Proceedings of the Linux Symposium

June 13th–15th, 2011
Ottawa, Ontario
Canada

Contents

X-XEN : Huge Page Support in Xen

Aditya Sanjay Gadre
Pune Institute of Computer Technology
adivb2003@gmail.com

Kaustubh Kabra
Pune Institute of Computer Technology
kabrakaustubh@gmail.com

Ashwin Vasani
Pune Institute of Computer Technology
vasani.ashwin@gmail.com

Keshav Darak
Pune Institute of Computer Technology
keshav.darak@gmail.com

Abstract

Huge pages are the memory pages of size 2MB (x86-PAE and x86_64). The number of page walks required for translation from a virtual address to physical 2MB page are reduced as compared to page walks required for translation from a virtual address to physical 4kB page. Also the number of TLB entries per 2MB chunk in memory is reduced by a factor of 512 as compared to 4kB pages. In this way huge pages improve the performance of the applications which perform memory intensive operations. In the context of virtualization, i.e. Xen hypervisor, we propose a design and implementation to support huge pages for paravirtualized guest paging operations.

Our design reserves 2MB pages (MFNs) from the domain's committed memory as per configuration specified before a domain boots. The rest of the memory is continued to be used as 4kB pages. Thus availability of the huge pages is guaranteed and actual physical huge pages can be provided to the paravirtualized domain. This increases the performance of the applications hosted on the guest operating system which require the huge page support. This design solves the problem of availability of 2MB chunk in guest's physical address space (virtualized) as well as the Xen's physical address space which would otherwise may be unavailable due to fragmentation.

1 Introduction

"The Xen hypervisor is a layer of software running directly on computer hardware replacing the operating system thereby allowing the computer hardware to run multiple guest operating systems concurrently. Support for x86, x86-64, Itanium, Power PC, and ARM processors allow the Xen hypervisor to run on a wide variety of computing devices and currently supports Linux, NetBSD, FreeBSD, Solaris, Windows, and other common operating systems as guests running on the hypervisor."[5]

A system running the Xen hypervisor contains three components:

1. *Xen Hypervisor*
2. *Domain 0, the Privileged Domain (Dom0)* – Privileged guest running on the hypervisor with direct hardware access and guest management responsibilities
3. *Domain U, Unprivileged Domain Guests (DomU)* – Unprivileged guests running on the hypervisor.

Types of Virtualization in Xen:

1.1 Paravirtualization

A term used to describe a virtualization technique that allows the operating system to be aware that it is running on a hypervisor instead of base hardware. In this type of virtualization, the operating system is modified in a way that it has direct access to a subset of hardware.

1.2 Hardware Virtual Machine (HVM)

A term used to describe an unmodified operating system that is running in a virtualized environment and is unaware that it is not running directly on the hardware. Special hardware support is required in this type of virtualization i.e. Intel-VT or AMD-V.

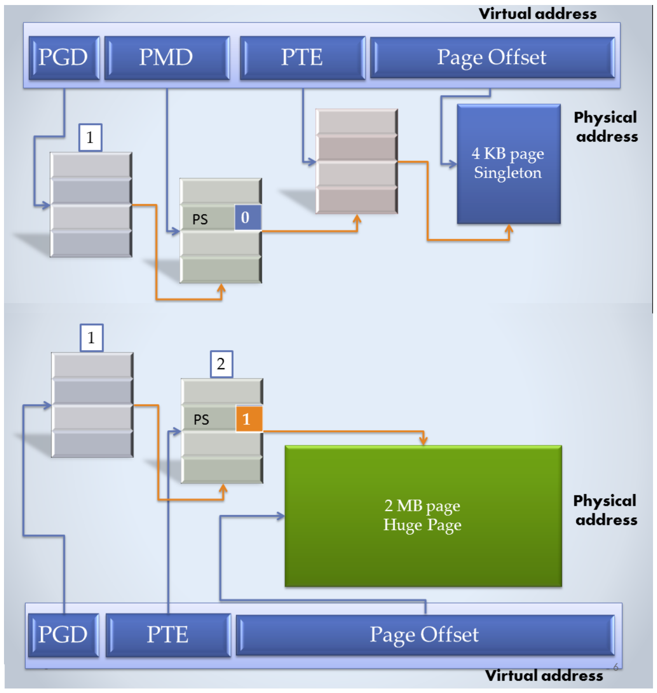
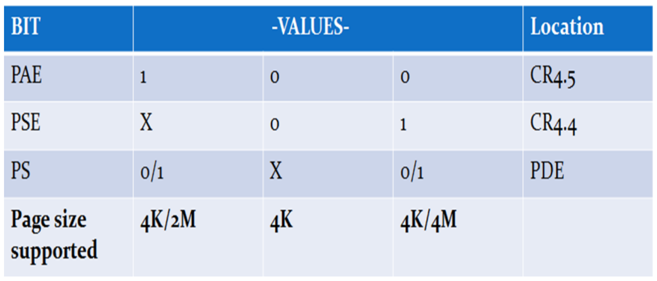


Figure 1: Huge Pages



BIT	-VALUES-			Location
PAE	1	0	0	CR4.5
PSE	X	0	1	CR4.4
PS	0/1	X	0/1	PDE
Page size supported	4K/2M	4K	4K/4M	

Figure 2: Register bits.

2 Huge Pages

Huge pages (Figure 1) are memory pages of size 2MB/4MB depending upon the bits PAE and PSE in CR4 register and PS bit (Figure 2) in the page directory entry.

These bits can be set or cleared using specific instructions in kernel code. The bits table shown above depicts the various page sizes supported according to the bits set or cleared. Huge pages can be reserved in Linux kernel by giving boot time parameter (`hugepages`) option or by executing following command:

```
# echo n > /proc/sys/vm/nr_hugepages
# cat /proc/meminfo | grep Huge
```

3 Problem with huge pages in virtualized environment

When the paravirtualized kernel is compiled with `CONFIG_HUGETLB_PAGE`, support for HugeTLBfs is enabled. Now when the request for hugepage reservation is made to the kernel, it reserves appropriate PFN range.

Problem 1: This contiguous PFN range might not correspond to a contiguous MFN range. When an application allocates a hugepage, the kernel makes the reservation, but when the application tries to access this hugepage, the kernel crashes. The reason for this crash is the MFN corresponding to first PFN of the allocated hugepage range is now treated as contiguous hugepage by the architecture. But the next MFN may not even belong to the domain itself, hence the kernel crashes.

Problem 2: Consider the case when the allocated contiguous PFN range corresponds to contiguous MFN range. Even then the first MFN may not be aligned to a 2MB boundary. In this scenario also the kernel crashes.

Hence, the prerequisites for hugepage allocation in virtualized environment are that allocated hugepages must be contiguous, and must be 2MB aligned in both the kernel address space (virtualized PFNs) and the hypervisor's address space (MFN).

4 Motivation

Huge pages are used by various applications such as JVM, Oracle, MySql for performance improvement. These applications use *mmap* system call for huge page allocation from huge page pool.

As the applications and data grow larger and larger, the need of huge pages (HugeTLB) increases. The database service providers have their databases deployed on dedicated servers where they utilize hugepage support for improvement in performance. When these providers now want to host these dedicated servers on the Xen based cloud, there is a need to provide huge page support in the virtualized environment such that it will be able to utilize the performance improvement of HugeTLB.

5 Earlier work

Solution of this problem was previously attempted by Dave McCraken. In that implementation there is a flag named superpages in the config file specified for DomU. If this particular flag is set to 1 in the config file, then during domain creation all allocations are aligned on 2MB boundaries and in 2MB chunks (i.e. 512 MFNs). Hence, the prerequisite for the domain creation in this case is that hugepages (or 2MB contiguous chunks) equivalent to the amount of domain's memory (memory/2) should be available with Xen's buddy allocator. If these chunks are not available then the domain denies booting. After the domain has booted and begins using memory, then due to pre-existing fragmentation problem the amount of contiguous PFNs required for hugepage allocation may not be available with the domains.

6 Innovation

In the server environment, when the database servers must handle large amount of data and users, they are usually hosted on the dedicated servers. To get the performance benefits they use hugepages instead of normal 4kB pages. When these dedicated servers are to be moved to Xen based cloud environment, that is where our design to solve the problem fits in. In our implementation we allow to specify the number of hugepages that will be required. When the domain is created, the appropriate MFN and PFN ranges for hugepage allocation are reserved. This implementation ensures that once the domain has booted it will always get that specified number of hugepages. The implementation will be further elaborated in Section 7.

7 Implementation

Our implementation works in three phases. In the first phase we reserve the MFNs for hugepage allocation from the Xen's allocator. In the second phase we reserve equivalent PFN range in the kernel for hugepage allocation. The third phase occurs when the request for hugepage reservation for the applications is made by the kernel. These three phases are further elaborated below.

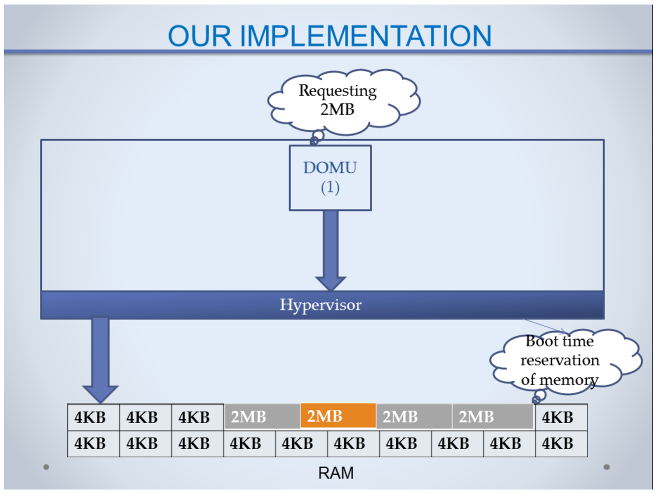


Figure 3: Solution.

7.1 Domain Creation

In our implementation we added a parameter `hugepage_num` which is specified in the config file of the DomU. It is the number of hugepages that the guest is guaranteed to receive when the kernel asks for hugepage using its boot time parameter or reserving after booting (eg. using `echo XX > /proc/sys/vm/nr_hugepages`).

We have introduced two variables in the domain's structure in the Xen hypervisor: `hugepage_num` is an integer which eventually stores the number of hugepages mentioned in the config file of the domain, and `hugepage_list` stores the MFNs of the pages to reserve for the domain. When the domain is being created, the number of hugepages is stored in the variable `hugepage_num`, and then calls to Xen's allocator for 2MB chunks (order 9) are made. The resulting MFNs are added to `hugepage_list`.

```
struct domain {
    :
    //Storing MFN list
    struct page_list_head hugepage_list;
    //Number of hugepages
    unsigned int hugepage_num;
    :
} d;
```

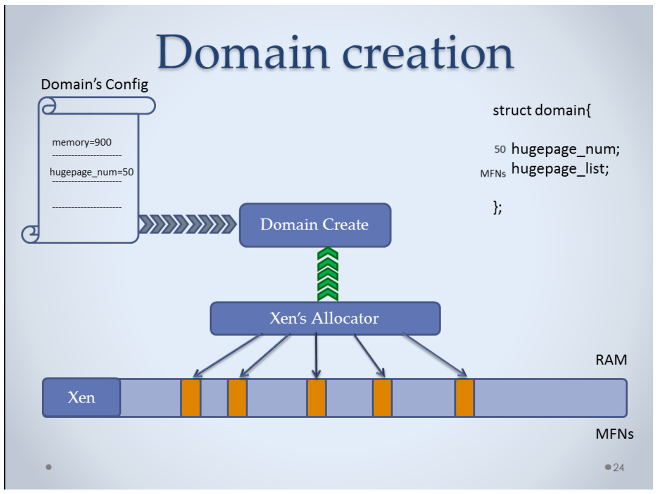


Figure 4: Domain Creation

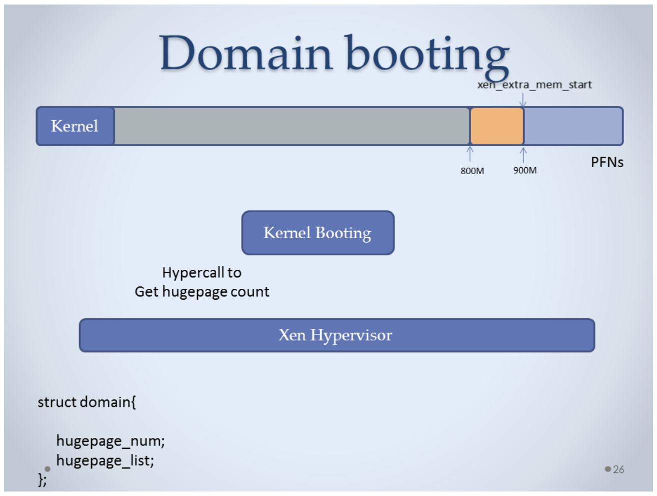


Figure 5: Domain Booting

7.2 Domain Booting

When the domain is booting, the memory seen by the kernel is reduced by the amount required for hugepages. The kernel then makes a hypercall to Xen to get the count of hugepages. Xen takes the count from `hugepage_num` and returns it to the kernel. The kernel now increments `xen_extra_mem_start` by the amount equivalent to the count of hugepages i.e. `count * 2MB`. This is required so that the functioning of ballooning driver is not hampered by our implementation. Now the kernel adds these PFNs (at 2MB intervals) in the `xen_hugepfn_list` which is a newly introduced variable in the kernel. Hence, the PFN range is reserved in the kernel for hugepage allocation.

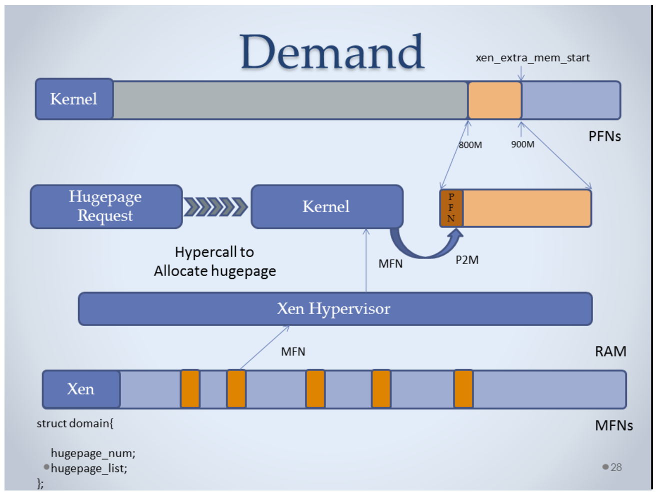


Figure 6: Demand Supply

7.3 Hugepage reservation Request

When a request for hugepage is made by using boot time parameter or reserving after booting (eg. Using `echo XX > /proc/sys/vm/nr_hugepages`), the kernel makes a hypercall to allocate hugepage. The hypervisor then removes one MFN from `hugepage_list` in the domain's structure and returns it to the kernel. The kernel removes one PFN from `xen_hugepfn_list` in the kernel. Then it maps 512 MFNs to the corresponding PFNs beginning from the PFN and MFN just retried. This process is repeated for the number of hugepages requested by the kernel. Hence, all the requirements for hugepage allocation are satisfied i.e. the hugepages are contiguous and 2MB aligned on both kernel (PFN) and hypervisor (MFN) side.

8 Performance

The performance of huge pages was measured by allocating different memory sizes as shown in Figures 7 and 8. Memory was allocated using *mmap* function call and flag `MAP_HUGETLB` was passed in the argument.

9 Constraints

1. Existing live migration techniques need to be modified to support huge page migration.
2. Our implementation may not support 1GB super pages.

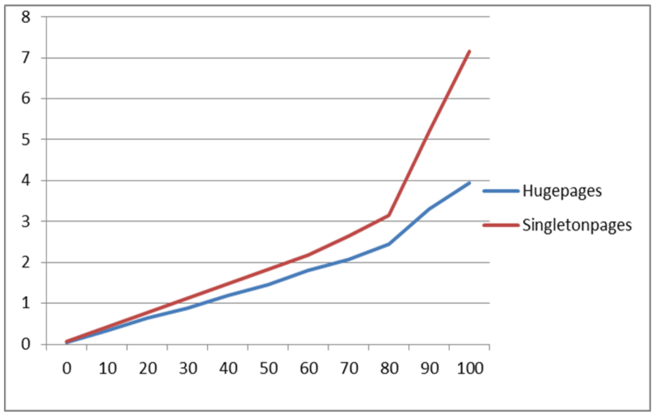


Figure 7: X-axis: Buffer Length & Y-Axis: Time required (Thread =4)

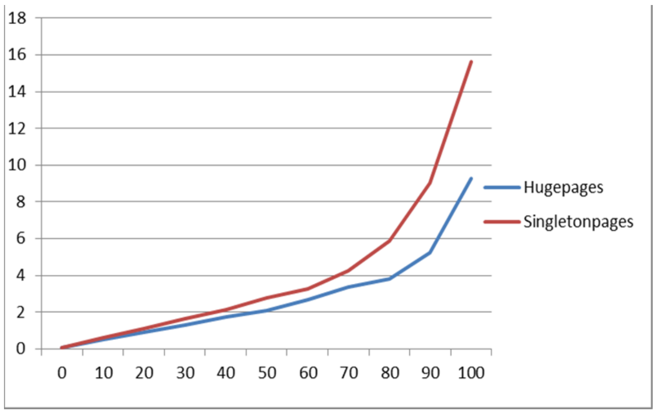


Figure 8: X-axis: Buffer Length & Y-Axis: Time required (Thread =6)

References

[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003.

[2] Abhishek Nayani, Mel Gorman & Rodrigo. Memory Management in Linux. http://www.ecsl.cs.sunysb.edu/elibrary/linux/mm/mm.pdf

[3] Tim Deegan, CITRIX Systems. Memory management in (x86) Xen. http://www.slideshare.net/xen_com_mgr/xen-memory-management

[4] Andrés Krapf. XEN Memory Management (Intel IA-32). http://www-sop.inria.fr/everest/personnel/Andres.Krapf/docs/xen-mm.pdf

[5] http://www.xen.org/files/Marketing/WhatisXen.pdf

NPTL Optimization for Lightweight Embedded Devices

2011 Linux Symposium

Geunsik Lim
Samsung Electronics
`geunsik.lim@samsung.com`

Hyun-Jin Choi
Samsung Electronics
`hj89.choi@samsung.com`

Sang-Bum Suh
Samsung Electronics
`sbuk.suh@samsung.com`

Abstract

One of the main changes included in the current Linux kernel is that, Linux thread model is transferred from LinuxThread to NPTL[10] for scalability and high performance. Each thread of user-space allocates one thread (1:1 mapping model) as a kernel for each thread's fast creation and termination. The management and scheduling of each thread within a single process is to take advantage of a multiple processor hardware. The direct management by the kernel thread can be scheduled by each thread. Each thread in a multi-processor system will be able to run simultaneously on a different CPU. In addition, the system service while blocked will not be delayed. In other words, even if one thread calls blocking a system call, another thread is not blocked.

However, NPTL made features on Linux 2.6 to optimize a server and a desktop against Linux 2.4 dramatically. However, embedded systems are extremely limited on physical resources of the CPU and Memory such as DTV, Mobile phone. Some absences of effective and suitable features for embedded environments needs to be improved to NPTL. For example, the thread's stack size, enforced / arbitrary thread priority manipulation in non-preemptive kernel, thread naming to interpret their essential role, and so on.

In this paper, a lightweight NPTL (Native POSIX Threads Library) that runs effectively on embedded systems, for the purpose of a way to optimize is described.

1 Introduction

Generally speaking, most existing embedded environments have been designed and developed for specific purposes, such as Mobile phone, Camcorder, DTV. However, in order to satisfy customer's diverse needs, recent embedded products are required to become smarter.

Many customers want to efficiently use embedded products in their life. To fulfill these needs, most embedded products offer their own app-stores. Customers download from these app-stores and install programs they need in their daily life. With the appearance of these app-stores, the number of applications for embedded products is dramatically increasing.

In mobile environments, where CPU and memory[12] resources are limited compared to desktop or server environments, the increase of applications make it more important to design and develop an efficient system without any resource expansion.

The higher performance hardware incurs higher cost in manufacturing; therefore, most companies prefer lower cost hardware solutions if they can achieve safe and reasonably fast run-time execution environment through efficient supports from underlying platform itself. Companies equipped with these solutions will have a big advantage in business especially in terms of cost competitiveness.

The performance and degree of integration of hardware improves every year. However, designing and developing optimized software especially for lightweight embedded environments is still a hard goal to achieve. Recently, the number of processes or threads running in user-space of Linux-based embedded products has reached at least 200 and sometimes exceeds 700.

For comparison, Microsoft Windows 7 running in desktop environment with a high performance CPU and high capacity memory, on average has more than 700 threads created and managed. These processes and threads have process condition cycles such as running and waiting depending on the conditions given.

Considering this, we may guess that the number of processes or threads running in embedded products with lower hardware specification is non-negligible, i.e. large enough to require an efficient solution using minimum memory footprint and achieving optimized performance.

The NPTL 1:1 thread mapping model, which appeared in Linux Kernel 2.6, dramatically improved the limits of scalability and performance as compatered with the existing LinuxThread. NPTL[1] [3] was originally provided as an alternative option for LinuxThread for compatibility, but with the popularity of NPTL model, recent Linux distribution releases only support NPTL, which obsoletes thepierce augments on the useless of NPTL model.

The NPTL model adapted in released versions of Debian/Ubuntu, Fedora/RHEL, and openSUSE need improvements in order to support lightweight development environments. This paper tackles this issue and proposes many ways of achieving the improvements.

2 Thread stack size for embedded system

In implementing multi-programs through user-space threads, the stack size of a single thread is directly proportional to the maximum number of threads which a developer can produce. When a thread is created, stacks in shared memory region are allocated to the thread.

In general, the basic stack size in latest Linux distributions is 10 MB per thread. The stack size of a thread has been increased from 2MB to 8MB and now 10MB to avoid stack overflow problems at various Linux distributions such as Ubuntu, Fedora, openSUSE.

Due to low power management and cost competitiveness, most mobile embedded systems provide limited physical resources. Moreover, the swap device, which is normally used to overcome physical memory shortage, is not supported in most embedded environments.

Therefore, in order to implement efficient applications, developers should obey good thread programming styles and find an optimal stack size for threaded applications. In this way, we can implement an efficient and economic system without additional hardware support.

Generally speaking, two main reasons of system panic caused by user-space thread libraries are 1) OS bugs, and 2) page segment fault error by abnormal program operations. In application developers' point of view, OS bugs are difficult to control, but page segment fault error by abnormal program can be avoided through the adjustment of stack size to the right size.

Other than memory expansion and swap space support, there are three software approaches to solve system problems effectively.

- Increase or decrease the available stack size with ulimit

  Check and adjust the stack size through ulimit command in running shell. Adjustments made with the ulimit command only affect the shell where the ulimit command is given and its children. Therefore, in order to alter the setting globally in the system, the ulimit command should be executed at the end of booting cycle (eg. `/etc/rc.sysinit`).

- Define individual stack size of a user-space thread (pthread_attr_)

  Thread attribute value that application developer defined must be initialized by pthread_attr_init(), when developer want to define a stack size using POSIX standards for multi-thread programming. Developers call pthread_attr_getstacksize() library function to determine the stack size, and they can dynamically control the stack size of a thread with pthread_attr_setstacksize() library function. Stack size may also be set when creating threads using the pthread_create() library function.

- Establish appropriate stack size policy for all threads in NPTL layer

  Many application developers use pthread_attr_getstacksize() with t_attr.__stacksize variable which is defined differently for each architecture internally. In the NPTL library, t_attr.__stacksize variable takes the value assigned to the thread's stack size. If this value is not set, the system set by default stack size ulimit command brings.

Figure 1 shows the internal operation flow of user-space thread created through any process under NPTL environment. We have to decide the default stack size of embedded system that want to select as a default stack size value about all threads that are created in specified

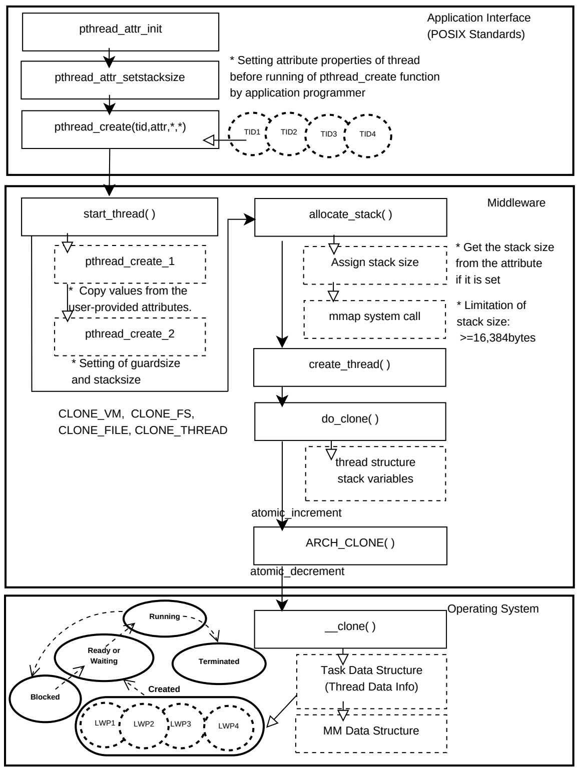

Application Interface
(POSIX Standards)
pthread_attr_init
pthread_attr_setstacksize
* Setting attribute properties of thread before running of pthread_create function by application programmer
pthread_create(tid,attr,*,*)
TID1
TID2
TID3
TID4
Middleware
start_thread( )
pthread_create_1
* Copy values from the user-provided attributes.
pthread_create_2
* Setting of guardsize and stacksize
allocate_stack( )
Assign stack size
* Get the stack size from the attribute if it is set
mmap system call
* Limitation of stack size: >=16,384bytes
create_thread( )
CLONE_VM, CLONE_FS, CLONE_FILE, CLONE_THREAD
do_clone( )
thread structure stack variables
atomic_increment
ARCH_CLONE( )
atomic_decrement
Operating System
__clone( )
Task Data Structure (Thread Data Info)
MM Data Structure
Running
Ready or Waiting
Terminated
Blocked
Created
LWP1
LWP2
LWP3
LWP4

Figure 1: Internal operation flow of thread

embedded system. Through it, we can manage consistent policy that adjust default stack size of thread at the middleware level automatically.

Since most threads created under embedded environment run light operations, it is typical to use stack sizes much less than the 10MB default used on servers. Thread stack size of embedded application is less than 1MB in general. Considering the minimum memory space of data structure for the creation of threads, Linux-based embedded system needs the stack size of more than 16,384 bytes per thread essentially. The stack size of 16,384 bytes includes guard size of 4,096 bytes during memory mapping practically. In other words, to prevent stack overflow, deduction 4,096 bytes from stack size by using pthread_attr_setstacksize() function is allocated to memory space for guard size as below.

```
4003b000 1127K r-x-- /lib/libc-2.12.so
40139000    8K ----- /lib/libc-2.12.so
4013b000   20K rw--- /lib/libc-2.12.so
40140000    8K r---- /lib/libc-2.12.so
40142000    8K rw--- /lib/libc-2.12.so
40144000    8K rw---   [ anon ]
40146000    4K -----   [ anon ]
40147000   12K rwx--   [ anon ]
402bf000    4K -----   [ anon ]
402c0000   12K rwx--   [ anon ]
402c3000    4K -----   [ anon ]
402c4000   12K rwx--   [ anon ]
 ..... remainder omitted ......
```

To decide default stack size in embedded system, we have to profile the distribution map of stack size that all threads are utilizing in embedded system. Otherwise, *Run-time error* may occur when application developer try to use stack size at run-time after reduce stack size of thread remarkably more than default stack size for normal operation.

When application developers compile threaded application source, they can compile entire source without any error problems. However, a segmentation fault error can occur when stack usage exceeds the size defined during running ELF binary code compiled like above.

In contrast, if the application allocates too large stack size for new thread using pthread_attr_setstacksize() function, then new thread creation will sometimes fail. As a result, system will return error code (number is 12) for the call to run pthread_create() function, and thereafter will not be able to create more threads. Therefore, in spite of Linux system different from uCLinux can support improved Virtual memory Management, it needs attention to allow a large stack size thoughtlessly under embedded environment.

3 Thread naming

It is very hard to identify the purpose of each thread when there hundreds of them in an embedded system. Optimization of user-space functions and SQA(Software Quality Assurance) is essential process before mass production for product competitiveness and reliability after complete entire development process using limited hardware resources.

If it is possible to monitor the purpose of individual thread for hundreds of threads run in embedded system at this time, we can improve development productivity rapidly by finding the part of thread's abnormal running and debugging and optimizing it. It is possible to distinguish main role of relevant child threads using the name of thread function set as the 3rd parameter during pthread_create() function call. Alternatively, the application developer may obtain a unique value for each thread with Thread Local Storage (TLS)[11] that is supported by CPU and cross-compiler. This value is used for the purpose of identifying which thread runs a specified function at some point.

However it is difficult to know the unique purpose of each thread executed by this method, when there are hundreds of threads calling the same function by using pthread_create() function. If we expand and support the "Thread Naming" interface at middleware layer, or user-space thread model like NPTL, it would be easy to understand the operational purpose of all threads in the platform.

In a large scale project, many teams participate and cowork to develop, and often cause bad effect by creating another task to check and understand the operational purpose of threads created by other teams. When several developers in each team produce thread using additional support like pthread_set_naming_np() and pthread_get_naming_np() for embedded system, the team to improve performance can analyze the detail information of all threads produced obviously by defining the role of additional thread by using pthread_set_naming_np() library call.

Table 1 below shows the sample output including the role of threads. When threaded applications using GDB (GNU Debugger) use named threads, system programmers can distinguish each other. Readability can be improved by understanding the detail flow of thread to analyze the purpose of creating, sleeping and finishing of each thread.

4 Thread profiling

We can find the optimal time point for booting the system by profiling the time interval between ahead thread and back thread and CPU share of all threads created before the initial GUI screen of embedded platform appears.

For example, when the share of CPU for specific region is low during the system booting, we can maximize the CPU usage and shorten the system boot time by separating independent functions as threads, and by moving some threads. If the generation of thread occurred long after specific thread's execution, Analyze CPU usage of thread executed for long time in detail. If CPU usage were not high, the research for system optimization would be possible. Despite high CPU usage during embedded system booting, if relevant scheduling work could be possible after initial screen appears, it would be effective to run those threads after initial screen appearance.

Table 2 shows information like Stack Size, Guard Size, Priority Value, Start Time, Time Gap which can be used to optimize system boot time. Performance optimization is possible by debugging internal operation information of user-space functions and identifying naming information of relevant threads because thread number 168 is executed almost for 1 second in the table below.

5 Extension of pthread_{set|get}_priority_np interface

When Linux kernel based on 2.6 version uses system call and library call, it consists of total 140 priorities with normal priority level using nice value from -20 to 19 and real-time priority level from 0 $\sim$ 99. Low number have high priority in Linux kernel-space.

Normal priority is defined in the file of `kernel/sched.c` and Linux kernel schedule this tasks with O(1) scheduler or CFS scheduler (since 2.6.23)[4]. Depending on Linux kernel version, after allocate one normal priority between bitmap 100 and bitmap 139 about a nice value between -20 and 19 by user-space application developers.

User-space real-time support and a few challenges for 100% POSIX compliance was written in the section "8. Remaining Challenges" in *Native POSIX Threads Library for Linux*[10] paper by Ulrich Drepper.

Infrastructure for POSIX compatible real-time support for user-space was improved by adding features such as Priority Queuing, Robust Mutex (`RT_MUTEX`)[8] and Priority Inheritance[5] [7] [9] to Linux and Glibc. This means application developer can realize real-time threaded programming in user-space. Table 3 shows the system call and library call for setting scheduling priority against the process/thread with normal priority and the process/thread with real-time priority.

The scheduling priority of an already-running normal priority thread can be changed by a system call like setpriority(), nice().

In case of tasks having real-time priority value, there are possible values for scheduling policy like `SCHED_RR`

STACK-ADDR (hex)	*PID* (process)	*TID* (thread)	*Thread Naming* (thread's role)	*Stack Size* (kbytes)	*Memory(RSS)* (kbytes)
[0x40a1b460]	[339]	[342]	Files Copy Extension	64	910
[0x40a2b460]	[339]	[343]	LifeCycle Controller	64	4,211
[0x40a3b460]	[339]	[344]	Micom Task	64	200
[0x40a4b460]	[339]	[345]	Event Dispatcher	64	355
[0x40a5b460]	[339]	[346]	Device Node Manager	64	305
[0x40a6b460]	[339]	[347]	Micom Task Extension #1	64	200
[0x40aeb460]	[339]	[348]	Media API	512	5,442
[0x412a8460]	[339]	[350]	Message Event Handler	64	34
[0x41328460]	[339]	[351]	Drawing Process	512	204
[0x41338460]	[339]	[352]	FlashTimerTask	64	382
[0x41ac5460]	[339]	[353]	UI Manager	512	566
[0x41ad5460]	[339]	[354]	Multi IPI Recovery	64	705
[0x41267460]	[339]	[355]	System Process	64	9,376
[0x41cbc460]	[339]	[356]	MediaCaptureComponent	64	202
[0x41ccc460]	[339]	[357]	MP4 MediaPlayer	512	7,109
[0x41cdc460]	[339]	[358]	JPEG Component	64	153
[0x41cec460]	[339]	[359]	Media Component	64	776
[0x41cfc460]	[339]	[360]	Async I/O	64	371
[0x41cdc460]	[339]	[361]	Video Output Component	64	2,221
[0x41cec460]	[339]	[362]	Service Manager	64	460
-	-	...	below omission	...	-

Table 1: Thread naming information of each user-space thread

(real-time round-robin policy), `SCHED_FIFO` (real-time FIFO policy), `SCHED_OTHER` (for regular non-real-time scheduling) and so on. The scheduling priority in user-space can be set from 1 to 99 for real-time scheduling policy. Therefore, the priority of normal non-real-time threads is counted as 0.

Considered real-time property under embedded environment `SCHED_RR` seems ideal, `SCHED_FIFO` is more useful to take advantage of performance practically because simple policy is good for performance and effective for management.

```
struct sched_param {
  . . . . . .
  int sched_priority;
  . . . . . .
};
```

Table 4 shows number of gettid() system call according to architecture.

Because the use of gettid() is CPU-architecture dependent in Linux kernel 2.6, system call number varies different among CPU architectures. You may utilize gettid() system call with the following definition because of non-implementation of gettid() in Linux system.

```
/* Using gettid syscall in user-space */
#define gettid() syscall(__NR_gettid)
```

We use gettid() instead of getpid() in NPTL thread model to find out the unique number of thread executed in the related function region to apply normal priority to threads are created as nice value. The gettid() function has to be made using syscall(__NR_gettid). And then, the use of gettid() function is available to utilize gettid() function by syscall(__NR_gettid) in the function of relevant thread.

Above _syscall() function returns kernel-space thread id that mapped about user-space thread id that is running by calling `include/asm-arm/unistd.h` header file. The gettid() system call is defined as follows in the file `kernel/timer.c`.

PID (process)	*TID* (thread)	*StackSZ* (byte)	*GuardSZ* (byte)	*Priority* (nice)	*StartTime*	*Time Gap* (msec)
160	162	262,144	4,096	5	1792015420	195
160	163	262,144	4,096	0	1792015423	3
160	164	262,144	4,096	0	1792015551	128
160	165	262,144	4,096	5	1792015666	115
160	166	262,144	4,096	5	1792015668	2
160	167	262,144	4,096	5	1792015670	2
160	168	262,144	4,096	5	1792016634	977
160	169	262,144	4,096	10	1792016637	3
160	170	262,144	4,096	5	1792016781	144
160	171	262,144	4,096	5	1792016783	2
160	172	262,144	4,096	5	1792016786	3
160	173	262,144	4,096	5	1792016788	2
160	174	262,144	4,096	-5	1792016873	85
160	175	262,144	4,096	-5	1792016874	1
160	176	262,144	4,096	5	1792016876	2
160	177	262,144	4,096	5	1792016878	2
160	178	262,144	4,096	5	1792016880	2
160	179	262,144	4,096	5	1792016917	37
160	180	262,144	4,096	5	1792016920	3
160	181	262,144	4,096	5	1792016922	2
160	182	262,144	4,096	-15	1792016925	3
160	183	262,144	4,096	5	1792017258	333
160	184	262,144	4,096	5	1792017260	2
160	185	262,144	4,096	5	1792017262	2
160	186	262,144	4,096	0	1792017264	2
160	187	262,144	4,096	0	1792017266	2
160	188	262,144	4,096	5	1792017284	18
-	-	...	below omission	...	-	-

Table 2: Thread profiling result

```
/* gettid syscall details in Linux */
asmlinkage long sys_gettid(void){
  return current->pid;
}
```

When you try to utilize gettid() system call using above method, it is recommended to add thread library function including system calls after considering the impact of embedded system's performance because of the cost of system calls. It is very useful to measure execution time, calls and errors for system calls of thread library function to be added to know the cost of CPU usage. The file `arch/arm/kernel/call.S` of the ARM Architecture defines sys_set_thread_aread() as sys_ni_syscall (224) and sys_get_thread_area as sys_ni_syscall (225).

Maintenance of source code of large scale project can be simplified by avoiding having many different functions preferred by developer in embedded platform. Instead a uniform common interface can be obtained by extending thread function of pthread_set_priority_np() or pthread_get_priority_np() additionally for the application developer to get ID value of a thread easily.

