change a file's permission bits, those changes will be preserved across upgrades.

Conary supports two types of change sets:

- The differences between two versions in a repository
- The complete contents of a version in a repository (logically, this is the difference between nothing at all and that version)

In the first case, where Conary is calculating the differences between two different versions, the result is a **relative changeset**. In the second case, where Conary is encoding the entire content of the version, the result is an **absolute changeset**. (If you use an absolute changeset to upgrade to the version provided in the absolute changeset, Conary internally converts the changeset to a relative changeset, thereby preserving your local changes.) Absolute changesets are convenient ways of distributing versions of troves and files to users who have various versions of those items already installed on their systems. In practice, they can be distributed just like package files created by traditional package management systems.

Conary can do two things with one of these changesets. It can update a system, either directly from a changeset file, or by asking the repository to provide a changeset and then applying that changeset. It can also store existing changesets in a repository. This capability will be used in the future to provide repository mirroring, and it can also be used to move changes from one repository to a branch in a different repository.

4.1 Representing Local Changes

Conary can also generate a **local changeset** that is a relative changeset showing the difference between the repository and the local system for the version of a trove that is installed. You can distribute a local changeset to another machine in two ways:

- You can distribute it to other machines

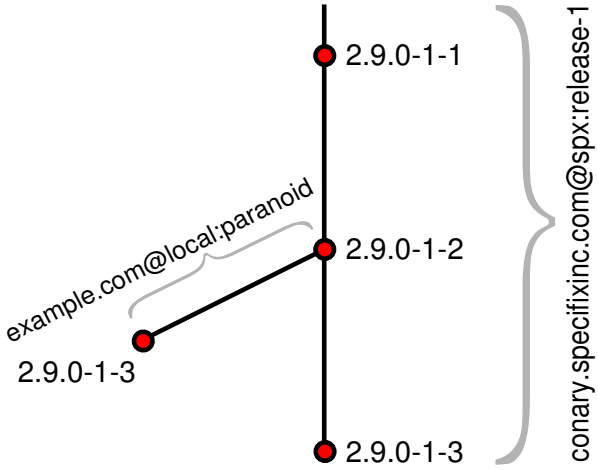
with the same version of the trove in question installed.

- You can commit the local changeset to a branch of a repository, and then update to that branch on target machines.

There is an important distinction between the two cases. In the first case, the machine that applies the changeset will act as if those changes had been made by the system's administrator; since those changes are not in a repository they are not versioned. In the second case, however, the machine gets those changes by updating the trove to the branch that contains those changes, and it can continue to track changes from that branch.

For example, assume that you have machines with troves from branches labeled `conary.specifixinc.com@spx:rel1` installed, and you have some local changes that you want to distribute to a group of machines. Let's say that after updating to version `2.9.0-1-2` of `tmpwatch`, you want to change the permissions of the /usr/sbin/tmpwatch binary because you are paranoid: `chmod 100 /usr/sbin/tmpwatch` Now, you record that change in a local changeset; that changeset is relative to `2.9.0-1-2`, and describes your local changes.

You then commit your local changeset to the `conary.example.com@local:paranoid` branch in your local repository. Now, on all the machines in the group, you can update `tmpwatch conary.example.com@local:paranoid`. Each machine will now look in the `conary.example.com` repository on the `paranoid` branch if you simply run `conary update tmpwatch`. This means that if you make further changes to the `tmpwatch` package, you can commit those changes to the `paranoid` branch on the `conary.example.com` repository, and each of the machines will update to the latest version you have committed to that branch. Every time a new version of `tmpwatch` is released on the `conary.specifixinc.com@spx:rel1` branch, you will have to apply the changeset to the `conary.example.com@local:paranoid` branch before the machines with your `paranoid` branch installed will update their copies of `tmpwatch`.



If rather than maintaining a branch, you merely want to distribute some changes that are local to the group of machines, you do not want to commit the local changeset to the repository. Instead, you want to copy the changeset file (let's call it paranoid.ccs) to each machine and run `conary localcommit paranoid.ccs` on each machine. Now, your change to permissions applies to each system, but `conary update tmpwatch` will still look at `conary.specifixinc.com@spx:rel1` and Conary will apply updates to `tmpwatch` from `conary.specifixinc.com@spx:rel1` without additional work required on your part, and it will preserve the change to the permissions of the /usr/sbin/tmpwatch binary on each machine.

Both ways of managing local change are useful. Committing local changesets to a repository is best for systems with entirely centralized management policy, where all sys-

tem changes must be cleared by some central agency, whereas distributing local changesets is best when individual systems are expected to autonomously update themselves asynchronously.

4.2 Merging

When Conary updates a system, it does not blindly obliterate all changes that have been made on the local system. Instead, it does a three-way merge between the currently installed version of a a file as originally installed, that file on the local system, and the version of the file being installed. If an attribute of the file was not changed on the local system, that attribute's value is set from the new version of the package. Similarly, if the attribute did not change between versions of the package, the attribute from the local system is preserved. The only time conflicts occur is if both the new value and the local value of the attribute have changed; in that case a warning is given and the administrator needs to resolve the conflict.

For configuration files, Conary creates and applies context diffs. This preserves changes using the the widely-understood diff/patch process.

4.3 Efficiency

Conary is more efficient than traditional packaging systems in several ways.

- By utilizing relative changesets whenever possible, Conary uses less bandwidth.
- By modifying only changed files on updates, Conary uses less time to do updates, particularly for large packages with small changes.
- By using a versioned repository, Conary saves space because unchanged files are stored once for the whole repository, instead of once in each version of each package.
- By enabling distributed repositories, Conary
  - saves the time it takes to maintain a modified copy of an entire repository, and
  - saves the space it takes to store complete copies of an entire repository.

4.4 Rollbacks

Because Conary updates systems by applying changesets, and because it is able to follow changes on the local system intrinsically, it easily supports **rollbacks**. If requested, Conary can store an inverse changeset that represents each **transaction** (a set of trove updates that maintains system consistency, including any dependencies) that it commits to the local system. If the update creates or causes problems, the administrator can ask Conary to install the changeset that represents the rollback.

Because rollbacks can affect each other, they are strictly stacked; you can (in effect) go backward through time, but you cannot browse. You have to apply the most recent rollback before you apply the next most recent rollback, and so forth.

This might seem like a great inconvenience, but it is not. Because Conary maintains local changes vigorously, including merging changes to configuration files, and because all the old versions you might have installed before are still in the repositories they came from, you can "update" to older versions of troves and get practically the same effect as rolling back your upgrade from that older version.

Applying rollbacks can be more convenient when you know that you want to roll back the

previous few transactions and restore the system to the state it was in, say, two hours ago. However, if you want to be selective, "upgrading" to an older version is actually more convenient than it would be to try to select a rollback transaction that contains the change you have in mind.

5 Other Concepts

5.1 Dynamic Tags

In place of the fragile script metadata provided by traditional package management systems, Conary introduces a concept called **dynamic tags**. Files managed by Conary can have sets of arbitrary text tags that describe them. Some of these tags are defined by Conary (for example, `shlib` is reserved to describe shared library files that cause Conary to update /etc/ld.so.conf and run `ldconfig`), and others can be more arbitrary. (In order to allow tag semantics to be shared between repositories, it is likely that Specifix will host a global tag registry in the future.)

By convention, a tag is a noun or noun phrase describing the file; it is not a description of what to do to the file. That is, *file* is-a *tag*. For example, a shared library is tagged as `shlib` instead of as `ldconfig`. Similarly, an info file is tagged as `info-file`, not as `install-info`.

Conary can be explicitly directed to apply a tag to a file, and it can also automatically apply tags to files based on a **tag description** file. A tag description file provides the name of the tag, a set of regular expressions that determine which files the tag applies to, the path of the **tag handler** program that Conary runs to process changes involving tagged files, and a list of actions that the handler cares about. Conary then calls the handler at appropriate times to handle the changes involving the tagged files.

Actions include changes involving either the tagged files or the tag handlers. Conary will pass in lists of affected files whenever it makes sense, and will coalesce actions rather than running all possible actions once for every file or component installed.

The current list of possible actions is:

- Tagged files have been installed or updated; Conary provides a list of all installed or updated tagged files.
- Tagged files are going to be removed; Conary provides a list of all tagged files to be removed.
- Tagged files have been removed; Conary provides a list of filenames that were removed.
- The tag handler itself has been installed or updated; Conary provides a list of all tagged files already installed on the system.
- The tag handler itself will be removed; Conary provides a list of all the tagged files already installed on the system to facilitate cleanup.

Because the tag description files list the actions they handle, the tag handler API can be expanded easily while maintaining backward compatibility with old handlers.

Avoiding duplication between packages by writing scripts once instead of many times avoids bugs in scripts. Practically speaking, it avoids whole classes of common bugs that cause package upgrades to break installed software, and even more importantly from a provisioning standpoint, bugs that would cause rollbacks to fail. It makes it much easier to fix

bugs when they do occur, without any need for "trigger" scripts that are often needed to work around script bugs in traditional package management. It also allows components to be installed across distributions—as long as they agree on the semantics for the tags, the actions taken for any particular tag will be correct for the distribution on which the package is being installed.

Calling tag handlers when they have been updated makes recovery from bugs in older versions of tag handlers relatively benign; Conary needs to install only a single new tag handler with the capability to recover from the effects of the bug. Older versions of packages with tagged files will use the new, fixed tag handler, which allows you to revert those packages to older versions as desired, without fear of re-introducing bugs created by old versions of scripts.

Furthermore, storing the scripts as files in the filesystem instead of as metadata in a package database means:

- they can be modified to suit local system peculiarities, and those modifications will be tracked just like other configuration file modifications;
- they are easier for system administrators to inspect; and
- they are more readily available for system administrators to use for custom tasks.

5.2 Groups and Filesets

There are two other kinds of troves that we did not discuss when we introduced the trove concept: groups and filesets.

Filesets are troves that contain only files, but those files come from components in the repository. They allow custom re-arrangements of any set of files in the repository. (They have no analog at all in the classical package model.) Each fileset's name is prefixed with `fileset-`, and that prefix is reserved for filesets only.

Filesets are useful primarily for creating small embedded systems. With traditional packaging systems, you are essentially limited to installing a system, then creating an archive containing only the files you want; this limits the options for upgrading the system. With Conary, you can instead create a fileset that references the files, and you can update that fileset whenever the components on which it is based are updated, and use Conary to update even very thin embedded images.

The desire to be able to create working filesets was a large motive for using file-specific metadata instead of trove-specific metadata wherever possible. For example, files in filesets maintain their tags, which means that exactly the right actions will be taken for the fileset. If Conary had package scripts like traditional package managers, it would be impossible to automatically determine which parts (if any) of the script should be included in the fileset. (As already discussed, scripts have other problems that tags solve; this is just another one of the architectural reasons that tags are preferable to scripts.)

Groups are troves that contain any other kind of trove, and the troves are found in the repository. (The task lists used by apt are similar to groups, as are the components used by anaconda, the Red Hat installation program.) Each group's name is prefixed with `group-`, and that prefix is reserved for groups only.

Groups are useful for any situation in which you want to create a group of components that should be versioned and managed together. Groups are versioned like any trove, including packages and components. Also, a group ref-

erences only specific versions of troves. Therefore, if you install a precise version of a group, you know exactly which versions of the included components are installed; if you update a group, you know exactly which versions of the included components have been updated.

If you have a group installed and you then erase a component of the group without changing the group itself, the local changeset for the group will show the removal of that component from the group. This makes groups a powerful mechanism administrators can use to easily browse the state of installed systems.

The relationship between all four kinds of troves is illustrated as follows:

Troves		built from	
		source	repository
contain	files	component	fileset
	troves	package*	group

*packages contain only components

Groups and filesets are built from `:source` components just like packages. The contents of a group or fileset is specified as plain text in a source file; then the group or fileset is built just like a package.

This means that groups and filesets can be branched and shadowed just like packages can. So if you have a local branch with only one modified package on it, and then you want to create a branch of the whole distribution containing your package, you can branch the group that represents the whole distribution, changing only one line to point to your locally changed file. You do not have to have a full local branch of any of the other packages or components.

Furthermore, when the distribution from which you have branched is updated, your modification to the group can easily follow the updates, so you can keep your distribution in sync without having to copy all the packages and components.

6 Further Work

An alpha release of Conary is now available from `http://www.specifixinc.com`, along with a Linux distribution built with Conary. While these releases allow users and developers to begin making use of Conary's features, there is significant work remaining.

The shadow design discussed in this paper has not yet been implemented.

Conary does not yet resolve dependencies. Although some dependency information is already generated and tracked on a per-file basis, no effort is made to ensure that those dependencies are resolved when components are installed.

As Conary and Conary-based distributions become more popular, there will be a need for both repository caches and repository mirrors. While some preliminary design work has been done for each of these, no implementation work has begun.

The implementation of flavors is preliminary, especially in regards to configuration settings. While limited testing has been done with troves built for varying architectures and Specifix's build scripts implement some configuration settings, Conary does not yet properly select the flavor to install on a system.

Conclusion

Conary was designed to address many of the limitations of the traditional packaging

metaphor. The enormous growth in the Linux developer base over the past decade has shown that packaging systems do not scale well to multiple repositories with conflicting content, and can make it difficult for large numbers of developers to coordinate package releases.

Conary provides flexible branching, which enables it to find both binaries and sources anywhere on the Internet, and allows the local administrator to preserve local changes and create local development branches of those packages. By providing a name space separator as part of the branch names, Conary allows many groups to use the same tool while building a single distributed version tree, without any formal collaboration between the groups.

Innovations such as shadows and versioning groups of packages and files (allowing those container objects themselves to be branched and shadowed) significantly reduce the difficulty of maintaining customized Linux distributions. Instead of being forced to accept complete responsibility for all aspects of the distribution, developers can now concentrate on maintaining just their changes. Those changes are represented in a concise way that can track upstream changes to the entire distribution.

Conary is designed to enable a loosely-coupled, Internet-based collaborative approach to building Linux distributions. By making branching and shadowing inexpensive operations that can change almost any aspect of a Linux system, we hope members of the Linux community will be able to build the Linux distribution they want, rather than use one that is merely close enough.

On-demand Linux for Power-aware Embedded Sensors

Carl Worth, Mike Bajura, Jaroslav Flidr, and Brian Schott
USC/Information Sciences Institute
{cworth,mbajura,jflidr,bschott}@east.isi.edu

Abstract

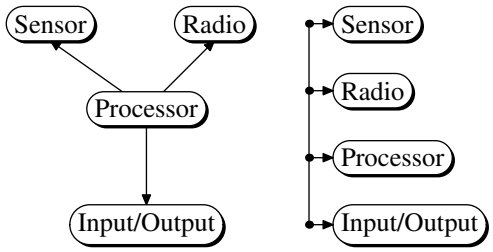
We introduce a distributed sensor architecture which enables high-performance 32-bit Linux capabilities to be embedded in a sensor which operates at the average power overhead of a small microcontroller. Adapting Linux to this architecture places increased emphasis on the performance of the Linux power-up/shutdown and suspend/resume cycles.

Our reference hardware implementation is described in detail. An acoustic beamforming application demonstrates a 4X power improvement over a centralized architecture.

1 Introduction

Traditional sensor platform architectures are based on a hub-and-spoke model with peripherals clustered around a central processor as shown in Figure 1(a). In this model, the lower-bound of total system power is set by the lowest active mode of the central processor which must be continually active to broker peripheral operations.