POSIX compatibility is very important in the view of standardization, but considering the characteristics of embedded platform, when we need additional thread API, application developers are able to know the extended thread API by making the name of function with the format of _np() to express "Non Portable" meaning

Scheduling Priority	*PID/TID*	*Function Name* (API)	*Call Interface* (classification)	*LinuxThread* (interface)	*NPTL* (interface)
Normal priority (from -20 to 19)	Process	setpriority() nice()	System call	getpid()	gettid()
	Thread	setpriority() nice()	System call	getpid()	gettid()
real-time priority (from 1 to 99)	Process	sched_setscheduler() sched_setparam()	System call	getpid()	gettid()
	Thread	pthread_setschedprio() pthread_setschedparam()	Library call	getpid()	gettid()

Table 3: LinuxThread VS. NPTL scheduling system call comparison

Architecture	*File name*	*gettid() syscall number*
i386	./arch/i386/kernel/entry.S ./arch/x86/kernel/syscall_table_32.S	224
ARM	./arch/arm/kernel/call.S	224
MIPS	./arch/mips/kernel/scall32-o32.S ./arch/mips/kernel/scall64-932.S	4,222
PPC	./arch/ppc/kernel/misc.S	207
SH	./arch/sh/kernel/entry.S	224

Table 4: The number of gettid() per CPU architecture

in the end of function.

In addition, the extension of these additional common interface and unified programming specification maintain consistently application interface management of embedded system that is extended the scale of platform more and more. We can continue software update easily and rapidly for new features while preserving without modifying existing code.

6 Controlling CPU scheduling of {self|another} thread

By increasing the speed of user's application under embedded system environment at specific time, users often want to get shorten application's waiting time. The support of these mechanisms raise the flexibility of scheduling priority for CPU usage when threads need higher CPU usage at specific time. Effective throughput of applications are possible by grouping thread applications based on the importance of processing speed and response speed in embedded system having limited CPU performance.

When new threads according to *Task scheduling importance hierarchy* are created, we need the thread dealing mechanism to realize the way to give suitable scheduling priority value of thread. Table 5 shows explanation about task classification and task meaning according to the Task scheduling importance hierarchy table.

We can minimize user's waiting time for embedded devices by self-adjusting a thread's scheduling priority at specific time, or by changing another thread's normal priority at run-time dynamically with pthread_setschedparam() library call in Linux 2.6 based NPTL environment.

Improvement is important, keeping POSIX compatibility with additional thread APIs to give different priority to many threads produced in one process. This means that reuse of the existing source is possible continually.

The pseudo code below shows the implementation of NPTL Library to control scheduling priority arbitrarily or by force for user-space threads based on normal priority that are created on non-preemptive Linux kernel 2.6.

Hierarchy of scheduling priority	Description
Busy Task (Urgent)	Busy task means the threads in the top of screen which interact with user or which occupy CPU usage under processing CPU.
Foreground Task (Normal)	Foreground task is thread that appear in the screen of user's embedded device but doesn't have activity to be processed immediately.
Service Task (Support)	Service task is middleware level component which supply important functions for processing of application and thread that occupies service of system.
Background Task (Hidden)	Background task is thread that occupies activity not visible to user.
Idle Task (Unlimited)	Idle task is thread that doesn't occupy component of any active application in embedded system.

Table 5: Task scheduling importance hierarchy

```
int __pthread_setschedparam(tid,
    policy, param)
  pthread_t tid;
  int policy;
  const struct sched_param *param;
 {
/* To support priority,if use SCHED_FIFO,
 * SCHED_RR,display notification message
 * ( @/usr/include/linux/sched.h ) */

/* Default value is a normal priority */
struct pthread *pd=(struct pthread *)tid;

if (policy == SCHED_OTHER){
/* Scheduling priority of thread */
int which = PRIO_PROCESS ;

/* Handling of SCHED_OTHER priority */
if ( param->sched_priority < -20 &&
     param->sched_priority > 19 ){
   printf("ERR! Nice range:-20~19\n");
   return errno;
}
/* Getting LWP(thread id of kernel) to
 * change scheduling priority about
 * assigned thread id.
 */
if (setpriority(which,unique_kernel_tid(),
    param->sched_priority) ){
   perror("setpriority() Error.\n");
   result = errno;
}
```

Mentioned above, after improving scheduling-related thread function of NPTL library, the way described below can control thread application's scheduling actively to apply different scheduling priority to many threads which are created in one process in embedded system.

```
/*
 * @Description: arbitrarily & by force
 * thread scheduling for urgent threads
 * @thread variables:(pthread_t thread[max])
 * If you want to affect priority about each
 * thread in a process in Linux 2.6 + NPTL,
 * We recommend that you use SCHED_OTHER
 * policy based on priority scheduling.
 * Or,If you need time critical performance
 * about threads, use real-time SCHED_RR
 * using pthread_setschedparam( ) syscall.
 */

struct sched_param schedp;
/* priority number of between -20~19. */
int priority = -20 ;
memset(&schedp, 0, sizeof(schedp));
schedp.sched_priority = priority;

/* for controlling self thread */
pthread_setschedparam(pthread_self(),
SCHED_OTHER, &schedp)

/* for controlling another thread */
pthread_setschedparam(thread[i],
SCHED_OTHER, &schedp)
```

7 Optimization: configurability and building with -Os

Glibc library[6] including NPTL has a lot of features. However embedded system do not need all features of glibc library. For this reason, it takes compilation time of more than 40 minutes to build the entire glibc source on average at the high-end Linux development computer (e.g: Intel Core2 Quad 9400, RAM 2GB).

We can consider how to compile entire glibc source with same configuration structure and build process like the compilation of Linux kernel. By doing so, we can compile only the necessary software components among a lot of components of glibc source for embedded system. How to compile this method can be modular as a functional unit reduce a long compilation time at software development step. Through this method, we do not select a unnecessary shared object libraries (e.g: NSS, NIS, DNS, CIDN, Locales, Segfault, Crypt, NSS, Resolv, etc) for lightweight rootFS. As a result, hard disk space and memory footprint can be minimized through configurable build method.

If you try to compile glibc sources with -Os option without -O2 through gcc compiler's optimization option interface[2] at the configurable build system menu of NPTL library, shared object binary files can be minimized to fit in the embedded system environment.

Without any optimization option, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Turning on optimization flags makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program.

- -Os: Optimize for size. -Os enables all -O2 optimization that do not typically increase code size. It also performs further optimization designed to reduce code size.
- -O0: Reduce compilation time and make debugging produce the expected results. This is the default.
- -O1: Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function. With -O, the compiler tries to reduce code size and execution time, without performing any optimization that take a great deal of compilation time.
- -O2: Optimize even more. GCC performs nearly all supported optimization that do not involve a space-speed trade-off. As compared to -O, this option increases both compilation time and the performance of the generated code.
- -O3: Optimize yet more. -O3 turns on all optimization specified by -O2 and also turns on the -finline-functions, -funswitch-loops, -fpredictive-commoning, -fgcse-after-reload, -ftree-vectorize and -fipa-cp-clone options.

8 Further work

Two issues of the current approach need to be addressed for future research direction. First, the performance gain issues of the CFS scheduler-based embedded Linux system, and second, the performance trade off issues when there are tasks with real-time priorities.

At latest version of the Linux kernel, the existing O(1) scheduler was replaced by the CFS scheduler[4] in order to maximize fairness of running tasks. This replacement was applied after 2.6.23 version. Our proposed approach shows some performance issues in evaluating application responsiveness at the CFS scheduler-based embedded Linux system.

For example, the time slot gap generated by our approach in the CFS scheduler is usually smaller than that of O(1) scheduler. This results not much performance gain in the CFS scheduler. CFS scheduler merged in Linux 2.6.23 by Ingo Molnar is that nice value entered by the application developer was replaced from time slice table to weight table. This is possible through vruntime (virtual run-time) which schedules tasks fairly. If two nice values produce a small gap like 0, instead of a large gap like 10, it is not easy to produce any performance gain. Therefore, we need a new approach for task scheduling policy which produces a wide gap for better performance gain even in the CFS scheduler-based embedded Linux system.

Second, the proposed approach doesn't consider cases when some tasks have real-time priorities. Any task with real-time priority share CPU time with others, so we need to find a preemptive way to replace these tasks with more emergent tasks.

9 Conclusions

We have shown through examples that current NPTL thread model, which was introduced in Linux 2.6 for improvements of performance and scalability, has several limitations. These limitations sometimes cause users to wait several seconds tediously after they launch an application downloaded from app-stores.

In general, the nature of embedded system environment has physical conditions with limited CPU and memory. Therefore, the existing embedded systems using NPTL are needed to improve by operating lightly and speedily with the best technical methods.

We introduced several approaches of improving the current NPTL thread model: a suitable thread stack size for embedded environments, thread naming interface expansion for optimization, supports of thread profiling and debugging components to minimize the boot time of embedded platform, a thread priority management method according to scheduling importance of thread application, an arbitrary or enforced thread scheduling control policy to speed up user application processing, selective source compile methods using modular configuration structure for minimizing memory footprint, etc.

We also have shown that the improved NPTL thread model has better performance and fewer memory footprint compared to the current model.

Our results confirm that the existing NPTL thread model in Linux can be utilized in embedded system through the several improvement features. This provides cost effective development opportunity for embedded developers to start developing thread models for embedded systems through existing open source like Linux.

10 Acknowledgments

We would like to thank Wonsuk Lee for his valuable reviews and comments. We would like to thank my Samsung colleagues KyungIm Jung and Jae-Min Ryu for some suggestions and review of the article. We'd also like to thank all those members of the Linux kernel development community who have contributed patches.

References

[1] Sebastien DECUGIS. Nptl stabilization project(nptl tests and trace). In *Ottawa Linux Symposium*, 2005.

[2] Free Software Foundation(FSF). GCC Online Manual(Options that control optimization). `http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options`.

[3] Steven J. Hill. Native posix threads library (nptl) support for uclibc. In *Ottawa Linux Symposium*, 2006.

[4] Ingo Molnar. CFS Scheduler Design. `http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt`.

[5] Ingo Molnar. PI-futex. `http://lwn.net/Articles/102216/`.

[6] Roland McGrath. GNU C Library(Glibc). `http://www.gnu.org/software/libc/`.

[7] Rusty Russell. Fast userlevel locking in linux. In *Ottawa Linux Symposium*, 2002.

[8] Steven Rostedt. RT-mutex subsystem with PI support. Linux kernel documentation: `kernel/Documentation/{rt-mutex.txt|rt-mutex-design.txt}`.

[9] Ulrich Drepper. Futexes Are Tricky. `http://www.akkadia.org/drepper/futex.pdf`.

[10] Ulrich Drepper. Native Posix Thread Library for Linux. `http://people.redhat.com/drepper/nptl-design.pdf`.

[11] Ulrich Drepper. [TLS]ELF Handler For Thread-Local Storage. `http://people.redhat.com/drepper/tls.pdf`.

[12] Ulrich Drepper. What Every Programmer Should Know About Memory. `http://www.akkadia.org/drepper/cpumemory.pdf`.

Analysis of Disk Access Patterns on File Systems for Content Addressable Storage

Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho
National Institute of Advanced Industrial Science and Technology
`{ k.suzaki | k-iijima | yagi-toshiki | c.artho } @aist.go.jp`

Abstract

CAS (Content Addressable Storage) is virtual disk with deduplication, which merges same-content chunks and reduces the consumption of physical storage. The performance of CAS depends on the allocation strategy of the individual file system and its access patterns (size, frequency, and locality of reference) since the effect of merging depends on the size of a chunk (access unit) used in deduplication.

We propose a method to evaluate the affinity between file system and CAS, which compares the degree of deduplication by storing many same-contents files throughout a file system. The results show the affinity and semantic gap between the file systems (ext3, ext4, XFS, JFS, ReiserFS (they are bootable file systems), NILFS, btrfs, FAT32 and NTFS, and CAS.

We also measured disk accesses through five bootable file systems at installation (Ubuntu 10.10) and at boot time, and found a variety of access patterns, even if same contents were installed. The results indicate that the five file systems allocate data scattered from a macroscopic view, but keep block contiguity for data from a microscopic view.

1 Introduction

Content Addressable Storage (CAS) is becoming a popular method to manage virtual disks for many instances of virtual machines [2, 6]. In CAS systems, data is managed in chunks, and it is addressed not by its physical location but by a name derived from the content of that data (usually a secure hash is used as a unique name). A CAS system can reduce the use of physical disk space by deduplication, which merges same-content chunks with a unique name.

CAS provides a universal virtual block device and accepts any file system on it. The performance depends on data allocations and their access patterns through the file system, because each file system has techniques to optimize space usage and I/O performance. The optimizations include data alignment, contiguous allocation, disk prefetching, lazy evaluation, and so on. These factors make the file system a key factor for the performance of CAS.

From the view of the disk, a file system works as a "filter" to allocate data. Even if the same contents are saved, access patterns differ between file systems. Especially Linux has many file systems, because Linux supports a wide variety of targets, from mobile devices to super computers. In this paper, we propose a method to evaluate the affinity between file system and CAS. The method evaluates the effect of deduplication when many same-content files are stored throughout a file system.

We also analyze the real behavior of bootable file systems on CAS. We measure access patterns at installation (write-centric processing) and boot time (read-centric processing). From the results, we investigate the affinity between file system and CAS behavior.

This paper is organized as follows. Section 2 reviews features of CAS systems and Section 3 describes features of Linux file systems. Section 4 proposes the method to measure the affinity between file system and CAS. Section 5 report the results, the affinity and real behavior at installation and boot time. Section 6 discusses future works. Section 7 summarizes our conclusions.

2 Content Addressable Storage

This section describes features of CAS (Content Addressable Storage) systems. A pioneering CAS system

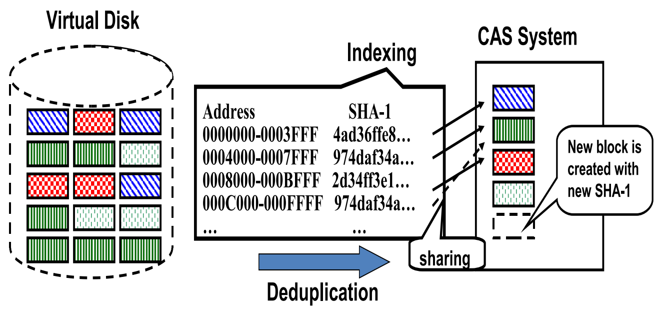

Virtual Disk
Indexing
CAS System
Address SHA-1
0000000-0003FFF 4ad36ffe8...
0004000-0007FFF 974daf34a...
0008000-000BFFF 2d34ff3e1...
000C000-000FFFF 974daf34a...
... ...
New block is created with new SHA-1
sharing
Deduplication

Figure 1: Virtual disk managed by CAS system.

is Venti developed for the Plan9 data archive [9]. Venti is a block storage system in which chunks are identified by a collision-resistant cryptographic hash.

Figure 1 shows a virtual disk managed by CAS. The virtual disk is divided by chunk for each region. Each chunk is named by its hash-value, and stored in a database of the CAS system. The chunks are managed by the mapping table, which translates from address to hash-value. If the contents of chunks are the same, the hash-value is same and the CAS system can reduce its storage space consumption. A chunk is also self-verifying with its hash digest and can keep data integrity.

In this paper we use LBCAS (LoopBack Contents Addressable CAS) version 2 [11, 12]. LBCAS offers a loopback block file (virtual disk) which is managed by FUSE (Filesystem in Userspace) [1]. Chunk data is created when a write access is issued to the virtual disk. The chunks are stored in a Berkeley DB [8] and managed by their SHA-1 hash-value. The size of a chunk is defined by the configuration (32 KB – 512 KB). The driver of LBCAS has a memory cache for 32 chunks. When a chunk overflows from the cache, the data is written to the Berkeley DB.

3 File Systems

Many file systems are developed for Linux, and each of them has its own advantages. In this paper, we treat 9 file systems: ext3, ext4, XFS, JFS, ReiserFS, NILFS, btrfs, FAT32 and NTFS. Unfortunately, not all of them can be used a root file system, because boot loader and installer have to recognize them. We used five file systems (ext3, ext4, XFS, JFS, ReiserFS) to investigate the behavior at installation and boot time.

3.1 Linux File Systems

This section describes the features of 9 file systems used in this paper.

Ext3 is the default file system on many Linux distributions. It extends ext2 with journaling. ext3 keeps compatibility with ext2, including some limitations, such as no extent allocation, no dynamic allocation of i-nodes, etc.

Ext4 [7] succeeds ext3 and extends it with *extent allocation* and *delayed allocation*. Extent allocation keeps contiguous physical blocks for a file and reduces fragmentation. Delayed allocation is a technique to reduce file fragmentation, which is also used by XFS.

JFS is a 64-bit journaling file system originally created by IBM for AIX. JFS uses a B+ tree to accelerate lookups in directories. JFS dynamically allocates space for i-nodes as necessary. JFS increases disk I/O performance by using *allocation groups* and extent allocation. An allocation group is a sub-volume in a file system that keeps track of free blocks and file data on its own. JFS contains effective methods to use allocation groups.

XFS is a high-performance journaling file system originally created by Silicon Graphics. XFS increases disk I/O performance by using allocation groups, extent allocation, delayed allocation, and *variable block size*. Variable block size allows XFS to be created with block sizes ranging between 512 B and 64 KB, increasing I/O bandwidth when large files are created. Delayed allocation makes it possible to allocate a contiguous group of blocks, reducing fragmentation.

ReiserFS (version 3) is the first journaling file system to be included in the standard Linux kernel. ReiserFS has the *tail packing* optimization which allocates last partial blocks of multiple files into a single block. The technique can reduce the internal fragmentation of files.

NILFS [4] is a stackable file system which is also called log-structured file system. NILFS allocates data in succession from the top of a disk. The sequential write achieves high I/O throughput on a real block device. The data on the log-structured format are only appended and never overwritten. In particular, a previous version of a file can be retrieved in the file system.

Btrfs is a new file system which features *copy-on-write*. Copy-on-write is used for creating a snapshot and for

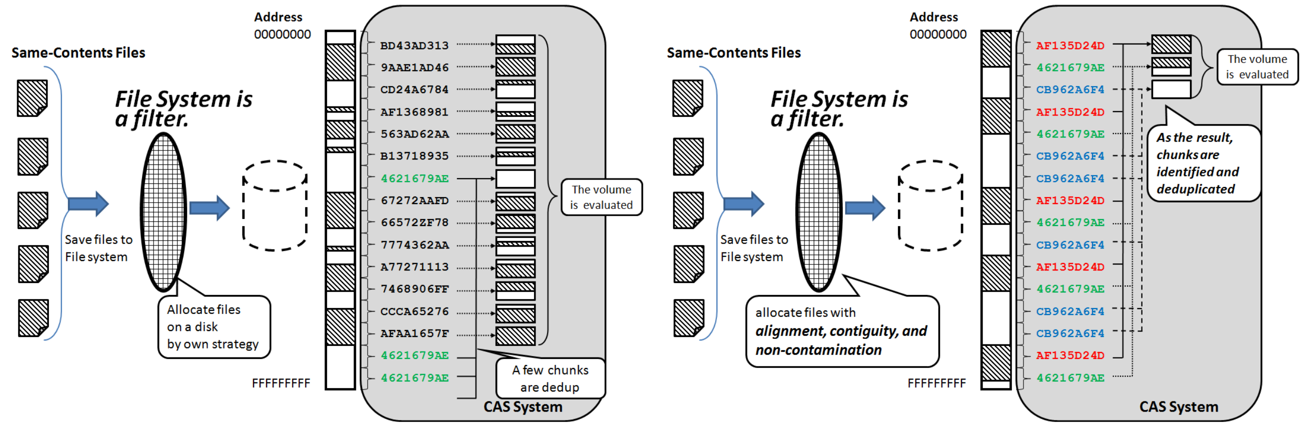


Figure 2: Affinity evaluation between file system and CAS. The left figure shows bad allocation and the right figure shows good allocation for CAS.

cloning. Btrfs has many new features which includes extent allocation implemented on ext4.

FAT32 and NTFS are file systems for Windows. They are not UNIX-style file systems and do not use i-nodes. The data is managed by a unit called "cluster", which is a contiguous groups of hardware sectors. The original development of FAT32 comes from floppy disks and has simple structure. FAT32 manages the clusters with an array, and does not perform well on large disks. NTFS developed for WindowsNT and has several improvements over FAT, which includes extent allocation. NTFS manage a file with the Master File Table (MFT) containing meta-data about every file and directory. The details of NTFS are not open, and the drivers for Linux are developed in many ways. Currently most Linux distributions use the NTFS-3G driver.

3.2 Bootable File System

The boot loader has to recognize a file system in order to load the kernel. The currently popular boot loader *GRUB* recognizes some file systems. In order to analyze the behavior at installation and boot time, we select five popular file systems (ext3, ext4, XFS, JFS, and ReiserFS version3) recognized by GRUB.

3.3 System Installation on a File System

Even if the same applications are installed on a file system, the installer of a Linux distribution recognizes the target file system, and customizes some files for it. For example, the initial ram disk image "initrd" is customized for a file system. A kernel and initrd are loaded by GRUB from a file system at boot time, and the kernel uses the configuration files to mount a file system on a disk as the root file system.

The root file system has to include additional files to maintain itself. Some of them are management tools for the file system. Furthermore, each file system has special features. For example ext3 and 4 file systems have a `lost+found` directory to retrieve lost files, which other file systems lack. However, the differences are small and they are negligible for installation and booting.

4 Affinity between File System and CAS

File systems allocate data on a disk; in doing so, the act as a filter. Each filter changes the location of data by its own strategy. Depending on the location, the effect of deduplication changes. We evaluate the difference from the view of deduplication.

We develop a method to evaluate the affinity between file system and CAS. The idea is simple. Ideally, even if many same-content files are saved on CAS, the total disk usage of CAS will be close to the size of one file, because all files are deduplicated. *Namely, the closer the total disk usage is to the size of one file, the better the allocation strategy for CAS.*

For example, when 1,000 files with 1MB same-content data are stored on a disk through a normal file system, it will use 1,000 MB. However, if deduplication of CAS works perfectly, the increase will amount to only 1MB.

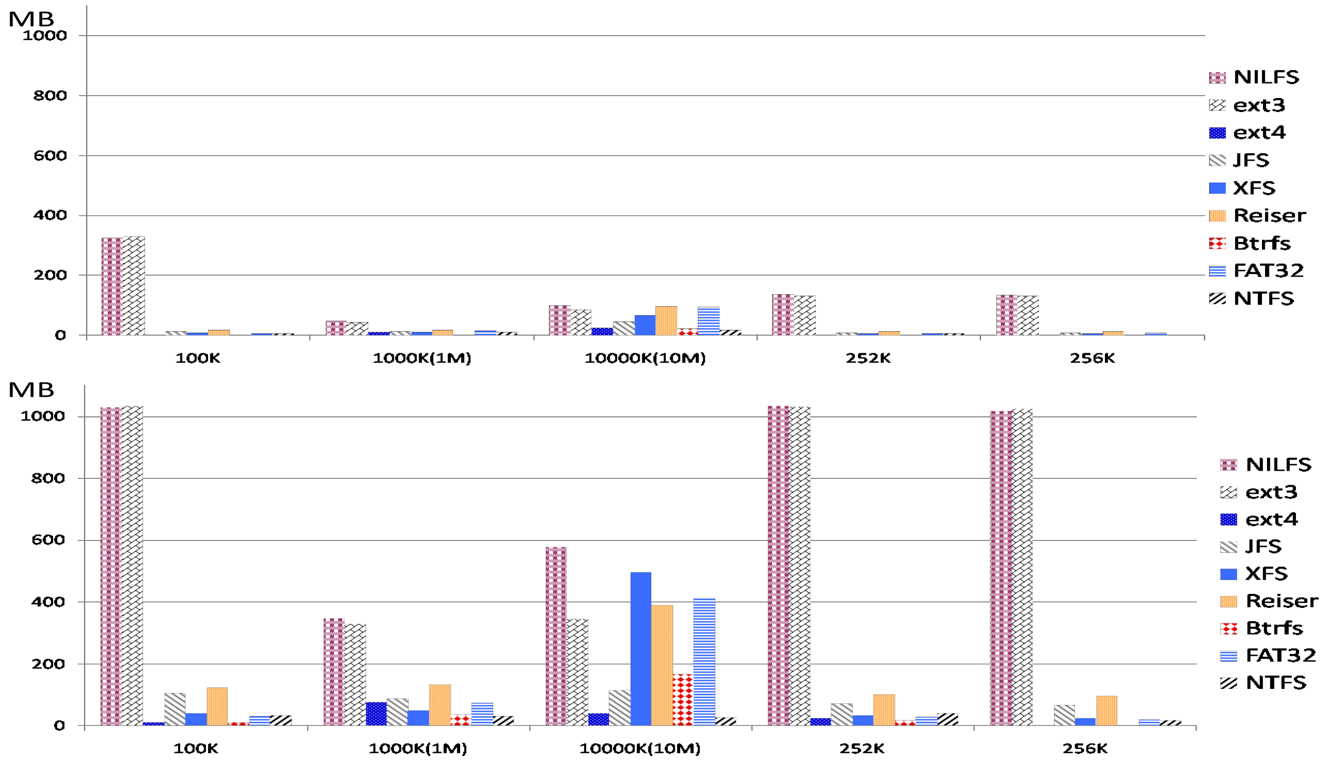


Figure 3: The increase of CAS space when 10,000 100 KB-files, 1,000 1,000 KB-files, 100 10,000 KB-files, 3,968 252 KB-files and 3,906 256 KB-files (which consume nominal 1GB) are stored on each file system (ext3, ext4, XFS, JFS, ReiserFS, NILFS, btrfs, FAT32 and NTFS). The upper figure shows the results on 32 KB LBCAS and the lower figure shows the results on 256 KB LBCAS. A small increase indicates good deduplication.

Figure 2 shows the image of the evaluation. The right of Figure 2 shows poor allocation by a file system. At that time, the locations of data are scattered and will not deduplicated, except the non-used (zero-cleared by initialization) region. The left of Figure 2 shows good allocation. The whole data is deduplicated and the consumption of CAS is close to the size of the file.

The affinity comes from (1) Alignment matching, (2) Contiguous allocation of data blocks, and (3) Non-contamination with other data in a CAS chunk. If the allocation strategy of a file system aligns data of a file at the alignment of a CAS chunk, it increases the chance of the file being deduplicated. Contiguous allocation of data blocks is also important to fill a chunk with same-content data. Non-contamination comes into play when a chunk is not entirely filled up with contiguous allocation of data blocks. At that time, the remainder of a chunk should not be filled with other data. However, some techniques pack data into a small empty region and reduce the chance of data to be deduplicated. For example, tail packing will contaminate a chunk.

The evaluation measures the affinity by determining the total volume used by CAS. Unfortunately, this methodology is still simple, because it does not care of the volume management mechanism (for example, bitmap table to manage free space) and meta-data which is used to identify the locations of contents with file names. A volume management mechanism is fixed-size and the update may be small and negligible. However, meta-data is created for each file and the volume consumption of meta-data is non-negligible when a file is small. On a real evaluation we must care about the use of meta-data.

5 Affinity and Performance Evaluation

This section evaluates the affinity and performance between file system and CAS. The affinity of deduplication is described in section 5.1, and the real performance of CAS on a file system is described in section 5.2.

5.1 Affinity evaluation between File System and CAS

We measure the effect of deduplication, when many same-content files are saved. We used random data as contents, because they are not deduplicated with other files. We tried to save 100 KB, 1,000 KB (1 MB), 10,000 KB (10 MB) random data files to fill 1 GB in 4 GB LBCAS system. Namely, 10,000 files for 100 KB, 1,000 files for 1 MB, 100 files for 10 MB were used.

The evaluation has to consider meta-data for each file. We assume that a meta-data consists of 256 bytes for a file. This means that the 10,000 files consume 256 KB on disk for meta-data. The consumption of meta-data is non-negligible when the deduplicated file is small. For example, at the 100 KB case, 256 KB is used for meta-data (256 B * 10,000) and the ideal consumption of CAS is 356 KB (100 KB + 256 KB). We have to care the increase. However, in the 1,000 KB case, the total is 1,025.6 KB and at 10,000 KB case, the total is 10,002.56 KB.

We also tried to save 256 KB and 252 KB random data files to check the suitable size for CAS deduplication. If file system allocates the files in succession, 256 KB (64 4 KB-file-system-blocks) files will fit to 256 KB and 32 KB chunks many times. We assume a stackable file system corresponds to this case. When 252 KB (63 4 KB-file-system-blocks) files are saved, 256 KB chunks will each fit to 64 allocations. If a block (4KB) is used between 2 contiguous files for meta-data or something, 252 KB data will also fit to 256 KB.

5.1.1 Experimental results

Figure 3 shows the increase of CAS when the files are stored on 32 KB LBCAS and 256 KB LBCAS. The results are the average of three trials. They show the different effect of deduplication on each file system. When the chunk size is larger than the size of test file, a file does not fill a chunk, even if a file is allocated continuously. At that time, a chunk is subject to contamination by other data. For example, a 256 KB chunk is not filled up with 100 KB, and 252 KB files. 256 KB file can fill up a 256 KB chunk, but there is no big difference between the 252 KB and 256 KB cases on any file systems.

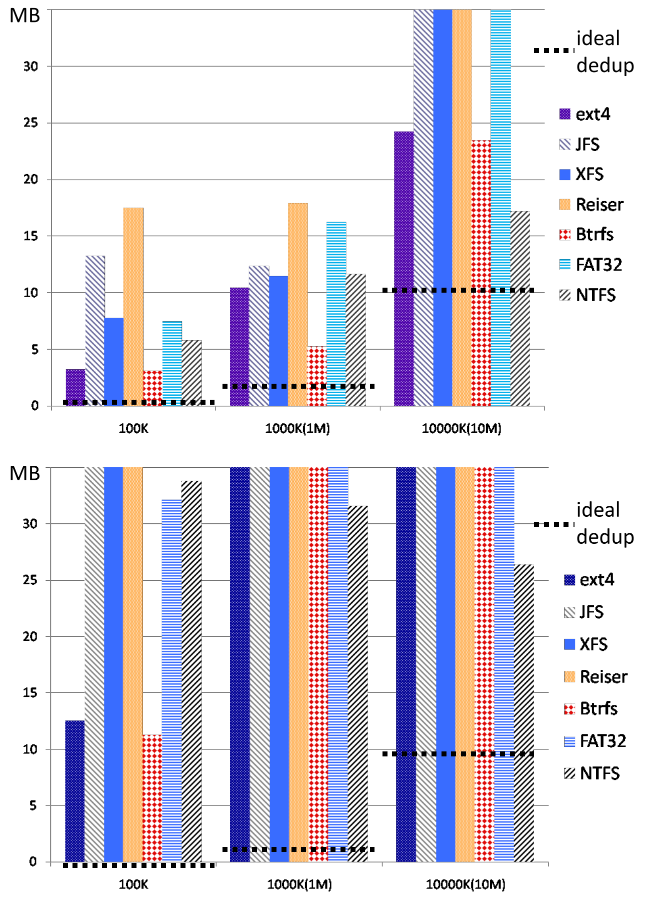


Figure 4: Details on the increase of CAS described in Figure 3. The maximum range is 35 MB. The figure includes line which indicates ideal deduplication. The upper figure shows the results on 32 KB LBCAS and the lower figure shows the results on 256 KB LBCAS. NTFS and ext3 are eliminated because they are out of range. The results of 252 KB and 256 KB files are also eliminated because we could not get alignment matching cased by contiguous allocation of data blocks.

Ext3 and NILFS show the worst results on 100 KB, 252 KB, and 256 KB files on 256 KB CAS. Their total space usage reaches 1 GB, which means no deduplication. The results become better on 32 KB CAS, but they are still worse than on the other file systems. The results of ext3 and NILFS on 100 KB files are about 2.5 times worse than for 252 KB and 256 KB on 32 KB CAS. We guess the results come from the ease of contamination when the chunk size is larger than a file. If a file is allocated at a contiguous region, such as a stackable file system, the chance to being contaminated is inversely proportional to the size of file.

XFS and FAT32 show good results except for the 10 MB file case on 256 KB CAS. We guess the data in a large file is not allocated contiguously, which hampers deduplication.

ReiserFS shows bad results on 256 KB CAS. We guess it comes from tail packing, which contaminates a chunk. 256 KB chunks are large and the penalty of not to being deduplicated has a big impact. ReiserFS also shows bad results on the 10 MB file case on 256 KB CAS. This is also caused by non-contiguous allocation for large files.

Figure 4 shows details of Figure 3, which limits the maximum (35 MB), eliminates two file systems (ext3 and NILFS) and 2 trials (252 KB and 256 KB files), and includes a line which indicates ideal deduplication. The results indicate ext4, btrfs, and NTFS on 32 KB CAS are close to the ideal case on any file size. They keep three features; alignment matching, contiguous allocation of data blocks, and non-contamination with other data. On 256 KB CAS, ext4 and btrfs show bad results on 10,000 KB and 100,000 KB files, although they show good results on 100 KB files. We guess ext4 and btrfs cannot keep contiguous allocation of data blocks, and suffer from contamination with other data on larger files.

5.1.2 Impact by chunk size

Figure 5 shows the ratio of consumption between 32 KB and 256 KB CAS. The ratio indicates the improvement by the smaller 32 KB chunk size. In the ideal case the ratio is 8 times. It means 256 KB chunks are not aligned or include slightly different data. For example, JFS in the 100 KB file case reduces the space consumption by 8 times on 32 KB CAS compared to 256 KB CAS. From another view, small ratios indicate that there is no improvement for small chunks when compared to

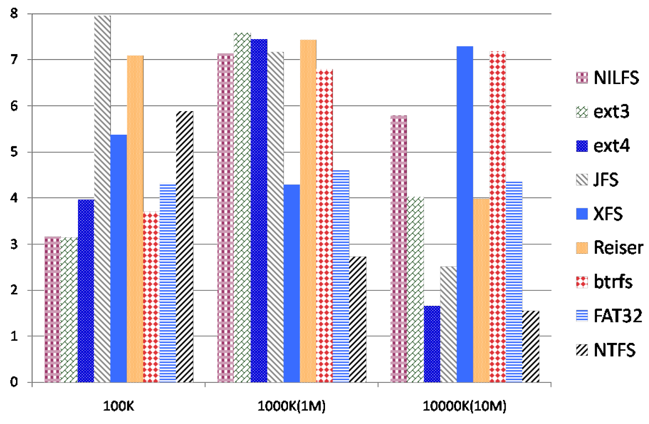


Figure 5: The ratio of space consumption on each file system. It shows the improvement of 32 KB CAS from 256 KB CAS.

larger 256 KB chunks. At that time, the user should use 256 KB chunks because the mapping table of 256 KB CAS is 8 times smaller than 32 KB CAS. For example, ext4 and NTFS show that there is a little impact on 10 MB file case.

5.1.3 Future work

The experiments were tried on an initial disk image which does not have fragmentation. We eliminate such evaluations, because of it is not clear which metrics to use to compare fragmented file systems, and we cannot decide factors for deduplication. We recognize these conditions to be important in real cases. Evaluations under these conditions are our next challenge.

5.2 CAS performance at installation and at boot time

We evaluated LBCAS performance at Linux installation and at boot time. We installed Ubuntu 11.04 desktop (Linux 2.6.38) on five file systems (ext3, ext4, XFS, JFS, and ReiserFS) of 4 GB LBCAS on KVM [3] virtual machine with 768 MB memory. KVM ran on a ThinkPAD T400 with an Intel Core2 Duo processor with 2 GB of memory. We compared the effect of 32 KB chunk and 256 KB chunk of LBCAS.

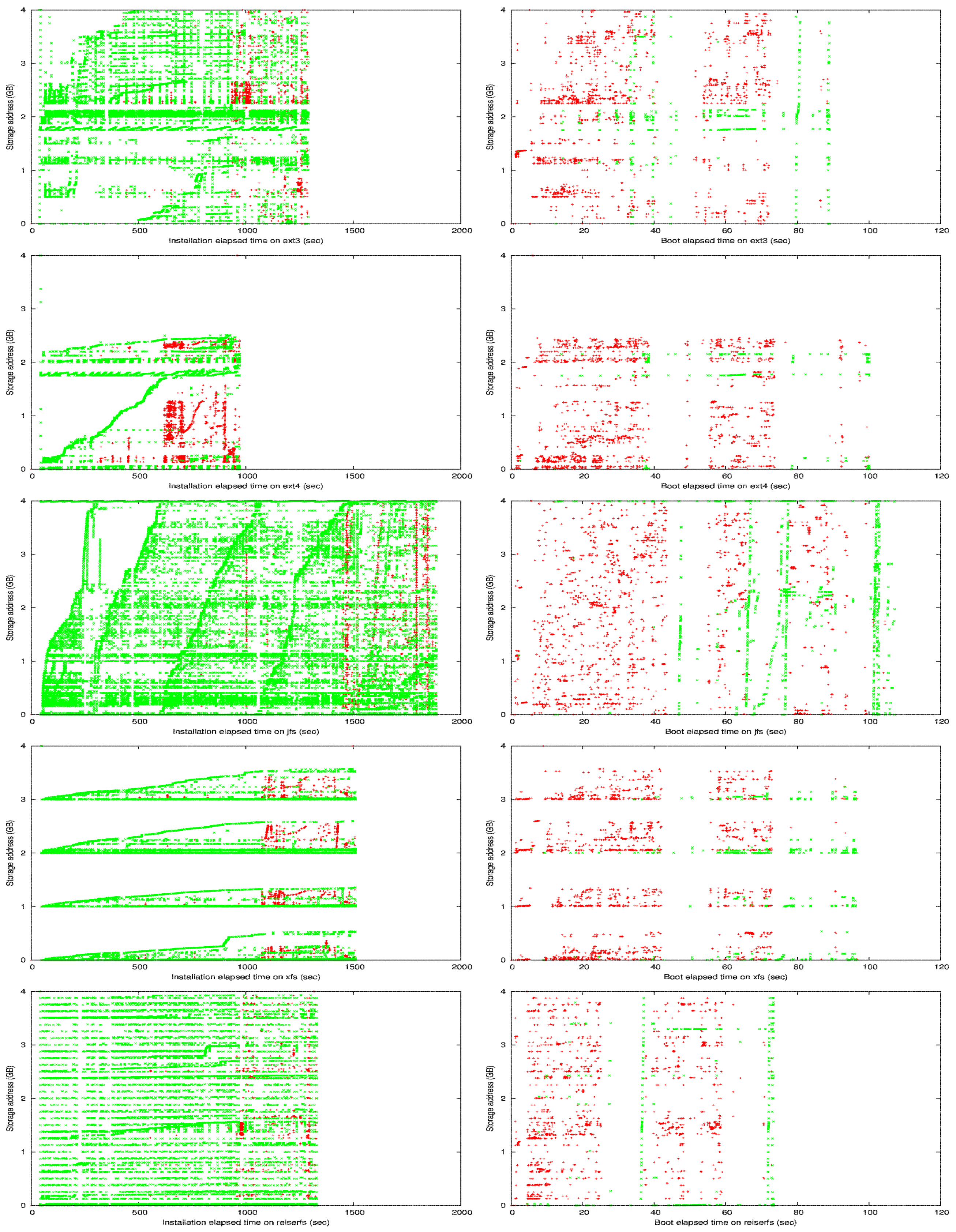


Figure 6: Access pattern at installation (left) and boot time (right) of Ubuntu on each file system. The X axis indicates elapsed time and Y indicates the address of the disk access (4GB). The green "X" plots indicate write accesses, and red "+" plots indicate read accesses. The file systems are ext3, ext4, JFS, XFS, and ReiserFS from top to bottom.