System power is typically reduced by using less-capable processors or microcontrollers in place of the central processor. Although sensor activity is mostly infrequent and bursty with low average computational requirements, peak processing requirements can still be quite high.



(a) Hub-and-spoke (b) Distributed

Figure 1: Alternate sensor node architectures

Within a system design, this creates tension between the desire for the high-performance processing capability of a larger processor and the low-power operation of a smaller one.

This tension is further complicated by the observation that while many large processors require significantly more power than small ones when inactive, they also often provide significantly more power-efficient computation when active. Another tradeoff in this design space weighs the strength of development and debugging tools, such as Linux, available for larger processors versus the constrained programming environments available for small ones.

Replacing the hub-and-spoke architecture with a distributed model as shown in Figure 1(b) can decouple processing from peripheral operation and create a system that combines the strengths of both large and small processors.

In this model, processor and peripherals become autonomous modules that are each powered independently. High-performance processing can be made available when needed, but without increasing the lower-bound of total system power. Low average system power can be achieved by operating for a majority of the time in extremely low-power modes with only essential modules active.

This distributed architecture places Linux in an unconventional role as a peer module rather than as a central processor. This emphasizes the performance of the power-up/shutdown and suspend/resume cycles as keys for achieving low average system power.

The remainder of this paper is organized as follows. Section 2 describes several popular research and commercial sensor platforms. Section 3 recounts the design challenges we faced as we built a reference sensor node with autonomous modules. As usual, real-world issues forced difficult engineering decisions. Section 4 details the modules we have built so far.

Our hardware is in a more complete state than our software. We have identified some aspects of the behavior of Linux that we need to investigate more fully. These issues are discussed in Section 5. Section 6 contains power results we have obtained with a vehicle tracking algorithm. Finally, Section 7 draws conclusion and Section 8 describes areas for future work.

2 Related Work

Applied research in wireless sensor networks has made use of a variety of platforms with varying processing capabilities and power requirements, but almost always with a hub-and-spoke model. Several platforms are described below in order from more-capable, higher-power platforms to less-capable, lower-power platforms.

2.1 PC/104

PC/104 [3] is a well-supported specification for PC-compatible, embedded systems consisting of stacking modules. Cerpa et al [2] chose PC/104 systems for their "high end" sensors in a tiered deployment for habitat monitoring. Their cited reasons for choosing PC/104 include the ability to run PC-compatible software (i.e. Linux) and the wide spectrum of available PC/104 modules.

PC/104 provides great flexibility and the power requirements are lower than that of desktop PCs. However, at 1-2 Watts per module[3], and with most sensors requiring at least two modules, this platform requires too much power for many sensor applications.

2.2 Embedded StrongARM devices/PDAs

Off-the-shelf devices based on low-power, embedded processors such as the Intel SA-1110 or PXA25x offer another convenient platform for sensor network research. Compared to x86-class processors, these processors offer a significant power savings along with a reduction in maximum clock rate and the absence of a hardware floating-point unit.

Representative devices of this class include the HP iPAQ, the CerfCube[4], and the Crossbow Stargate[14]. These devices are extensible via Compact Flash, Bluetooth, etc. but have less flexibility than PC/104. And while the power requirements of these systems can be less than a comparable PC/104 stack, Mainwaring et al[9] found that at 2.5W active power, the power usage of the CerfCube was excessive for long-term use in a sensor network.

Several research sensors have been developed with architectures similar to these off-the-shelf platforms. These include the μAMPS[10] and WINS[1] nodes. For example, the WINS node

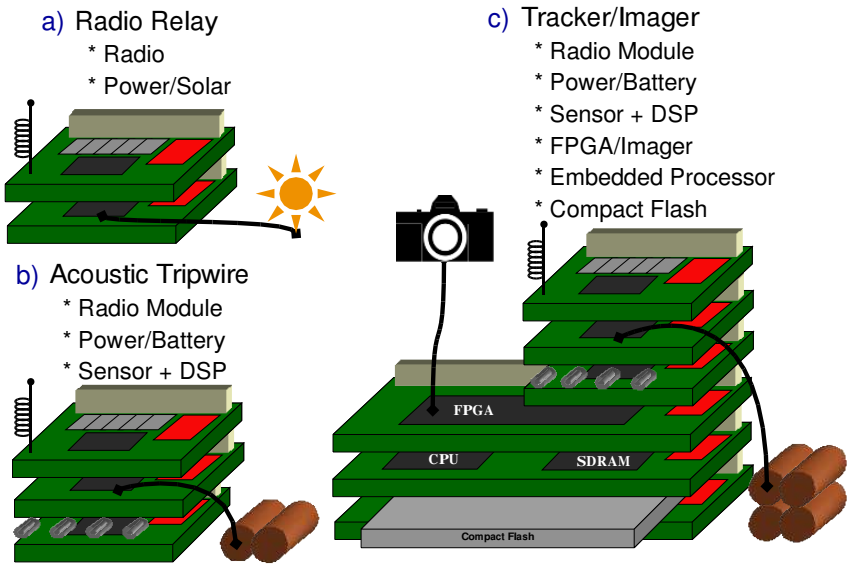


Figure 2: Power-aware sensor concept

has a central 133MHz StrongARM SA-1100 processor along with radio and sensor peripherals. As measured by Raghunathan et al[12] the WINS node operates in the range of 360-1080 mW and can also be placed into a 64 mW sleep mode.

2.3 Motes

An example of a very low-power sensor architecture is that of the Berkeley Motes[6, 7, 8]. Current Motes are based around a central microcontroller (MCU) such as the ATmega 90LS8535, an 8-bit MCU with 128KB Flash and 8KB SRAM. The Mote includes a radio and has serial connections and 10-bit analog ADC ports to various sensors on expansion modules. Typical power consumption for this sensor when active is in the 10-100 mW range. Sleep power is about 60 μW.

Mote-class sensors demonstrate that a wide variety of low-bandwidth sensing applications can be accomplished with very small processors and with very little memory. The limitation of these systems is encountered when an application doesn't fit within the memory and processing footprint of the MCU. High bandwidth sensor processing is beyond the capabilities of these small-scale sensors and there is little room for expansion.

3 Implementation

Our primary system design goal was to construct a family of interchangeable processor, sensor, and communication modules that can be mixed and matched according to the application requirements. Ideally, our architecture would be able scale from simple sensors as shown in Figure 2(a) and Figure 2(b), to complex sensors as shown in Figure 2(c) without having to learn and port to a new platform at every scale.

Other goals were driven by practical experiences of using other platforms in the field. Rapid prototyping is an important concern. The ability to create testbeds using COTS peripherals is a strength of platforms such as PC/104. Availability of Linux device drivers

and a friendly programming environment were strong motivators in our implementation decisions. Data collection is an important step in sensor network algorithm development. We wanted lots of data storage and data networking options in our new platform.

Primary constraints on the system design include size and power. We targeted the size of the Berkeley Mote, while still supporting a Linux-capable processor in the stack. In the end, the size was dictated by the minimum footprint of a Compact Flash socket and our chosen stack connector. Power in our system needs to be able to scale from 1 mW to a few Watts. The low power target limited many of our implementation choices.

An early design decision was how the autonomous modules would communicate. Interfaces such as ethernet were quickly dismissed due to power requirements. In small embedded devices, the most power-efficient communication is available with hardware-supported interfaces such as Serial Peripheral Interface (SPI), Controller Area Network (CAN), Universal Asynchronous Receiver Transmitter (UART), and Inter-Integrated Circuit (IIC or I2C). Each of these interfaces has strengths and weaknesses. I2C supports multi-master operation, but has a limit of 100kbps on most devices. SPI has faster transfers, (up to a few megabits per second), but has limited multi-master support in most devices and little flow control. UART is widely supported, but requires clock agreement on both interfaces. CAN is multi-master, but the bus drivers in the supporting devices are relatively high power, (since CAN is designed for long cables).