5.2.1 Access Trace at installation and boot time

Figure 6 shows the trace of access on each file system at installation and boot time. The graphs indicate that access patterns are different on each file system.

The left graphs show the installations. Most accesses are write operations. The installation includes creating the file system on CAS. The creation of a file system is started after 50 sec. Before 50 sec, the installation requires some preparations in memory.

The graphs show that the accesses of the five file systems are scattered in the 4 GB disk space. This property remains the same even if the disk size is changed to 2 GB or 8 GB. The results indicate that the five file systems have different allocation strategies and allocate data scattered from the macroscopic view of disk fragmentation.

The right graphs show booting. Booting is a read-centric process, but there are write operations at the end of boot, because several configuration files are updated. The access distributions follow the written data at installation, and there are no localities of reference from the macroscopic view.

5.2.2 Installation Time (Dynamic Feature)

Table 1 shows statistics of read and write accesses on each file system at installation time. Installation is a write-centric process and writes about 10 times as much data as it reads.

The upper three rows show the accesses issued by file system, access times, total volume of accesses, and the average. The results show the ability of a file system. Lower access times and fewer accesses are better. Ext4 shows the fewest access times and largest average access on write operations. This result may be the effect of delayed allocation, and it yields the fastest installation time (Figure 6). JFS shows the smallest total volume on write operations, but installation time is the worst, because the number of accesses is large. A CAS system is sensitive to access times, because the access unit is the chunk size even if an access is only 1 bit. The boot times of five file systems are almost proportional to the write access times, and do not follow the total volumes. Fewer accesses are better on a CAS system.

The lower four rows in Table 1 show the number of read and write chunks of 32 KB and 256 KB size. Fewer chunks are better. ReiserFS shows the best performance on read and write, respectively, on both 32 KB and 256 KB chunks. This indicates ReiserFS is good at locality of access, but accesses the same chunks many times.

5.2.3 Disk Image of LBCAS (Static Feature)

Table 2 shows statistics of a static LBCAS disk image with 32 KB and 256 KB chunks.

ReiserFS shows the fewest chunks, the smallest total volume used by chunks, and the largest volume of zero-cleared chunks on 32 KB and 256 KB chunks. The results of the number of chunks and total volume is about 10% less than other file systems, which might come from tail packing to reduce disk consumption. This is a good feature for a normal disk but it ruins the effect of deduplication mentioned in Section 5.1.1.

The efficiency, which indicates the ratio of effective data in a chunk, is more than 99% on all file systems using 32 KB chunks, although Figure 6 shows that data accesses look to be scattered. It shows that the allocation strategies of the file systems pack data in small region. However, the efficiencies of ext4 and XFS decrease to less than 94% when using 256 KB chunks. These results indicate that ext4 and XFS allocate data discretely for larger units. This feature suggests a suitable chunk size of a file system.

On deduplication, ext4 is the best on 32 KB and 256 KB chunks, which indicates that many same-content chunks are created. We guess the effect comes from alignment matching, contiguous allocation of data blocks, and non-contamination. This effect is predicted by the results mentioned in Section 5.1.1 Figure 4. Currently btrfs and NTFS are not bootable file systems and they are out of scope on current experiments.

We compared the ratio of same chunks between 2 CAS images which are installed same OS. Figure 7 shows the image of comparison. The ratios are measured on two types. One type is the ratio between 2 CAS images of different file systems. The results indicate the affinity between file systems from the view of CAS. It helps the understanding the effect of mixture of CAS images which has different file systems.

File system	ext3		ext4		JFS		XFS		ReiserFS	
	read	write	read	write	read	write	read	write	read	write
Access times	13,618	182,971	14,945	**164,124**	13,409	247,981	11,650	265,981	**10,739**	186,432
Total volume(MB)	385.6	3,808.2	413.3	3,745.1	393.3	**3,194.8**	385.4	4,740.6	**327.0**	3,946.4
Average (KB)	29.0	21.3	28.3	**23.4**	30.0	13.2	**33.9**	18.3	31.2	21.7
32KB LBCAS										
Number of chunks	13,152	68,759	14,978	70,401	14,209	65,141	15,260	66,337	**11,784**	**57,087**
Total volume (MB)	411.0	2,148.7	468.1	2,200.0	444.0	2,035.7	476.9	2,073.0	**368.3**	**1,784.0**
256KB LBCAS										
Number of chunks	2,314	8,684	2,712	9,222	2,687	8,207	2,839	8,499	**2,066**	**7,170**
Total volume (MB)	578.5	2,171	678.0	2,305.5	671.8	2,051.8	709.8	2,124.8	**516.5**	**1,792.5**

Table 1: Statistics of read/write accesses on each file system at installation time (dynamic feature). The bold figures indicate the best performance.

File system	ext3	ext4	JFS	XFS	ReiserFS
32 KB LBCAS					
Number of chunks	67,157	67,819	64,770	65,415	**56,671**
Total volume (MB)	2,148.1	2,192.3	2,035.0	2,059.2	**1,783.7**
Zero-cleared chunk (MB)	1,947.9	1,903.7	2,061.0	2,036.8	**2,312.3**
Effectiveness (%)	99.88	99.93	99.90	**99.99**	99.94
Dedupication (Total MB / Unique MB)	49.47/8.75	**73/19.16**	10.94/3.75	15.03/7.06	12.69/5.44
256 KB LBCAS					
Number of chunks	8,554	9,020	8,190	8,475	**7,156**
Total volume (MB)	2,169.8	2304.0	2,050.5	2,124.0	**1,792.0**
Zero-cleared chunk (MB)	1,926.3	1,792.0	2,045.5	1,972.0	**2,304.0**
Effectiveness (%)	98.40	93.13	99.12	92.10	**99.47**
Dedupication (Total MB / Unique MB)	31.25/2.0	**49.0/8.25**	3.0/1.75	5.25/1.75	3.0/0.5

Table 2: Statistics of a virtual disk for each file system (static feature). The row of effectiveness shows the ratio of blocks of file system in a chunk. It shows coverage of effective region in a chunk. The row of deduplication shows two data; total and unique. The total indicates the summation of same-content chunks. The unique indicates the summation of merged chunks with deduplication. The bold figures indicate the best performance.

File system	ext3		ext4		JFS		XFS		ReiserFS	
	read	write	read	write	read	write	read	write	read	write
Access times	6,115	3,653	5,663	3,458	6,260	2,894	6,199	4,383	**5,195**	**2,625**
Total volume (MB)	209.7	39.7	228.0	40.7	198.1	**17.2**	216.8	22.7	**187.7**	27.2
Average (KB)	35.1	11.1	41.2	**12.1**	32.4	6.1	35.8	5.31	**37.0**	10.6
32 KB LBCAS										
Number of chunks	8,065	1,491	8,548	1,522	8,130	1,204	8,507	**1,094**	**7,402**	1,295
Total volume (MB)	252.0	46.6	267.1	47.6	254.1	37.6	265.8	**34.2**	**231.3**	40.5
256KB LBCAS										
Number of chunks	1,508	292	1,624	**247**	1,941	712	1,767	372	**1,500**	367
Total volume (MB)	377.0	73.0	406.0	**61.8**	485.3	178	441.7	93.0	**375.0**	91.8

Table 3: Statistics of read/write accesses on each file system at boot time (dynamic feature). The bold figures indicate the best performance.

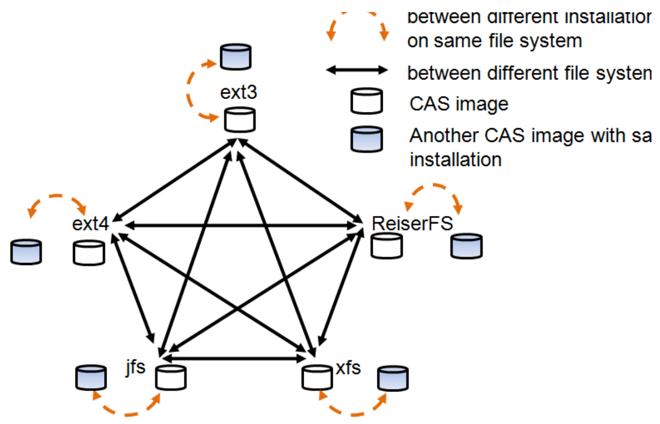

between different installation on same file system
between different file system
CAS image
Another CAS image with same installation
ext3
ext4
ReiserFS
jfs
xfs

Figure 7: Compare ratio of same chunks between 2 CAS images which are installed same OS. The ratios are measured on 2 types. One type is the ratio between 2 CAS images of different file systems. Another type is the ratio between 2 CAS images which install same OS on same file system at different time.

Another type of measurement is the ratio between 2 CAS images which install same OS on same file system at different time. The results indicate the suitable file systems which reduce consumption of physical resources, when some users install same OS on the CAS system with same file system.

Figure 8 shows the results. We measure the ratio on different chunk sizes from 4KB to 256KB, in order to know the effect. The upper figure shows the ratio between different file systems, which are illustrated in Figure 7 with solid lines. The results indicate that there are small differences on any combination. It means there are not strong affinities among the 5 file systems. The ratios of same chunks depend on chunk size. On 4KB chunk size, the ratio is very high from 80% to 90%, because chunk size matches the block size of most file systems, and most data blocks having the same contents will be same. The most difference comes from meta-data and file system management data, except ReiserFS which has tail packing. Tail packing reduced the consumption of storage 10% more than other file systems in Table 2. Unfortunately, it was known to cause negative impact on deduplication, because it contaminates a block for a file. The effect was measured in a single CAS which had many same files, mentioned in Section 5.1.1. However, the ratio of same chunks between 2 CAS images on 4KB chunk size is almost same to other file systems. It means that tail packing assigned same fractions of files in a block and keeps same chunk on different installa-

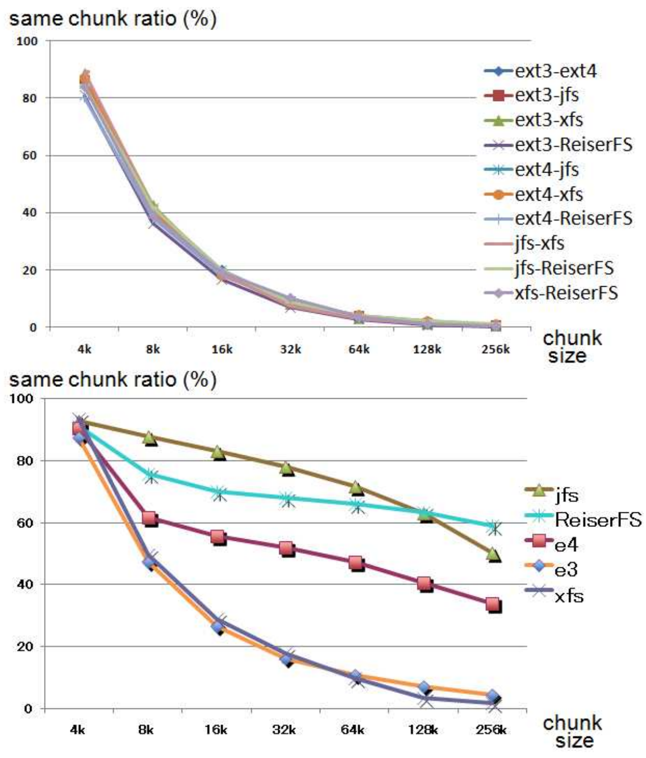


Figure 8: Ratio of same chunks between 2 CAS images on different chunk size from 4KB to 256KB. The upper figure shows the ratio between different file systems. The lower figure shows the ratio between different installations on same file system.

tion.

More than 8KB chunks show a ratio reduced inversely proportional to chunk size. 8KB and 16KB chunk sizes have 40% and 20% same chunks respectively on any file systems. The results indicate it is difficult to get same chunks between different file systems. It means we should not use different file systems on CAS system.

The lower in Figure 8 of shows ratio of same chunks between 2 CAS images which install same OS on same file system at different time. The ratios also depend on chunk size but the effects are different. Both jfs and Reiser do not reduce the ratio inversely proportional to chunk size. They keep high ratio of same chunk on larger chunk sizes. Especially jfs and ReiserFS keep 50% same chunks on 256KB chunk size. The results indicate that jfs, Reiser and ext4 allocate most files at same addresses on an installation, but ext3 and xfs allocate different addresses. We will investigate block allocation repeatability in next challenge. The results means we should use same file system on CAS system and the file system is one of jfs or ReiserFS.

Figure 9 shows the statistics of installed files on ext3. The upper figure indicates the number of files classified by size, and lower figure indicates total volume occupied by files classified by size. The number of files shows the case on ext3 but the results on different file systems are almost same. The lower figure is calculated using a minimum unit of 4KB block and each file is rounded up by 4KB. The calculation causes the big difference on ReiserFS case measured in Table 2, because tail packing reduces the consumption.

The upper figure indicates 77.9% files are less than 4KB. Less than 4KB file use only 1 block and do not affect contiguous allocation of data blocks. The result implies that we do not need to care about contiguous allocation, but less than 4KB files use only 20.1% of the storage showed in lower figure. The remaining portion, consisting of files larger than 4KB, requires contiguous allocation in order to achieve high deduplication. The investigation of the relation of file size and deduplication is not finished. We will continue the research.

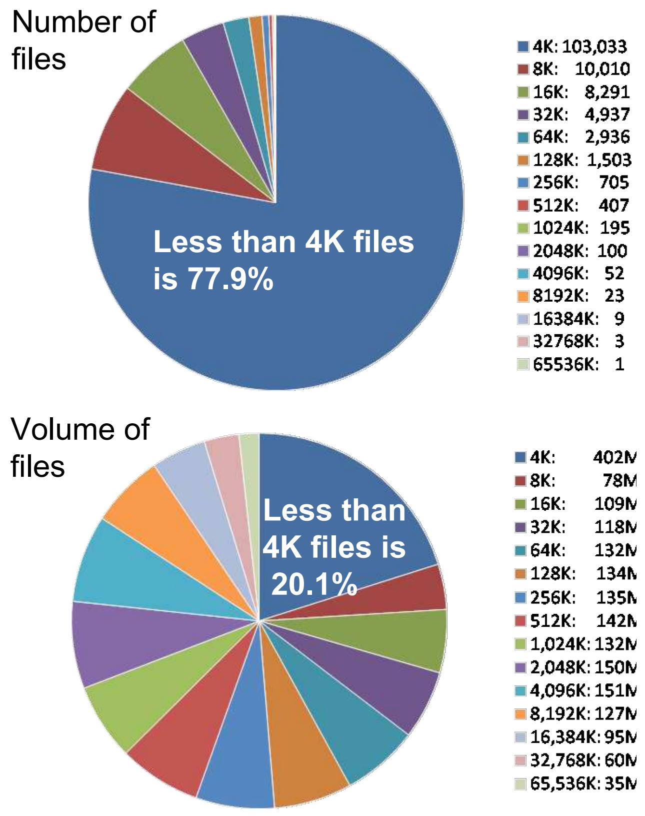


Figure 9: Statistics of installed files on ext3. The upper figure indicates the number of files classified by size, and lower figure indicates total volume occupied by files classified by size. Circular graphs show the percentage of each items (total 100%).

5.2.4 Boot Time (Dynamic Feature)

Table 3 shows statistics of read and write accesses for each file system at boot time. Boot is a read-centric process and has about twice as many read than write operations. The table format is the same as Table 1.

From the view of disk accesses (upper three rows), ReiserFS and XFS are the best in read and write operations, respectively. These features may be responsible for the fastest boot time shown in Figure 6. The largest average access size, however, occurs under ext4 for both operations. It might be a result of disk-prefetching contiguous data blocks allocated by extent allocation.

The lowest number of chunks on 32 KB occurs on ReiserFS and XFS on read and write operations, respectively. The lowest number of chunks on 256 KB occurs on ReiserFS and ext4 on read and write operations, respectively. The lowest number of read operations on ReiserFS explains the fast boot time.

6 Discussions

In this paper we treat CAS, which offers block-level deduplication, but there is another level of deduplication. We compare them in Section 6.1. The results in Section 5.2 lead us the importance of optimization on file system and CAS. We discuss two type of optimization in Sections 6.2 and 6.3.

6.1 Deduplication on file system level

CAS offers block-level deduplication, but deduplication is not limited to the block level. Deduplication can be applied at the file system level, as implemented by lessfs [5] and SDFS [10]. In this case, file system is limited to the original one, and there is no affinity problem with the file system.

File system deduplication means that file system includes the function of deduplication. It detects identical content in files, and merges the same content at the file system level. It does not care about block level restrictions. Namely, it does not care about block alignment matching, contiguous allocation, and contamination by other files.

The evaluation we proposed in this paper is not applied on file system deduplication, because all same-content files are deduplicated perfectly. We already confirm the effect of our evaluation method on lessfs and SDFS. They deduplicate all files well. In order to evaluate file system deduplication, we should use partially-similar-content files. For example, we tried files in which 256-bytes or 257-bytes of random data are repeated. The case with 256-bytes is deduplicated well, but files containing 257-bytes of repeated random data are not deduplicated well on lessfs and SDFS. It means they offer fixed-length deduplication. On fixed-length type, location of same-content data in a file is very important. Variable-length deduplication does not care about location and deduplicates both files well, but requires more comparison time.

File system deduplication has another disadvantage. A file system which has deduplication is usually a pseudo file system and is not usable as a bootable file system, because it is not recognized by boot loader. An operating system on a virtual machine has to use a loop-back file which is a pseudo block device, to install bootable file system. Therefore the affinity problem between file system on a virtual machine and loop-back file supported by file system deduplication will occur again.

6.2 FS Optimization for CAS

Boot time optimization for CAS is proposed in paper [12]. It takes a trace of block accesses on ext3 and reallocates data blocks in the file system. The data blocks in ext3 which are required to boot are arranged in line on the disk. This increases the read-ahead coverage of kernel prefetching. As a result, both the number of accesses and the number of CAS chunks are reduced.

This optimization is necessary for each file system on CAS. Optimization should consider the access profile as well as storage deduplication. Storage deduplication could be further increased by using a binary patch technique. We will investigate a delta encoding method which reuses existing block data.

6.3 CAS Optimization for FS

In cloud computing, the storage system can optimize a virtual disk for the file system used. Classically file systems have been optimized for a disk device. However, a virtual disk on cloud computing, which is managed by key-value storage, could change its behavior for a file system. For example, when a file system prefetches extra data, virtual storage could push the data to memory in advance. We will investigate an intelligent virtual storage based on the analysis of file system features.

7 Conclusions

We analyzed the affinity between nine Linux file systems (ext3, ext4, XFS, JFS, ReiserFS, which are bootable file systems, and NILFS, btrfs, FAT32 and NTFS) and CAS with 32 KB and 256 KB chunks. We proposed a method to evaluate the degree of deduplication by storing many same-content files through a file system and showed the affinity between file system and CAS. We also evaluated file systems on CAS by measuring the access patterns at installation and boot time.

The evaluations with same-content files indicate the degree of deduplication in a file system and show the affinity between file system and CAS. We estimate the effects come from the alignment matching, contiguous allocation of data blocks, and non-contamination with other data. Ext4, btrfs, and NTFS show good affinity for CAS.

At installation and boot time, ReiserFS shows good results, attributable mainly to reduced read and write accesses. The effect of deduplication on ReiserFS is not so high in a single image. ext4 shows good results on deduplication. The affinities between different file systems are little from the view of same chunks in CAS, but jfs and ReiserFS have many same chunks between different installations respectively. The results suggest that there is block allocation repeatability on jfs and ReiserFS. We will investigate it as next challenge.

The results of two types of experiments suggest the possibility of optimization of a file system and a virtual disk. On cloud computing, an intelligent storage system could change its behavior for a file system.

References

[1] FUSE: Filesystem in userspace, *http://fuse.sourceforge.net/*

[2] K. Jin and E. L. Miler, *The Effectiveness of Deduplication on Virtual Machine Disk Images*, The Israeli Experimental Systems Conference, SYSTOR 2009.

[3] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, *kvm: the Linux Virtual Machine Monitor*, Proceedings of Linux Symposium 2007, Volume 1, pp. 225–230, June 2007.

[4] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai, *The Linux implementation of a log-structured file system*, ACM Operating Systems Review, Vol. 40 Issue 3, 2006.

[5] lessfs: open Source data deduplication for less, *http://www.lessfs.com/*

[6] A. Liguori, E.V. Hensbergen, *Experiences with Content Addressable Storage and Virtual Disks*, First Workshop on I/O Virtualization (WIOV), December, 2008.

[7] A. Mathur, M.Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, *The new ext4 filesystem: current status and future plans*, Proceedings of Linux Symposium 2007, Volume 2, pp. 21–34, June 2007.

[8] M.A. Olson, K. Bostic, and M. Seltzer, *Berkeley DB*, Proceedings of USENIX FREENIX 1999, pp. 183–191, June 1999.

[9] S. Quinlan and S. Dorward, *Venti: A New Approach to Archival Storage*, Proceedings of the 1st USENIX Conference on File and Storage Technologies, Monterey CA, January, 2002.

[10] SDFS: A user space deduplication file system, *http://www.opendedup.org/*

[11] K. Suzaki, T. Yagi, K. Iijima, and N.A. Quynh, *OS Circular: Internet Client for Reference*, Proceedings of the 21st Large Installation System Administration Conference, pp.105-116, Dallas TX, November, 2007.

[12] K. Suzaki, T. Yagi, K. Iijima, C. Artho and Y. Watanabe, *Effect of Disk Prefetching of Guest OS on Storage Deduplication*, ASPLOS Workshop RESoLVE 2011.

Verifications around the Linux kernel

Alexandre Lissy
Mandriva
alissy@mandriva.com
Laboratoire d'Informatique de l'Université de Tours
alexandre.lissy@etu.univ-tours.fr

Stéphane Laurière
Mandriva
slauriere@mandriva.com

Patrick Martineau
Laboratoire d'Informatique de l'Université de Tours
patrick.martineau@univ-tours.fr

Abstract

Ensuring software safety has always been needed, whether you are designing an on-board aircraft computer or next-gen mobile phone, even if the purpose of the verification is not the same in both cases. We propose to show the current state of the art of work around the verification of the Linux kernel, and by extension also present what has been done on other kernels. We will conclude with future needs that must be addressed, and some way of improvements that should be followed.

1 Introduction

The Linux kernel is an important piece of code, both in terms of size (it is currently evaluated at 5 million lines of code) and of role. Like any software, it needs to be tested. Testing generally occurs through users running the kernel and reporting the bugs they identify. Achieving good coverage of the code with this manual approach is however not feasible: code analysis can be used to ease verification and bring it to a new level. These techniques are known as *static analysis* and have been used for several years, not only for the Linux kernel. Linux distributions do not use exactly the vanilla kernel and may have patches added, so it is important to be able to check not only upstream code but also downstream code. Having a clear view of what has already been done in the field of Linux kernel checking is a first but important step before deciding what kind of new verifications to work on. First, a more precise description of our objectives is available in section 2, followed by a presentation of some of the tools and related papers starting with one of the main reference in section 3, then presenting the SLAM initiative in section 4. The next section 5 provides a specific look at the Linux case, presenting SATURN in section 5.1, then some work around model checking in section 5.2. Coccinelle and Undertaker related work is presented respectively in section 5.3 and 5.4 just before presenting another way to approach the problem in section 5.5. Before concluding, some work being done to entirely verify an OS kernel in section 6 is presented.

2 Objectives

Our main goal is to bring more stability from a runtime point of view to the kernel, and in fact to any software that is packaged and available in Linux distributions in general, and in the Mandriva Linux distribution in particular. The kernel is an interesting piece of code, because it evolves rapidly (so that complex quality assurance is hard) and the codebase differs between the vanilla one and the one used by distributions, which apply patches relating typically to hardware compatibility, extended features, backports, etc. This part of a Linux distribution is also a good candidate because of its aura: is it easier to find people interested in augmenting the quality of this kind of code than for other parts of other critical code, even though some parts of the kernel are less likely to bring attention. Thus the question: "can we model-check the Linux kernel?", which led to a first part of the work: what is the current status of checking techniques in Linux kernel and also in any other kernel component? What is the literature on model-checking techniques applied to the same case? What has been already tested? What are the current known limits of model-checking? What are the current known limits of tools applying model-checking? Even "what does model checking exactly mean in the context of the Linux kernel?". So the objectives of the present article is to summarize what has been collected as part of this digging work, trying to give a view of the current state of the model checking techniques applied to Linux kernel and other kernel (Windows, seL4). It is in no way completely exhaustive but a hopefully good compilation of several "related work" section in many papers and communications that have been found.

3 Foundation: From Stanford to Coverity

The ability to verify the source code of a kernel has been studied by Engler et al., in [9] in which they present some new way to enforce system reliability by using rules in the compiler written by the programmers themselves. By "systems", they do not only target kernel, but also libraries or embedded systems. They intend to check interfaces usage, i.e. whether APIs are used correctly. First, they describe why in their opinion model-checking is not a viable option for this problem with the following arguments:

- Specifications are costly to build, hard to write
- They may not exactly abstract what they should
- Real life shows that they often miss some points and are over-simplifying

So the authors introduce *Meta-Level Compilation (MC)* to allow checking programmer-written rules with an extended-version of GCC, **xg++** implementing the *metal* language, which is high-level and state-machine based. The technique is not new, and the authors mention previous work aiming at similar objectives like *ctool* and *Open C++*. The lack of data flow information within these tools is however identified by the authors as a key limitation which make them harder to use and less powerful than Meta-Level Compilation. State-machines in the *metal* language work by matching interesting features (using C++-written patterns) in the analyzed code and then causing transitions between states. Meta-Level Compilation can be used not only to find bugs, but also for optimizing code: automatically detecting if a shared variable is never written enables identification of excessive locking usage; and the reverse might be used to detect non-protected variables. Of course, those kind of rules cannot be generic and are project-specific. *Metal*-written rules are first compiled using the `mcc` compiler, before being loaded into `xg++`. Several *checkers* are proposed as usage examples: assertions side-effects, compile-time assertions, correct usage of APIs (malloc/free, userspace pointer usage in kernel code), null pointer dereference, etc.

Coverity, a widely known tool to analyze source code, which has a partnership with the Department of Homeland Security (USA) [1] and was used to check many FLOSS projects, has its roots in the Stanford Checker (which is the implementation of Engler et al. work described in [9]). This checker is not available as free software. Dan Carpenter committed *Smatch*[2] to provide the same functionality in a FLOSS compatible way. A couple of years later, the same author committed a new paper, [10], which gives a feedback on both static analysis and model checking after several experiences with both. The paper consists mainly of real case studies in both areas, not to incriminate one or the other technique but to describe and compare. They first explain how they used both approaches:

- Using classical explicit state model checkers, with two approaches for the specifications problem: one automating the extraction of slice of functionalities translated into model-checking language, another model-checking directly the C code.
- Using meta-level compilation approach, classified as static analysis, which according to them reduces the work needed to find bugs. However, this method is unsound: errors can be missed. Also, they did not model the heap, tracked most variables values nor do alias analysis. They did their best to avoid the use of annotations in the code.

Regarding model-checking, they state the assumption "model-checking will find more bugs" is false: when checking an embedded system (FLASH), this technique found fewer (four times) bugs than the static analysis method, even though the model-checking approach identified some bugs that the previous method was not able to. The main reasons for the differences are: the code needs to be executed and the environment has to

[1] `http://scan.coverity.com/`
[2] `http://repo.or.cz/w/smatch.git`

be modeled. The discussion section brings up several interesting issues.

- They thought it would be hard to find many bugs in a complex system, but it turned out false: a codebase of at least one million lines of code not showing bugs is more a tip that there is a bug in the bug finder.
- It is easier to write code on how a property must be checked than why the property was violated.
- Not being too general is easier to handle for checking
- Hard to inspect errors may be left unfixed simply because users ignore the warning and does not understand it (they give an example from commercial *PREfix* tool)
- More analysis can result in useless analysis, because it will expose complex bugs.
- Another myth they destroy is that "all bugs matter": more bugs reported does not imply more fixes committed, regardless whether it is free software, because no importance ordering is given.

From their point of view, model checking is less powerful, mainly because it needs a good model of the environment (which is difficult, requiring weeks or months of work), and it cannot analyze more efficiently such large codebase than static analysis can do. However, model-checking has several advantages over static analysis which all result in stronger correctness results.

4 Microsoft: From SLAM Project to SDV

Starting in 1999, as a result of a brainstorming inside the Software Productivity Tools group at Microsoft Programmer Productivity Research Center[3], three projects were launched aiming at improving the quality of software, especially, but not limited to, drivers. All these projects shared a unique goal: checking interface usage. Only the approach to solve the problem is different, and a brief presentation of the two that did not last as long as SLAM is provided:

Vault A new programming language with pre/post conditions on the types, thus enabling definition of rules that the program must follow; it is now used as part of MSIL byte-code for CLR virtual machine, and is available as a Visual Studio plugin: [8].

ESP is closer to SLAM, but does not take the same course for the implementation and the trade-offs of the static analysis.

Regarding our topic, only SLAM is interesting, first because this project has been successful, and also because of the lack of information available on ESP project. The first major contribution to the problem of checking interface usage is the ability to abstract C program as Boolean programs[6, 5]. The Boolean program is created from the C programs by taking the Control Flow Graph, and pruning variables with Boolean variables. Boolean programs are interesting as reachability and termination is decidable. Building those is done using *Counter-Example Guided Abstraction Refinement (CEGAR)* [5, 7, 2]. Along with their Boolean Program model, the author contributes a model-checking algorithm[5] to check this model.

The authors introduce an interface description language SLIC[7] and present the CEGAR process articulated over three tools: `C2BP` which transforms C programs to boolean programs, `Bebop` allows model-checking of boolean programs and `Newton` which checks path feasibility between C program and boolean program. *Bebop* makes uses of Context-Free Language Reachability results [32]. As of 2002-2003, the SLAM project was working well and could be used as a basis for "Static Driver Verifier" [3]: the goal is to have an easy-to-use SLAM, to be distributed for use by driver developers. Joint work with Drivers Teams led to a lot of new checking rules being written not by SLAM developers. In [4], the authors show that over the 470 rules, only 60 were written by checking "experts": this validates the widespread possibility for developers of interfaces to provide checking rules. In the same paper, they also note that the use of SDV and rules complexity has been decisive in the new driver model design for latest Windows releases: too complex rules meant complex checking, and thus it has been decided to re-design the interface to ease checks. Work to improve SLAM into SLAM2 has been done as stated in [2]: less false-positives (from 19.7% to 0.4% in the worst case reported), less timeouts (CEGAR loop unable to finish, from 6% to 3.2% in the worst case reported).

5 Linux SDV?

The Linux kernel has also been a basis for verification work, because it has widespread diffusion and is easily fixable. An overview of the work that has been done and the projects evolving around this topic is proposed. Engler et al. [10] already started some important work toward checking the Linux kernel, which they continue to do with their Coverity tool.

5.1 Saturn

In [31], the authors describe a generic framework aimed at detecting errors in large codebase. Source code is translated into boolean constraints that can be used to check properties thanks to SAT-solving tools. The advantages of the method as claimed by the authors are the following:

Precision no abstraction is being done

Flexibility boolean constraints does not put any requirements on the language used

Compacity boolean formulas can be simplified when doing intra-procedural analysis. For inter-procedural, a summary of the function is compiled and used to process the verification. It is required as the authors aims at large code base and SAT is NP-complete

SATURN processing can be parallelized, the authors claiming that an 80-processor cluster reduces the processing from 23 hours to 50 minutes, when analyzing the Linux kernel. They also contribute experimental results of the running against the previous codebase: more errors found than in previous literature's work and fewer false positives. Another interesting contribution is the autoslicing, allowing to limit the codebase to relevant parts regarding the property analyzed. They contribute two case studies:

- Checking finite state properties, also known as temporal safety properties (as in [5, 6, 7]), with an example on verifying locks
- Checking for memory leaks (not only on the Linux kernel, but also on samba, openssl, postfix, openssh and binutils)

5.2 Model Checking Concurrent Linux Device Drivers

The authors present[30] an approach to model-check shared memory programs, thus enabling the automated verification of Linux device drivers. Their tool, `DDVerify`, uses predicate abstraction and their technique introduces concurrent software verification with predicate abstraction. They justify their approach of targeting shared memory because the resulting bugs are very hard to discover and understand. They also contribute a concurrent model of the needed parts of the Linux kernel API. `DDVerify` generates an annotated version of the source code to be checked with a SAT-solver (`SatAbs`), using assertions given by the user. Dealing with concurrency implies being able to deal with threads: it has implications on the state space resulting of the model. To ensure finite space (to be able to use the `SMV checker`), their tool uses a static and bounded number of threads. Further work is needed to allow the use of infinite dynamic number of threads and the `Boppo` checker. To automate the process, they implement a *CEGAR* loop as described in [5, 7, 2]. However, they conclude with two important facts:

- In their approach, the performance of the model-checker is critical: most of the execution time is spent on verifying the abstracted program. They used three tools, `Cadence SMV`, `Boppo` and `Bp`
- Model-Checkers performances for Concurrent Device Drivers needs to be improved significantly to be able to cope with real world drivers

They also claim to have a better (more accurate) model for the operating system than in SLAM or BLAST projects. They provide results of their benchmarks in two cases: sequential and concurrent, using a small benchmark driver, machwzd (less than 500 lines of code, using locks, IO and timers), and running on Intel Xeon 3GHz, 4GB RAM. The `bp` model-checker executes in about one second, while `Boppo` and `SMV` are around 30 seconds, in the sequential case. When going concurrent, only `SMV` is available, and runs around 400 seconds on average, with peaks at 1800 seconds. Despite poor performances, their tool found one new bug related to memory regions usage (`request_region()` function).

5.3 Coccinelle

Coccinelle is a French word for a kind of beetle that eats bugs. The goal of the Coccinelle project is to kill bugs before they are introduced. It started with [14] in which the authors describe and analyze recent *collateral evolutions* and contribute requirements of a helper tool: those evolutions are API changes that impact potentially a huge number of files in the whole kernel. They give three examples:

- Elimination of the `check_region()` function, which started in Linux 2.4.2 (released on February 2001) and that was still not completed by Linux 2.6.10 (released on December 2004).
- Addition of an extra argument to the `usb_submit_urb()` function, started in Linux 2.5.4 (released on February 2002) and that was still not completed by Linux 2.6.10.
- New function for copying data from userspace to kernel space `video_usercopy()`. Worse than previous examples, this new method was introduced in a driver for Linux 2.6.3 but the old one remained in place, and it resulted in the bug never being fixed, and the **new** method being **removed** as of Linux 2.6.8.

Thanks to this study highlighting the defects of API changes in the whole kernel, the authors contribute *semantic patches* (classical patches with augmented information to describe the context, and being able to go further in the manipulation of the code), which have to comply with three requirements:

- Identify the code to modify
- Description of the new code
- Existing context impact

A more detailed description of the collateral evolution problem is also available in [18, 20]. This research report also contributes an interesting comparative study of the evolution of APIs in the Linux kernel, from releases 2.2 to 2.6, showing the huge increase of APIs available inside the kernel. Focusing on the Semantic Patches notion, the authors presents a detailed overview of their contribution in [19]. They give a motivating example with the changes that were done to the SCSI stack. They also justify why they chose to base their Semantic Patches approach on classical patches: it is human readable, which is important for the acceptation of the tool, yet it allows to be much more general.

In [15], the authors contribute design information on the heart of the Coccinelle tool: basically, they are using model-checking under the hood. Source code is parsed into a model, and semantic patches are translated to a CTL formula which can be used in a model-checking algorithm against the previous model. However, Coccinelle is able to **change** the source code after successful matching of CTL formula. They contribute experimental results of runs of their tool, showing fast processing. Much of the modifications have been automatically applied successfully, and for the remaining ones, changes in the Semantic Patches were needed but proved to be successful too. In [16, 21] the authors present a more kernel developers-oriented explanation of the Semantic Patches approach, with many collateral evolution examples addressed by Coccinelle.

In [17], the notion of "mega collateral evolution" is introduced, as a very huge evolution, touching everywhere in the Linux sources, and they propose an analysis of 23 out of 35 identified: number of files changed, impact (lines of code), number of maintainers involved, duration of the changes, misses and errors. And they compare the same collateral evolution being processes by hand-written Semantic Patches applied by Coccinelle, using criterion such as Sematic Patch size (lines of code), average execution time, misses and percent ok. The same analysis is introduced for "error-prone evolutions," i.e. evolutions that resulted in errors (mistakes, conflicts). In both case, the semantic patches approach is much more reliable, even if not perfect.