We decided to support the three major interface standards (I2C, SPI, and UART). We allocated six 8-bit channels on our connector and specified two preferred channels for each standard interface. In order to prevent bus contention and power leakage when modules are off, all modules have bus isolation switches between themselves and the connector. We also introduced a separate I2C control network for module discovery and coordination of the switches.

A small microcontroller (MCU) is standard on most modules to control the bus isolation switches, a power switch, and any module-specific functions. The MCUs are intended to be always on when the node is operating, (with a power overhead as low as .05 mW). They network with each other over I2C to facilitate module discovery and coordinate access to the channels using a common messaging protocol.

Figure 3 contains a diagram of the features common to each module, as well as an optional processor expansion bus so that high-speed, high-power peripherals, (USB, Compact Flash, LCD, and AC97 audio), can be used in the stack or removed for low-power operation.

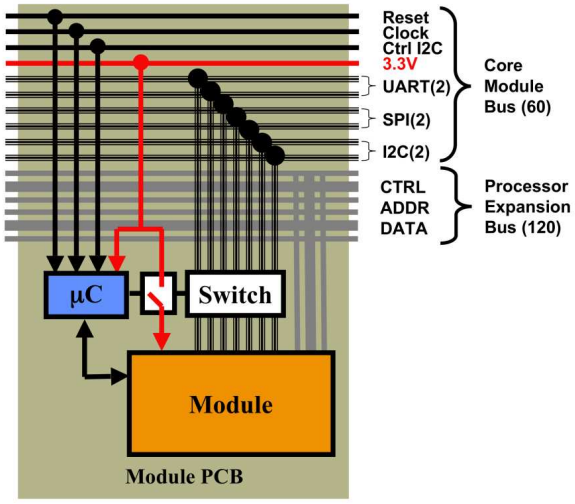


Figure 3: Power-aware module diagram

4 Available Modules

Our hardware modules are small boards approximately 6.5×4.5cm (2.5×1.75"), with a

180-pin connector on either side. So far, we have designed, built, and tested 4 modules, 3 of which are shown in Figures 4 and 5.

PXA A module including an Intel PXA255 XScale processor, 64MB of SDRAM and 32MB of Flash. This board supports dynamic voltage scaling, an active clock rate range of 100-400MHz, and a 33MHz idle mode. An SA-1111 coprocessor provides support for USB master and two Compact Flash cards. All interface lines are routed to the stack connector.

ADC A four-channel, 12-bit analog-to-digital converter module. The MCU has sufficient memory for a dedicated 7kB sample buffer. Basic signal processing can be performed by the local MCU or samples can be efficiently transmitted over an SPI interface to the PXA module for advanced processing. An important point is that the PXA255 processor can be off or suspended during data sampling.

IOB The power & I/O board is the only required module in the stack. It provides the primary power supply and contains most of the digital I/O connectors, (USB master and slave, SPI, I2C, and UART).

FPGA This module was developed as an emulation board for a low-power DSP being developed at MIT[15], but is interesting for other high-performance applications. It contains a Xilinx Virtex-II 3000 FPGA, 2MB of synchronous SRAM, and 32MB of SDRAM. This module operates as a coprocessor on the memory bus of the PXA255 and supports the two SPI channels on the stack.

We also have a Compact Flash (CF) board, two of which can be placed into the stack. This board connects to the processor expansion

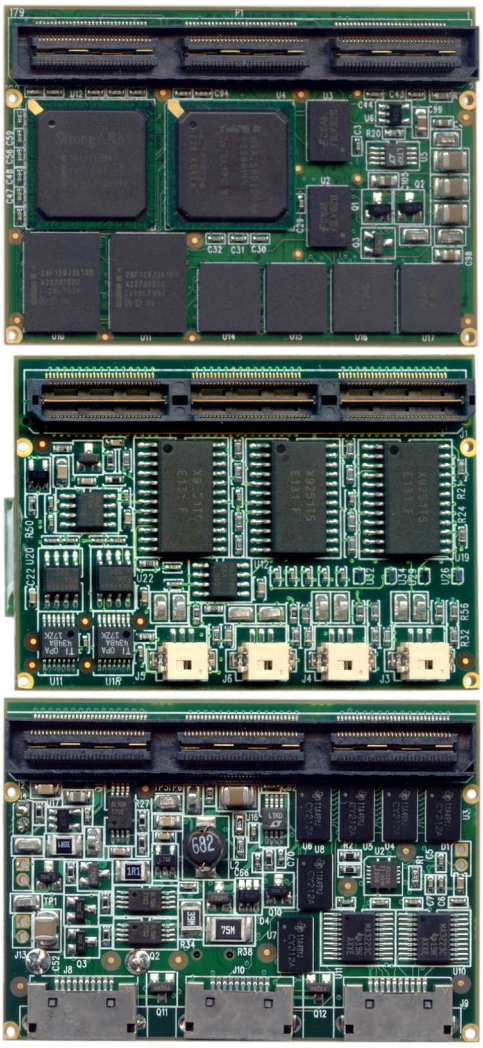


Figure 4: PXA, ADC, and IOB modules at actual size

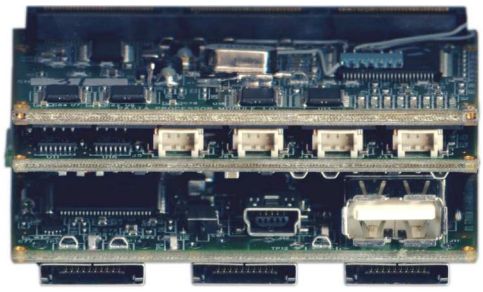


Figure 5: Stacked modules

bus so that it acts as a daughter-board of the PXA module rather than an independent module. Figure 6 shows a stack consisting of IOB and PXA modules along with a CF board populated with a CF ethernet adapter.

Figure 6: Stack including CF board

Table 1 shows design-time estimates of the power consumed by each module for various operational modes. In the "off" mode everything on the module is powered off except for the power-control MCU. The PXA module has the widest operational power range due to the the range of processor clock rates and various possibilities in processor and memory utilization. Figure 7 provides more details of the how the PXA module power can scale from 0.05 mW to 1.5 W. The innermost portion of this diagram includes figures for the overhead of power conversion and the 32kHz MCU on the IOB module.

Module	Mode	Power
PXA	off	0.05 mW
PXA	suspended	2.5 - 7.5 mW
PXA	active	150 - 1530 mW
IOB	active	0.1 mW
ADC	off	0.05 mW
ADC	active	40 mW

Table 1: Power modes for various modules

5 Impact on Linux

In a conventional hub-and-spoke model, Linux runs on a central processor and manages some number of peripheral devices. In contrast, our distributed architecture places Linux on an autonomous module which is a peer to other modules. Any other module might request a power transition of the Linux module, from off to powered, from suspended to active, etc.

The efficiency of these power transitions is a critical component of the average system power. The distributed platform is designed to achieve low average system power through aggressive duty-cycling of high-powered components. The time and energy spent during power-mode transitions is overhead that must be amortized, imposing limits on practical duty cycles that can be used.

We are currently using Linux version 2.4.21 with the standard ARM and PXA patches as well as customizations for our PXA module. The user-level software distribution is derived primarily from the handhelds.org[11] Familiar distribution. Our reference sensor application (see Section 6) does not turn the PXA module off, but does suspend/resume the processor aggressively to achieve active operation for a few milliseconds once per second. The time spent during suspend/resume is divided between time spent in driver callbacks and time spent in the kernel proper. Table 2 shows the times we have measured for these transitions on our PXA module.