The major contribution at this point is real-life proof of the effectiveness of Coccinelle. With [28], the authors make another use of their code matcher and transformer, targeting at finding bugs using semantic patches. The contribution not only finds bugs, but also, with examples, shows how to automatically workaround them when it is possible. In [13] and in a more detailed version of this work available in [12], three case studies are presented: consistency of errors checking (with functions returning `NULL` or `ERR_PTR`), detection of allocation and deallocation protocols, and bad interaction with freeing memory. The authors also contribute a "protocol finding", which allows to automatically discover the correct usage of an API.

Another use of the bug finding aspect, but not limited to the Linux kernel source code is given in [25] where several open source software projects are analyzed with Coccinelle. More recent contributions in [1] aim at helping the creation of semantic patches: the idea is to let the developer fix one driver as an example, and then infer a semantic patch that will be applicable on all the other drivers. The main contribution at this level is the `SPFIND` algorithm, that is the heart of the tool.

Another use of the Coccinelle tool and its companion is described in [22, 23], where Herodotos is introduced. This contribution aims at following bugs found in software over several versions, and the authors are able to show life cycle of classes of bugs. An interesting example is the *notand* class: misuse of boolean and bit operators. As of post 2008 kernel, a large number of these defects were dropped: this has been a target of the authors since Linux version 2.6.24, showing that their patches have improved the situation.

5.4 Undertaker

With Undertaker, the authors aim at another kind of verification around the Linux kernel: configurability. In [27], the authors introduce the LIFE (*LInux Feature Explorer*), and provide a much more detailed description of their work in [29]. In the first paper, they contribute a first important element: variability model and variability implementation. The model consists of the *Kconfig* part of the Linux kernel, i.e. where configurations options are defined and can be enabled or disabled. The implementation concerns the usage of those options inside the sourcecode. Contributions of the LIFE are mainly on three parts: **source2rsf**, a tool used to extract compilation information; **kconfigextractor**, a tool to generate a boolean formula-based model of the configuration options defined by Kconfig; and **undertaker**, a tool which analyzes the `RSF` files previously extracted. Once everything is setup, satisfiability between variability model and variability implementation is checked, and thus consistency between both. Later, in [29], the authors give more details about logic behind all steps of the analysis. Also, they explain why the current `grep`-based tool that checks for configurability defects is not enough: it is obvious that the `checkkconfigsymbols.sh` script is not being used by kernel maintainers; running this tool exposes issues which have been not fixed for several months.

The given example deals with CPU Hotplug capabilities: a typo between Kconfig and usage in source code lead to hotplug missing, for more than six months. Speculative reasons why maintainers do not use the script are: too many false positives, huge processing time, false negatives. To cope with the huge number of configurable functionalities in the Linux kernel (8000 in the first paper, 10000 in the second), the authors contribute a model-slicing algorithm for Kconfig, otherwise the boolean formulas become too complex and cannot be treated. Detected defects can either be dead code or undead code. Another contribution in this paper is the study of introduced/fixed defects among several kernel releases: they studied versions between 2.6.30 (rc1 and stable) and 2.6.36 (rc1 and stable). A huge peak of "fixed defects" can be seen on the 2.6.36-rc1 release, which the authors explain by the merge of the patches they sent to maintainers to fix the issues they found.

5.5 Integrated Static Analysis for Linux Device Driver

In this paper [24], the authors aim at porting the verifications techniques developed by Microsoft for SDV (*Static Driver Verifier*) [2] to the Linux kernel project. They come up with several contributions. First, they propose an extension of the SLIC language [7] from Microsoft: SLICx. This is both needed because of the lack of available SLIC tool and to fit their needs better:

- Reducing the set of possible types for state fields
- No more parallel assignments
- Allowing any C statement and expressions

For the same reason, lacking source code, they use the CBMC model-checker. A second contribution is a model for Linux, implementing life cycle for a device driver, i.e. `module_init()` and `module_exit()`. As another contribution, they give an example of checking with the RCU API. In their tool, the authors are able to get further than previous tools such as `CBMC` or `SDV`: absence of memory leaks, simulating preemption, deadlocks, race conditions. Testing occurs by annotating drivers. They also propose some experimental results over their solution. A first result is that they argue about *modular analysis* being faulty when dealing with functions calls specifications. It is the classical issue of modeling the whole environment in model-checking. The verification process given is not automated, many steps being manual.

6 Verification of a kernel: seL4

Work has been done to verify a (almost) full kernel: seL4[11]. It is a secure version of the L4 micro-kernel. They present pretty robust results with a formal proof of a very large part of the micro-kernel, only the boot code is not yet proven. Regarding our goal, this must be looked scarcely: the codebase is "just" 8700 lines of C and 600 lines of assembly language, and both are verified. However, the authors states that performance impact is null or low enough to be discarded. The proof is machine-assisted and machine-checked, but it requires human intervention: implementation is strictly proven against the specifications.

One contribution is a rapid kernel design and implementation, that helps designers balance between formal methods and high performance, through a Haskell prototyping phase that helps to prove the final C implementation. Proof is contributed thanks to the Isabelle/HOL theorem prover, which requires human interaction. The specification part is important: abstract specifications (describing functionalities of the system), and executable specifications (describing how the system works from a low-level point of view). The C implementation is included in the verification toolchain, meaning the authors had to describe C semantics in a model: in fact, they modeled a subset of the C language (C99: types, memory model, structure padding, unsafe pointer handling, pointer arithmetic); limitations towards the full C99 language are: no use of & operator, avoiding as much as possible references (to comply with absence of ordering in expressions evaluation), functions pointers are not allowed, nor `goto` or `switch`.

The authors consider, in their goal to prove security properties, that both the compiler (GCC) and the hardware (ARMv6-based) can be trusted; however GCC is not trusted for predictable bit field compilation and optimization. The authors also contribute a model for the machine, which is needed because assembly code has to be handled and proved. Codebase size and proof size are interesting figures: in Haskell/C, the number of lines of code (LOC) is 8700, once translated in Isabelle's language, it gives 15,000 LOC. And then the proof is 5500 LOC long. The authors also contributes a project-management view of their work, and one interesting point is that their approach is not that bad: in fact, they claim it is better than EAL6 Certification and that it provides stronger guarantee.

7 Conclusion

In this paper we presented an overview of the state of the art in the field of checking the Linux kernel and beyond. We presented work that has been initiated by Engler et al. on verifying Linux, OpenBSD and the FLASH embedded system, and a comparative study of static analysis versus model checking. We presented the work that has been done and published by Microsoft around the automated verification of the usage of APIs by device drivers, which led to a powerful static analyzer tool being available in the latest Driver Development Kit. We presented work done on the Linux kernel that shares some aspects with the Microsoft approach, and also several projects (Coccinelle, Undertaker) that are quite mature. We also presented a fully proven micro-kernel analysis showing some methodology for complete correctness, but which would not scale to the size of the Linux kernel (which is on the order of 500 times bigger). The "verification" topic toward Linux kernel is not a new field, but we can observe that all the approaches in the literature are static analysis. Critical analysis made by Engler et al. [10] on Static Analysis versus Model-Checking gives some enlightenment on the reasons.

However, an interesting way to handle the model-checking approach would be to slice the Linux kernel source code into several sub-parts that can be checked independently: thus, the state explosion issue related to model-checking could be avoided. Autoslicing as described in [29] could be a good starting point. A first step might be accomplished with the Atomic API available in the kernel: it is concise, critical for some parts of the kernel, it mixes C and assembly language, and it has strict usage rules. Plus, it lives inside the `arch` directory, which according to [22] is now the more error-prone directory, before the `drivers`. Generally speaking, this last paper shows that the effort should be emphasized on `arch`. Following the Engler's Meta-Level Compilation idea expressed in [9], it would also be interesting to work inside GCC thanks to the recent availability of plugins: the MELT plugin[26] allows easier manipulation of the GCC internals especially for pattern matching operations.

References

[1] Jesper Andersen and Julia L. Lawall. Generic patch inference. In *23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 337–346, L'Aquila, Italy, 2008.

[2] T. Ball, E. Bounimova, V. Levin, R. Kumar, and J. Lichtenberg. The static driver verifier research platform. In *International Conference on Computer Aided Verification*, 2010.

[3] T. Ball, B. Cook, V. Levin, and S.K. Rajamani. Slam and static driver verifier: Technology transfer of formal methods inside microsoft. In *Integrated Formal Methods*, pages 1–20. Springer, 2004.

[4] T. Ball, V. Levin, and S.K. Rajamani. A decade of software model checking with slam. 2010.

[5] Thomas Ball and Sriram K. Rajamani. Boolean programs: A model and process for software analysis. Technical report, Microsoft Research, February 2000.

[6] Thomas Ball and Sriram K. Rajamani. Checking temporal properties of software with boolean programs. In *In Proceedings of the Workshop on Advances in Verification*, 2000.

[7] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN '01: Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 103–122, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[8] Robert DeLine and Manuel FÃd'hndrich. The fugue protocol checker: Is your software baroque? Technical report, 2003.

[9] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. pages 1–16, 2000.

[10] Dawson Engler and Madanlal Musuvathi. Static analysis versus software model checking for bug finding. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 405–427. Springer Berlin / Heidelberg, 2004.

[11] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[12] Julia L. Lawall, Julien Brunel, René Rydhof Hansen, Henrik Stuart, and Gilles Muller. WYSIWIB: A declarative approach to finding protocols and bugs in Linux code. Technical Report 08/1/INFO, Ecole des Mines de Nantes, Nantes, France, 2008.

[13] Julia L. Lawall, Julien Brunel, Nicolas Palix, René Rydhof Hansen, Henrik Stuart, and Gilles Muller. WYSIWIB: A declarative approach to finding protocols and bugs in Linux code. In *The 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 43–52, Estoril, Portugal, 2009.

[14] Julia L. Lawall, Gilles Muller, and Richard Urunuela. Tarantula: Killing driver bugs before they hatch. In *The 4th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, pages 13–18, Chicago, IL, 2005.

[15] Yoann Padioleau, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. Semantic patches for documenting and automating collateral evolutions in linux device drivers. In *PLOS 2006: Linguistic Support for Modern Operating Systems*, San Jose, CA, 2006.

[16] Yoann Padioleau, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. Towards documenting and automating collateral evolutions in linux device drivers. Research Report 6090, INRIA, 01 2007.

[17] Yoann Padioleau, Julia L. Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. In *EuroSys 2008*, pages 247–260, Glasgow, Scotland, 2008.

[18] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. Understanding collateral evolution in linux device drivers (long version). Research report 5769, INRIA, Rennes, France, 2005.

[19] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. SmPL: A domain-specific language for specifying collateral evolutions in Linux device drivers. In *International ERCIM Workshop on Software Evolution (2006)*, Lille, France, 2006.

[20] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. Understanding collateral evolution in Linux device drivers. In *The first ACM SIGOPS EuroSys conference (EuroSys 2006)*, pages 59–71, Leuven, Belgium, 2006.

[21] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. Semantic patches, documenting and automating collateral evolutions in Linux device drivers. In *Ottawa Linux Symposium (OLS 2007)*, Ottawa, Canada, 2007.

[22] Nicolas Palix, Julia Lawall, and Gilles Muller. Herodotos: A Tool to Expose Bugs' Lives. Research Report RR-6984, INRIA, 2009.

[23] Nicolas Palix, Julia Lawall, and Gilles Muller. Tracking code patterns over multiple software versions with herodotos. In *AOSD '10: Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, pages 169–180, New York, NY, USA, 2010. ACM.

[24] H. Post and W. Küchlin. Integrated static analysis for linux device driver verification. In *Integrated Formal Methods*, pages 518–537. Springer, 2007.

[25] Sune Rievers. Finding bugs in open source software using coccinelle, jan 2010.

[26] Basile Sarynkévitch. Extending the gcc compiler with melt to suit your needs. In *RMLL 2010*, 2010.

[27] Julio Sincero, Reinhard Tartler, Christoph Egger, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Facing the Linux 8000 Feature Nightmare. In ACM SIGOPS, editor, *Proceedings of ACM European Conference on Computer Systems (EuroSys 2010), Best Posters and Demos Session*, 2010.

[28] Henrik Stuart, René Rydhof Hansen, Julia L. Lawall, Jesper Andersen, Yoann Padioleau, and Gilles Muller. Towards easing the diagnosis of bugs in OS code. In *4th Workshop on Programming Languages and Operating Systems*, pages 1–5, Stevenson, Washington, 2007.

[29] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature Consistency in Compile-Time Configurable System Software. In European Chapter of ACM SIGOPS, editor, *Proceedings of the EuroSys 2011 Conference (EuroSys '11)*, 2011.

[30] T. Witkowski, N. Blanc, D. Kroening, and G. Weissenbacher. Model checking concurrent linux device drivers. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 501–504. ACM, 2007.

[31] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Trans. Program. Lang. Syst.*, 29(3):16, 2007.

[32] Hao Yuan and Patrick Eugster. An efficient algorithm for solving the dyck-cfl reachability problem on trees. In *ESOP '09: Proceedings of the 18th European Symposium on Programming Languages and Systems*, pages 175–189, Berlin, Heidelberg, 2009. Springer-Verlag.

Faults in Patched Kernel

Alexandre Lissy
Mandriva
`alissy@mandriva.com`
Laboratoire d'Informatique de l'Université de Tours
`alexandre.lissy@etu.univ-tours.fr`

Stéphane Laurière
Mandriva
`slauriere@mandriva.com`

Patrick Martineau
Laboratoire d'Informatique de l'Université de Tours
`patrick.martineau@univ-tours.fr`

Abstract

Tools have been designed to detect for faults in the Linux Kernel, such as Coccinelle, Sparse, or Undertaker, and studies of their results over the vanilla tree have been published. We are interested in a specific point: since Linux distributions patch the kernel (as other software) and since those patches might target less common use cases, it may result in a lower quality assurance level and fewer bugs found. So, we ask ourselves: is there any difference between upstream and distributions' kernel from a faults point of view ? We present an existing tool, Undertaker, and detail a methodology for reliably counting bugs in patched and non-patched kernel source code, applied to vanilla and distributions' kernels (Debian, Mandriva, openSUSE). We show that the difference is negligible but in favor of patched kernels.

1 Introduction

A survey of the number of "faults" in the Linux kernel has been conducted in 2001 by [1]. In this paper, authors showed the relatively large number of bugs in drivers of the kernel: the `drivers` subdirectory of the kernel accounts for 7 times the number of bugs found in source files compared to the other directories. And this is not a matter of code length, as the figure is given by the following ratio:

$$r = \frac{err\_rate_{drivers}}{err\_rate_{non-drivers}}$$

Ten years later, an updated version of the survey on more recent kernels was contributed in [4]. One of the major outcomes of this second paper is that it shows drivers are less prone to errors: the first paper pointed out methods of improvement in the kernel, and a large effort has been conducted in this area to make things better indeed. Now, the `arch` subdirectory is the one where faults are most concentrated! No reason other than "people and institutions focus on drivers quality" is given to explain the improvement. This result also emphasizes the need to periodically check and follow the status of bugs: basically, actions need a continuous assessment of their usefulness. So the community (be it either researchers, companies, etc.) needs feedback to know where the effort is needed. Some effort in this area is already proposed by Phoronix.com[1]: the website regularly checks the status of the current versions of the kernel using its Phoronix Test Suite, checking mainly for regressions (performance, power consumption, etc.). It should also be noted that the authors of [4] have a tool, Heorodotos [3], which they use for tracking detected bugs life cycle: this is already a first step towards the tracking of "where we need to work". Our motivations are further detailed in section 2, then the tool used in section 3. The methodology is presented and explained in section 4 and we present then discuss results in section 5.

2 Motivations

As we just exposed, there have been several studies of the fault density in the kernel, with several tools. Chou et al. [1] used `xgcc`, itself introduced by [2], which is able to find 12 types of errors in the source code of Linux (and OpenBSD). The output has been inspected

[1] `http://www.phoronix.com`

by hand to check for false positives, etc., and the kind of errors checked for are: deadlock of spinlock, NULL pointer dereference, large stack variable, NULL pointer assumptions, array boundary checking, lock release and no double-lock, use of already-free'd memory, no floats, memory leak, user pointer dereference and correct size allocation.

As stated in [4], the Linux kernel has evolved a lot since this first study, and thus the authors propose to update the results: one first work was to reuse the same checkers, by re-implementing them after explaining how they understood what the original authors meant. To find the faults corresponding to those checks, they used a tool of their own, Coccinelle. They also used Herodotos to track the faults among the versions of the kernel. One common point of those two studies is that they only follow the upstream Linux kernel, they have no interest in *forks* or derivatives. This is the case of kernel used by distributions: they backport features, they fix bugs, etc., and in our opinion, it is interesting to study whether, from a faults point of view, there is a difference between distribution's kernels and the upstream one.

Another motivation is to study another type of faults, those measured by Undertaker: a tool that checks for configurations faults. Of course, in their papers [6, 5], the authors show results but once again, only on upstream kernel. Following the same idea as previous authors, we would like to have a clear point of view of the status of our distribution's kernel and where it needs focus for improvements: there is no obvious reasons that it will be the same than upstream.

3 Undertaker: Finding Configurations Defects

Software configurability in the Linux kernel is a giant and growing space: currently, more than 10000 *features* are available. A vast majority of software allows for compile-time configuration, which in the case of Linux is handled as preprocessor macros. Many of the options have dependencies between themselves, and tools (Kconfig) allow the user to manipulate the *variability model*. These macros are then used in the C code to enable/disable some features. In [6, 5] the authors present Undertaker, a tool that is able to analyze and check for consistency between configurations models and their use in the source code; not only because it can leave dead code blocks, but also because of incorrect selection of code. Imagine a code path that *must not be followed* in case of `CONFIG_NOT_FOLLOW` is set, but source code uses `#ifdef CONFIG_FOLLOW_NOT`, this will lead to inconsistencies. The authors link a *real-life* case with CPU hotplug: this leads to the feature, hotplug of CPU, which has been broken for more than six months before being fixed. Undertaker makes uses of LIFE (*LInux Feature Explorer*) to extract configurations options as a model from *Kconfig* files, making re-use of *Sparse* (static analysis tool for Linux). Both configuration and implementation models are then transformed to Boolean formulas which can be checked thanks to SAT solvers (Undertaker use *PicoSAT*). Thanks to this, they identified, reported and fixed several issues: 14 were accepted and waiting to be applied to 2.6.34 and a total of 90 have been identified. Please note that we have not been able to work with kernel prior to 2.6.30 due to a bug in Undertaker.

4 Methodology

As stated before, our goal is to be able to check whether quality hypothesis that are considered for upstream kernel versions are also applicable to kernel source code used to create a distribution's package: it can differ in a non-insignificant way. For example, Mandriva packages the Kerrighed kernel (Kerrighed is a SSI system on top of Linux), so basically, it is a giant patch over the upstream kernel. Several distributions also package XEN (which allows efficient virtualisation and paravirtualisation) which is also itself a big patch over the kernel. To measure the faults, we decided to use Undertaker for several reasons:

- It is easy to use, so we can have results rapidly
- It aims at an interesting and newer problem compared with other tools

However, as explained in the section 3, Undertaker only treats the problem of configurability. Although the kernel has a lot of configurable features, configurability is not the only ground for errors, and so we would like to continue and extend this study by integrating other tools to have a better view of the error status for the whole kernel. Using Undertaker is a preliminary step towards wider experiments. Several kernel sources from different distributions (Mandriva, openSUSE and Debian) were used to run the tool over as similar as possible versions. When it analyzes the kernel, Undertaker identifies several kinds of issues:

- Dead code: no configuration item can enable this code
- Undead code: no configuration item can disable this code. Of course, only code between `#if...#endif` is considered

The resulting issues can have different scopes: code is either dead or undead globally, i.e. on all architectures, or it can be dead or undead on several architectures but good on at least one.

In our case, the dead or undead differentiation is not very interesting, as it is a fault in both cases. However, the architecture scope becomes interesting: distributions often target several architectures so we limit ourselves to globally dead or undead code. As in previous papers [1, 4, 6, 5] we consider the number of faults per subdirectories, e.g., `kernel/`, `drivers/`, `mm/`, etc. However, since we are interested in non-upstream kernel, there is one more thing to consider: when analyzing a patched kernel, how can we take into account the differences brought by patches? We do not wish to analyze each kernel patch by hand. The number of lines of ANSI C code present in each subdirectory analyzed by Undertaker is counted, thanks to the *sloccount* tool, and we consider the number of faults in a directory divided by the number of ANSI C lines of code:

$$faultRate_{dir} = \frac{Faultsindir}{ANSI\ C\ LOCindir}$$

This provides a fault rate per single line of code, which gives an idea of the "quality" status (limiting ourselves to what Undertaker is able to diagnose) in the kernel, independent of the size of the code base. This allow us to compare the impact quality of patches.

When two kernel versions are compared, for each directory, it is simply the difference between the first kernel's values and the second one. So, for a chart comparing 2.6.33.7 and 2.6.33.7-mdv1 (Figure 4), the computed difference is as follows:

$$diff = faultrate_{2.6.33.7} - faultrate_{2.6.33.7-mdv1}$$

Hence a **positive** difference means an **improvement** in Mandriva's kernel, whereas a **negative** difference suggests **new bugs introduced**. Also, it should be noted that the naming of kernel versions follows some distinct conventions for openSUSE and Mandriva: in the first case, it uses the pattern `rpm-kver` where `kver` is the version of the kernel; for Mandriva the pattern is `vkver-mdvX` where `kver` is the version of the kernel and `X` is the release number of the kernel package.

5 Results and discussion

Some preliminary results are now presented and discussed regarding our objectives. In section 5.1 a first run of some "verification against upstream and literature" is done, to check that the methodology gives comparable results to what has already been done, especially in [4, 6, 5]. Then some specific cases are studied: Debian kernel in section 5.2, Mandriva kernel in section 5.3 and openSUSE 11.2 kernel in section 5.4.

5.1 Verification with Upstream

First, the goal is to verify that the approach gives comparable results with [4], not because exact figures are necessary (since measurements are not done over exactly the same things and in the same way, it is meaningless compare directly), but to check that it "reflects" the same tendencies. Our results are present in Figure 1 and target kernel from 2.6.32 to 2.6.38. Similar data is visible in [4, p. 13, Figure 9] where "fault rate per directory" is plotted for kernel 2.6.5 to 2.6.33. It is possible to estimate the fault rate of the `drivers` directory at 0.3, and the fault rate of `arch` at 0.4, compared with [4, p. 13, Figure 9]. In our results, values are respectively of 0.57 and 0.76. This leads to the following ratios:

$$ratio_{coccinelle} = \frac{0.3}{0.4} = 0.75$$

and

$$ratio_{undertaker} = \frac{0.57}{0.76} = 0.75$$

Even if the values are not the same, what they indicate is similar. Also, they lead to convergent interpretations:

- The overall fault rate is decreasing
- `arch` directory shows a higher fault rate than `drivers`

5.2 Debian's kernel

Starting with the Debian kernel, we focus on two 2.6.37 releases provided within the distribution: 2.6.37-1 and

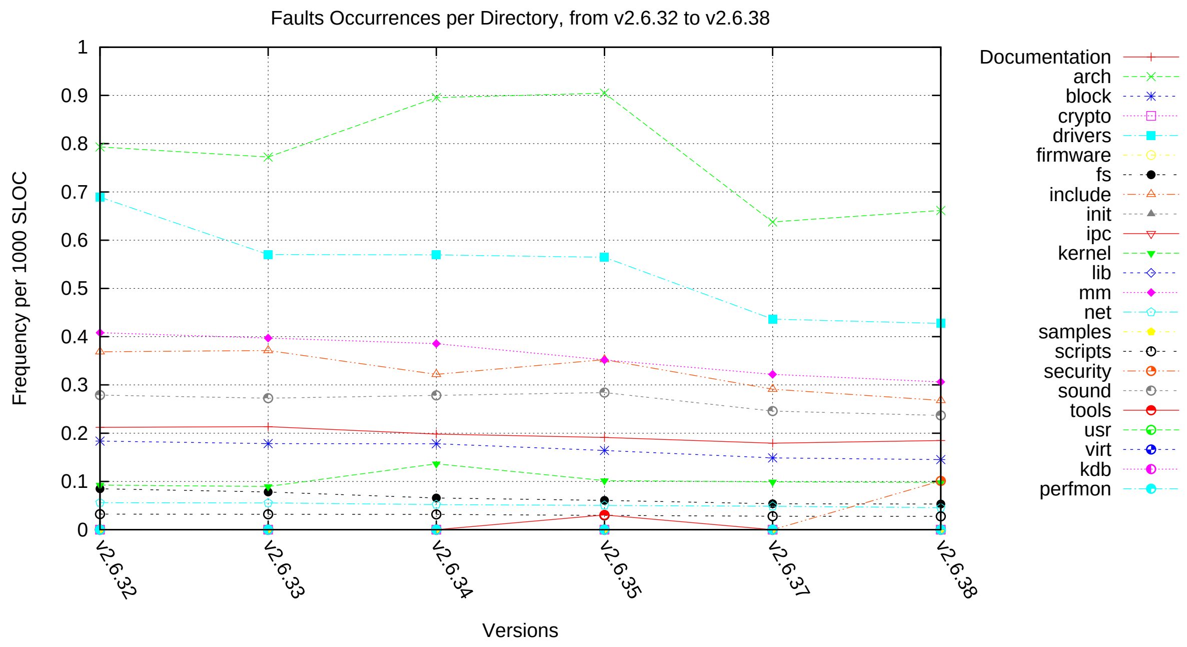


Figure 1: Evolution of the fault rate in the Vanilla kernel, from 2.6.32 to 2.6.38

2.6.37-2. A first look at the chart given in Figure 2 shows that both releases do not present the same fault rate: 2.6.37-2 has a higher fault rate in `kernel` directory than the 2.6.37-1 release. Taking a look at the difference between 2.6.37-1 and 2.6.37 from upstream, we can see in Figure 3 that there is a general **improvement**.

5.3 Mandriva's Kernel

For the Mandriva case, we have made a direct comparison of fault rate for each directory on only one version of the kernel, 2.6.33.7, whose results are available in Figure 4. As stated before, this is computed by doing the difference of fault rate between upstream and the distribution's kernel. From this chart, we can get two interesting points: the number of faults is generally lower in the patched kernel, and the fault rate is generally lower too. But the differences in fault rate are very tight, and even if they are in favor of patched versions, it can be considered as negligible.

5.4 openSUSE's Kernel

Looking in the openSUSE release gives another insight. Figure 5 shows the status over all RPM released kernels of the openSUSE 11.2 distribution, going from 2.6.30

	2.6.33.7		2.6.33.7-mdv1	
Directory	Faults	Rate	Faults	Rate
Documentation	2	0.213493	2	0.213061
arch	1240	0.772214	1241	0.772731
block	2	0.178380	2	0.178380
drivers	2558	0.570437	2552	0.567989
fs	50	0.078163	52	0.080378
include	92	0.371393	92	0.370956
kernel	9	0.089347	9	0.089326
mm	18	0.397263	18	0.396922
net	23	0.055458	23	0.055406
scripts	1	0.031829	1	0.031809
sound	115	0.272438	115	0.271721

Table 1: Mandriva Kernel 2.6.33.7 versus Upstream 2.6.33.7

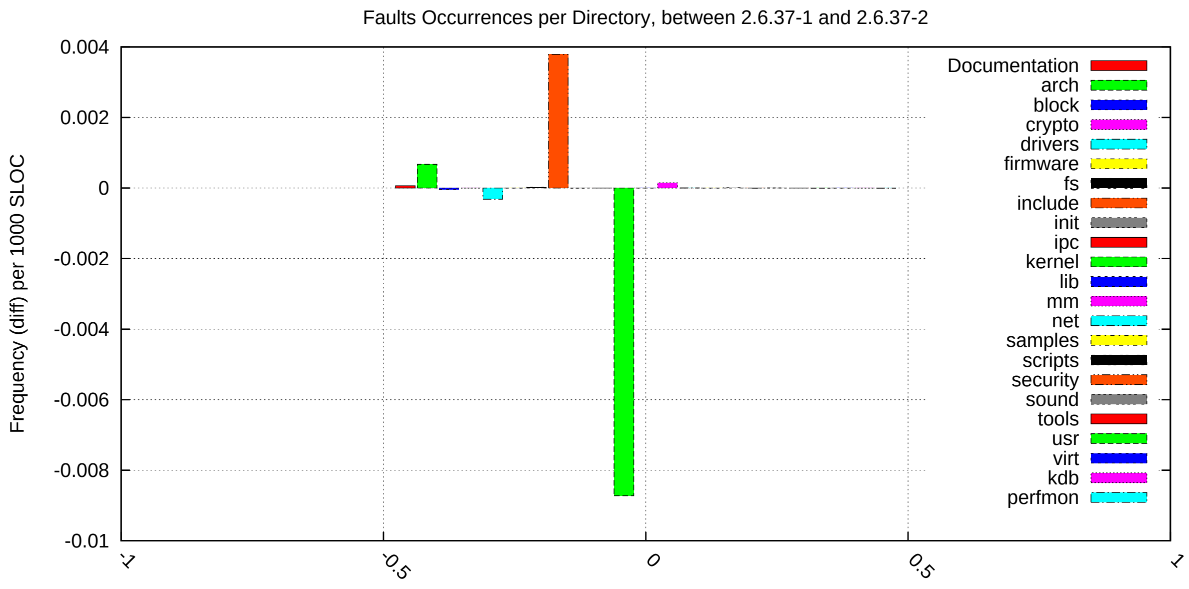


Figure 2: Fault rate differences between Debian 2.6.37-1 and 2.6.37-2

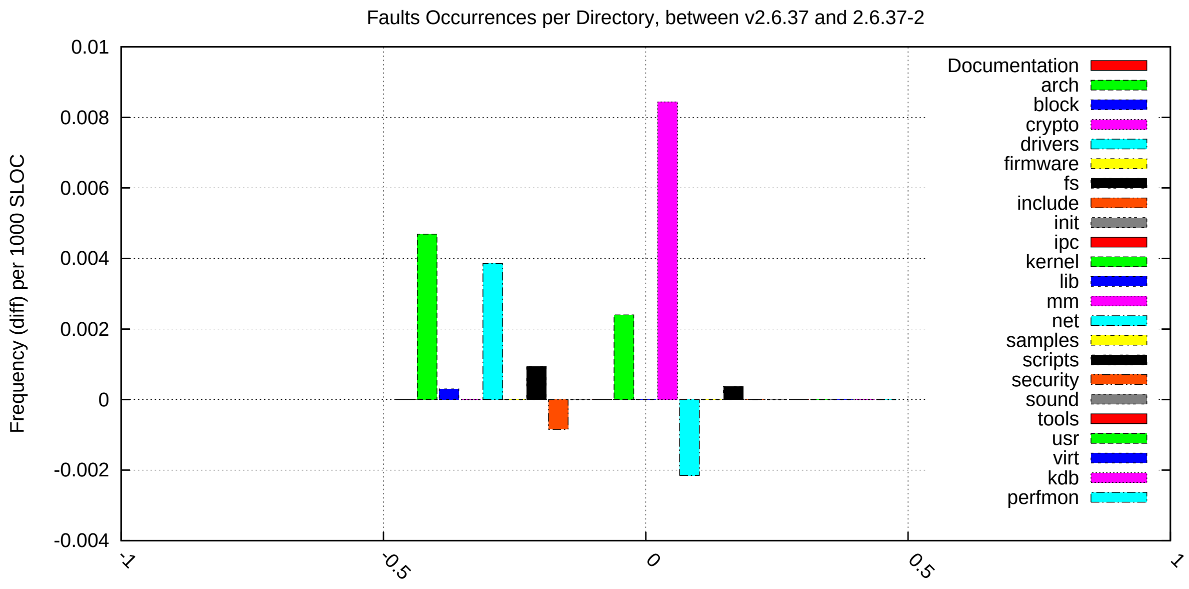


Figure 3: Vanilla 2.6.37 and Debian 2.6.37-2

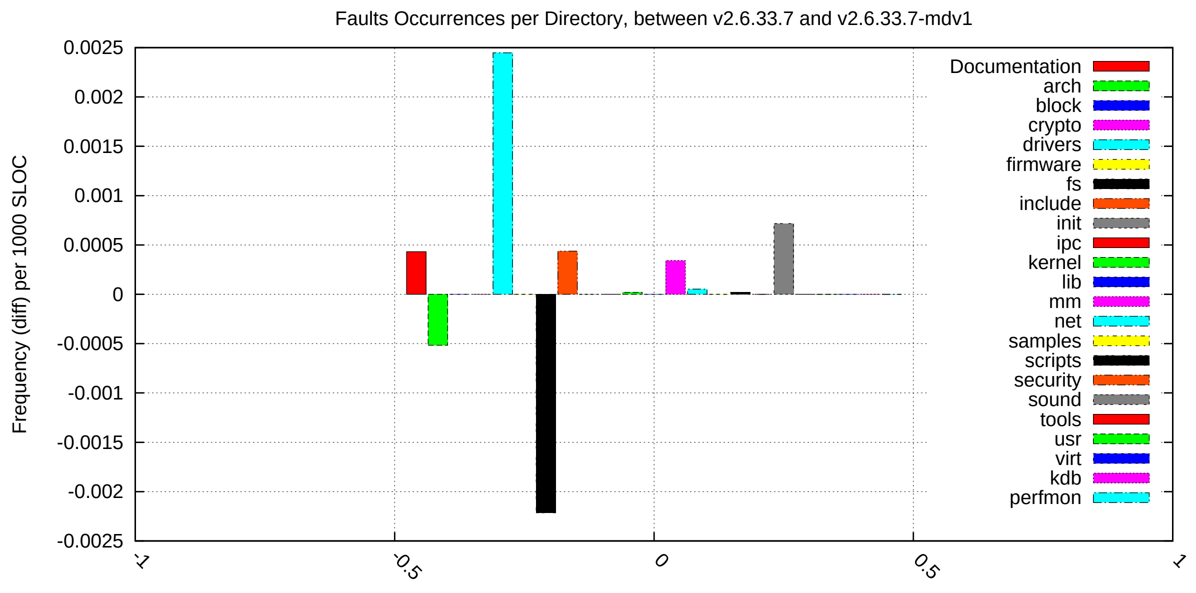


Figure 4: Mandriva versus Vanilla, 2.6.33.7

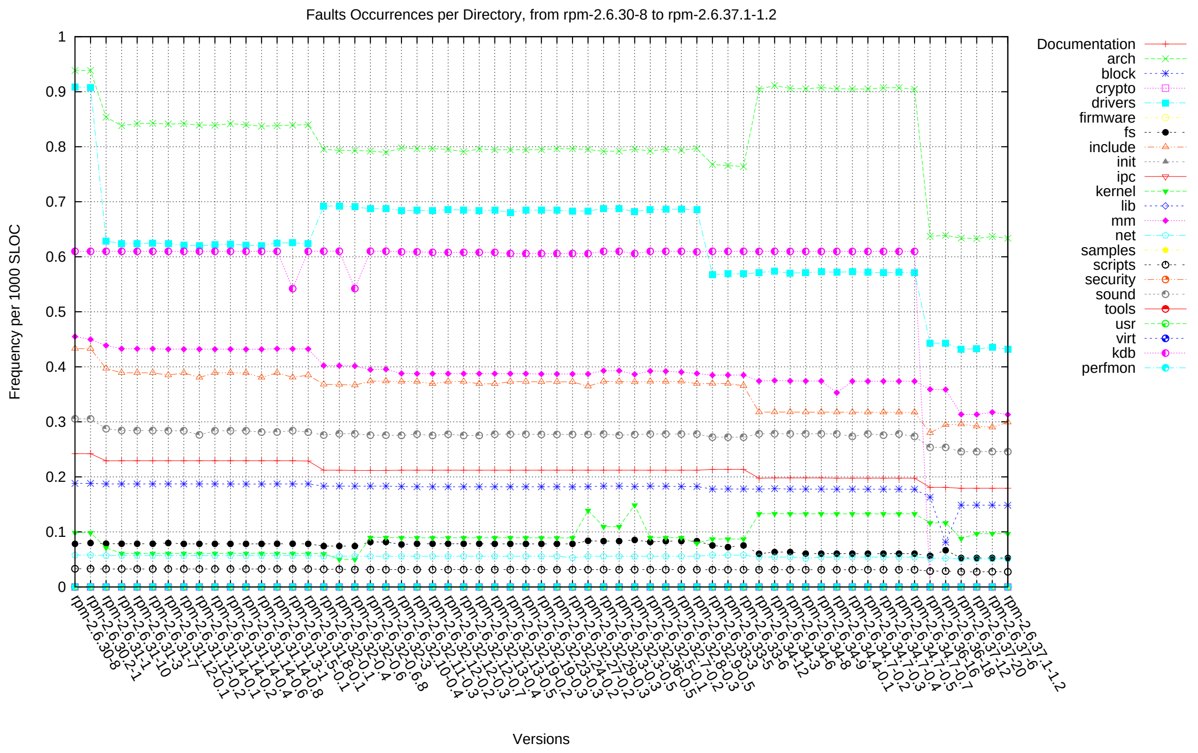


Figure 5: openSUSE Kernel, from 2.6.30 to 2.6.37

	rpm-2.6.37-20		2.6.37	
Directory	Faults	Rate	Faults	Rate
Documentation	2	0.179308	2	0.179308
arch	1145	0.633151	1133	0.637841
block	2	0.148566	2	0.148865
drivers	2261	0.432644	2256	0.436499
fs	36	0.052706	36	0.053648
include	81	0.292085	79	0.291239
kernel	11	0.096745	11	0.099146
mm	16	0.313535	16	0.321970
net	23	0.050789	22	0.048632
scripts	1	0.027578	1	0.027952
sound	118	0.245929	118	0.245929

Table 2: openSUSE 2.6.36-20 versus Vanilla 2.6.37

to 2.6.37 releases: thus, we can compare it roughly to Figure 1 targeting upstream releases between 2.6.32 and 2.6.38. It can be seen that the evolution is quite similar once again. Figure 2 show that the number of added fault is similar to what has been observed in previous cases, and that the fault rate is also lower.