Transition	Time
Suspend drivers	507 ms
Suspend kernel	13 ms
Resume kernel	78 μs
Resume drivers	25 ms

Table 2: Linux transitions with driver callbacks

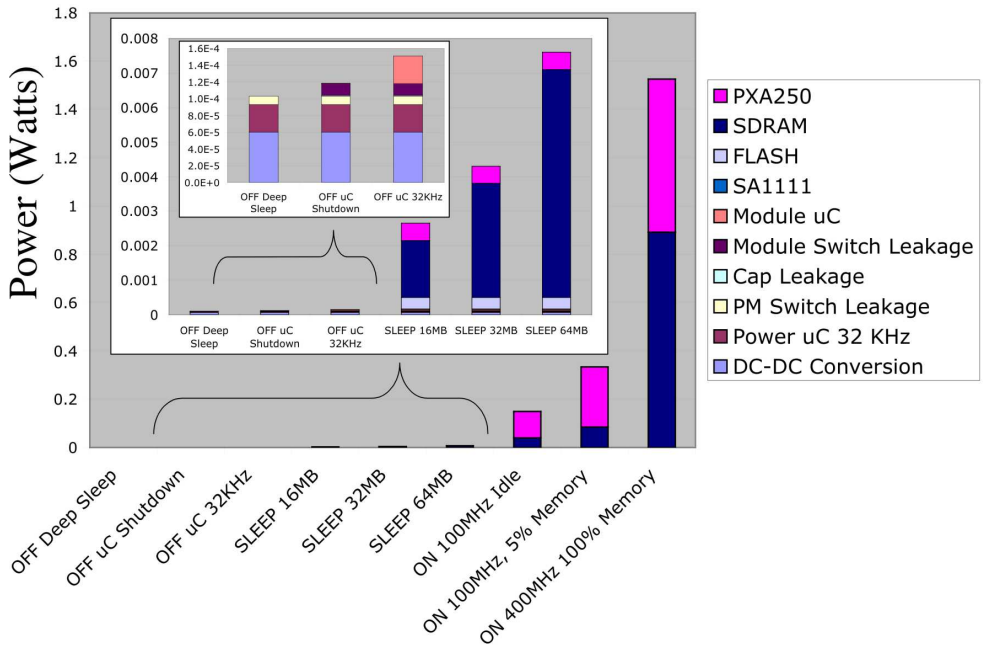


Figure 7: PXA module power states

We expect the suspend/resume transitions of the Linux kernel itself to behave in a symmetric fashion. However, we measured a very responsive resume time of 78 μs and a much slower suspend time of 13 ms. We do not yet have a complete explanation for why the suspend process is so much slower, although we have accounted for a 1 ms delay that is caused by the MCU on our PXA board, and therefore not a feature of the standard Linux kernel.

A much more significant problem is the time spent in the power management callbacks of various subsystems and drivers. A suspend time of 520ms spells disaster for an application such as the one described in Section 6.

We quickly tracked down the source of this long suspend time to the USB OHCI driver (hcd). An ill-fated decision in our design was the choice of the SA-1111 coprocessor. One difficulty we encountered was that we had to provide our own suspend/resume callbacks as they do not exist for the SA-1111-based hcd in the standard kernel. To simplify the task, we have ported the PCI-based OHCI code which contains a call to `mdelay (500)` along with the comment, "Suspend chip and let things settle down a bit." This single 500ms delay accounts for over 98% of the time required to suspend drivers. We suspect that this constant can be safely reduced so that much of the time lost during suspend can be recovered. Even so, USB-related timeouts, etc. are on the order of milliseconds—orders of magnitude more than the time required by the kernel.

Clearly, this poses a serious problem for applications with a high duty cycling requirement. We are currently working around the long driver suspend times by simply removing drivers for non-essential devices and subsystems, (such as SA-1111), prior to running an application with a restricted power budget. This allows the convenience of things such as using a USB 802.11 adapter during development and debugging without the long suspend times of the USB drivers during execution.

6 Results

We implemented a vehicle tracking algorithm using 4-channel acoustic beamforming. In this application, data is continually sampled at a rate of 1kHz, but signal processing only needs to be performed at a maximum rate of 1Hz.

In previous work[13], this algorithm was implemented on a successor to the WINS node, (a hub-and-spoke platform with an Intel SA-1110 processor). Although efficient signal processing software was developed, system power savings were modest since the processor had to remain active (yet mostly idle) at all times simply to drive data collection.

We have ported the algorithm to the distributed platform described in this paper. On this platform, the signal processing for one second's worth of data can be completed in 3 ms by the PXA255 processor running at 100MHz. Since this is a newer processor than the SA-1110 of the WINS platform, direct comparison of power numbers between the two platforms would be unfair. Instead, we estimate the power needed for two implementations of the algorithm on the distributed node.

The first version is intended to behave as if in a hub-and-spoke system. The processor remains active at all times to store samples into main memory. The second version takes advantage of the distributed nature of the platform. Linux on the PXA module is suspended as much as possible while the ADC module continues to sample and buffer data. This approach adds the overhead needed to suspend/resume Linux and to transfer data from the ADC module to the PXA module over the SPI channel. The amount of data to be transferred is 8192 bytes, (4 channels $*$ 1024 samples/s $*$ 2 bytes/sample $*$ 1 s). The SPI transfer rate is 1.8 Mbps yielding a total transfer time of 36.4 ms.

We measured the power consumed by the PXA module at two different stages in the algorithm. During active computation the PXA module consumes 528 mW. When mostly idle, (e.g. when transferring 1kHz data from the ADC module), it consumes 370 mW. We have not yet measured the average power consumed during the suspend or resume transitions, but we use an estimate of 370 mW. This estimate should be conservative as the actual power usage should ramp down to less than 10 mW during the transition.

Combining these measurements with estimates from Table 1 and the time measurements from Table 2, we compute the total energy spent to compute one result per second. From this we can determine the average power necessary for the complete algorithm. These results are given for both versions of the algorithm in Tables 3 and 4.

Module/Mode	Power	Time	Energy
IOB active	0.1 mW	1 s	0.1 mJ
ADC active	40 mW	1 s	40.0 mJ
PXA active	528 mW	3 ms	1.6 mJ
PXA idle	370 mW	997 ms	368.9 mJ
Estimated energy per second			410.6 mJ
Estimated system power: 411 mW			

Table 3: Hub-and-spoke power requirements for beamforming

Module/Mode	Power	Time	Energy
IOB active	0.1 mW	1.0 s	0.1 mJ
ADC active	40 mW	1.0 s	40.0 mJ
PXA suspended	7.5 mW	948 ms	7.1 mJ
PXA resuming	370 mW	78 μs	28.9 μJ
PXA transferring	370 mW	36.4 ms	13.5 mJ
PXA processing	528 mW	3 ms	1.6 mJ
PXA suspending	370 mW	13 ms	4.8 mJ
Estimated energy per second			95.9 mJ
Estimated system power: 96 mW			

Table 4: Distributed power requirements for beamforming

7 Conclusion

The 96 mW beamforming result marks a success for the distributed sensor platform—a 4X power reduction over the 411 mW required for the hub-and-spoke platform. This shows that it is possible to take advantage of 32-bit, Linux processing without average power exceeding the 10-100 mW range of a less-capable sensor based on a 8-bit microcontroller, (i.e. a Mote).

8 Future Work

The field of power-aware sensing is rich, and we have only just begun to explore the possibilities, even within our own platform. Many applications require a much smaller power budget than the 96 mW result we have demonstrated. Our long-term goal is to design sensors capable of operating entirely from scavenged energy, (e.g. solar), which requires operation in the range of 1 mW[5].

We are currently building a low-power "tripwire" module which will implement single-channel acoustic vehicle detection. This will allow the PXA module to be completely off when a vehicle is not present. We anticipate that this will allow beamforming within a power budget as low as 10 mW.