5.5 Overall analysis

A general tendency can be drawn from these results: distributions' kernels, even if patched, shows a slightly lower fault rate. It is interesting, not because it shows that the packaged kernels are somewhat "better" than upstream, but because it shows that they are not worse and that patches do not have a real impact, regarding the configurability faults measured by Undertaker. It should be noted that the difference in fault rate is not important, so there is an impact of patches in those distributions, but it is negligible if considering its importance. As shown in [4], the faults in Linux are decreasing over the time thanks to the efforts being put on producing tools to identify as much as possible of these faults, allowing to fix them.

6 Conclusion

Our initial objective was to study whether, regarding the Undertaker analyzing tool and its target, there were differences between a distribution-patched kernel source code and its upstream counterpart. We presented the way Undertaker works, and we also explained how we used it to measure fault rates. The methodology given also explains the notion of fault rate we used. After checking that we obtain comparable results with previous studies, confirming that Undertaker's specifics can be avoided and that the fault rate it computes gives a similar idea of the quality status of the code base as Coccinelle [4], we ran the process over some kernels from different distributions: Debian, Mandriva and openSUSE. We have been able to show that, whatever distribution we consider, there is a constant pattern: the computed fault rate is often lower on distribution-patched kernels, while the number of faults is often higher. However, the figures also give an important detail: if we look at the deltas, the differences are always very low; for example, in the `arch` directory of openSUSE's 11.2 kernel release 2.6.37-20, we have 1145 faults were identified, while 1133 were in the corresponding upstream version, as illustrated by Table 2. Figures presented in Table 1 are roughly the same for Mandriva's 2.6.33.7 versus Vanilla 2.6.33.7. And regarding the number of faults, the deltas are negligible.

Further work should target a more precise analysis of the faults: it would be interesting to check whether the faults identified and which are more present in distributions' kernels are evenly distributed or whether they concentrate around hot spots. Another improvement would be to integrate other detailed measures such as what is done from Coccinelle in [4]: that would allow us to perform the same checks, i.e. whether there are hot spots in patched-code, not limiting ourselves to the study of configurability.

References

[1] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating system errors. pages 73–88, 2001.

[2] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. pages 1–16, 2000.

[3] Nicolas Palix, Julia Lawall, and Gilles Muller. Tracking code patterns over multiple software versions with herodotos. In *AOSD '10: Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, pages 169–180, New York, NY, USA, 2010. ACM.

[4] Nicolas Palix, Suman Saha, Gaël Thomas, Christophe Calvès, Julia Lawall, and Gilles Muller.

Faults in Linux: Ten Years Later. Research Report RR-7357, INRIA, 08 2010.

[5] Julio Sincero, Reinhard Tartler, Christoph Egger, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Facing the Linux 8000 Feature Nightmare. In ACM SIGOPS, editor, *Proceedings of ACM European Conference on Computer Systems (EuroSys 2010), Best Posters and Demos Session*, 2010.

[6] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature Consistency in Compile-Time Configurable System Software. In European Chapter of ACM SIGOPS, editor, *Proceedings of the EuroSys 2011 Conference (EuroSys '11)*, 2011.

Towards Co-existing of Linux and Real-Time OSes

Hitoshi Mitake, Tsung-Han Lin, Hiromasa Shimada, Yuki Kinebuchi, Ning Li, Tatsuo Nakajima
Department of Computer Science and Engineering, Waseda University
{mitake, johnny, yukikine, lining, tatsuo}@dcl.info.waseda.ac.jp

Abstract

The capability of real-time resource management in the Linux kernel is dramatically improving due to the effective contribution of the real-time Linux community. However, to develop commercial products cost-effectively, it must be possible to re-use existing real-time applications from other real-time OSes whose OS API differs significantly from the POSIX interface. A virtual machine monitor that executes multiple operating systems simultaneously is a promising solution, but existing virtual machine monitors such as Xen and KVM are hard to used for embedded systems due to their complexities and throughput oriented designs. In this paper, we introduce a lightweight processor abstraction layer named SPUMONE. SPUMONE provides virtual CPUs (vCPUs) for respective guest OSes, and schedules them according to their priorities. In a typical case, SPUMONE schedules Linux with a low priority and an RTOS with a high priority. The important features of SPUMONE are the exploitation of an interrupt prioritizing mechanism and a vCPU migration mechanism that improves real-time capabilities in order to make the virtualization layer more suitable for embedded systems. We also discuss why the traditional virtual machine monitor design is not appropriate for embedded systems, and how the features of SPUMONE allow us to design modern complex embedded systems with less efforts.

1 Introduction

Modern real-time embedded systems like smart phones become highly functional along with the enhancements of CPUs targeting their market. But their functional features introduced significant engineering cost. The main difficulty in the development of such devices comes from the conflicting requirement of them: low latency and high throughput must be established in one system. This requirement is hard to satisfy with existing OSes, because all of them are categorized as either *Real-Time Operating System (RTOS)* or *General Purpose Operating System (GPOS)*. RTOSes, like eCos [2] or TOPPERS[1] [3], are designed and developed for executing real-time tasks such as processing wireless communication protocols. In the typical case, such a task runs periodically for short time. The feature of executing such deadline-sensitive tasks imposes limitation on RTOSes. For example, most RTOSes cannot change the number of tasks dynamically. On the other hand, GPOSes such as Linux, are designed and developed for executing tasks which consist of significant amount of computation. Of course some of them in desktop computers are latency sensitive, to offer a comfortable user experience, but missing a deadline is not fatal for them. The contribution from the real-time Linux community has significantly improved the real-time resource management capability of Linux [7]. However, there is always a tradeoff between satisfying real-time constraints and achieving maximum throughput [8].

In order to develop a modern real-time embedded system which satisfies conflicting requirements, combining multiple OSes on virtual machine monitors can be an effective approach. A virtual machine monitor, e.g. KVM [6], Xen [4] and VMware [5], is traditionally used in the area of data center or desktop computing for executing multiple OS instances in one physical machine. Their capability of executing multiple OSes is also attractive for embedded systems because they make it possible to implement a system which has multiple OS personalities. If there is a virtualization layer which has a capability of executing GPOS and RTOS in one physical machine, developing real-time embedded systems can be simpler.

Armand and Gien [9] presented several requirements for a virtualization layer to be suitable for embedded systems:

[1]TOPPERS is an open source RTOS that offers μITRON interface, and it is used in many Japanese commercial products.

1. It should execute an existing operating system and its supported applications in a virtualized environment, such that modifications required to the operating system are minimized (ideally none), and performance overhead is as low as possible.

2. It should be straightforward to move from one version of an operating system to another one; this is especially important to keep up with frequent Linux evolutions.

3. It should reuse native device drivers from their existing execution environments with no modification.

4. It should support existing legacy often real-time operating systems and their applications while guaranteeing their deterministic real-time behavior.

Unfortunately, there is no open source virtualization layer that satisfies all the above requirements. VirtualLogix[2] VLX [9] is a virtualization layer designed for combining RTOS and GPOS, but it is proprietary software. OKL4 microvisor [12] is a microkernel based virtualization technology for embedded systems, but performs poorly as the nature of microkernels [9]. In addition, we found that there are fatal performance degradation of guest OSes when RTOS and SMP GPOS share a same physical CPU. This performance problem comes from the phenomenon called *Lock Holder Preemption*(LHP) [11]. It is a general phenomenon of virtualization layers, hence a solution for this problem was already proposed. However these existing solutions only focus on the throughput of guest OSes, therefore the virtualization layers that execute RTOSes cannot adopt these solutions. To the best of our knowledge, there is no virtualization layer that can execute RTOS and GPOS on a multicore processor without performance degradation caused by LHP, and is distributed as open source software.

Our laboratory is developing an open source virtualization layer for combining RTOS and Linux on embedded systems that adopt multicore processors, named SPUMONE (Software Processing Unit, Multiplexing ONE into two or more). During the development of this virtualization layer, we faced many difficulties specific to embedded systems. They come from the limitation of hardware resources, the requirement of engineering cost, or scheduling RTOS and SMP GPOS on the same CPU. Because of these difficulties, we believe that virtualization layers for real-time embedded systems should be developed as open source software for incorporating various insights from a wide range of community.

[2]VirtualLogix, Inc. was acquired by Red Bend Software in Sep. 2010

This paper is structured as follows: in Section 2, the detailed motivation of our project is described. Section 3 describes the basic architecture of SPUMONE. Section 4 and Section 5 are catalogs of the problems we encountered. Section 4 describes the difficulties of dealing with real-time virtualization layers which adopt multicore processors. Section 5 describes the method for isolating OSes spatially in embedded systems. Section 6 shows related work and the differences between SPUMONE and them. Finally Section 7 concludes this paper and mentions about future directions of this project.

2 Why Virtualization

This section presents three advantages of using virtualization layers in embedded systems. The first advantage is that control processing can be implemented as application software on RTOS. Embedded systems usually include control processing like mechanical motor control, wireless communication control or chemical control. Using software-based control techniques enables us to adopt a more flexible control strategy, so recent advanced embedded systems contain microprocessors instead of hardware implemented controllers for implementing flexible control strategies. On the other hand, recent embedded systems need to process various information. For example, applications which require significant computation, such as multimedia players and full featured web browsers, are crucial elements of modern smart phones. Therefore, recent embedded systems have to contain both control and information processing functionalities. In traditional embedded systems, dedicated processors are assigned for respective processing. A general purpose processor with sufficient computational capability offers a possibility to combine these multiple processing on a single processor. A virtualization layer can host RTOS and GPOS on one system, therefore this approach requires less hardware controllers and reduces the cost of embedded systems hardware.

The second advantage is that a virtualization layer makes it possible to reuse existing software. Even if the virtualization layer is based on the para-virtualization technique (which requires the modification of guest OSes), application programs running on the guest OSes do not need to be modified. In a typical case of developing embedded systems, vendors have their own OSes and applications running on them. The virtualization layer can execute such in-house software with standard OS platforms like Symbian or Android. If the in-house software is developed against such a standard platform, it should be modified when a standard platform is replaced. Actually, the standard platform is frequently replaced according to various business reasons. On the other hand, if the in-house software is developed as application programs that run on the vendor specific OSes, porting application programs is not required even if a standard platform is replaced.

The third advantage is the isolation of source code. For example, proprietary device drivers can be mixed with GPL code without license violation. This may solve various business issues when adopting Linux in embedded systems.

3 Basic Architecture

3.1 User-Level Guest OS vs. Kernel Level Guest OS

There are several traditional approaches to execute multiple operating systems on a single processor in order to compose multiple functionalities. Microkernels execute guest OS kernels at the user level. When using microkernels, various privileged instructions, traps and interrupts in the OS kernel need to be virtualized by replacing their code. In addition, since OS kernels executed as user level tasks, application tasks need to communicate with the OS kernel via inter-process communication. Therefore, many parts of the OS need to be modified.

VMMs are another approach to execute multiple OSes. If a processor offers a hardware virtualization support, all instructions that need to be virtualized trigger traps to VMM. This makes it possible to use any OSes without any modification. But if the hardware virtualization support is incomplete, some instructions still need to be complemented by replacing some code to virtualize them.

Most of the processors used for the embedded systems only have two protection levels. So when kernels are located in the privileged level, they are hard to isolate. On the other hand, if the kernels are located in the user level, the kernels need to be modified significantly. Most of embedded system industries prefer not to modify a large amount of the source code of their OSes, so it is desirable to put them in the privileged level. Also, the virtualization of MMU introduces significant overhead if the virtualization is implemented by software. Therefore, we need reorder mechanisms to reduce the engineering cost, to ensure the reliability of the kernels and to exploit some advanced characteristics of multicore processors.

The following three issues are most serious problems, when a guest OS is implemented in the user level.

1. The user level OS implementation requires heavy modification of the kernel.
2. Emulating an interrupt disabling instruction is very expensive if the instruction cannot be replaced.
3. Emulating a device access instruction is very expensive if the instruction cannot be replaced.

In a typical RTOS, both the kernel and application code are executed in the same address space. Embedded systems have dramatically increased their functionalities in every new product. To reduce the development cost, the old version of application code should be reused and extended. The limitation of hardware resources is always the most important issue to reduce the product cost. Therefore, the application code sometimes uses very ad-hoc programming styles. For example, application code running on RTOS usually contains many privileged instructions like interrupt disable/enable instructions to minimize the hardware resources. Also, device drivers may be highly integrated into the application code. Thus, it is very hard to modify these applications to execute at the user level without changing a significant amount of application code, even if their source code is available. Therefore, it is hard to execute the application code and RTOS in the user level without violating the requirements described in Section 1. Therefore, executing RTOS is very hard if the processor does not implement the hardware virtualization support. Even if there is a proper hardware virtualization support, we expect that the performance of RTOS and its application code may be significantly degraded. Our approach chooses to execute both guest OS kernels and a virtualization layer at the same privileged level. This decision makes the modification of OS kernels minimal, and there is no performance degradation by introducing a virtualization layer. However, the following two issues are very serious in the approach.

1. Instructions which disable interrupts have serious impact on the task dispatching latency of RTOS.
2. There is no spatial protection mechanism among OS kernels.

The first issue is serious because replacing interrupt disable instructions is very hard for RTOS and its application code as described above. The second issue is also a big problem because executing guest OS kernels in virtual address spaces requires significant modification on them. SPUMONE proposes a technique presented in Section 4 and Section 5 to overcome the problems.

3.2 SPUMONE: A Multicore Processor Based Virtualization Layer for Embedded Systems

SPUMONE is a thin software layer for multiplexing a single physical CPU (pCPU) core into multiple virtual CPU (vCPU) cores [19, 20]. The current target processor of SPUMONE is the SH4a architecture, which is very similar to the MIPS architecture, and is adopted in various Japanese embedded system products. Also, standard Linux and various RTOSes support this processor. The latest version of SPUMONE runs on a single and multicore SH4a chip. Currently, SMP Linux, TOPPERS [3], and the L4 [12] are running on SPUMONE as a guest OS.

The basic abstraction of SPUMONE is vCPU as depicted in Figure 1. In this example, SPUMONE hosts two guest OSes, Linux and RTOS. Linux has two vCPUs, vCPU0 and vCPU1. vCPU0 is executed by pCPU0 and vCPU1 is executed by pCPU1. RTOS has one vCPU, vCPU2. This is executed by pCPU1. So both of vCPU1 and vCPU2 are executed on pCPU1. Unlike typical microkernels or VMMs, SPUMONE itself and guest OS kernels are executed in the privileged level as mentioned in Section 3.1. Since SPUMONE provides an interface slightly different from the one of the underlying processor, we simply modify the source

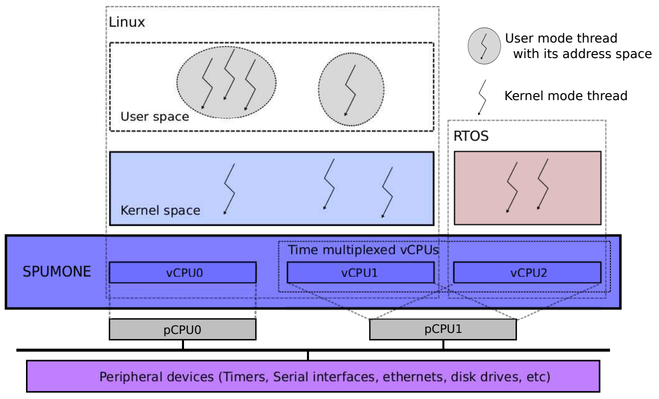

Linux
User space
Kernel space
User mode thread with its address space
Kernel mode thread
RTOS
SPUMONE
Time multiplexed vCPUs
vCPU0
vCPU1
vCPU2
pCPU0
pCPU1
Peripheral devices (Timers, Serial interfaces, ethernets, disk drives, etc)

Figure 1: An Overview of SPUMONE

code of guest OS kernels, a method known as paravirtualization. This means that some privileged instructions should be replaced to *hypervisor calls*, function calls to invoke SPUMONE API, but the number of replacements is very small. Thus, it is very easy to port a new guest OS or to upgrade the version of a guest OS on SPUMONE.

To spatially protect multiple OSes, if it is necessary, SPUMONE may assume that underlying processors support the mechanisms to protect physical memories used by respective OS like VIRTUS [24]. The approach may be suitable for enhancing the reliability of guest OSes on SPUMONE without significantly increasing overhead. In section 5, we propose an alternative novel approach to use a functionality of multicore processor to realize the spatial protection among guest OSes. The approach does not assume that the processor provides an additional hardware support to spatially isolate guest OSes.

SPUMONE does not virtualize peripheral devices because traditional approaches incur significant overhead that most of embedded systems could not tolerate. In SPUMONE, since device drivers are implemented in the kernel level, they do not need to be modified when the device is not shared by multiple OSes.

Multicore processor version of SPUMONE is designed on the distributed model similar to the Multikernel approach [18]. A dedicated instance of SPUMONE is assigned to each physical core. Therefore, data structures used in SPUMONE need not to be protected by using synchronization mechanisms. This design is chosen in order to eliminate the unpredictable overhead of synchronization among multiple physical cores. This may simplify the design of SPUMONE.

They communicate with each other via the specially allocated shared memory area and the inter-core interrupt (ICI) mechanism. First, a sender stores data on a specific memory area, then it sends an interrupt to a receiver, and the receiver copies or simply reads the data from the shared memory area.

3.2.1 Interrupt/Trap Delivery

Interrupt virtualization is a key feature of SPUMONE. Interrupts are intercepted by SPUMONE before they are delivered to each guest OS. When SPUMONE receives an interrupt, it looks up the interrupt destination table to make a decision to which OS it should be delivered. Traps are also delivered to SPUMONE first, then are directly forwarded to the currently executing guest OS.

To allow interrupts to be intercepted by SPUMONE, the interrupt entry point of the guest OSes should not be registered to hardware directly. The entry point of each guest OS must notify SPUMONE via a hypervisor call to registering their real vector table. An interrupt is first examined by the interrupt handler of SPUMONE in which the destination vCPU is determined, and the corresponding scheduler is invoked. When the interrupt triggers OS switching, all the registers including MMU state of the current OS are saved into the stack, then the registers in the stack of the previous OS are restored. Finally, the execution is switched to the entry point of the destination OS. The processor initializes the interrupt just as if the real interrupt occurred, so the source code of the OS entry points does not need to be changed.

3.2.2 vCPU Scheduling

Multiple guest OSes run by multiplexing a physical CPU. The execution states of the guest OSes are managed by data structures that we call vCPUs. When switching the execution of vCPUs, all the hardware registers are stored into the corresponding register table of vCPU, and then restored from the table of the next executing vCPU. The mechanism is similar to the process implementation of a typical OS, however the vCPU saves the entire processor state, including the privileged control registers.

The scheduling algorithm of vCPUs is the fixed priority preemptive scheduling. When RTOS and Linux share the same pCPU, the vCPU owned by RTOS would gain a higher priority than the vCPU owned by Linux in order to maintain the real-time responsiveness of RTOS. This means that Linux is executed only when the vCPU of RTOS is in an idle state and has no real-time task to be executed. The process scheduling is left up to OSes so the scheduling model for each OS need not to be changed. Idle RTOS resumes its execution when it receives an interrupt. The interrupt to RTOS should preempt Linux immediately, even if Linux has disabled the execution of its interrupt handlers. The details of this requirement and the solution for it is described in Section 4.1.1.

3.2.3 Modifying Guest OS Kernels

Each guest OS is modified to be aware of the existence of the other guest OSes, because hardware resources other than the processor are not multiplexed by SPUMONE as described below. Thus those are exclusively assigned to each OS by reconfiguring or by modifying their kernels. The following describes how the guest OS kernels are modified in order to run on the top of SPUMONE.

- Interrupt Vector Table Register Instruction: The instruction registering the address of a vector table is replaced to notify the address to the interrupt manager of SPUMONE. Typically this instruction is invoked once during the OS initialization.
- Bootstrap: In addition to the features supported by the single-core SPUMONE, the multicore version provides the virtual reset vector device, which is responsible for resetting the program counter of the vCPU that resides on a different pCPU.
- Physical Memory: A fixed size of physical memory area is assigned to each guest OS. The physical address for the OSes can be simply changed by modifying the configuration files or their source code. Virtualizing the physical memory would increase the size of the virtualization layer and the substantial performance overhead. In addition, unlike the virtualization layer for enterprise systems, embedded systems need to support a fixed number of guest OSes. For these reasons we simply assign a fixed amount of physical memory to each guest OS.

- Idle Instruction: On a real processor, the idle instruction suspends a processor until it receives an interrupt. On a virtualized environment, this is used to yield the use of real physical core to another OS. We prevent the execution of this instruction by replacing it with the hypervisor call of SPUMONE. Typically this instruction is located in a specific part of the kernel, which is fairly easy found and modified.

- Peripheral Devices: Peripheral devices are assigned by SPUMONE to each OS exclusively. This is done by modifying the configuration of each OS not to share the same peripherals. We assume that most of the devices can be assigned exclusively to each OS. This assumption is reasonable because, in embedded systems, multiple guest OSes are usually assigned different functionalities and use different physical devices. It usually consists of RTOS and GPOS, where RTOS is used for controlling special purpose peripherals such as a radio transmitter and some digital signal processors, and GPOS is used for controlling generic devices such as various human interaction devices and storage devices. However some devices cannot be assigned exclusively to each OS because both systems need to share them. For instance, the processor we used offers only one interrupt controller. Usually a guest OS needs to clear some of its registers during its initialization. In the case of running on SPUMONE, a guest OS booting after the first one should be careful not to clear or overwrite the settings of the guest OS executed first. For example, we modified the Linux initialization code to preserve the settings done by TOPPERS.

3.2.4 Dynamic Multicore Processor Management

As described in the previous section, SPUMONE enables multiplexing of virtual CPUs on physical CPUs. The mapping between pCPUs and vCPUs is dynamically changed to balance the tradeoffs among real-time constraints, performance and energy consumption. In SPUMONE, a vCPU can be migrated to another core according to the current situation. The mechanism is called the vCPU migration mechanism. In SPUMONE, all kernel images are located in the shared memory. Therefore, the vCPU migration mechanism just moves the register states to manage vCPUs, and the cost of the migration can be reduced significantly. Actually, the round trip time of the vCPU migration in the current version of SPUMONE on the RP1 platform[3] is about 50 μs when a vCPU is moved to anther pCPU and back to the original pCPU.

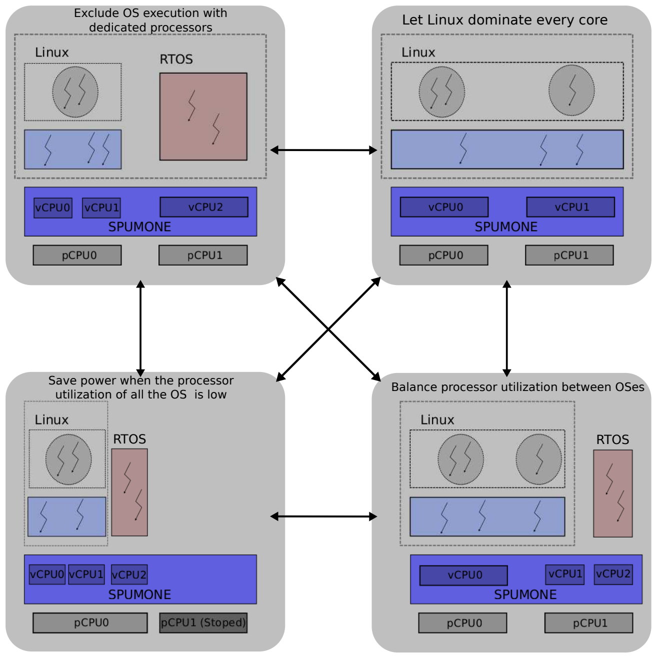

Exclude OS execution with dedicated processors
Linux
RTOS
vCPU0
vCPU1
vCPU2
SPUMONE
pCPU0
pCPU1
Let Linux dominate every core
Linux
vCPU0
vCPU1
SPUMONE
pCPU0
pCPU1
Save power when the processor utilization of all the OS is low
Linux
RTOS
vCPU0
vCPU1
vCPU2
SPUMONE
pCPU0
pCPU1 (Stoped)
Balance processor utilization between OSes
Linux
RTOS
vCPU0
vCPU1
vCPU2
SPUMONE
pCPU0
pCPU1

Figure 2: Dynamically Changing the Mapping Between Virtual CPUs and Physical CPUs

There are several advantages of our approach. The first advantage is to change the mapping between vCPUs and pCPUs to reduce energy consumption. As shown in Figure 2, we assume that a processor offers two p-CPUs. Linux uses two vCPUs and the real-time OS uses one vCPU. When the utilization of RTOS is high, two vCPUs of Linux are mapped on one pCPU (Left Top). When RTOS is stopped, each vCPU of Linux uses a different pCPU (Right Top). Also, one pCPU is used by a vCPU of Linux and another pCPU is shared by Linux and RTOS when the utilization of RTOS is low (Right Below). Finally, when it is necessary to reduce energy consumption, all vCPUs run on one pCPU (Left Below). This approach enables us to use very aggressive policies to balance real-time constraints, performance, and energy consumption.

[3]The RP1 platform is our current hardware platform that contains a multicore processor. The processor has four SH4 CPUs and they are communicated with a shared memory. The platform is developed by Hitach and Renesas.

Configuration	Time	Overhead
Linux Only	68m 5.9s	-
Linux and TOPPERS	69m 3.1s	1.4%

Figure 3: Linux kernel build time

OS(Linux version)	Added LoC	Removed LoC
Linux/SPUMONE(2.6.24.3)	161	8
RTLinux 3.2(2.6.9)	2798	1131
RTAI 3.6.2(2.6.19)	5920	163
OK Linux (2.6.24)	28149	-

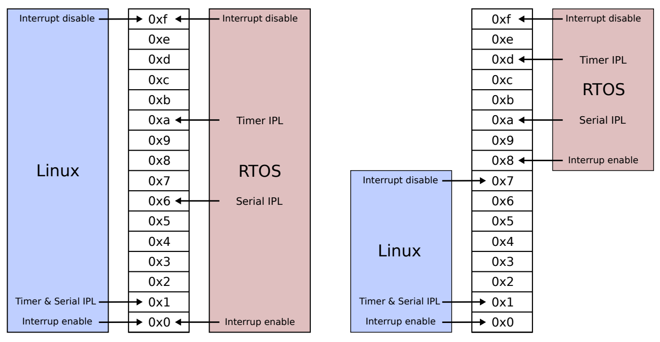
Figure 4: The total number of modified LoC in *.c, *.h, *.S and Makefile

3.2.5 Performance and Engineering Cost

Figure 3 shows the time required to build the Linux kernel on native Linux and modified Linux executed on the top of SPUMONE together with TOPPERS. TOPPERS only receives the timer interrupts every 1ms, and executes no other task. The result shows that SPUMONE and TOPPERS impose the overhead of 1.4% to the Linux performance. Note that the overhead includes the cycles consumed by TOPPERS. The result shows that the overhead of the existence of SPUMONE to the system throughput is sufficiently small.

We evaluated the engineering cost of reusing RTOS and GPOS by comparing the number of modified lines of code (LoC) in each OS kernel. Figure 4 shows the LoC added and removed from the original Linux kernels. We did not count the lines of device drivers for inter-kernel communication because the number of lines will differ depending on how many protocols they support and how complex they are. We did not include the LoC of utility device drivers provided for communication between Linux and RTOS or Linux and servers processes because it depends on how many protocols and how complex those are implemented.

The table also shows the modified LoC for RTLinux, RTAI and OK Linux, all of which are previous approaches to support multiple OS environments. Since we could not find RTLinux, RTAI, OK Linux for the SH4a processor architecture, we evaluated them developed for the Intel architecture. OK Linux is a Linux kernel virtualized to run on the L4 microkernel. For OK Linux, we only counted the code added to the architecture dependent directory `arch/l4` and `include/asm-l4`. The results show that our approach requires



(a) Without IPL separation (b) With IPL separation

Figure 5: Separating Interrupt Priorities Between Guest OSes

only small modifications to the Linux kernel. The result shows that the strategy of SPUMONE, virtualizing processors only, succeeds in reducing the number of modification of guest OSes and to satisfy the requirements described in Section 1.

4 Real-Time Resource Management in SPUMONE

4.1 Reducing RTOS Dispatch Latency

In order to minimize the dispatch latency of RTOS tasks during concurrent activities of Linux on a single device, we propose the following two techniques in SPUMONE.

4.1.1 Interrupt Priority Level Separation

The first technique is to replace the interrupt enabling and disabling instructions with the hypervisor calls. A typical OS disables all interrupt sources when disabling interrupts for the atomic execution. For example, `local_irq_enable()` of Linux enables all interrupt and `local_irq_disable()` disables all interrupt. On the other hand, our approach leverages the interrupt priority mechanism of the processor. The SH4a processor architecture provides 16 interrupt priority levels (IPLs). We assign the higher half of the IPLs to RTOS and the lower half to Linux as shown in Figure 5. When Linux tries to block the interrupts, it modifies its interrupt mask to the middle priority. RTOS may therefore preempt Linux even if it is disabling the interrupts. On the other hand, when RTOS is running, the interrupts for

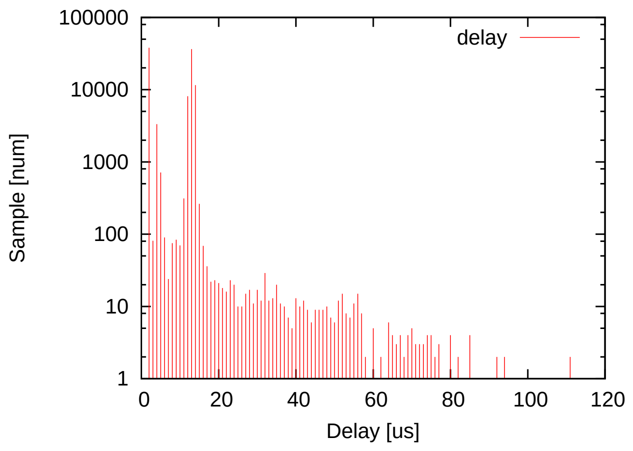


Figure 6: Interrupt dispatch latency of TOPPERS without IPL separation

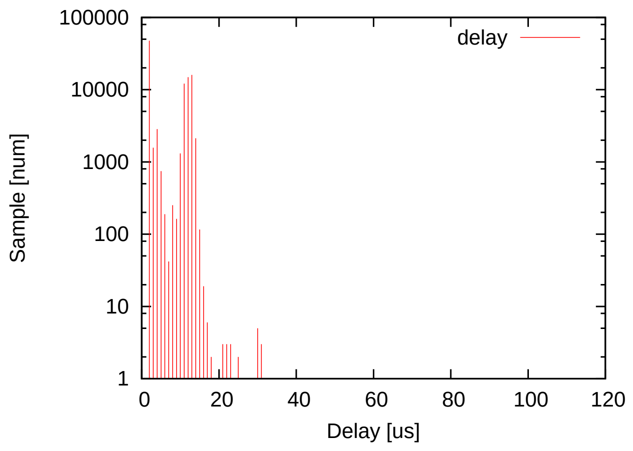


Figure 7: Interrupt dispatch latency of TOPPERS with IPL separation

Linux are blocked by the processor. These blocked interrupts could be delivered immediately when Linux is dispatched.

The instructions for enabling and disabling interrupts are typically provided by the kernel internal API like `local_irq_enable()` and `local_irq_disable()`. They are typically coded as inline functions or macros in the kernel source code. For Linux, we replace `local_irq_enable()` with the hypervisor call which enables entire level of interrupts and `local_irq_disable()` with the other hypervisor call which disables the lower priority interrupts. For RTOS, we replace the API for interrupt enabling with the hypervisor call enabling only high priority interrupts, and the API for interrupt disabling with the other hypervisor call disabling the entire level of interrupts. Therefore, interrupts assigned to RTOS are immediately delivered to RTOS, while interrupts assigned to Linux are blocked during execution of the RTOS. Figure 5 shows the interrupt priority levels assignment for each OS, which we used in the evaluation environment.

Figures 6 and 7 show the task dispatch latency of TOPPERS under two configurations of SPUMONE. In Figure 6, the evaluation result of SPUMONE without the IPL separation executing Linux and TOPPERS is depicted. Linux executes `write()` on the file stored on a Compact Flash card repeatedly and TOPPERS measures the task dispatch latency of the interrupts from time management unit. In Figure 7, the result of SPUMONE with the IPL separation is shown. The guest OSes and their workloads are the same as the condition used in a case when the IPL separation is not used.

As these results show, the workload of Linux heavily interference with the task dispatch latency of RTOS if the IPL separation is not configured. Therefore we can say that separating IPL is an effective method to guarantee the low interrupt dispatch latency of RTOS.

However, the approach assumes that all activities in TOPPERS are processed at the higher priority than the activities of Linux. The current version of Linux is improving real-time capabilities. So, in the near future, some applications that requires to satisfy real-time constraints will be developed on Linux. In this case, the approach described here cannot be used. Also, the approach increases the number of modifications of Linux. It is desirable not to replace interrupt enable/disable instructions in terms of the engineering cost. Therefore, we have developed an alternative method described in the next section.

4.1.2 Reducing Task Dispatching Latency with vCPU migration

The second technique is based on the vCPU migration mechanism introduced in Section 3.2.4. The first technique, replacing API for interrupt enabling/disabling requires slight but non-trivial modification of Linux. In addition, the technique may not work correctly when the device drivers or kernel modules are programmed in a bad manner, which enable or disable interrupts with a non-standard way. The second technique exploits the vCPU migration mechanism. Under this technique, SPUMONE migrates a vCPU, which is assigned to Linux and shares the same pCPU with the vCPU of RTOS, to another pCPU when it traps into kernel mode, or when interrupts are received. In this way, only the user level code of Linux is executed concurrently on the shared pCPU, which will never change the priority levels. Therefore, RTOS may preempt Linux immediately without separating IPL used in the first technique.

4.2 Increasing the Throughput of SMP Linux

Generally speaking, porting OSes to virtualization layers produces semantic gap because the assumptions which guest OSes rely on may not be preserved. For example, OSes assume that they dominate CPU, memory, and storage. In the ordinary environment where OSes run directly on the real hardware, this assumption is true. But when virtualization layers execute guest OSes, this assumption is no longer held. CPU and memory are shared by multiple OSes.

The semantic gap produced by virtualization layers can cause some new problems. One of the typical problems is called the Lock Holder Preemption (LHP) problem [11].

The LHP problem occurs when the vCPU of the guest OS is preempted by the virtualization layer during the execution of critical sections protected by mutex based on busy waiting (e.g. `spinlock_t`, `rwlock_t` in Linux). Figure 8 depicts the typical scenario of LHP in SPUMONE. On SPUMONE, the execution of vCPU2 belonging to RTOS is started immediately even if vCPU1 executing Linux is currently running on the same processor, because the activities of RTOS are scheduled at a higher priority than the activities in Linux. Let us assume that the execution of the Linux kernel is preempted while the kernel keeps a lock. In this case, other vCPUs owned by Linux and running on other pCPUs may wait to acquire the lock via busy waiting.

4.2.1 Existing Solutions of LHP

This performance degradation problem caused by LHP is a general one of every virtualization layer. So, there are existing solutions for solving the problem. Uhlig, et al. pointed this problem [11]. They also introduced the methods to avoid the problem. The method is named as *Delayed Preemption Mechanism (DPM)*. DPM is suitable for a virtualization layer based on the para-virtualization technology because it does not waste CPU time and can be implemented with a less effort. However, this solution increases the dispatch latency of guest OSes, making it unsuitable for embedded systems that need to satisfy real-time constraints.

VMware ESX employs a scheduling algorithm called co-scheduling [15] in its vCPU scheduler [16]. This solution wastes lots of CPU time. VMware ESX employs the technique because it is the full-virtualization technique. Also, it does not assume to execute multiple vCPUs for a guest OS on one pCPU. Sukwong and Kim introduced the improved co-scheduling algorithm named *balance scheduling* and implemented it on KVM [17].

Wells, et al. introduced a hardware based solution called *spin detection buffer (SDB)* for detecting meaningless spin of vCPUs produced by LHP [13]. They found that the execution pattern can be distinguished when a CPU is spinning before acquiring a lock. SDB inspects the number of store instructions and counts the number of updated memory address if a thread is executed in kernel mode. If the number of counted addresses does not exceed the threshold (they set it as 1024), SDB judges the thread is spinning in vain. This hardware information can be used by a virtualization layer to avoid the LHP problem.

Friebel and Biemueller introduced a method for avoiding LHP on Xen [14]. Respective threads in the guest OSes count the number of spinning on a busy wait mutex. When the count exceeds the threshold, the spinning thread invokes the hypervisor call in order to switch to another vCPU.

4.2.2 Solving LHP in SPUMONE

The methods described in Section 4.2.1 improve the throughput of SMP Linux on traditional VMMs. But all of them assume that there are no real-time activities.

In this section, we propose a new method for avoiding LHP. In our approach, the vCPU of Linux, which shares pCPU with the vCPU of RTOS, is migrated to another pCPU when an interrupt for RTOS is received. Then, it returns to the original pCPU when RTOS yields pCPU and becomes idle. When two vCPUs for Linux are executed on the same pCPU, they also cause the

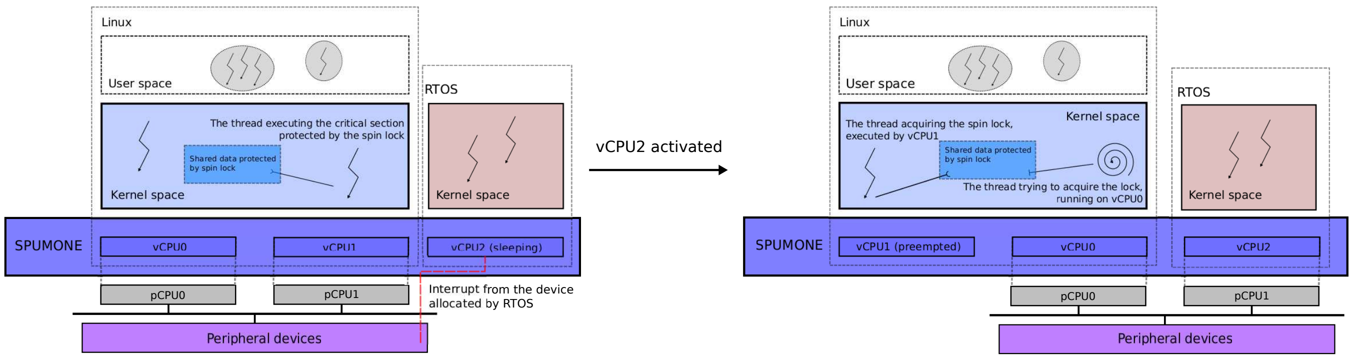

Linux
User space
The thread executing the critical section protected by the spin lock
Shared data protected by spin lock
Kernel space
RTOS
Kernel space
SPUMONE
vCPU0
vCPU1
vCPU2 (sleeping)
pCPU0
pCPU1
Interrupt from the device allocated by RTOS
Peripheral devices
vCPU2 activated
Linux
User space
The thread acquiring the spin lock, executed by vCPU1
Kernel space
Shared data protected by spin lock
The thread trying to acquire the lock, running on vCPU0
RTOS
Kernel space
SPUMONE
vCPU1 (preempted)
vCPU0
vCPU2
pCPU0
pCPU1
Peripheral devices

Figure 8: Typical example of Lock Holder Preemption in SPUMONE

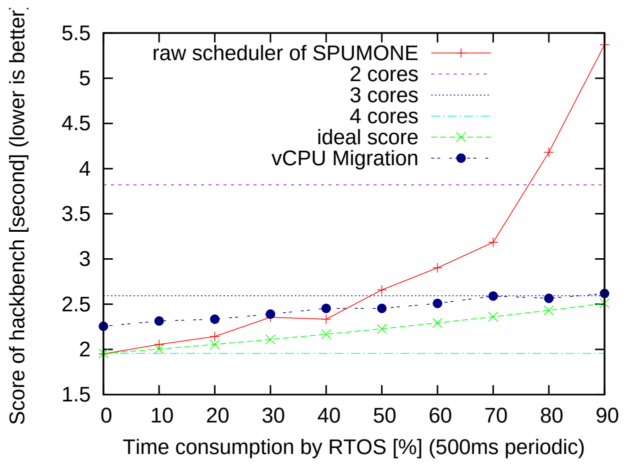


Figure 9: Result of hackbench on Various Configuration

LHP problem. But, in this case, we assume that the delayed preemption mechanism can be used since Linux does not have real-time activities. The vCPU migration mechanism is similar to the thread migration in normal OSes, but in the case of SPUMONE, interrupt assignments have to be reconfigured because peripherals devices are not virtualized. In our evaluation environment, timer interrupts and ICI should be taken into account. Let us assume that vCPU0 is migrated from pCPU0 to pCPU1 while executing an activity on vCPU1. The timer device raising interrupts periodically for vCPU0 on pCPU1 should be stopped before the vCPU migration. Then, the timer device on pCPU1 should be multiplexed for both vCPU0 and vCPU1. Also, ICI for vCPU0 on pCPU0 should be forwarded to vCPU0.

Figure 9 shows the hackbench score on various configurations of SPUMONE. In this evaluation environment, four pCPUs execute five vCPUs. Therefore two vCPU share one pCPU. One vCPU belongs to TOPPERS and four vCPUs belong to Linux. TOPPERS executes a task which consumes CPU time in the 500ms period. Linux executes hackbench for measuring its throughput. The X axis means the CPU consumption rate of the task on TOPPERS, and the Y axis means the score of hackbench. Three horizontal lines describe the score of hackbench under the case that Linux dominates pCPUs.

The line indicated as "ideal score" describes the score which we expected at first. When RTOS consumes the time of $f (0 \leq f < 1)$ on one pCPU, Linux should exploit the rest of CPU resources: $4 - f$ (When $f = 1$, which means RTOS never yields pCPU, the rest of CPU resources is not equal to 3. Because this is the same as the situation when 1 CPU stops execution suddenly from the perspective of Linux). The line of the ideal score is calculated as: $I(f) = \frac{S_1}{4-f}$ where f means the CPU consumption rate of RTOS and S_1 means the score of Linux dominating one core. hackbench has enough parallelism, therefore we predicted that the score might be linear according to the CPU consumption rate of RTOS.

The score actually measured is presented as the line indicated as "raw scheduler of SPUMONE". We noticed that the rapid degradation of the performance is caused by LHP. So we designed and implemented the new method described above. The score measured when using the new method is described as the line indicated as "vCPU migration". This score is still worse than the ideal score, but it sufficiently utilizes the CPU resource because it is better than or nearly equal to the case when Linux dominates three cores.

Current score when using our new method is still worse than the ideal score, so more optimization or better vCPU scheduling policy is required. We are planning to apply the method described in Section 4.1.2. In modern systems, mutexes based on busy wait mechanism are only used in kernel space. Therefore if the vCPU of Linux, shareing pCPU with the vCPU of RTOS, is migrated to another pCPU when the thread invokes system calls or the interrupts for Linux rises, LHP can be avoided.

4.3 Real-Time Task Aware Scheduler

One of the ongoing projects of SPUMONE, we plan to use these additional resources to further improve the real-time capability of guest OSes, especially Linux, by dynamically scheduling the vCPU of the guest OSes on top of the SPUMONE. In the original design strategy of SPUMONE, we gave a high priority to the vCPU of RTOS which is higher than the priority of the vCPU of Linux. But this is not always the case; there might exist some real-time processes that have quicker response time requirements than the RTOS processes. In this situation, we can mark one of the vCPUs of Linux as rt-vCPU and schedule it against vCPU of RTOS. When the priority of this rt-vCPU is higher than that of the vCPU of RTOS, it can gain the control of the pCPU, but simultaneously, because we have some other pCPU in multicore system, we can migrate the vCPU of RTOS to another core and compete with other vCPUs, so the overall performance will not be harmed too much. But the overhead of this migration operation has to be carefully taken care of.

5 Offering Spatial Protection in SPUMONE

As described in Section 3, SPUMONE locates guest OS kernels and SPUMONE in the same privileged level. However, the Linux kernel might contain security holes because of its huge source code, and there are possibilities to infect the Linux kernel. For increasing the reliability of the entire system, a virtualization layer should offer mechanisms to protect a virtualization later and co-existing RTOSes. In traditional approaches, strict memory isolation is used for the protection, but our approach cannot rely on the traditional solution because it is too expensive for typical embedded systems. SPUMONE offers two mechanism to increase the reliability of the entire system. In the following sections, we will explain the mechanisms in detail.

5.1 Protecting SPUMONE and RTOS

In the virtualization environment of SPUMONE, guest OS kernels are running side by side with SPUMONE in the most privileged level. This means that these kernels and SPUMONE could be affected by one and another. In order to further improve the security of the system, we try to better protect these kernels without implementing too much functionality in SPUMONE. We did so by taking advantage of the distributed design of SPUMONE in the multicore platform [21]. Multicore design of SPUMONE is different from traditional VMMs in that each physical core has its own dedicated virtualization layer instance, while the traditional ones have only one instance across all available physical cores. We then simply install each SPUMONE instance into the local memory area of each physical core. Because the content of local memory is only accessible from its own physical core, the attacks or intrusions from the other cores are therefore prohibited as shown in Figure 10. This also means that the attacks will not propagate. When one part of the system is broken and tries to affect others, it will not make it. So the remaining part of the systems can operate normally.

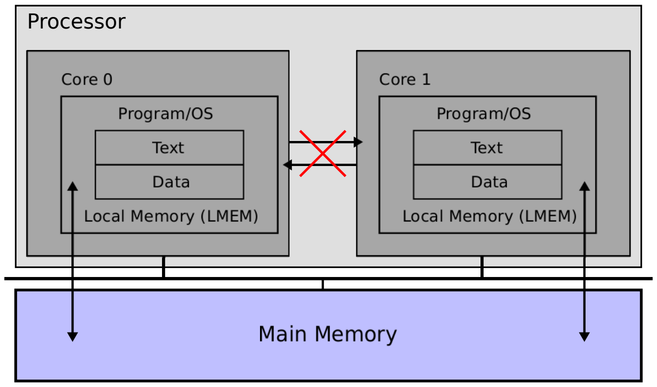


Figure 10: Separated Core-Local Memory and Its Application for Security

5.1.1 Core-Local Memory

Let us assume that two OS kernels running on top of a dual-core processor where each core has an independent core-local memory. If the following assumptions are satisfied, an OS kernel is protected from others.

1. The size of an OS kernel is small enough to fit in core-local memory.
2. Each core should be restricted to reset other core where the reset cleans up the content of the core-local memory.

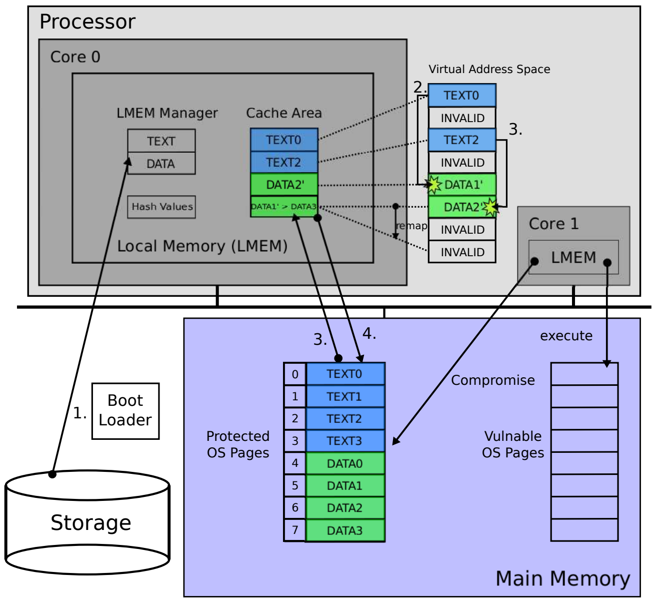


Figure 11: An Overview of Hash-Based Integrity Management

3. The boot image of an OS kernel should not be infected, and a secure boot loader can load the kernel image in the shared memory correctly.

4. Each core should be restricted to access I/O devices. I/O devices that are managed by a core should not be accessed from other cores.

5.1.2 Hash-Based Integrity Management

The problem of the solution presented in the previous section is the size of core-local memory. Currently they are a few hundred KBs. It is too small to load a modern RTOS. In order to virtually extend the size of a local memory, we propose a hash-based integrity management mechanism assisted by the core-local memory protection. The original kernel image is stored in the shared main memory, and a subset of the kernel image is copied to the core-local memory before execution by the core. When a part of the kernel image is loaded in the core-local memory, this part is verified every time to make sure that it is not corrupted or infected.

We present how the hash-based integrity management works in Figure 11. The page allocation in a core-local memory and the calculation of cryptographic hash values are managed by the local memory (LMEM) manager that resides permanently in the core-local memory.

An OS kernel image that can be protected from other OS kernels is called a protected OS (pOS). An OS kernel that may be infected by malicious activities is called a vulnerable OS (vOS). pOS and vOS must run on different cores.

1. The boot loader loads the LMEM manager into the core-local memory. The OS kernel images of pOS and vOS are loaded at the same time into the main memory. LMEM calculates the hash value of each page of pOS, and stores it in a hash table also located in the core-local memory. The manager loads a memory page that contains the entry point of pOS into the core-local memory. Then the other core may start to execute vOS.

2. The pages of pOS are mapped in a virtual address space, and a page table for managing the virtual address space should be in the core-local memory. When the size of the page table is bigger than the size of the core-local memory, LMEM can swap out unused page tables to the shared memory. LMEM also manages the hash table for maintaining the integrity of the swapped page tables. LMEM manages page faults when the page table does not contain a corresponding page table entry.

3. When LMEM handles a page fault, a corresponding page is copied from the shared memory to the core-local memory. LMEM calculates the hash value of the page and compares it with the pre-calculated value stored in the core-local memory. A mismatch of the hash value means that the image of pOS in the shared memory is corrupted. If there is no mismatch, the page fault is correctly completed and the execution of pOS is resumed.

4. When there is no space available in the core-local memory, LMEM swaps out some pages to the shared memory. LMEM checks whether the page is updated or not, and if it is updated, LMEM recalculates the hash value of the page and updates the hash table entries. The pages will be used for loading other pages.

In this approach, the image of pOS in the shared memory may be corrupted by vOS. Our current policy is to restart pOS by reloading a new undamaged kernel image by a secure loader. We are also considering a technique to protect a kernel image by using a memory error correction technique and an encryption technique.

6 Related Work

Virtualization technologies are already used in the area of the desktop and data center computing today [4, 5, 6]. It is also becoming a strong technique for constructing real-time embedded systems because of an enhancement of processors targeting embedded systems market.

RTLinux [22] is a well known hard real-time solution for Linux, but it is also known with its patent problems. RTAI [23] is another real-time extension for Linux and is distributed as free software, but it requires significant modification of the source code of Linux.

KVM for ARM [10] is a KVM based lightweight para-virtualization technology for ARM processors. This might be a strong candidate of virtualization technology for real-time embedded systems because it only requires automated modification of the guest Linux.

OKL4 [12] is a hypervisor based on the micro-kernel design. Armand and Gien introduced the poorness of its performance come from the design of micro-kernel [9]. VirtualLogix VLX [9] is a practical designed VMM for real-time embedded systems.

And in our best knowledge, none of them can handle LHP on multicore processors while guaranteeing real-time responsiveness when the guest OSes have asymmetrical priorities and roles.

7 Conclusion and Future Directions

Before concluding our paper, we would like to share our experiences with the difficulties to promote open source software in Japanese embedded system industries.

We have been discussing various aspects of open source software with embedded system industries for a long time. We found that a lot of people in industries who are working on open source software are aware of the the merits. Especially, asking questions to communities is very useful to find good solutions to problems. However, their bosses who were hardware engineers before do not understand the merits because in their cases, the solutions should be solved by themselves inside of their industries. The cultural gaps between the generations inside industries becomes one of the biggest obstruction to work with open source communities.

Now, we already know that social networks have significantly strong power on sharing knowledge. Open source communities are kind of social networks to share knowledge about open source software, and engineers who ask questions to communities also need to answer question of other people in the communities, but it sometimes too difficult to make time for discussing open source communities while they are working.

Also, it is not easy that embedded system industries contribute their software on open source communities because they sometimes use the old version of software. Because the modification of the old version is not easy to be integrated into the current version of the open source software.

In this paper, we introduced SPUMONE that is a virtualization layer for multicore processor based embedded systems. We described an overview of SPUMONE and showed how SPUMONE reduces the RTOS dispatch latency and protects SPUMONE and RTOS from malicious attacks on the Linux kernel. Currently, we are preparing to distribute SPUMONE as open source software.

References

[1] Tatsuo Nakajima, Yuki Kinebuchi, Hiromasa Shimada, Alexandre Courbot, Tsung-Han Lin. Temporal and Spatial Isolation in a Virtualization Layer for Multi-core Processor based Information Appliances. In *Proceedings of the 16th Asia and South Pacific Design Automation Conference*, 2011.

[2] eCos. http://ecos.sourceware.org

[3] TOPPERS Project. http://www.toppers.jp/en/index.html

[4] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003.

[5] VMware. http://www.vmware.com

[6] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin Qumranet, Anthony Liguori. kvm: the Kernel-based Virtual Machine. In *Proceedings of Ottawa Linux Symposium*, 2007.

[7] Ingo Molnar. RT-patch. http://www.kernel.org/pub/linux/kernel/projects/rt/

[8] Paul E McKenney. 'Real Time' vs. 'Real Fast': How to Choose? In *Proceedings of the Ottawa Linux Symposium*, 2008.

[9] François Armand and Michel Gien. A Practical Look at Micro-Kernels and Virtual Machine Monitors. In *Proceedings of the 6th IEEE Conference on Consumer Communications and Networking Conference*, 2009.

[10] Christoffer Dall and Jason Nieh. KVM for ARM. In *Proceedings of Ottawa Linux Symposium*, 2010.

[11] Volkmar Uhlig, Joshua LeVasseur, Espen Skoglund, and Uwe Dannowski. Towards Scalable Multiprocessor Virtual Machines. In *VM'04: Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium*

[12] Open Kernel Labs. OKL4 Microvisor. http://www.ok-labs.com/products/okl4-microvisor

[13] Philip M. Wells, Koushik Chakraborty, and Gurindar S. Sohi. Hardware Support for Spin Management in Overcommitted Virtual Machines. In Proc. *of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT-2006),* Sept. 2006, Seattle, WA

[14] Thomas Friebel and Sebastian Biemueller. How to Deal with Lock Holder Preemption. http://www.amd64.org/fileadmin/user_upload/pub/2008-Friebel-LHP-GI_OS.pdf

[15] J. K. Ousterhout. Scheduling Techniques for Concurrent Systems. *Proceedings of Third International Conference on Distributed Computing Systems, 1982*

[16] VMware, Inc. VMware vSphere(TM) 4: The CPU Scheduler. in VMware(R) ESX(TM) 4 http://www.vmware.com/files/pdf/perf-vsphere-cpu_scheduler.pdf

[17] Orathai Sukwong and Hyong S. Kim. Is Co-scheduling Too Expensive for SMP VMs? In *Proceedings of the ACM European conference on Computer systems*, 2011.

[18] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris Rebecca Isaacs, Simon Peter Timothy Roscoe, Adiran Schüpbach, Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, 2009.

[19] Yuki Kinebuchi, Takushi Morita, Kazuo Makijima, Midori Sugaya, Tatsuo Nakajima. Constructing a Multi-OS Platform with Minimal Engineering Cost. In *proceedings of Analysis, Architectures and Modelling of Embedded Systems*, 2009.

[20] Tatsuo Nakajima, Yuki Kinebuchi, Alexandre Courbot, Hiromasa Shimada, Tsung-Han Lin, Hitoshi Mitake. Composition kernel: a multi-core processor virtualization layer for rich functional smart products. In *Proceedings of the 8th IFIP WG 10.2 international conference on Software technologies for embedded and ubiquitous systems*, 2010.

[21] Tsung-Han Lin, Yuki Kinebuchi, Alexandre Courbot, Hiromasa Shimada, Takushi Morita, Hitoshi Mitake, Chen-Yi Lee and Tatsuo Nakajima. Hardware-assisted Reliability Enhancement for Embedded Multicore Virtualization Design. In *the Proceedings of 14th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing*, 2011.

[22] Victor Yodaiken. The RTLinux Manifesto. In *the Proceedings of the 5th Linux Expo, March 1999, in Raleigh North Carolina*

[23] P. Mantegazza, E. L. Dozio, S. Papacharalambous. RTAI: Real Time Application Interface. In *Linux Journal, volume 2000. Specialized Systems Consultants, Inc. Seattle, WA, USA*, 2000.

[24] Hiroaki Inoue, Junji Sakai, Masato Edahiro. Processor virtualization for secure mobile terminals. In *ACM Transactions on Design Automation of Electronic Systems*, Volume 13 Issue 3, July 2008.

Comparing different approaches for Incremental Checkpointing: The Showdown

Manav Vasavada, Frank Mueller
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-7534
e-mail: mueller@cs.ncsu.edu

Paul H. Hargrove, Eric Roman
Future Technologies Group
Lawrence Berkeley National Laboratory
Berkeley, CA 94720

Abstract

The rapid increase in the number of cores and nodes in high performance computing (HPC) has made petascale computing a reality with exascale on the horizon. Harnessing such computational power presents a challenge as system reliability deteriorates with the increase of building components of a given single-unit reliability. Today's high-end HPC installations require applications to perform checkpointing if they want to run at scale so that failures during runs over hours or days can be dealt with by restarting from the last checkpoint. Yet, such checkpointing results in high overheads due to often simultaneous writes of all nodes to the parallel file system (PFS), which reduces the productivity of such systems in terms of throughput computing. Recent work on checkpoint/restart (C/R) has shown that incremental C/R techniques can reduce the amount of data written at checkpoints and thus the overall C/R overhead and impact on the PFS.

The contributions of this work are twofold. First, it presents the design and implementation of two memory management schemes that enable incremental checkpointing. We describe unique approaches to incremental checkpointing that do not require kernel patching in one case and only require minimal kernel extensions in the other case. The work is carried out within the latest Berkeley Labs Checkpoint Restart (BLCR) as part of an upcoming release. Second, we evaluate the two schemes in terms of their system overhead for single-node microbenchmarks and multi-node cluster workloads. In short, this work is the final showdown between page write bit (WB) protection and dirty bit (DB) page tracking as a hardware means to support incremental checkpointing. Our results show savings of the DB approach over WB approach in almost all the tests. Further, DB has the potential of a significant reduction in kernel activity, which is of utmost relevance for proactive fault tolerance where an immanent fault can be circumvented if DB-based live migrations moves a process away from hardware about to fail.

1 Introduction

With the number of cores increasing manifold at a rapid rate, high performance computing systems have scaled up to thousands of nodes or processor cores. Also, with the increase in the availability of off-the-shelf components, parallel machines are no more a niche market. Huge scientific applications and even non-scientific applications with highly parallel patterns exploit such machines, and hence provide faster time-to-solution. Even with the high amount of processing power available, such high-end applications experience execution times in the order of hours or even days in some cases. Examples of such applications are general scientific applications, climate modeling, protein folding and 3D modeling. With the use of off-the-shelf components, the Mean Time Between Failure (MTBF) has also been reduced substantially [12], which indicates an increasing probability of hardware failure on such machines. After a failure, the current process would need to be restarted from the scratch. This approach would not only waste CPU cycles and power in duplicated work but also delay the results by a substantial amount of time. To address these problems, fault tolerance is needed.

There have been many approaches to support fault tolerance in HPC. One of the approaches is checkpoint/restart (C/R). This approach involves checkpointing the application on each node at regular intervals of time to non-local storage. Upon failure, the checkpoint is simply shifted to a spare node and the checkpoint is restarted from the last checkpoint instead of restarting the application from the scratch. Checkpointing involves saving the state of the process at a point in time and then using the same data at the time of restart. There have been various frameworks for application as well as system-level C/R.

The checkpoint restart framework which this paper revolves around is Berkeley Labs Checkpoint Restart (BLCR) [6]. BLCR is a hybrid kernel/user implementation of C/R for Linux developed by the Future Technologies Group at Lawrence Berkeley National Laboratory. It is a robust, production quality implementation that checkpoints a wide range of applications without requiring any changes made to the code. The checkpoint process involves saving the process state including registers, virtual address space, open files, debug registers etc., and using the data to restart the process. BLCR support has been tightly integrated into various MPI implementations like LAM/MPI, MVAPICH, OpenMPI and others to enable checkpointing of parallel applications that communicate through MPI.

Researchers at North Carolina State University (NCSU) have been working on various extensions for BLCR. One of the extensions for the BLCR was incremental checkpointing. BLCR's current naive approach checkpoints the entire state of the process at every checkpoint period. In most cases, in accordance with the 90/10 law, the process might sit in a tight loop for the entire period between two checkpoints and only modify a subset of application state. In such cases, checkpointing the entire process not only wastes memory but also time. With large applications, write throughput to disk can rapidly become the bottleneck for large checkpoints. Hence, reducing write pressure on the time-critical path of execution through incremental checkpointing can become quite important.

With the incremental checkpointing approach, the key virtue is the detection of modified data. The most convenient approach would be to detect modifications at page granularity. However, there can be various methods to detect modifications on a page. The previous approach taken by researchers at NCSU was to propagate the dirty bit in the page table entry to user level by using a kernel patch [15].

Contributions:

This paper presents the design, implementation and evaluation of two different approaches to incremental checkpointing. Our contributions are as follows:

- We present an approach for the detection of modified data pages that does not require patching the kernel as in previous work and can instead be used on vanilla kernels.
- We compare and contrast the two approaches for performance and establish the pros and cons of each. This helps the users decide which approach to select based on their constraints.
- We compare the performance of the two approaches against base checkpointing to assess the benefits and limitations of each.
- We show that our lower overhead dirty-bit tracking has the potential of a significant reduction in kernel activity. When utilized for proactive fault tolerance, an immanent fault could more likely be circumvented by dirty bit-based live migration than by a write protection-based scheme due to these overhead. As a result, a process could be migrated from a node about to fail to healthy node with a higher probability under dirty-but tracking than under write protection.

2 Related Work

C/R techniques for MPI jobs frequently deployed in HPC environments can be divided into two categories: coordinated checkpointing, such as LAM/MPI+BLCR [13, 6] and CoCheck [14], and uncoordinated checkpointing, such as MPICH-V [4, 5]. Coordinated techniques commonly rely on a combination of operating system support to checkpoint a process image (e.g. via the BLCR Linux module [6]) or user-level runtime library support. Collective communication among MPI tasks is used for the coordinated checkpoint negotiation [13]. Uncoordinated C/R techniques generally rely on logging messages and their temporal ordering for asynchronous non-coordinated checkpointing, e.g. by pessimistic message logging as in MPICH-V [4, 5]. The framework of OpenMPI [3, 10] is designed to allow both coordinated and uncoordinated types of protocols. However, conventional C/R techniques checkpoint the entire process image, leading to high checkpoint overhead, heavy I/O bandwidth requirements and considerable hard drive pressure, even though only a subset of the process image of all MPI tasks changes between checkpoints. With our incremental C/R mechanism, we mitigate the cost by checkpointing only the modified pages.

Incremental Checkpointing: Recent studies focus on incremental checkpointing [7, 9]. TICK (Transparent Incremental Checkpointer at Kernel Level) [7] is a system-level checkpointer implemented as a kernel thread. It supports incremental and full checkpoints. However, it checkpoints only sequential applications running on a single process that do not use inter-process communication or dynamically loaded shared libraries. In contrast, our solution transparently supports incremental checkpoints for an entire MPI job with all its processes. Pickpt [9] is a page-level incremental checkpointing facility. It provides space-efficient techniques for automatically removing useless checkpoints aiming at minimizing the use of disk space. Yi et al. [17] develop an adaptive page-level incremental checkpointing facility based on the dirty page count as a threshold heuristic to determine whether to checkpoint now or later, a feature complementary to our work that we could adopt within our scheduler component. However, Pickpt and Yis adaptive scheme are constrained to C/R of a single process, just as TICK was, while we cover an entire MPI job with all its processes and threads within processes. Agarwal et al. [1] provide a different adaptive incremental checkpointing mechanism to reduce the checkpoint file size by using a secure hash function to uniquely identify changed blocks in memory. Their solution not only appears to be specific to IBMs compute node kernel on BG/L, it also requires hashes for each memory page to be computed, which tends to be more costly than OS-level dirty-bit support as caches are thrashed when each memory location of a page has to be read in their approach. A prerequisite of incremental checkpointing is the availability of a mechanism to track modified pages during each checkpoint. Two fundamentally different approaches may be employed, namely a page protection mechanism for the write bit (WB) or a page table dirty bit (DB) approach. Different implementation variants build on these schemes. One is the bookkeeping and saving scheme that, based on the DB scheme, copies pages into a buffer. Another solution is to exploit page write protection, such as in Pickpt and checkpointing for Grids under XtreemOS [11], to save only modified pages as a new checkpoint. The page protection scheme has certain draw-backs. Some address ranges, such as the stack, can only be write protected if an alternate signal stack is employed, which adds calling overhead and increases cache pressure.

We present two different approaches to incremental checkpointing in this work. The first approach exploits the write bit (WB) to detect modifications on a page level. This approach does not require the kernel to be patched (unlike Grid checkpointing under XtreemOS, which required a patch [11]). This is different than the prior work since it uses innovative approaches to handle corner cases for detecting modifications on pages. The second approach uses the dirty bit (DB) for tracking writes on page. This approach shadows the DB from the kernel within the user level and captures the modification status of the page. Both our approaches work for entire MPI jobs.

3 Design

This section describes the design of incremental checkpointing in BLCR. The main aim of the incremental checkpointing facility is to integrate it seamlessly with BLCR with minimal modifications to the original source code. The enhanced code should also have a minimal overhead while taking incremental checkpoints. When incremental checkpointing is disabled, it should allow BLCR to checkpoint without any additional complexity. For this purpose, we have divided the checkpoints into three categories.

- Default Checkpoint: checkpointing sans incremental code;
- Full Checkpoint: Fully checkpointing of the entire process despite of any modifications;
- Incremental Checkpoint: Checkpointing of only modified data pages of a process.

In the above list, Default and Full checkpoints would be identical in their output but different in their initialization of various data structures, which is detailed later.

The main criteria of the design of incremental checkpointing is to provide a modular approach. The most critical task in incremental checkpointing is to detect the modification of data pages in order to determine whether it should be checkpointed (saved) or not. Currently, we support two approaches. Based on previous work done at NCSU, the first approach is called the dirty bit (DB) approach. The details of this approach are discussed below. This approach requires users to patch their kernels and recompile it. Another approach we designed avoids the patching of the kernel. It instead uses the currently existing mechanisms in the kernel to detect modifications to pages.

In addition to the above approaches, other solutions may be designed in future depending on the features provided by the Linux kernel and the underlying hardware. To efficiently support different algorithms with minimal code modifications, we designed an interface object for incremental checkpointing that unifies several of the essential incremental methods. Algorithms simply "plug in" their methods, which are subsequently called at appropriate places. Hence, BLCR remains agnostic to the underlying incremental implementation. This interface needs to encompass all methods required for incremental checkpointing.

3.1 Incremental Interface

The incremental interface uses BLCR to call the incremental checkpointing mechanism in a manner agnostic to the underlying implementation. This enables various incremental algorithms to be implemented without major code changes in the main BLCR module. The interface object is depicted in Figure 1.

```
int (*init)(cr_task_t *, void *);
int (*destroy)(cr_task_t *, void *);
int (*register_handlers)(cr_task_t *cr_task, struct vm_area_struct *map);
int (*page_modified)(struct mm_struct *mm, unsigned long addr, struct vm_area_struct *map);
int (*shvma_modified)(struct vm_area_struct *vma);
int (*clear_vma)(struct vm_area_struct *vma);
int (*clear_bits) (struct mm_struct *mm, unsigned long addr);
```

Figure 1: BLCR incremental object interface

With this object, existing BLCR code is converted to function calls. If they are not defined, BLCR will behave as it would without any incremental checkpointing. At the first checkpoint, this object would be created per process and associated with a process request. The high level design is depicted in Figure 2.

The initialization function allows a specific incremental approach to set up the data structures (if any), initialize pointers etc. Similarly, the destroy function lets the specific module free up used memory and/or unregister certain handlers. The detection of modified data pages might utilize existing kernel handlers or hooks that need to be registered. The register_handler function is used for registering specific hooks. This function is utilized here to register hooks for memory mapping and shared

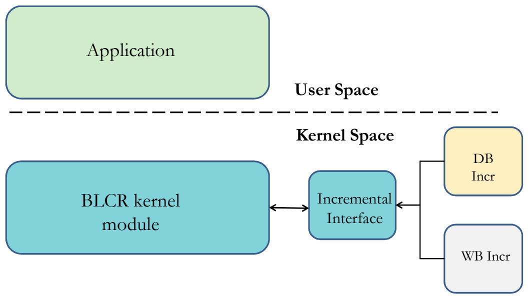


Figure 2: BLCR incremental design

writes. The mmap hooks keep track of mapping and unmapping of the memory pages to ensure that the newly mapped pages are not skipped as described in one of the cases. The page_modified function is the heart of this interface object. It returns a boolean value indicating whether the page has been modified or not. Similarly, shvma_modified returns a boolean for whether a shared page has been modified or not. After each incremental checkpoint, clear_vma and clear_bits can be used to reset the bits for the next checkpoint

3.2 Write Bit Approach

The WB approach is inspired by work by Mehnert-Spahn et al. [11] and tracks the modified data pages. However, they implemented their mechanism on Kerrighad Linux/SSI through source code modifications. One of the main criteria behind the design of this approach was to ensure that no modifications of kernel code were required. Therefore, in addition to the WB, additional mechanisms were utilized for incremental checkpointing.

In this approach, the WB is cleared at each checkpoint. At the next checkpoint, we check whether the WB is set or not. If the page whose WB is cleared is written to, the Linux kernel generates a page fault. Since the segment permission for writing would be granted, the kernel will simply set the write bit of the associated page table entry and return. The WB serves as an indicator that, if set, implies that the page was modified between checkpoints. If it is not set, the page was not modified between the checkpoints. However, this approach does not work for a number of corner cases. We shall look at those cases and discuss how they can be handled in the following.

3.2.1 VM Area Changes

One of the major issues with the above WB approach is its requirement to track changes in the virtual memory area. Many memory regions might be mapped or unmapped between two successive checkpoints. Some memory regions may be resized. We need to cover all such cases in order to ensure correctness. We have assigned a data structure for each page that tracks the status of the page. The structure and design of this tracking approach will be discussed in the next section. Map tracking includes:

- A page is unmapped: If a page is unmapped between two successive checkpoints, then the corresponding tracking structure for that page needs to be invalidated or removed. To this end, we need to be alerted when a page was unmapped while the process runs. We used the close entry provided in the vm_area structure, which is a hook called when a virtual memory area is being "closed" or unmapped. With that hook, we associate required steps when a memory area is unmapped.
- New regions are mapped: This case describes the instance in which new memory regions are added between two checkpoints. For example, consider an incremental checkpoint 1 written to disk. Before incremental checkpoint 2 is taken, page A is mapped into the process address space. At the next checkpoint, if page A was not modified, it will not be checkpointed since the WB would not be set. However, this would be incorrect. To handle this case, we do not allocate the tracking structure for newly mapped regions. Hence, at the next checkpoint on detecting the missing tracking structure, page A will be checkpointed regardless of the status of the WB in its page table entry.

3.2.2 Corner Cases

One of the more serious cases is posed by the system call mprotect. For a VM area protected against writes, the kernel relies on the cleared write bit to raise page faults and then checks the VM permissions. This case can also give erroneous output. For example, assume page A was modified by the user thus setting the WB. Before the next incremental checkpoint, the user protects the page allowing only reads, effectively clearing the WB. When the next checkpoint occurs, the checkpoint mechanism fails to identify the modification on the data page and, hence, discounts it as an unmodified page. We have handled this case by using the DB. The mprotect function, while setting permission bits, masks the DB. Hence, if the page is modified then we can detect it through the DB.

The other corner case is that of shared memory. In BLCR only one of the processes will capture the shared memory. However, we may miss the modification if the process capturing the shared memory has not modified the data page. To handle this, we reverse map the processes through the cr_task structures and check for modifications in each process tracking structure for the page. If even one of them is modified, then the shared page is dirty and should be checkpointed.

3.2.3 Tracking Structure

The tracking structure for incremental checkpointing is a virtual page table maintained by the BLCR module. This is done for two purposes: (1) to track VM area changes like unmapping, remapping, new mapping etc; (2) to detect writes to shared memory. Only two bits suffice to maintain the tracking state of the page. Initially, the design was to replicate a page table structure in BLCR to maintain the state of each page. Since this will have to be performed for the entire process, using a long type variable would waste a significant amount of memory. We have optimized this tracking structure to use only 4 bits per page. This results in an almost eight-fold reduction in memory usage as compared to maintaining a properly mirrored page table.

3.3 Dirty Bit Approach

The second approach taken by previous work uses the DB for detecting page modifications. It uses an existing Linux kernel patch to copy the PTE DB into user level [15]. The problem with using the DB is that the kernel uses the DB for its own purpose, which might introduce an inconsistency if BLCR and the Linux kernel were both using it simultaneously. The patch introduces redundant bits by using free bits in the PTE and maintaining a correct status of the dirty bit for a given page. This approach requires the kernel to be patched. More significantly, this approach prevents page faults from being raised at every write as in the WB approach but still allows dirty page tracking.

4 Framework

We conducted our performance evaluations on a local cluster. This cluster has 18 compute nodes running Fedora Core 12 Linux x86 64 (Linux kernel- 2.6.31.9-174.fc12.x86_64) connected by a two Gigabit Ethernet switches. Each node in the cluster is equipped with four 1.76GHz processing cores (2-way SMP with dual-core AMD Opteron 265 processors) and 2 GB memory. A large RAID5 array provides shared file service through NFS over one Gigabit switch. Apart from the modifications for incremental checkpointing in BLCR, we also instrumented the code for the BLCR library to measure the time across checkpoints. OpenMPI was used as the MPI platform since BLCR is tightly integrated in its fault tolerance module.

5 Experiments

We designed a set of experiments to assess the overheads and analyze the behavior of two different approaches of incremental checkpointing, namely (i) the WB approach and (ii) the DB approach. The experiments are aimed at analyzing the performance of various test benchmarks for these two approaches in isolation and measuring their impact on the performance of application benchmarks.

Various NAS Parallel Benchmarks [2] (NPB) as well as a microbenchmark have been used to evaluate the performance of above two approaches. From the NPB suite, we chose SP, CG, and LU as their runtimes are long enough for checkpoints. In addition, we devised a microbenchmark that scales from low to high memory consumption in order to evaluate the performance of the incremental approaches under varying memory utilization.

5.1 Instrumentation Techniques

For getting precise measurement of time, the method of instrumentation is quite important. The BLCR framework has been modified to record timings of two levels. Figure 3 depicts the block diagram of an application. In the context of the NPB suite, this would be an MPI program with cross-node communication via message passing [8]. We can issue an ompi-checkpoint command so that the OpenMPI framework will engage in a coordinated checkpoint [10]. To assess the performance, we could simply measure the time across the ompi-checkpointing call. However, this would required modifications to OpenMPI. It would also include the timing for the coordination of MPI processes due to an implicit barrier, which would skew our results. Instead, we modified the BLCR library. We measure the timing across the do_checkpoint call in each of the processes. The processes then output their time to a common file (see Figure 3).