As mentioned in Section 5, there remains a fair amount of engineering and research with regards to the role of Linux within a distributed sensor. This includes reducing the time required in the power management callbacks of all relevant drivers as much as possible. An additional task is to move from Linux version 2.4 to 2.6. The dynamic nature of the new unified device model in 2.6 should make a natural fit with a platform consisting of autonomous modules that can be powered on and off at any point.

Additionally, new power-scheduling research could better take advantage of the wide range of power modes in this platform. For example, Linux could monitor the frequency and duty cycles of the power-up/shutdown and suspend/resume cycles. It would then be possible to provide intelligent feedback into these cycles based on the relative overhead of each transition. This work would complement current efforts that dynamically adjust voltage and clock rate based on system load.

9 Availability

This work has been developed as part of the *Power-Aware Sensing Tracking and Analysis (PASTA)* project, but we hope that it will be useful to researchers and hobbyists with a wide range of applications. To that end we are working toward making the hardware modules available at cost. Further details will be made available at the PASTA website, `http://pasta.east.isi.edu`. All of the software developed under PASTA is also available there under the terms of the GNU General Public License (GPL).

10 Acknowledgements

This research is sponsored by the Defense Advanced Research Projects Agency and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F33615-02-2-4005. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, the Air Force Research Laboratory, or the U.S. Government.

References

[1] Jonathan R. Agre, Loren P. Clare, Gregory J. Pottie, and Nikolai P. Romanov. Development platform for self-organizing wireless sensor networks. In *Proceedings of Aerosense*, pages 257–268. International Society of Optical Engineering, 1999.

[2] Alberto Cerpa, Jeremy Elson, Deborah Estrin, Lewis Girod, Michael Hamilton, and Jerry Zhao. Habitat monitoring: Application driver for wireless communications technology. In *ACM SIGCOMM Workshop on Data Communications in Latin America and the Caribbean*, April 2001.

[3] PC/104 Consortium. PC/104 Embedded-PC Modules, 2004. `http://www.pc104.org/`.

[4] Intrinsic Corporation. CerfCube embedded strongarm system. `http://www.intrinsyc.com/products/cerfcube/`.

[5] L. Doherty, B.A. Warneke, B. Boser, and K.S.J. Pister. Energy and performance considerations for smart dust. In *International Journal of Parallel and Distributed Sensor Networks*, 2001.

[6] Jessica Feng, Farinaz Koushanfar, and Miodrag Potkonjak. System-architectures for sensor networks issues, alternatives, and directions. In *International Conference on Computer Design: VLSI in Computers and Processors*, page 226. IEEE, 2002.

[7] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S.J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[8] J.M. Kahn, R.H. Katz, and K.S.J. Pister. Next century challenges: mobile networking for smart dust. In *Proceedings of the fifth annual ACM/IEEE international conference on Mobile computing and networking*, pages 271–278. ACM Press, 1999.

[9] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless sensor networks for habitat monitoring. In *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97. ACM Press, 2002.

[10] R. Min, M. Bhardwaj, S. Cho, A. Sinha, E. Shih, A. Wang, and A. Chandrakasan. An architecture for a power-aware distributed microsensor node, 2000.

[11] The Handhelds.org Project. Handhelds.org (web document). `http://www.handhelds.org`.

[12] V. Raghunathan, C. Schurgers, S. Park, and M. Srivastava. Energy aware wireless microsensor networks, 2002.

[13] R. Riley, S. Thakkar, J. Czarnaski, and B. Schott. Power-aware acoustic beamforming. In *Proceedings of the Fifth International Military Sensing Symposium*, 2002.

[14] Crossbow Technology. Stargate xscale processor platform. `http://www.xbow.com`.

[15] A. Wang and A. Chandrakasan. Energy-efficient dsps for wireless sensor networks, 2002.

Virtually Linux

Virtualization Techniques in Linux

Chris Wright
OSDL
chrisw@osdl.org

Abstract

Virtualization provides an abstraction layer mapping a virtual resource to a real resource. Such an abstraction allows one machine to be carved into many virtual machines as well as allowing a cluster of machines to be viewed as one. Linux provides a wealth of virtualization offerings. The technologies range in the problems they solve, the models they are useful in, and their respective maturity. This paper surveys some of the current virtualization techniques available to Linux users, and it reviews ways to leverage these technologies. Virtualization can be used to provide things such as quality of service resource allocation, resource isolation for security or sandboxing, transparent resource redirection for availability and throughput, and simulation environments for testing and debugging.

1 Introduction

Virtualization has many manifestations in computer science. At the simplest level it can be viewed as a layer of abstraction which helps delegate functionality—typically handling resource utilization. This abstraction layer often helps map a *virtual* resource to a *physical* or *real* resource. The virtual resource is then presented directly to the resource consumer obscuring the existence of the real resource. This can be implemented through hardware[1] or software [16, 21, 19], may include any subset of a machine's resources, and has a wide variety of applications. Such usages include machine emulation, hardware consolidation, resource isolation, quality of service resource allocation, and transparent resource redirection. Applications of these usage models include virtual hosting, security, high availability, high throughput, testing, and ease of administration.

It is interesting to note that differing virtualization models may have inversely correlated proportions of virtual to physical resources. For example, the method of carving up a single machine into multiple machines—useful in hardware consolidation or virtual hosting—looks quite different from a single system image (SSI) [15]—useful in clustering. This paper primarily focuses on providing multiple virtual instances of a single physical resource, however, it does cover some examples of a single virtual resource mapping to multiple physical resources.

Modern processors are sufficiently powerful to provide ample resources to more than one operating environment at a time. Of course, timesharing systems have always allowed for concurrent application execution. However, there are many ways in which these concurrent applications may effect one another. Because the

[1] For example, an MMU helps with translation of virtual to physical memory addresses.

operating system provides access to shared resources such as the CPU, memory, I/O devices, file system, network, etc., one application's use of the system's resources may effect another's. This can have negative effects on both quality of service and security. Carving a single machine into a series of independent virtual machines can eliminate the quality of service and security issues.

At the same time, modern computing systems, inclusive of both hardware and software, are subject to failures and scalability problems. The application of virtualization can hide these shortcomings by distributing computing loads across a cluster of physical systems which may present a single *virtual* interface to an application.

The remainder of this paper is organized as follows. Section 2 presents a variety of virtualization techniques. Section 3 gives a detailed comparison of some of these techniques. Section 4 presents conclusions drawn from the comparisons.

2 Virtualization Techniques

The term "virtual" is one of those horribly overloaded terms in computing. For the purpose of this paper, we will define virtualization as a technique for mapping virtual resources to real resources. These virtual resources are then used by the resource consumer, fully decoupled from any real resources that may or may not exist. As discussed in Section 1 the virtual resource may be some or all of a system's resources.

There are many virtualization techniques available to Linux users, and these techniques can be leveraged through a variety of applications. The techniques reviewed in this paper fall roughly into two categories: *complete* virtualization, Section 2.1, which provides all or nearly all of a system's resources; and *partial* virtualization, Section 2.2, which provides only a specialized subset of resources. Understanding the different techniques helps identify which technique is the best given a specific set of requirements.

2.1 Complete Virtualization

Complete virtualization techniques involve creating a fully functional and isolated virtual system which can support an OS. This instance of the OS may have no indication that it is not being run natively on real hardware, and it is often referred to as the *guest*. Host-based virtual systems run atop an existing *host* OS. Others run atop a thin supervisor which just helps multiplex resources to the virtual systems. Typically the host machine is capable of supporting many concurrent virtual systems, each with its own guest OS instance. These virtual systems can be created by simple software emulation or by more complicated methods. These types of complete virtualization techniques differ in terms of efficiency and performance, portability for either the host or the guest OS, and functional goals.

The rest of this section is organized at follows. Section 2.1.1 is a look at pure software processor emulation techniques. Section 2.1.2 looks at the virtual OS approach taken by User-mode Linux. Finally, Section 2.1.3 reviews techniques using virtual machines and virtual machine monitors.