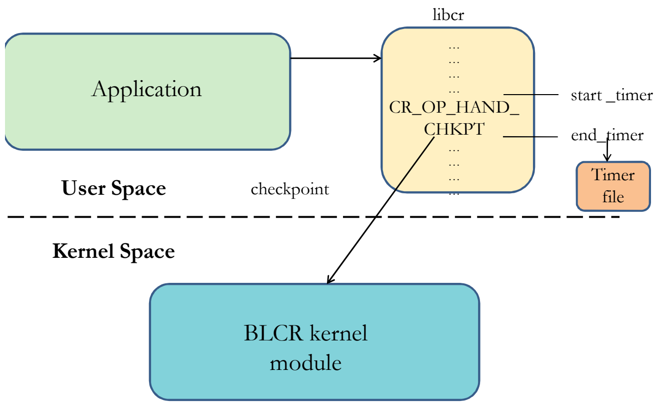


Figure 3: BLCR library timer design

There is one small caveat with the above approach. Our initial tests showed very low variations between the two incremental approaches. After studying timings for various phases, it was found that most of the checkpoint time was dominated by writes to the context file on the file system. This overhead was dominating any other time like, including the time to detect page modifications. Our approach thus was aimed at excluding the write time from the total time. We wanted to only measure the time to detect page modifications. To this end, we enhanced the BLCR kernel module to measure only the modification commands. The design is as depicted in Figure 4. We accrue the timing measurements for modification detection across each page chunk. As a post processing step, we calculate the maximum, minimum and average of all checkpoint timings.

Automated checkpoint scripts enable regular checkpointing of various MPI and non-MPI processes.

5.2 Memory Test

We have split the test suite into two parts. The first part, the memory test, measures the difference between two checkpointing approaches on a single machine. The sec-

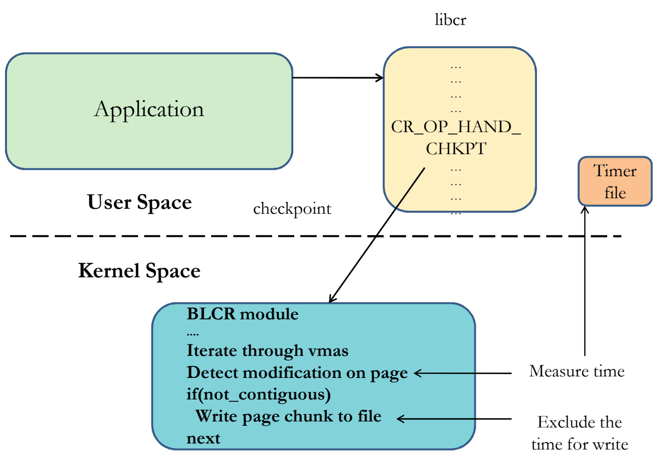


Figure 4: BLCR kernel module timer design

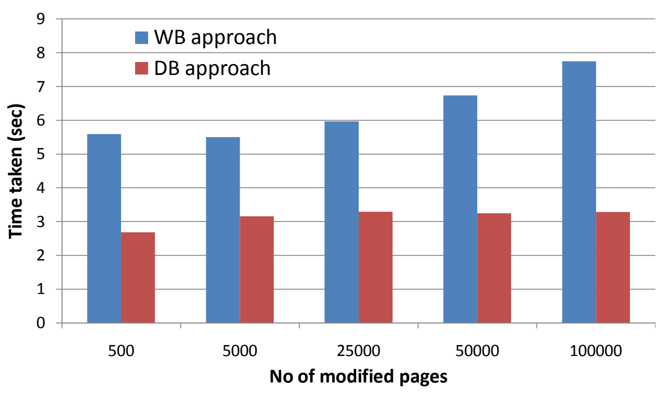


Figure 5: Micro benchmark Test

ond part of experiments measures the impact of performance on multi-node MPI benchmarks as the number of nodes and the memory consumption scales. We discuss the first set of experiments in this section. We have devised a microbenchmark for measuring the performance difference between the two approaches of WB and DB. This benchmark allocates a specified number of memory pages and, depending on the configuration specified, alters a certain number of pages per checkpoint. This allows a comparison of the performance under varying memory utilization.

The experiment is conducted on a large data set. We create a map of 200,000 memory pages within a single process. We constrain the number of checkpoints at 20 with a ratio of incremental to full checkpoints at 4:1. This means a full checkpoint is always followed by four incremental checkpoints as such a hybrid scheme was shown to be superior to only incremental checkpointing [16]. We vary the number of modified pages by large increments. The data points for this graph are at 500, 5k, 25k, 50k, and 100k modified pages. The results are depicted in Figure 5.

Figure 5 indicates that the difference between the performance of DB and WB is low when the set of modified pages is low. As the number of modified pages increases, the difference also increases. When the modified data set reaches 100,000 pages, the difference is almost twice that of WB. We can conclude from this experiment that using the DB approach has significant potential to improve performance.

To understand this result, let us explain the mechanism first. BLCR iterates through every page and checks for modified pages. For each page, the WB and DB approach will use their own mechanisms for checking modified pages. In the WB approach, BLCR has to check its own data structure for mappings of the page (mapped or not). It then fetches the PTE from the address passed to it. After detecting whether a page has been modified or not, the clear bit function clears the WB in the PTE for the next round. For this, the WB approach has to map the PTE again to access it. In DB, on the other hand, the testing for modification and clearing the bit on the PTE happens in a single sweep within the test-and-clear function. In addition to it, the DB approach does not have to manipulate any internal data structures to keep track of mappings. These factors make DB a much faster approach then WB in the above experiment.

We devised a second case using alternate pages to provide insight into the performance difference of incremental vs. default full checkpointing. In this case, alternate pages from the process address space are modified and the performance is assessed. We provided a fixed-size data set of 100k pages here. By writing to ever other page, 50k pages will be modified between checkpoints. We observed that incremental checkpointing takes significantly longer than full default checkpointing. It seems counter intuitive that saving a smaller data set for incremental checkpointing takes more time than saving the full checkpoint data, yet the explanation to this anomaly lies in the way BLCR saves a process' memory space. BLCR iterates through each virtual memory area (VMA) structure to gather contiguous chunks of pages before committing them to stable storage. Upon a full checkpoint, the entire mapped space becomes one chunk written to disk through a single system call. When we modify alternate pages, we encounter an unmodified page after each modified page, where the former is discarded by BLCR as it is unmodified. Since the chunk breaks there, BLCR will have to issue a write to disk for each single modified page. Therefore, we issue significantly more write calls in incremental checkpointing than in full checkpointing. Notice that this deficiency is being addressed by aggregating non-contiguous chunks before committing them to stable storage, but such an approach comes at the cost of additional meta-data to describe the internal check structure.

5.3 NAS Benchmarks

In this section, we analyze the performance of multi-node MPI benchmarks for the WB and DB approaches. We selected the MPI version of the NPB benchmark site [2]. We constrained the selection of benchmarks from the NPB set to those with sufficiently long run-times to take a suitable number of checkpoints.

We devised a set of experiment using strong scaling by increasing the number of processors. With such increase in computing resources, we decrease the per-node and overall runtime. However, this renders some benchmarks unusable for our experiments as the run-time was not sufficient to issue a checkpoint, particularly for smaller input sizes (classes A and B) of the NPB suite. Yet, using large input sizes (class D) under NPB on fewer processors (1 to 4) is not practical either due to excessively long checkpointing times. Hence, we settled for input sizes of class C for the experiments.

Considering all benchmarks and assessing their run-times, we selected three suitable benchmarks for our tests: SP, CG and LU. We present the following experiments and results for the same.

We assessed the performance for the SP benchmark on 4, 9, 16 and 36 processor. Notice that SP requires the number of processors to be a perfect square. The experiments were performed on class C inputs with a checkpoint interval of 60 seconds over a varying number of nodes. Figure 6 depicts the runtime spent inside the Linux kernel portion of BLCR. We observe that the DB approach incurs less overhead in the kernel than the WB approach in all of the cases. We see a downwards slope and a decrease in the difference between DB and WB from 4 processors to 9 processors to 16 processors. The reason for the decrease in time spent in the kernel is that as we increase resources the application is more distributed among nodes. This implies less data per node to checkpoint and, hence, less time spent on checkpointing in the kernel.

In the case of 36 processors, we see a sudden spike in kernel runtime. This anomaly is attributed to the fact we only have 16 physical nodes but deploy the application across 36 processes. Thus, multiple processes are vying for resources on the some nodes. The processes are contending for cache, networking and disk. Hence, the 36-processor case (and any oversubscription case studied in the following) should only be considered by itself and not in comparison to lower number of processes.

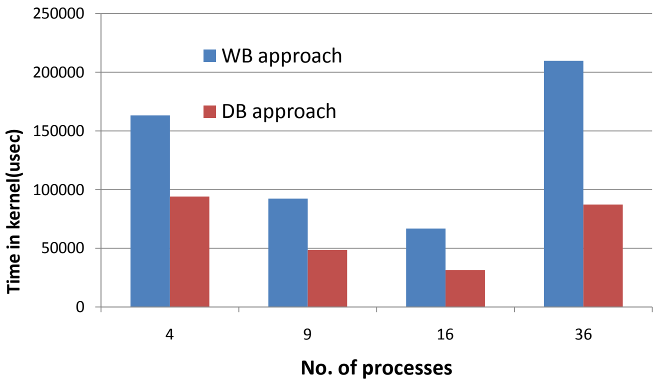


Figure 6: SP benchmark

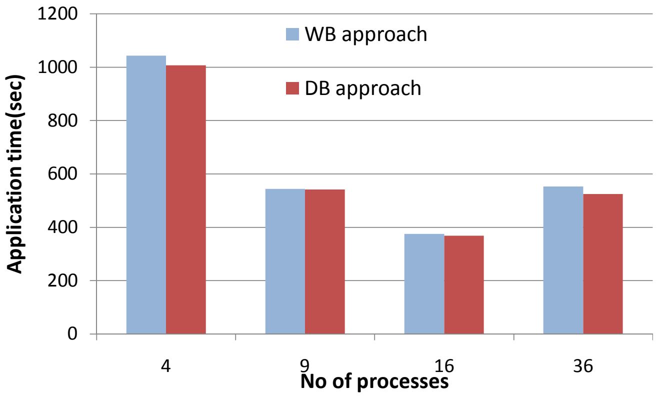


Figure 7: SP benchmark (Application time)

Figure 7 depicts the overall application time for the SP benchmark for different numbers of nodes. We see that the DB approach slightly outperforms the WB approach in all cases. As the number of processes (and processors) increases from 4 over 9 to 16, we see a decrease in total application time. Recall that the work gets distributed between various nodes as the number of processors increases while the application time decreases. This happens under both the configurations, WB and DB. For 36 processes, the application time goes. Again, we are oversubscribing with 36 processes on 16 physical nodes. This causes contention for memory, networking and disk. The DB approach shows slightly higher performance gains for this oversubscription case likely due to mutual exclusion inside the kernel (kernel locks), which impacts WB more due to more time spent in the kernel.

The next NPB program tested was the CG benchmark for class C inputs. We varied the number of nodes from 4 over 8 and 16 to 32 processors. The incremental to full checkpoint ratio is kept at 4:1. Checkpoints are taken every 10 seconds.

Figure 8 depicts the kernel runtime for CG. These results indicate considerable savings for DB over WB for 4 processors and smaller savings for 8 processors. At 16 processors, more savings materialize for DB. In contrast, the overhead of WB increases drastically. This anomaly can be explained as follows. The total running time of CG is low. Checkpoints were taken at an interval of 10 seconds. Since the savings due to the DB approach exceed 10 seconds, the benchmark run under DB resulted in fewer checkpoints, which further decreased the application runtime. In contrast, WB ran past the next checkpoint interval, which incurred one additional checkpoint. Thus, we may gain by issuing fewer checkpoints under DB due to the lower kernel overhead of the latter.

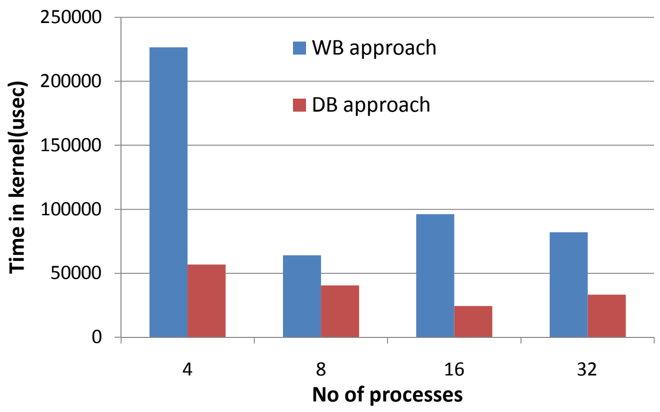


Figure 8: CG benchmark

Figure 9 depicts the total application time for CG. We see considerably more savings of DB over WB in the case of 16 nodes than for 4 or 8 nodes due to the lower number of checkpoints for DB. In all other cases, DB slightly outperforms WB. The higher overall runtime for 32 processes is caused by node oversubscription again.

Next, we assessed the performance under the LU benchmark for 4,8,16 and 32 processors under class C inputs.

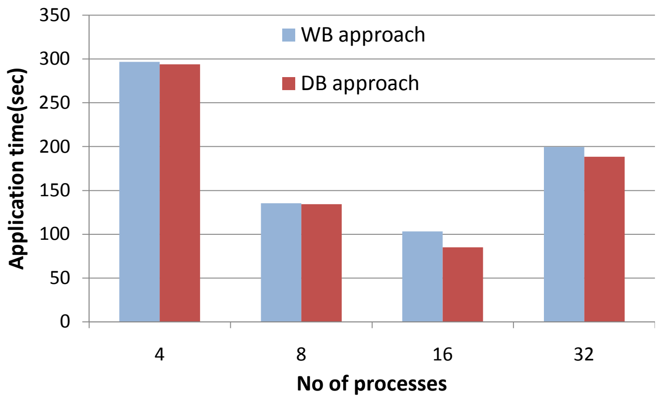


Figure 9: CG benchmark (Application time)

Checkpoints were taken every 45 seconds, and the incremental to full checkpoint ratio was 4:1.

Figure 10 depicts the kernel time. As in the previous experiment, there are significant savings in time spent in the kernel. The total time decreases as the number of nodes increases from 4 over 8 to 16. Under 32 processes, we see an increase of total time relative to the prior process counts due to node oversubscription. The savings of the DB compared to WB are significant in all cases. As in the previous savings, these savings in the order of microseconds only materialize in minor overall application runtime reductions as application runtime is in the order of seconds. The results for total application time for LU are depicted in Figure 11.

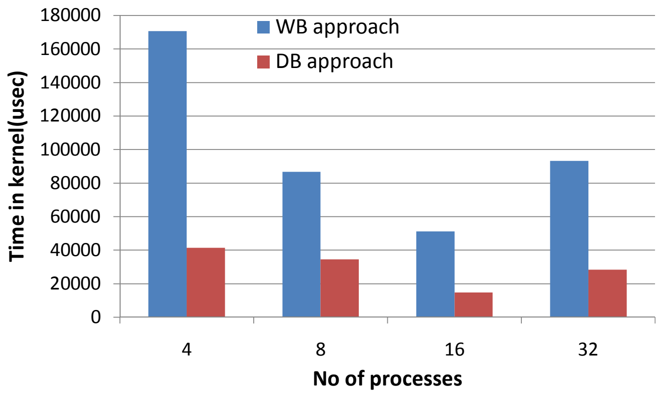


Figure 10: LU benchmark

In terms of overall runtime for LU, we see that DB is at par or slightly outperforms WB. We also see that the percent are quite low compared to the percent savings for kernel runtime in the previous graph. As explained before, this is due to the fact that the time spent in kernel is measured in microseconds while application time is measured in seconds.

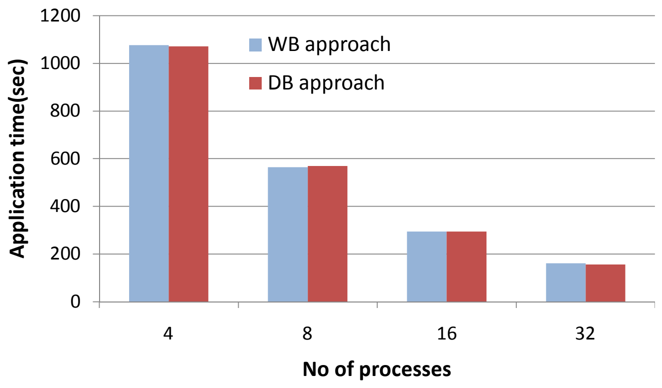


Figure 11: LU benchmark (Application time)

In summary, we observe that the DB approach incurs significantly less kernel overhead but only slightly less overhead than WB approach for most test cases. Larger savings are observed for longer-running applications as a slight reduction in DB overhead may aggregate so that fewer overall checkpoints are taken at the same checkpoint interval.

6 Future Work

We have developed and evaluated two different approaches to incremental checkpointing. One (WB) required patching the Linux kernel to detect modifications at page level while the other (DP) did not require any patches. We have further quantitatively compared the two approaches. We are currently investigating if patching of the kernel for DB can be omitted when swap is turned off. This would alleviate the user from the tedious kernel patching and recompilation of the kernel. This approach is particularly promising for high-performance computing under MPI as swap tends to be disabled. We are currently DB usage within the kernel beyond swap functionality to determine if utilization of DB by the BLCR would create any side effects for the kernel. We are also considering dynamic activation and deactivation of swap while a process is running. In that case, the DB functionality bit should be gracefully handed over to the kernel without affecting ongoing checkpoints. These issues are currently being investigated and we aim to implement them in the future. Furthermore, we are considering to integrate both incremental checkpointing mechanisms, DB and WB, with the latest BLCR release. The mechanism are already in the BLCR repository and the integration work is under way.

7 Conclusion

In this paper, we outlined two different approaches to incremental checkpointing and quantitatively compared them. We conducted several experiments with the NPB suite to determine the performance of the DB and WB approaches in head-to-head comparison them. We make the following observations from the experimental results. (i) The DB approach is faster significantly than the WB approach than DB with respect to kernel activity. (ii) DB also slightly outperforms WB for overall application in nearly all cases, and particularly for long-running application where DB may result in fewer checkpoints than WB. (iii) The WB approach does not required kernel patching or kernel recompilation. (iv) The difference in performance between the WB and the DB approach increases with the amount of memory utilization within a process. (v) The advantage of DB for kernel activity could be significant for proactive fault tolerance where an immanent fault can be circumvented if DB-based live migrations moves a process away from hardware about to fail.

8 Acknowledgement

This work was supported in part by NSF grants 0937908 and 0958311 as well as by subcontract LBL-6871849 from Lawrence Berkeley National Laboratory.

References

[1] Saurabh Agarwal, Rahul Garg, Meeta S. Gupta, and Jose E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 277–286, New York, NY, USA, 2004. ACM.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[3] B. Barrett, J. M. Squyres, A. Lumsdaine, R. L. Graham, and G. Bosilca. Analysis of the component architecture overhead in Open MPI. In *European PVM/MPI Users' Group Meeting*, Sorrento, Italy, September 2005.

[4] G. Bosilca, A. Boutellier, and F. Cappello. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In *Supercomputing*, November 2002.

[5] Bouteiller Bouteiller, Franck Cappello, Thomas Herault, Krawezik Krawezik, Pierre Lemarinier, and Magniette Magniette. MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. In *Supercomputing*, 2003.

[6] J. Duell. The design and implementation of berkeley lab's linux checkpoint/restart. Tr, Lawrence Berkeley National Laboratory, 2000.

[7] Roberto Gioiosa, Jose Carlos Sancho, Song Jiang, and Fabrizio Petrini. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *Supercomputing*, 2005.

[8] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[9] Junyoung Heo, Sangho Yi, Yookun Cho, Jiman Hong, and Sung Y. Shin. Space-efficient page-level incremental checkpointing. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 1558–1562, New York, NY, USA, 2005. ACM.

[10] Joshua Hursey, Jeffrey M. Squyres, and Andrew Lumsdaine. A checkpoint and restart service specification for Open MPI. Technical report, Indiana University, Computer Science Department, 2006.

[11] John Mehnert-Spahn, Eugen Feller, and Michael Schoettner. Incremental checkpointing for grids. In *Linux Symposium*, July 2009.

[12] Ian Philp. Software failures and the road to a petaflop machine. In *HPCRI: 1st Workshop on High Performance Computing Reliability Issues, in Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA-11)*. IEEE Computer Society, 2005.

[13] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. In *Proceedings, LACSI Symposium*, October 2003.

[14] G. Stellner. CoCheck: checkpointing and process migration for MPI. In IEEE, editor, *Proceedings of IPPS '96. The 10th International Parallel Processing Symposium: Honolulu, HI, USA, 15–19 April 1996*, pages 526–531, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. IEEE Computer Society Press.

[15] Luciano A. Stertz. Readable dirty-bits for ia64 linux. Internal requirement specification, Hewlett-Packard, 2003.

[16] C. Wang, F. Mueller, C. Engelmann, and S. Scott. Hybrid full/incremental checkpoint/restart for mpi jobs in hpc environments. In *International Conference on Parallel and Distributed Systems*, December 2011.

[17] Sangho Yi, Junyoung Heo, Yookun Cho, and Jiman Hong. Adaptive page-level incremental checkpointing based on expected recovery time. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1472–1476, New York, NY, USA, 2006. ACM.

User-level scheduling on NUMA multicore systems under Linux

Sergey Blagodurov
Simon Fraser University
sergey_blagodurov@sfu.ca

Alexandra Fedorova
Simon Fraser University
alexandra_fedorova@sfu.ca

Abstract

The problem of scheduling on multicore systems remains one of the hottest and the most challenging topics in systems research. Introduction of non-uniform memory access (NUMA) multicore architectures further complicates this problem, as on NUMA systems the scheduler needs not only consider the placement of threads on cores, but also the placement of memory. Hardware performance counters and hardware-supported instruction sampling, available on major CPU models, can help tackle the scheduling problem as they provide a wide variety of potentially useful information characterizing system behavior. The challenge, however, is to determine what information from counters is most useful for scheduling and how to properly obtain it on user level.

In this paper we provide a brief overview of user-level scheduling techniques in Linux, discuss the types of hardware counter information that is most useful for scheduling, and demonstrate how this information can be used in an online user-level scheduler. The Clavis scheduler, created as a result of this research , is released as an open source project.

1 Introduction

In the era of increasingly multicore systems, memory hierarchy is adopting non-uniform distributed architectures. NUMA systems, which have better scalability potential than their UMA counterparts, have several memory nodes distributed across the system. Every node is physically adjacent to a subset of cores, but physical address space of all nodes is globally visible, so cores can access memory in a local as well as remote nodes. Therefore, the time it takes to access data is not uniform and varies depending on the physical location of the data. If a core sources data from a remote node, performance may suffer because of remote latency overhead and delays resulting from interconnect contention, which occurs if lots of cores access large amounts of data remotely [11]. These overheads can be mitigated if the system takes care to co-locate the thread with its data as often as possible [11, 24, 15, 23, 27, 9, 17, 22]. This can be accomplished via NUMA-aware scheduling algorithms.

Recent introduction of multicore NUMA machines into High-performance computing (HPC) clusters also raised the question whether the necessary scheduling decisions can be made at user-level, as cluster schedulers are typically implemented at user level [25, 6]. User-level control of thread and memory placement is also useful for parallel programming runtime libraries [10, 20, 14, 16], which are subject to renewed attention because of proliferation of multicore processors.

The Clavis user level scheduler that we present in this paper is a result of research reflected in several conference and journal publications [11, 12, 28, 29]. It is released as an open source [3]. Clavis can support various scheduling algorithms under Linux operating system running on multicore and NUMA machines. It is written in C so as to ease the integration with the default OS scheduling facilities, if desired.

The rest of this paper is organized as follows: Section 2 provides an overview of NUMA-related Linux scheduling techniques for both threads and memory. Section 3 describes the essential features that have to be provided by an OS for a user level scheduler to be functional, along with the ways to obtain them in Linux. Section 4 demonstrates how hardware performance counters and instruction-based sampling can be used to dynamically monitor the system workload at user level. Section 5 introduces Clavis, which is built on top of these scheduling and monitoring facilities.

2 Default Linux scheduling on NUMA systems

Linux uses the principle *local node first* when allocating memory for the running thread.[1] When a thread is migrated to a new node, that node will receive newly allocated memory of the thread (even if earlier allocations resulted on a different node). Figure 1 illustrates Linux memory allocation strategy for two applications from SPEC CPU 2006 suite: *gcc* and *milc*. Both applications were initially spawned at one of the cores local to memory node 0 of a two-node NUMA system (AMD Opteron 2350 Barcelona), and then in the middle of the execution were migrated to the core local to the remote memory node 1.

It is interesting to note in Figure 1 that the size of thread's memory on the old node remains constant after migration. This illustrates that Linux does not migrate the memory along with the thread. Remote memory access latency, in this case, results in performance degradation: 19% for *milc* and 12% for *gcc*. While *gcc* allocates and uses memory on the new node after migration (as evident from the figure), *milc* relies exclusively on the memory allocated before migration (and left on the remote node). That is why, *milc* suffers more from being placed away from its memory.

Linux Completely Fair Scheduler (CFS) tries to compensate for the lack of memory migration by reducing the number of thread migrations across nodes. This is implemented via the abstraction of *scheduling domains*: a distinct scheduling domain is associated with every memory node on the system. The frequency of thread migration across domains is controlled by masking certain events that typically cause migrations, such as context switches [13, 5, 4]. With scheduling domains in place, the system reduces the number of inter-domain migrations, favouring migrations within a domain.

Thread affinity to its local scheduling domain does improve memory locality, but could result in poor load balance and performance overhead. Furthermore, memory-intensive applications (those that issue many requests to DRAM) could end up on the same node, which results in contention for that node's memory controller and the associated last-level caches. Ideally, we need to: (a) identify memory intensive threads, (b) spread them across memory domains, and (c) migrate memory along with the threads. Performance benefits of this scheduling policy were shown in previous work [11, 12, 28].

[1]From now on we assume 2.6.29 kernel, unless it is explicitly stated otherwise.

Section 3 describes how to obtain the necessary information to enforce these scheduling rules on user level. Section 4 shows how to identify memory intensive threads using hardware performance counters. Section 5 puts it all together and presents the user-level scheduling application.

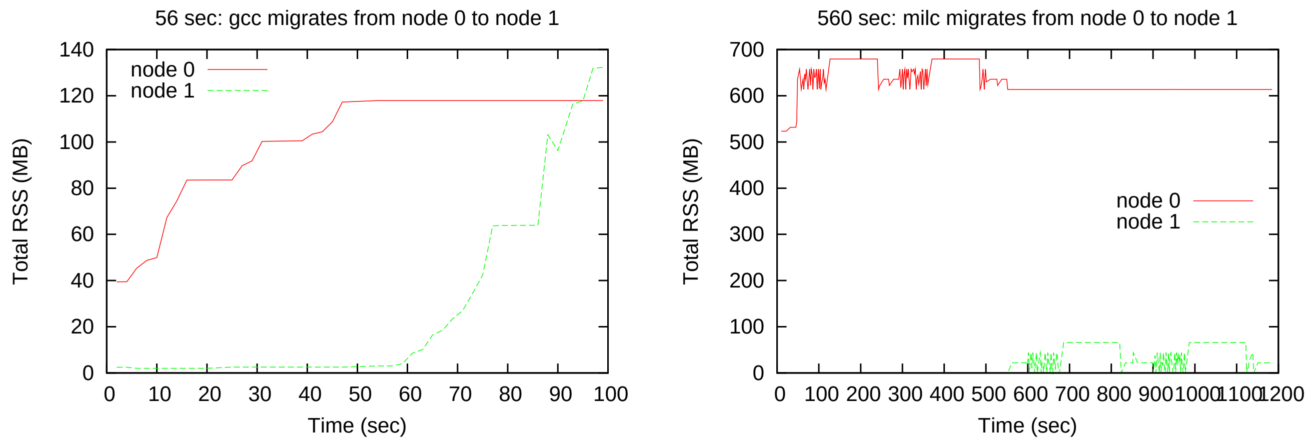
3 User-level scheduling and migration techniques under Linux

Linux OS provides rich opportunities for scheduling at user level. Information about the state of the system and the workload, necessary to make a scheduling decision, can be accessed via *sysfs* and *procfs*. Overall, scheduling features available at user level can be separated into two categories: those that provide information for making the right decision – we call them *monitoring features*, and those that let us enforce this decision – *action features*. Monitoring features provide relevant information about the hardware, such as the number of cores, NUMA nodes, etc. They also help identify threads that show high activity levels (e.g., CPU utilization, I/O traffic) and for which user level scheduling actually matters. Table 1 summarizes monitoring features and presents ways to implement them at user level. Action features provide mechanisms for binding threads to cores and migrating memory. They are summarized in Table 2.

As can be seen from the tables, many features are implemented via system calls and command-line tools (for example, binding threads can be performed via `sched_setaffinity` call or `taskset` tool). Using system calls in a user level scheduler is a preferred option: unlike command-line tools they do not require spawning a separate process and thus incur less overhead and do not trigger recycling of PIDs. Some command line tools, however, have a special *batch mode*, where a single instantiation remains alive until it is explicitly terminated and its output is periodically redirected to a file or to `stdout`. In Clavis, we only use system calls and command-line tools in batch mode.

4 Monitoring hardware performance counters

Performance counters are special hardware registers available on most modern CPUs as part of Performance



(a) gcc (b) milc

Figure 1: Memory allocation on Linux when the thread is migrated in the middle of its execution and then stays on the core it has migrated on. New memory is always allocated on the new node, old memory stays where it was allocated.

Monitoring Unit (PMU). These registers obtain the information about certain types of hardware events, such as retired instructions, cache misses, bus transactions, etc. PMU models from Intel and AMD offer hundreds of possible events to monitor covering many aspects of microarchitecture's behaviour. Hardware performance counters can track these events without slowing down the kernel or applications. They also do not require the profiled software to be modified or recompiled [18, 19].

On modern AMD and Intel processors, the PMU offers two modes in which profiling can be performed. In the first mode, the PMU is configured to profile only a small set of particular events, but for many retired instructions.[2] This mode of profiling is useful for obtaining *a high level* profiling data about the program execution. For example, a PMU configured in this mode is able to track the last level cache (LLC) miss rate and trigger an interrupt when a threshold number of events have occurred (Section 4.1). This mode, however, does not allow us to find out which particular instruction caused a cache miss.

In the second mode, the PMU works in the opposite way: it obtains detailed information about retired instructions, but the number of profiled instructions is very small. The instruction sampling rate is determined by a sampling period, which is expressed in cycles and can be controlled by end-users. On AMD processors with Instruction-Based Sampling (IBS), execution of one instruction is monitored as it progresses through the processor pipeline. As a result, various information about it becomes available, such as instruction type, logical and physical addresses of the data access, whether it missed in the cache, the latency of servicing the miss, etc. [19, 2] In Section 4.2 we provide an example of using IBS to obtain logical addresses of the tagged load or store operations. These addresses can then be used by the scheduler to migrate recently accessed memory pages after migration of a thread [11]. On Intel CPUs similar capabilities are available via Precise Event-Based Sampling (PEBS).

4.1 Monitoring the LLC miss rate online

As an example of using hardware performance counters to monitor particular hardware events online, we will show how to track LLC miss rate per core on an AMD Opteron systems. Previous research showed that LLC miss rate is a good metric to identify memory intensive threads [21, 11, 12, 28]. Threads with a high LLC miss rate will perform frequent memory requests (hence the term memory-intensive) and so their performance will strongly depend on the memory subsystem. LLC misses have a latency of hundreds of cycles, but can take even longer if the necessary data is located on

[2]The exact number of events that can be tracked in parallel depends on available counter registers inside the PMU and usually varies between one and four. A special monitoring software like `perf` or `pfmon`, however, can monitor more events than there are actual physical registers via event multiplexing.

Monitoring feature	*Description, how to get on user level*
Information about core layout and memory hierarchy of the machine (which caches are shared, cache size, etc)	Directories with the necessary files for each core on the system are located at /sys/devices/system/cpu/, including: ./cpu<ID>/cpufreq/cpuinfo_cur_freq - current frequency of the given core. ./cpu<ID>/cache/index<CID>/shared_cpu_list - cores that share a <CID>-th level cache with the given core. ./cpu<ID>/cache/index<CID>/size - cache size.
Information about which cores share every NUMA memory node on the system	Can be obtained via sysfs by parsing the contents of /sys/devices/system/node/node<NID>/cpulist The same information is also available with the `numactl` package by issuing `numactl --hardware` from the command line.
Information about which core id the given thread is currently running on	The latest data is stored in the 39-th column of the /proc/<PID>/task/<TID>/stat file, available via proc pseudo fs.
Detection of multi-threaded applications	For the purpose of scheduling, it is often necessary to identify, which threads belong to the same multithreaded application. All threads of such program share a single memory footprint and often benefit from co-scheduling together on the same shared cache or memory node. In Linux, threads are mostly treated as separate processes. To determine, which of them belong to the same application, the scheduler can read /proc/<PID>/task/<TID>/status file, which contains TGID field common to all the threads of the same application. The thread for which TID = PID = TGID is the main thread of the application. If it terminates, all the rest of the threads are usually terminated with it.
The amount of memory stored on each NUMA memory node for the given application	The file /proc/<PID>/numa_maps contains the node breakdown information for each memory range assigned to the application in number of memory pages (4K per page). In case of multithreaded programs, the same information can also be obtained from /proc/<PID>/task/<TID>/numa_maps.
Detection of compute bound threads	These are the threads that consume a significant portion of machine's computational resources (more than 30% of a core usage in our implementation). The threads can be detected by measuring the number of *jiffies* (a jiffy is the duration of one tick of the system timer interrupt) during which the given thread was scheduled in user or kernel mode. This information can be obtained via /proc/<PID>/task/<TID>/stat file (columns 13th and 14th). The `top` command-line tool provides similar data, if invoked with -H option that shows per-thread statistics (not aggregated for the entire multithreaded application) and if its "K" field is enabled.
Detection of I/O bound threads	These threads spend a significant portion of their execution time waiting for the data from the storage to process. The `iotop` command-line tool provides the information about read and write traffic from hard drive per specified interval of time for every such thread on the system.
Detection of network bound threads	Just like I/O bound, these threads are often waiting for the data, this time from the network. The `nethogs` command-line tool is able to monitor the traffic on the given network interface and break it down per process.
Detection of memory intensive threads	Refer to Section 4.

Table 1: Scheduling features for monitoring as seen from user level.

Action feature	*Description, how to get on user level*
Thread binding	To periodically rebind the workload threads, user level scheduler can use `sched_setaffinity` system call that takes cpu mask and rebinds the given thread to the cores from the mask. The thread will then run only on those cores as is determined by the default kernel scheduler (CFS). The same action can be performed by the `taskset` command line tool.
Specifying memory policy per thread	Detailed description is provided in the Linux Symposium paper by Bligh et al. also devoted to running Linux on NUMA systems [13].
Memory migration	Memory of the application can be migrated between the nodes in several ways: A *coarse-grained migration* is available via `numa_migrate_pages` system call or `migratepages` command line tool. When used, they migrate *all* pages of the application with the given PID from old-nodes to new-nodes (these 2 parameters are specified during invocation). *Fine-grained migration* can be performed with `numa_move_pages` system call. This call allows to specify logical addresses of the pages that have to be moved. The feature is useful if the scheduler is able to detect what pages among those located on the given node are "hot" (will be used by the thread after its migration to the remote node). *Automatic page migration*. Linux kernel since 2.6.12 supports the *cpusets* mechanism and its ability to migrate the memory of the applications confined to the cpuset along with their threads to the new nodes if the parameters of a cpuset change. Schermerhorn et al. further extended the cpuset functionality by adding an automatic page migration mechanism to it: if enabled, it migrates the memory of a thread within the cpuset nodes whenever the thread migrates to a core adjacent to a different node. The automatic memory migration can be either coarse-grained or fine-grained, depending on configuration [26]. Automigration feature requires kernel modification (it is implemented as a collection of kernel patches).

Table 2: Scheduling features for taking action as seen from user level.

the remote memory node. Accessing remote memory node requires traversing the cross-chip interconnect, and so LLC-miss latency would increase even further if the interconnect has high traffic. As a result, an application with higher LLC miss rate could suffer higher performance overhead on NUMA systems than an application which does not access memory often.

Many tools to gather hardware performance counter data are available for Linux, including `oprofile`, `likwid`, `PAPI`, etc. In this paper we focus on two tools that we use in our research: `perf` and `pfmon`. The choice of a tool depends on the Linux kernel version. For Linux kernels prior to 2.6.30, `pfmon` [19] is probably the best choice as it supports all the features essential for user level scheduling, including detailed description of a processor's PMU capabilities (what events are available for tracking, the masks to use with each event, etc), counter multiplexing and periodic output of intermediate counter events (necessary for online monitoring). `Pfmon` requires patching the kernel in order for the user level tool to work. The support for `pfmon` was discontinued since 2.6.30 in favour of the vanilla kernel profiling interface PERF_EVENTS and a user-level tool called `perf` [18]. `Perf` generally supports the same functionality as `pfmon` (apart from a periodic output of intermediate counter data, which we added). PERF_EVENTS must be turned on during kernel compilation for this tool to work.