2.1.1 Processor Emulation

Processor emulation is one technique used to provide complete virtualization. In this case, the CPU is emulated entirely in software. Additionally, it is typical to find peripheral devices such as keyboard, mouse, VGA, network, timer chips, etc. supported by the emulator.

The emulation is done in user-space software, which makes it a rich environment for debugging system level software running in the emulator. Also, this technique has great advantages for portability at the cost of runtime performance. The emulator may easily run on various hardware architectures, as all emulation is done in software. Further, because these are hardware emulators, there is often little to no restriction on what OS software can be executed. However, the dynamic translation required to translate hardware instructions from the emulated processor to the native processor is pure overhead, and thus can be hundreds of times slower than native instructions [22].

An exhaustive survey of processor emulators is beyond the scope of this paper. Here we take a brief look at a few of the prevalent emulators often used to host virtual Linux instances:

- QEMU CPU emulator
- Bochs
- PearPC
- Valgrind.

QEMU [21] is a CPU emulator that does dynamic instruction translation. It maintains a translation cache for efficiency. It can be used as a user-mode emulator which will run Linux binaries compiled for the CPU that QEMU is emulating regardless of the host platform. Also, QEMU can do full system emulation, which allows one to boot an OS on the QEMU emulated CPU. While the QEMU user-mode is available for many architectures, the complete system emulation mode is only available for x86 and is in testing for PowerPC. The x86 emulator provides all the PC peripheral devices needed to boot an OS, and can easily run an unmodified Linux kernel. It also features debugger support which can be quite useful for debugging a Linux kernel.

Bochs [2] is an IA-32[2] CPU emulator. It does dynamic compilation and is often cited as being rather slow [3]. Similar to QEMU, Bochs provides full platform emulation sufficient for running an OS, and it can boot an unmodified Linux kernel. While Bochs is highly portable, it targets only the IA-32 processor.

PearPC [17] is a PowerPC CPU emulator. The generic PearPC CPU emulator can be ported and is slow. PearPC also provides a PowerPC CPU emulator that is specific to x86 hosts. This version uses dynamic instruction translation and caching techniques (similar to QEMU) which improve the speed substantially.

Valgrind [14] is worthy of mentioning as it is both a very useful tool and contains an x86-to-x86 just-in-time (JIT) compiler, thus emulating the x86 CPU. However, this tool has been historically used like Purify [10] as a memory checker, and not typically used for bringing up a virtual instance of Linux on the emulated CPU[3]. It handles user-space emulation, but not full system emulation. Valgrind is developed as an instrumentation framework around the JIT, so it can been expanded to be a general purpose “meta-tool for program supervision.” [14]

2.1.2 Virtual OS

The virtual OS is rather specific to User-mode Linux (UML) [6]. In this case, the physical machine is controlled by a normal Linux kernel. The host kernel provides hardware resources to each UML instance. The UML kernel provides virtual hardware resources to all the processes within a UML instance. The processes on a UML instance can run native code

[2] IA-32 and x86 are used interchangeably in this paper.

[3] Efforts have been made to run UML under Valgrind.

on the processor, avoiding pure emulation, and UML kernel traps all privileged needs. The UML kernel is, in fact, just an architectural port—*ARCH=um*—of the normal Linux kernel. The architecture specific code in UML is actually user-space code which uses the host Linux kernel system call interface. In other words, it is a port of the Linux kernel to the Linux kernel. This form of virtualization can be used for security[4], debugging, or virtual hosting.

2.1.3 Virtual Machine

The virtual machine (VM) has been studied for well over thirty years [8, 9]. It is a powerful abstraction that gives the illusion of running on dedicated real hardware without such physical requirements. In its early incarnations it provided a safe and convenient way to share expensive hardware resources. The well-known IBM VM/370 [5] simulated the System/390 hardware, presenting multiple independent VM's to the user. The VM/370 was aided by the System/370 hardware design, a luxury which is often not available to the modern world of low-priced, powerful commodity processors based on the x86 architecture [23]. However, it is precisely this type of environment which can benefit from consolidating multiple hardware servers to a single amply powered machine.

The typical architecture includes a physical platform which runs a virtual machine monitor (VMM). This monitor carves up the physical resources and makes them available to each virtual machine. In some cases, the VMM is *host-based* requiring a host OS, host specific drivers and user-space code to launch a VM [13, 7]. As with processor emulation in Section 2.1.1, it is beyond the scope of this paper to give an exhaustive survey of virtual machine technologies. Here we take a brief look at a few of the prevalent projects which can be used to run Linux in a virtual machine:

- Plex86
- VMware[5]
- Xen

Plex86 [18] is one project that provides an x86 virtual machine. This project provides a hosted virtual machine monitor, requiring a host OS to run the plex86 VMM. Plex86 is quite specific to Linux. The host OS may be Linux (although other host kernels are supported) and requires a kernel module to help implement the VMM. It also makes some key assumptions regarding usage of the virtual x86 hardware and patches the guest Linux kernel to conform to these assumptions. Plex86 does very little to virtualize hardware I/O. Instead, Plex86 uses a Hardware Abstraction Layer (HAL) to handle virtual I/O to the hardware devices. This eliminates the need to provide any kind of virtual devices in the VM, and being host-based eliminates the need for the VMM to understand all the possible hardware on the host. I/O which is started in the guest OS is passed through the HAL using fairly simple guest kernel drivers which issue an `int $0xff`—which must not be used for other purposes on the host OS. The host VMM traps that software interrupt and handles the request accordingly. As noted by the project's author, Plex86 is still in a prototype state, and not really ready for meaningful benchmarking yet.

VMware [7] is worthy of mention, despite the fact that it is a commercial product. VMware™ Workstation [12] provides an x86 virtual machine and is in some ways similar to Plex86. It is a hosted virtual machine monitor, however,

[4] To be secure, UML must run in `skas` mode which requires a small patch to the host kernel

[5] VMware is a commercial product.

the goals of VMware Workstation include the ability to run a complete x86 OS without making any modifications. Therefore, it makes no assumptions about the guest OS. By emulating very standard hardware such as the PS/2 keyboard and mouse, the AMD PCnet™ network interface card or the Soundblaster 16 sound card the VM provides virtual hardware devices that can be run by standard guest OS drivers. Another x86 virtual machine from VMware is the ESX Server [26]—a pure virtual machine monitor that is not host-based. This method eliminates some of the overhead involved with running atop a host OS at the cost of requiring more hardware support in the VMM itself. As with Workstation, ESX requires no modifications to the guest OS. The lower overhead of ESX makes it a contender for a data center virtual hosting environment, where it could easily run multiple VM's on a single physical system.

Xen [16] is an x86 virtual machine monitor that provides a virtual hardware interface to the virtual machine. Typically, the virtual machine provides a hardware interface which is identical to the underlying hardware. However, the Xen VM hardware abstraction is similar but not identical to the underlying x86 hardware. This allows the VMM to overcome some of the shortcomings of the x86 architecture which make it difficult to virtualize [23]. A similar method was used for the Denali [1] isolation kernel. However, unlike Denali, the Xen VM supports a notion of a virtual address space. So the guest OS and applications may share resources just like a normal OS environment. In addition, guest kernels running in a Xen VM preserve the ABI to their applications. So, while there is a need to port the guest OS kernel to the Xen VM virtual hardware abstraction, the porting effort ends there. Further, given the similarity to the x86 architecture, the effort to port to Xen results in a very small amount of new OS code—well below 2% of the OS code base [16]. This method has proven to be quite effective when considering the minimal porting effort coupled with the impressive performance benchmarks [16].

2.2 Partial Virtualization

Partial virtualization techniques create virtualized resources that are a specialized subset of a complete system's resources rather than a complete virtual machine. These methods are typically used to present a virtual interface to clients or applications when limited isolation or virtualization is sufficient. Partial virtualization can have very different applications depending on the resource which is being virtualized. These techniques vary widely in the problems they solve, and in some cases can be used with alongside of complete virtualization. The remainder of this Section reviews these techniques.

2.2.1 Linux-Vserver

The Linux-Vserver [20] project takes some of the basic ideas of isolation from a virtual machine and implements them in a single host OS. The Linux kernel is patched to allow for multiple concurrent execution contexts, often called Virtual Private Servers[6] (VPS). This method eliminates any overhead associated with running multiple operating systems, multiple VM's and the supervisor VMM. Each context can have its own file system, its own network addresses, its own set of Linux Capabilities [25], and its own set of resource limits. With this level of software isolation, it is possible to run two concurrent contexts that are unable to interact with each other directly. It may still be possible to generate some indirect QoS degradation from *crosstalk* [24], however these effects should be largely mitigated by

[6] This is also the name given to Ensim's commercial product [4].

proper setting of each context's resource limits. While this solution does require a reasonably large kernel patch (a 337K patch against Linux 2.6.6), it is a very thin virtualization layer that efficiently isolates execution contexts.

2.2.2 Linux Virtual Server

The Linux Virtual Server Project [11] takes a very different view of server virtualization from Linux-Vserver, Section 2.2.1. Rather than creating a virtual operating environment for each server, it behaves as a network load balancer. The Linux Virtual Server, also referred to as IP Virtual Server (IPVS), presents a single network address for the network service and distributes client requests transparently to a hardware cluster of network servers. With IPVS, the client can be redirected to the next available resource using a variety of algorithms such as round robin and least connected. This is an example of virtualization used to provide enhanced availability throughput, or scalability. Further, this project in contrast with Linux-Vserver helps illustrate the difficulty in defining a "Virtual Server."

2.2.3 File system and Disks

The UNIX file system provides the basic namespace that applications use to interact with significant portions of the system. The root of a file system can be relocated in Linux using `chroot()`. This may be a stretch of the definition of virtualization, but this technique does allow a single server to give different views into the system global namespace. Tools like `chroot()` or the BSD `jail()` system[7] allow multiple applications to have completely private file system namespaces, which becomes an effective tool towards system virtualization. In fact, Linux-Vserver, Section 2.2.1, makes use of `chroot()` as key to its file system isolation. Linux has native support for per process private namespaces. This gives each process its own virtual or logical view of the system's global namespace, in a more powerful, flexible and secure manner than `chroot()`. Linux-Vserver is considering moving to namespaces as a replacement for `chroot()` isolation [20]. It would not be surprising to find other virtualization systems using the same technique for file system isolation.

Another layer of virtualization can be found in the disk or block device layer of the Linux kernel. The device-mapper allows administrators to create a virtual block device which is backed by one or more physical block devices. This type of virtualization is typically used for ease of administration.

3 Comparisons

Having reviewed a variety of virtualization techniques in Section 2, it is now useful to pick a representative subset and see how they compare with one another. For the sake of comparison, this paper will focus on QEMU, Usermode Linux, Xen, and Linux-Vserver. All four of these technologies can provide a virtual execution environment comprehensive enough to run either a complete OS, or at a minimum user-space applications.

3.1 QEMU

Pros:

- Portable to numerous architectures.
- Can be used to cross platforms.
- Can run guest OS unmodified.
- Can run on unmodified host OS.

[7] An implementation of BSD jail has been ported to Linux.

- Flexible, can run a full system or just isolated user-space programs.
- Very easy to debug system software.
- Security through isolation.

Cons:

- Processor emulation is much slower than virtualization.

3.2 User-mode Linux

Pros:

- Portable to numerous architectures.
- Can run on unmodified host OS.
- Efficient enough to run multiple instances on single host in virtual hosting environment.
- Very easy to debug system software.
- Security through isolation.

Cons:

- Still slower than a virtual machine.
- The guest OS kernel is not the same as a native one.

3.3 Xen

Pros:

- True virtual machine monitor for best performance.
- The guest OS user-space applications are binary compatible.
- No host OS, very clean virtual machine separation.
- Security through isolation.
- Ideal for virtual hosting environment, can scale up to 100 virtual machines.

Cons:

- The guest OS kernel must be ported to Xen virtual hardware architecture.

3.4 Linux-Vserver

Pros:

- Highly efficient way to isolate resources.
- Can conserve on disk and memory by sharing basic resources like shared libraries.
- Security through context separation.

Cons:

- Only one kernel instance, so quality of service may be hard to guarantee.

4 Conclusions

Virtualization is an old yet resurging technology. Virtual machine research is alive and well, and Linux provides a great testbed for new virtualization technologies. With a wealth of choices, Linux users are sure to find a virtualization technique that suits their requirements. From running as a guest OS on a virtual machine, to providing thin isolation environments for applications, to single system image clusters, Linux is thriving in this virtual reality.

References

[1] M. Shaw A. Whitaker and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[2] Bochs IA-32 Emulator Project. `http://bochs.sourceforge.net/`.

[3] Bochs FAQ. `http://bochs.sourceforge.net/doc/docbook/user/faq.html#AEN273`.

[4] Ensim Corporation. Ensim Virtual Private Server, June 2004. `http://www.ensim.com/products/privateservers/index.html`.

[5] R. J. Creasy. The Origin of the VM/370 Time-Sharing System. *IBM Journal of Research and Development*, 25(5):483–490, September 1981.

[6] Jeff Dike et al. User-Mode Linux, July 2000. `http://user-mode-linux.sourceforge.net/`.

[7] Mendel Rosenblum et al. VMWare. `http://www.vmware.com/`, February 1998.

[8] R. Goldberg. *Architectural Principles for Virtual Computer Systems*. PhD thesis, Harvard University, 1972.

[9] R. P. Goldberg. Architecture of Virtual Machines. In *Proceedings of the Workshop on Virtual Computer Systems*, pages 74–112, 1973.

[10] Reed Hastings and Bob Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Winter USENIX Conference*, 1992. Also available at `http://www.rational.com/support/techpapers/fast_detection/`.

[11] The Linux Virtual Server Project. `http://www.linuxvirtualserver.org/`.

[12] Ganesh Venkitachalam Jeremy Sugerman and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *USENIX Annual Technical Conference*, June 2001.

[13] Mac-on-Linux. `http://www.maconlinux.org/`.

[14] Julian Seward Nicholas Nethercote. Valgrind: A Program Supervision Framework. In *Electronic Notes in Theoretical Computer Science 89 No. 2*, 2003.

[15] Single System Image Clusters (SSI) for Linux. `http://openssi.org`.

[16] K. Fraser S. Hand T. Harris A. Ho R. Neugebauer I. Pratt P. Barham, B. Dragovic and A. Warfield. Xen and the Art of Virtualization. In *Symposium on Operating Systems Principles (SOSP)*, 2003.

[17] PearPC - PowerPC Architecture Emulator. `http://pearpc.sourceforge.net/`.

[18] Plex86 x86 Virtual Machine Project. `http://plex86.sourceforge.net/`.

[19] Herbert Poetzl. Linux-VServer Project, October 2001. `http://www.linux-vserver.org`.

[20] Herbert Poetzl. Linux-VServer Technology. In *LinuxTAG 2004*, June 2004.

[21] QEMU CPU Emulator. `http://fabrice.bellard.free.fr/qemu/`.

[22] QEMU Benchmarks. `http://fabrice.bellard.free.fr/qemu/benchmarks.html`.

[23] J. S. Robin and C. E. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *9th USENIX Security Symposium*, pages 129–144, August 2000.

[24] D. L. Tennenhouse. Layered multiplexing considered harmful. In

Rudin and Williamson, editors, *Protocols for High Speed Networks*. May 1989.

[25] Winfried Trumper. Summary about POSIX.1e. `http://wt.xpilot.org/publications/posix.1e`, July 1999.

[26] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.