The server we used has two AMD Opteron 2435 Istanbul CPUs, running at 2.6 GHz, each with six cores (12 total CPU cores). It is a NUMA system: each CPU has an associated 8 GB memory block, for a total of 16 GB main memory. Each CPU has 6 MB 48-way L3 cache shared by six cores. Each core also has a private unified 512 KB 16-way L2 cache and a private 64 KB 2-way L1 instruction and data caches. The client machine was configured with a single 76 GB SCSI hard drive. To track the LLC miss rate (number of LLC misses per instruction), the user-level scheduler must perform the following steps:

1) Get the layout of core IDs spread among the nodes of the server. On a two socket machine with 6 core AMD Opteron 2435 processors, the core-related output of `numactl` would look like:

```
# numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 2 4 6 8 10
node 1 cpus: 1 3 5 7 9 11
```

2) Get the information about L3_CACHE_MISSES and RETIRED_INSTRUCTIONS events provided by the given PMU model (the event names can be obtained via `pfmon -L`):

```
# pfmon -i L3_CACHE_MISSES
Code      : 0x4e1
Counters : [ 0 1 2 3 ]
Desc      : L3 Cache Misses
Umask-00 : 0x01 : [READ_BLOCK_EXCLUSIVE] :
Read Block Exclusive (Data cache read)
Umask-01 : 0x02 : [READ_BLOCK_SHARED] :
Read Block Shared (Instruction cache read)
Umask-02 : 0x04 : [READ_BLOCK_MODIFY] :
Read Block Modify
Umask-03 : 0x00 : [CORE_0_SELECT] :
Core 0 Select
Umask-04 : 0x10 : [CORE_1_SELECT] :
Core 1 Select
<...>
Umask-08 : 0x50 : [CORE_5_SELECT] :
Core 5 Select
Umask-09 : 0xf0 : [ANY_CORE] :
Any core
Umask-10 : 0xf7 : [ALL] :
All sub-events selected

# pfmon -i RETIRED_INSTRUCTIONS
Code      : 0xc0
Counters : [ 0 1 2 3 ]
Desc      : Retired Instructions
```

As seen from the output, L3_CACHE_MISSES has a user mask to configure. The high 4 bits of the mask byte specify the monitored core, while the lower ones tell `pfmon` what events to profile. We would like to collect all types of misses for the given core. Hence, all three meaningful low bits should be set. We will configure the "core bits" as necessary, so that, for example, core 1 user mask will be 0x17.

While RETIRED_INSTRUCTIONS is a core-level event and can be tracked from every core on the system, L3_CACHE_MISSES is a Northbridge (NB), node-level event [2]. Northbridge resources, including memory controller, crossbar, HyperTransport and LLC events are shared across all cores on the node. To monitor them from user level on AMD Opteron CPUs, the profiling application must start *only one session per node from a single core on the node* (any core on the node can be chosen for that purpose). Starting more than one profiling instance per node for NB events will result in a monitoring conflict and the profiling instance will be terminated.

3) To get periodic updates on LLC misses and retired instructions for every core on the machine, the scheduler needs to start two profiling sessions on each memory node. One session will access a single core on the node (let it be core 0 for the first node and core 1 for the second) and periodically output misses for all cores on the chip and instructions for this core by accessing NB miss event and this core's instruction event. Another instance will access the rest of the cores from the node and collect retired instruction counts from the other cores. The two sessions for node 0 would then look like so:

```
pfmon --system-wide --print-interval=1000 \
--cpu-list=0 --kernel-level --user-level \
--switch-timeout=1 \
-e L3_CACHE_MISSES:0x07,L3_CACHE_MISSES:0x17,\
L3_CACHE_MISSES:0x27,L3_CACHE_MISSES:0x37 \
-e L3_CACHE_MISSES:0x47,L3_CACHE_MISSES:0x57,\
RETIRED_INSTRUCTIONS

pfmon --system-wide --print-interval=1000 \
--cpu-list=2,4,6,8,10 --kernel-level \
--user-level \
--events=RETIRED_INSTRUCTIONS
```

In the first session, there are two event sets to monitor, each beginning with `--events` keyword. The maximum number of events in each session is equal to the number of available counters inside PMU (four, according to `pfmon -i` output above). `Pfmon` will use event multiplexing to switch between the measured event sets with the frequency `--switch-timeout` milliseconds. Monitoring is performed per core as is designated by `--system-wide` option[3] in kernel and

[3]`Pfmon` and `perf` can monitor the counters in two modes: system-wide and per-thread. In per-thread mode, the user specifies a command for which the counters are monitored. When the

user level for all events. Periodic updates will be given at 1000 ms intervals.

The scheduler launches similar profiling sessions on the rest of the system nodes, but replaces the core IDs as is seen in `numactl --hardware` output.

4) At this point, scheduler has the updated information about LLC misses and instructions for every core on the system, thus it can calculate the miss rate for every core. The data collected with `perf` and `pfmon` on each node contains "LLC missrate – core" pairs that characterize the amount of memory intensiveness within each node online. In order to make a scheduling decision, we need to find out the id of the thread that is running on a given core so the pair will turn into "LLC missrate – thread ID". This can be done via *procfs* (see Table 1). While it is possible to tell which threads are executing on the same core, there is currently no way to attribute individual miss rate to every thread due to limitation of measuring NB events on user level[4]. Fortunately, we did not find this to be a show stopper in implementing the user-level scheduler: the LLC miss rate as a metric of memory intensiveness is only significant for compute bound threads, and those threads are usually responsible for most activity on the core (launching a workload with more than one compute-bound thread per core is not typical).

The above steps also apply when using `perf` instead of `pfmon` under the latest kernel versions (we used 2.6.36 kernel with `perf`). The only challenge is that `perf` only includes several basic counters (cycles, instructions retired and so on) into its symbolic interface by default. The rest of the counters, including NB events and their respective user masks have to be accessed by directly addressing a special Performance Event-Select Register (PerfEvtSeln) [1]. Below are the invocations of `perf` with raw hardware event descriptors for the two sessions on node 0:

```
perf stat -a -C 0 -d 1000 \
-e r4008307e1 -e r4008317e1 -e r4008327e1 \
-e r4008337e1 -e r4008347e1 -e r4008357e1 \
-e rc0

perf stat -a -C 2,4,6,8,10 -d 1000 -e rc0
```

As can be seen, the names or raw hardware events in `perf` begin with an "r". Bits 0-7, 32-35 of the register are dedicated to the event code. Bits 8-15 are for the user mask. Bits 16-31 are reserved with the value 0x0083. If the event code is only 1 byte long (0xC0 for RETIRED_INSTRUCTIONS), there is no need to specify the rest of the code bits and, hence, mention all the reserved bytes in between.

4.2 Obtaining logical address of a memory access with IBS

IBS is AMD's profiling mechanism that enables the processor to select a random instruction fetch or micro-op after a programmed time interval has expired and record specific performance information about the operation. The IBS mechanism is split into two modules: instruction fetch performance and instruction execution performance. Instruction fetch sampling provides information about instruction TLB and instruction cache behavior for fetched instructions. Instruction execution sampling provides information about micro-op execution behavior [2]. For the purpose of obtaining the address of the load or store operation that missed in the cache, the instruction execution module has to be used as follows:

1) First of all, the register MSRC001_1033 (IbsOpCtl, Execution Control Register) needs to be configured to turn IBS on (bit 17) and set the sampling rate (bits 15:0). According to the register mnemonic, IbsOpCtl is in MSR (Model Specific Registers) space with the 0xC0011033 offset. MSR registers can be accessed from user level in several ways: (a) through x86-defined RDMSR and WRMSR instructions, (b) through command-line tools `rdmsr` and `wrmsr` available from `msr-tools` package, (c) by reading or writing into /dev/cpu/<CID>/msr file (MSR support option must be turned on in the kernel for that)[5].

process running that command is moved to another core, the profiling tool will switch the monitored core accordingly. In the system-wide mode, the tool does not monitor a specific program, but instead tracks all the processes that execute on a specific set of CPUs. A system-wide session cannot co-exist with a per-thread session, but a system wide session can run concurrently with other system wide sessions as long as they do not monitor the same set of CPUs [8]. NB events can only be profiled in system-wide mode.

[4]We are currently working on kernel changes that will allow measuring per-thread LLC at user level.

[5]Although accessing MSR registers from user level is straightforward, they are not the only CPU registers that can be configured that way. For example, turning a memory controller prefetcher on/off can only be done via F2x11C register from PCI-defined configuration space. For that, command line tools `lspci` and `setpci` from `pciutils` package can be used under Linux [7].

2) After IBS is configured, execution sampling engine starts the counter and increments it on every cycle. When the counter overflows, IBS tags a micro-op that will be issued in the next cycle for profiling. When the micro-op is retired, IBS sets the 18th bit of IbsOpCtl to notify the software that new instruction execution data is available to read from several MSR registers, including MSRC001_1037 (IbsOpData3, Operation Data 3 Register).

3) At that point, the user level scheduler determines if the tagged operation was a load or store that missed in the cache. For that, it checks the 7th bit of IbsOpData3. If the bit was set by IBS, the data cache address in MSRC001_1038 (IbsDcLinAd, IBS Data Cache Linear Address Register) is valid and ready to be read from.

4) After the scheduler gets the linear address, it needs to clear the 18th bit of IbsOpCtl that was set during step 2, so IBS could start counting again towards the next tagged micro-operation.

5 Clavis: an online user level scheduler for Linux

Clavis is a user-level application that is designed to test efficiency of scheduling algorithms on real multicore systems[6]. It is able to monitor the workload execution online, gather all the necessary information for making a scheduling decision, pass it to the scheduling algorithm and enforce the algorithm's decision. Clavis is released as an Open Source project [3]. It has three main phases of execution:

- *Preparation.* During this phase, Clavis starts the necessary monitoring programs in batch mode (`top`, `iotop`, `nethogs`, `perf` or `pfmon`, etc.) along with the threads that periodically read and parse the output of those programs. In case the workload is predetermined, which is useful for testing, Clavis also analyzes a launch file with the workload description and places the information about the workload into its internal structures (see below).
- *Main loop.* In each scheduler iteration, Clavis monitors the workload, passes the collected information to the scheduling algorithm and enforces algorithm's decision on migrating threads across cores and migrating the memory. It also maintains various log files that can be used later to analyze each scheduling experiment. The main cycle of execution ends if any of the following events occur: the timeout for the scheduler run has been reached; all applications specified in the launch file have been executed at least *NR* times, where *NR* is a configuration parameter specified during invocation.
- *Wrap-up.* In this stage, the report about the scheduler's work and the workload is prepared and saved in the log files. The report includes average execution time of each monitored application, the total number of pages that were migrated, the configuration parameters used in this run and so on.

[6]The word *clávis* means "a key" in Latin. In the past, Clavis greatly helped us to "unlock" the pros and cons of several scheduling algorithms that we designed in the systems lab at SFU.

Clavis can either detect the applications to monitor online or the workload can be described by the user in a special launch file. In the first case, any thread on the machine with high CPU utilization (30% as seen in the `top` output), high disk read/write traffic (50 KB/sec) or high network activity (1MB/sec on any interface) will be detected and its respective process will be incorporated into scheduler's internal structures for future monitoring. All the thresholds are configurable. Alternatively, the user can create a launch file in which case the scheduler will start the applications specified in it and monitor them throughout its execution. Launch file can contain any number of records with the following syntax:

```
<label> <launch time> <invocation string>
***rundir <rundir>
***thread 0 [<CPU ID>] -or-
***numa thread 0 [<CPU ID>, <NODE ID>]
<...>
***thread N [<CPU ID>] -or-
***numa thread N [<CPU ID>, <NODE ID>]
```

Each record describes a single application, possibly multithreaded. In the record, the user can specify a label that will be assigned to the application, which will then represent the application in the final report. If no label is specified, or if the application was detected at runtime, the binary name of the executable is used as a label. The launch time of the application since the start of the scheduler is entered next. This field is ignored when Clavis was started with "random" parameter, in which case the scheduler randomizes workload start time. The invocation string and run directory for each program are mandatory fields. In case of multithreaded applications, user can specify additional parameters that will be associated with the program threads. Usually, they are core and node IDs the given thread and its memory should be pinned to. The user, however, can utilize these fields to pass any data to the devised scheduling algorithm (e.g. offline signatures for each program thread).

Clavis is a multithreaded application written in C. It has the following file structure:

- *signal-handling.c* - implementation of the scheduler's framework: monitoring, enforcing scheduling decisions and gathering info for the logs.
- *scheduler-algorithms.c* - the user defined implementation of the scheduling algorithms is located here. This file contains several examples of scheduling algorithm implementations with different complexity to start with.
- *scheduler-tools.c* - a collection of small helpful functions that are used throughout the scheduler work.
- *scheduler.h* - a single header file.

Possible modes of Clavis execution will depend on the number of implemented scheduling algorithms. Clavis supports two additional modes on top of that: (1) a simple binding of the workload to the cores and/or nodes specified in the launch file with the subsequent logging and monitoring of its execution; (2) monitoring the workload execution under the default OS scheduler. Table 3 lists the log files that Clavis maintains throughout its execution. The source code of the scheduler, samples of the log files, algorithm implementation examples and the user level tools modified to work with Clavis are available for download from [3].

6 Conclusion

In this paper we discussed facilities for implementing user-level schedulers for NUMA multicore systems available in Linux. Various information about the multicore machine layout and the workload is exported to user space and updated in a timely manner by the kernel. Programs are also allowed to change workload thread schedule and its memory placement as necessary. Hardware performance counters, available on all major processor models, are capable of providing additional profiling information without slowing down the workload under consideration. The Clavis scheduler introduced in this paper is an Open Source application written in C that leverages opportunities for user level scheduling provided by Linux to test the efficiency of scheduling algorithms on NUMA multicore systems.

References

[1] AMD64 Architecture Programmer's Manual Volume 2: System Programming. *[Online] Available:* `http://support.amd.com/us/Processor_TechDocs/24593.pdf`.

[2] BIOS and Kernel Developer's Guide (BKDG) for AMD Family 10h Processors. *[Online] Available:* `http://mirror.leaseweb.com/NetBSD/misc/cegger/hw_manuals/amd/bkdg_f10_pub_31116.pdf`.

[3] Clavis: a user level scheduler for Linux. *[Online] Available:* `http://clavis.sourceforge.net/`.

[4] Linux load balancing mechanisms. *[Online] Available:* `http://nthur.lib.nthu.edu.tw/bitstream/987654321/6898/13/432012.pdf`.

[5] Linux scheduling domains. *[Online] Available:* `http://lwn.net/Articles/80911/`.

[6] Maui scheduler administrator's guide. *[Online] Available:* `http://www.clusterresources.com/products/maui/docs/mauiadmin.shtml`.

[7] PCI/PCI Express Configuration Space Access. *[Online] Available:* `http://developer.amd.com/Assets/pci%20-%20pci%20express%20configuration%20space%20access.pdf`.

[8] Pfmon user guide. *[Online] Available:* `http://perfmon2.sourceforge.net/pfmon_usersguide.html`.

[9] VMware ESX Server 2 NUMA Support. White paper. *[Online] Available:* `http://www.vmware.com/pdf/esx2_NUMA.pdf`.

Log filename	*Content description*
scheduler.log	The main log of the scheduler, contains information messages about the changes in the workload (start/termination of the eligible programs and threads), migration of the memory to/from nodes, the information about a scheduling decision made by the algorithm along with the metrics the algorithm based its decision on, etc. The final report is also stored here upon program or scheduler termination.

mould.log	Information about what core each workload thread has spent the run on and for how long it was there. The log format:
	
	`<time mark since the start of the scheduler (in scheduler iterations)>:`
	`<program label>`
	`sAppRun[<program ID in scheduler>].aiTIDs[<thread ID in scheduler>]`
	`(\#<run number>) was at <core ID>-th core for <N> intervals`

numa.log	Contains the updated information about node location of the program's memory footprint in the format:
	
	`<time mark>: <label> sAppRun[<progID>].aiTIDs[<threadID>] (\#<run number>)`
	`<number of pages on the 0-th node>`
	`<...>`
	`<number of pages on the last node> for <N> intervals`

vector.log	This log contains the updated data about the resources consumed by each program that is detected online or launched by the scheduler. The log format is:
	
	`<time mark>: <PID> <label> CPU <core utilization in %>`
	`MISS RATE <LLC miss rate>   MEM <Memory utilization in %>`
	`TRAFFIC SNT <Network traffic sent> RCVD <Network traffic received>`
	`IO WRITE <Disk traffic wrote> READ <Disk traffic read>`

systemwide.log	The updated information about the monitored hardware counters is dumped here in every scheduling iteration.
time.log	Number of seconds every program was running along with the time it has spent on user and kernel level.

Table 3: Log files maintained by Clavis during its run.

[10] Micah J. Best, Shane Mottishaw, Craig Mustard, Mark Roth, Alexandra Fedorova, and Andrew Brownsword. Synchronization via Scheduling: Managing Shared State in Video Games. In *HotPar*, 2010.

[11] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. A Case for NUMA-Aware Contention Management on Multicore Systems. In *USENIX ATC*, 2011.

[12] Sergey Blagodurov, Sergey Zhuravlev, and Alexandra Fedorova. Contention-Aware Scheduling on Multicore Systems. *ACM Trans. Comput. Syst.*, 28, December 2010.

[13] Martin J. Bligh, Matt Dobson, Darren Hart, and Gerrit Huizenga. Linux on NUMA Systems. *[Online] Available:* `http://www.linuxinsight.com/files/ols2004/bligh-reprint.pdf`, 2004.

[14] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Journal of parallel and distributed computing*, pages 207–216, 1995.

[15] Timothy Brecht. On the Importance of Parallel

Application Placement in NUMA Multiprocessors. In *USENIX SEDMS*, 1993.

[16] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[17] Julita Corbalan, Xavier Martorell, and Jesus Labarta. Evaluation of the Memory Page Migration Influence in the System Performance: the Case of the SGI O2000. In *Proceedings of Supercomputing*, 2003.

[18] Arnaldo Carvalho de Melo. Performance counters on Linux, the new tools. *[Online] Available:* `http://linuxplumbersconf.org/2009/slides/Arnaldo-Carvalho-de-Melo-perf.pdf`.

[19] Stéphane Eranian. What Can Performance Counters Do for Memory Subsystem Analysis? In *MSPC*. ACM, 2008.

[20] Wooyoung Kim and Michael Voss. Multicore Desktop Programming with Intel Threading Building Blocks. *IEEE Softw.*, 28:23–31, January 2011.

[21] Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn. Using OS Observations to Improve Performance in Multicore Systems. *IEEE Micro*, 28(3), 2008.

[22] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. K42: Building a Complete Operating System. In *EuroSys*, 2006.

[23] Richard P. LaRowe, Jr., Carla Schlatter Ellis, and Mark A. Holliday. Evaluation of NUMA Memory Management Through Modeling and Measurements. *IEEE Transactions on Parallel and Distributed Systems*, 3, 1991.

[24] Tong Li, Dan Baumberger, David A. Koufaty, and Scott Hahn. Efficient Operating System Scheduling for Performance-Asymmetric Multi-core Architectures. In *SC*, 2007.

[25] David Jackson Quinn, David Jackson, Quinn Snell, and Mark Clement. Core Algorithms of the Maui Scheduler. In *JSSPP*, 2001.

[26] Lee T. Schermerhorn. Automatic Page Migration for Linux. *[Online] Available:* `http://lca2007.linux.org.au/talk/197.html`, 2007.

[27] David Tam, Reza Azimi, and Michael Stumm. Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors. In *EuroSys*, 2007.

[28] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing Contention on Multicore Processors via Scheduling. In *ASPLOS*, 2010.

[29] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. AKULA: a Toolset for Experimenting and Developing Thread Placement Algorithms on Multicore Systems. In *PACT*, 2010.

Management of Virtual Large-scale High-performance Computing Systems

Geoffroy Vallée
Oak Ridge National Laboratory
valleegr@ornl.gov

Thomas Naughton
Oak Ridge National Laboratory
naughtont@ornl.gov

Stephen L. Scott
Tennessee Tech University and Oak Ridge National Laboratory
sscott@tntech.edu

Abstract

Linux is widely used on high-performance computing (HPC) systems, from commodity clusters to Cray supercomputers (which run the Cray Linux Environment). These platforms primarily differ in their system configuration: some only use SSH to access compute nodes, whereas others employ full resource management systems (e.g., Torque and ALPS on Cray XT systems). Furthermore, the latest improvements in system-level virtualization techniques, such as hardware support, virtual machine migration for system resilience purposes, and reduction of virtualization overheads, enable the usage of virtual machines on HPC platforms.

Currently, tools for the management of virtual machines in the context of HPC systems are still quite basic, and often tightly coupled to the target platform. In this document, we present a new system tool for the management of virtual machines in the context of large-scale HPC systems, including a run-time system and the support for all major virtualization solutions. The proposed solution is based on two key aspects. First, Virtual System Environments (VSE), introduced in a previous study, provide a flexible method to define the software environment that will be used within virtual machines. Secondly, we propose a new system run-time for the management and deployment of VSEs on HPC systems, which supports a wide range of system configurations. For instance, this generic run-time can interact with resource managers such as Torque for the management of virtual machines.

Finally, the proposed solution provides appropriate abstractions to enable use with a variety of virtualization solutions on different Linux HPC platforms, to include Xen, KVM and the HPC oriented Palacios.

1 Introduction

Virtual machines are widely used for server virtualization aiming at consolidating physical resources and, ultimately, increase the resource usage. As a result, many tools are available for the definition, deployment, and management of virtual machines (VMs) on servers or a small set of servers. Furthermore, over the past few years, two new trends appeared and were the focus of many research efforts: the deployment of cloud infrastructures, and the deployment of "virtual clusters". In fact, it has been shown that virtual clusters are an interesting solution for high-performance computing (HPC).

Even if these computational platforms are very different in nature, they are most of time running the Linux kernel, even if the Linux distribution on top of it can be very different and/or customized. For instance, most servers in the business world are running Linux, in the context of server consolidation, many VMs are Linux based. In the context of HPC, the trend is even more clear since more than 90% of HPC systems on the Top500 list [5] are Linux based.

This document is focused on the HPC context, for which it is preferable to have a few VMs running on the nodes of the HPC platforms (typically one VM per core), rather than running many virtual machines on a given core (over-subscription). This is mainly because in the context of HPC, input/output (I/O) operations are crit-

[1]The submitted manuscript has been authored by a contractor of the U.S. Government under Contract No. DE-AC05-00OR22725. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

ical (because they are involved in communication between the execution entities of a parallel application).

To the best of our knowledge, all ongoing efforts for the design and development of tools that aims at managing a large number of VMs are focusing on the "many VMs on a few nodes" paradigm rather than the "a few VMs on many nodes" paradigm. As a result, existing solutions are not adapted for the management of many VMs on HPC systems and ultimately the execution of parallel applications within these VMs.

In this document, we present the architecture of a new system-level solution for the deployment and management of many VMs that are used for the execution of large-scale parallel applications on HPC platforms. Because the primary target of the proposed system is large-scale HPC systems, the following characteristics are critical for its design:

- **Scalable bootstrapping**: the bootstrapping on many VMs running on many nodes of HPC systems is a challenging tasks, from the staging of both the VM image and the application, to the initialization of the VMs and the launch of parallel applications within the running VMs. Based on the large number of nodes and VMs, it is necessary to implement some advanced methods for startup of VMs and applications, with linear approaches being too expensive.

- **Portability**: HPC systems may have very different hardware configurations, as well as very different software configurations. Furthermore, different virtualization solutions will most certainly be deployed on different HPC systems. As a result, the proposed solution must support all major virtualization solutions, and abstract the underlying virtualization solution away from the users (so they can easily run their applications on various virtualized HPC platforms).

- **Customization**: one of the benefits of system-level virtualization is to allow users to define their own execution environments within the VMs (typically software configuration). Tools are already available for the specification and deployment of customized environments that will perfectly match the requirements of parallel applications. The proposed tool must support such customization capabilities.

- **Fault tolerance**: because the system is composed of many distributed hardware components, the probability of a failure during the execution of a parallel application at scale increases accordingly. As a result, even if the goal of this study is not to provide fault tolerance mechanisms for parallel applications, we have to ensure that the proposed system and its overall infrastructure will tolerate failures to some extent, at least to allow users to cleanly terminate the execution of their applications. For that, it is necessary to **detect failures**, and **guarantee communications** even in the context of link failures.

To address these challenges, the proposed solution is based on three different abstractions:

1. a set of tools and methods for the specification and instantiation of customized execution environments,

2. a control infrastructure that will be used for the management of VMs (startup, monitoring, termination), as well as the control of the execution of the parallel application; this "run-time" system must be scalable and fault tolerant,

3. a tool for the abstraction of the underlying virtualization solution so the user will not have to deal with technical details specific to the virtualization solution deployed on a given HPC system, which means that this abstraction must support all major virtualization solutions.

The remainder of this document is organized as follows: Section 2 presents how users can specify a customized execution environment for their applications that will be used for the deployment of VMs on HPC systems. Section 3 presents the control infrastructure used to control and orchestrate many VMs used in the context of the execution of a large-scale parallel applications. Section 4 presents the abstraction layer used to allow the user to implicitly switch between different virtualization solutions. Finally, Section 6 concludes.

2 Customization of Execution Environments

Based on the 36th edition of the Top500 list (September 2010), 91% of the 500 most powerful HPC systems are based on Linux. However, the software configuration of these systems vary greatly, from customized kernels and Linux distributions to out-of-the-box Linux distributions and the kernel they provide by default. Furthermore, each of the HPC systems have a well-defined set of available software, including scientific libraries and tools. So far, the users had to modify their application to fit the configuration of the target HPC platforms, leading to wasted resources and redundant effort (scientists should focus on the science gathered in their applications and not on modifications because of technical details of the HPC system).

An approach to address this challenge is to allow the users to define their execution environments based on the requirements of their applications. In a previous work, we introduced the concept of *Virtual System Environment* (VSE) [6] that enables the description of the software requirements of a given applications. We also proposed a set of tools for the instantiation of a VSE on a given HPC platform.

As a result, it is possible to specify the "static" requirements of the scientific application; requirements that are not specific to a given run of the application on a given platform. For instance, the user can specify requirements such as the Linux distribution to be used (e.g., Red Hat Enterprise Linux), the version of the Linux kernel, a set of scientific libraries. The specification is translated into terms of "packages" available from repositories. During the instantiation of a VSE on a given HPC platform, the list of packages is used to create a new image, which can then be used to setup a VM. Note that the tools associated with the VSE ensure that the image can be deployed independently of the virtualization solution that is ultimately used. For instance, the users do not have to know whether KVM or Xen will be used as the virtualization solution, the provided tools create a VM image that is agnostic to the different virtualization solutions. Note that the tool that abstracts the virtualization solution (presented in Section 4) ensures that the image is correctly "loaded" based on the target virtualization solution.

3 Control Infrastructure

The previous section presents how a user can customize the execution environment for their applications. This task can typically be done off-line and is independent from the target HPC system. Once on the HPC systems, the users must deploy the required VMs and start the application execution. A typical way to see the execution of a parallel application is the concept of *job*: a job is the combination of the application and an allocation for its execution (typically a set of nodes). Unfortunately, HPC systems can be used with very different configurations: some provide tools that assign an allocation to a given job (based on the number of requested nodes, the *job manager* allocates nodes to the job); while some other systems allow direct access to compute nodes (e.g., via SSH). This heterogeneity directly impacts how VMs will be deployed.

Furthermore, because we target large-scale HPC systems, it is not efficient to setup VMs and start the application execution in a linear fashion, more advanced startup methods are required.

Finally, even if failures occur during the execution of a parallel application running within VMs, we must continue to keep control on the running VMs. In the context of this study, our goal is not to provide fault tolerance capabilities for the VMs or even for the application, but to guarantee that even if compute nodes or VMs fail, it will still be possible to control remaining VMs and let the user decide the best solution (e.g., cleanly terminate VMs that are still alive and therefore, terminate the job).

3.1 Architecture Overview

In order to deploy VMs on compute nodes and control the execution of applications within these VMs, we need to have control on each compute node of the job allocation. Furthermore, in order to separate the system aspects (such as resource allocation) from the job management, the proposed architecture is based on the concept of agents, and five different types of agents have been defined: root agents (typically system agents), session agents (specific to a job), and tool agents (specific to a "tool", a tool being a self-contained part of a job, e.g., one of the binaries of a job when the parallel application is composed of different sub-applications).

- **Root agent**: agent in charge of resource allocation and release. Thus, this agent is a privileged agent. Only one root agent is on each compute node and is used to deploy other agents (both session and tool agents). Root agents are not specific to a job.

- **Session agent**: agent in charge of instantiating a job on allocated compute nodes. This is not a privileged agent and it acts on behalf of a user. A single session agent is deployed on compute nodes of a given job allocation.

- **Tool agent**: agent that instantiates the job itself; multiple tool agents can be deployed on compute nodes of a job allocation, and all tool agents act on behalf of the users. In the context of this paper, the tool agents are used to manage VMs. For that, we developed a specific tool agent that can be used to drive the tool that implements the abstraction of the underlying virtualization solution (presented in Section 4). For instance, the dedicated tool agent can instantiate a VM, pause it, or terminate it.

- **Controller agent**: agent in charge of creating an internal representation of a job and of coordinating the deployment of the different agents and the creation of communication channels between the agents. The communication channels are organized based on *topologies* (e.g., trees, meshes) that describe how the controllers, the root agents, the session agents, and the tool agents can communicate. In this example, root agents are running on different compute nodes, and both session and tool agents, that are children of a given root agent, actually run on the same compute nodes. Figure 1 presents an example of a tree-based topology. Topologies are also used to set routing tables up (which are then used to send messages from one agent to another), and to stage files, including the VM image.

- **Front-end agent**: agent that runs on the user's machine or on the HPC system login node. The front-end provides a MPI-like user interface to submit a job where the user specifies the VSE specification file and the number of nodes required.

3.2 Scalable Bootstrapping

To efficiently startup agents, we define a boot topology (the initial implementation is based on a binary tree but any k-ary tree could be used). This tree allows us to start the different agents in parallel, and provides good scalability. This approach is used in various HPC specific run-times and has proven to be efficient.

If failures occur during the bootstrapping phase, the different agents are designed and implemented to automatically terminate. This is implemented using a handshake mechanism with the agent's parent within the bootstrapping topology, as well as timers. Typically, if the handshake does not succeed within a window of time, we assume a failure and the agent terminates. On the other hand, if the handshake succeeds, the bootstrapping phase is assumed successful; the agent's state switch to running, and the parent assumes that the agent is running and reachable. As such, the failure detection is then in charge of detecting and reporting agent failures.

3.3 Fault Tolerance

For fault tolerance purposes, we provide two capabilities: failure detection and a fault tolerant topology. These two capabilities ensure that even if a node fails or if an agent fails, it will still be able to send/receive messages between agents that are still alive. This allows the user to decide the best policy to apply in the context of failure, for instance, triggering the clean termination of remaining agents and ultimately VMs.

3.3.1 Fault Detection

A key point to tolerate failures is to first detect failures. By detecting failures, it is possible to update routing tables and eventually re-establish failed communication channels to ensure that we can still control live agents. For this context, we propose a set of *detectors*. For instance, a mesh-based detector establishes connections between root agents and reports an error if the connection is closed. Another detector establishes connections between root agents based on a mesh topology and perform periodic ping-pong probes. If the ping-pong fails, a failure is reported. Finally, we provide a signal-based detector that can be used on compute nodes to detect the failure of any local session or tool agent (by catching the SIGCHLD signal).

3.3.2 Communication Fault Tolerance

Since our boot topology is a tree-based topology, the failure of any agent will prevent communications between different parts of the tree, leading to unreachable agents. To address this issue, we setup a topology based

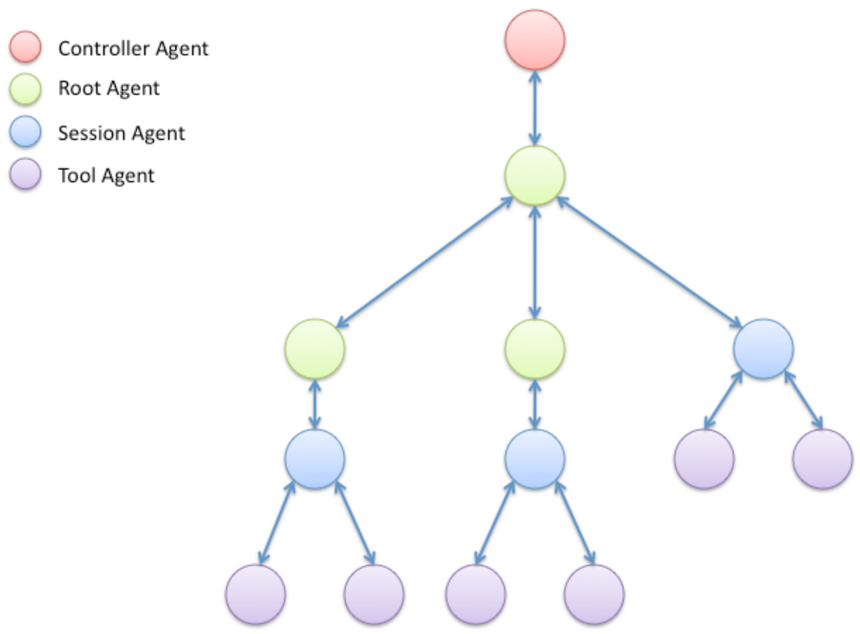


Figure 1: Example of Topology

on a binomial graph (BMG) [1] that provides redundant communication links between agents. As a result, even if a communication channel is closed because of a failure, it is possible to find another route to reach the destination.

4 Abstraction of the Underlying Virtualization Solution

Many tools are available for the management of virtual machines, such as libvirt [3], However, these tools, to the best of our knowledge, try to represent the union of all the capabilities of all the virtualization solutions, leading to overly complex tools. In the context of our study, we only require a lightweight tool that provides a simple API, typically start, stop, pause, un-pause a given VM (we may support migration in a future version of the system). For that, we propose the V2M tool from a previous study [7]. This tool abstracts the underlying virtualization solution via the implementation of plug-ins. Each plug-in is in charge of translating management tasks to commands that are specific to the underlying virtualization solution. The tool is also in charge of making sure that the VM image is correctly setup to be used with the target virtualization solution.

V2M is based on the concept of *profiles*, which specify how to deploy a VM based on VSE image. This profile is automatically created based on job data such as the allocation specification.

5 Use Case: the Palacios Virtualization Solution

Palacios is a virtualization solution specifically designed for HPC [2, 4]. For that, Palacios is focusing on minimizing its resource footprint and optimizing I/O (since efficient I/O is critical for HPC applications). Figure 2 presents an overview of the architecture for Palacios.

Palacios proved to be very scalable and is therefore a good candidate for experimentation at scale. In other terms, by selecting Palacios, we can setup an experimental configuration that is scalable and fault tolerant.

To support Palacios, a new V2M plug-in is created in order to interface with our infrastructure; no other modifications or extensions are required.

6 Conclusion

In this document we present the architecture for a new system-level infrastructure for the management of many virtual machines to support the execution of parallel applications on large-scale high-performance computing

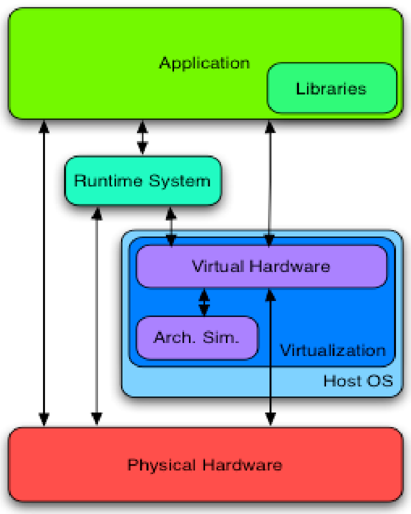


Figure 2: Overview of the Architecture of the Palacios Virtualization Solution

systems. The proposed architecture focuses on scalability and fault tolerance. Furthermore, the proposed solution abstract the underlying virtualization solution and can therefore be used with most of the current virtualization solutions such as Xen or KVM. Finally, our solution enables the customization of the execution environment that is deployed inside the virtual machines, which ultimately allows scientists to focus on science rather than technical details associated with the configuration and execution of their application on elaborate high-performance computing systems.

To implement these capabilities, we propose three distinct abstractions: the concept of VSE for the customization of the execution environment; a scalable and fault tolerant control infrastructure for the coordination of the VMs running across the compute nodes, and finally an abstraction of the underlying virtualization solution.

The implementation of the proposed architecture is still ongoing but initial experimentation shows that the design of the control infrastructure is scalable and maintains connectivity between the different nodes involved in the execution of a given application even in the event of failures.

Finally, Palacios, a virtualization solution designed for high-performance computing, is able to scale to a few thousand VMs, and we are working with the Palacios development team to perform experiments at scale using our tool on some of the world's larger HPC systems.

Acknowledgments

We would like to thank the Application Performance Tools group from ORNL, and especially Richard L. Graham, for their support on the development of the control infrastructure.

References

[1] Thara Angskun, George Bosilca, and Jack Dongarra. Binomial graph: A scalable and fault-tolerant logical network topology. In *International Symposium on Parallel and Distributed Processing and Applications*, pages 471–482.

[2] John Lange, Kevin Pedretti, Trammell Hudson, Peter Dinda, Zheng Cui, Lei Xia, Patrick Bridges, Steven Jaconette, Mike Levenhagen, Ron Brightwell, and Patrick Widener. Palacios and kitten: High performance operating systems for scalable virtualized and native supercomputing.

[3] The virtualization api. `http://libvirt.org/`.

[4] Palacios – an os independent embeddable vmm. `http://v3vee.org/palacios/`.

[5] Top 500 supercomputer sites. `http://top500.org/`.

[6] Geoffroy Vallée, Thomas Naughton, Hong Ong, Anand Tikotekar, Christian Engelmann, Wesley Bland, Ferrol

Aderholdt, and Stephen L. Scott. Virtual system environments. In *Systems and Virtualization Management. Standards and New Technologies*, volume 18 of *Communications in Computer and Information Science*, pages 72–83. Springer Berlin Heidelberg, October 21-22, 2008.

[7] Geoffroy Vallée, Thomas Naughton, and Stephen L. Scott. System management software for virtual environments. In *Proceedings of ACM Conference on Computing Frontiers 2007*, Ischia, Italy, May 7-9, 2007.

The Easy-Portable Method of Illegal Memory Access Errors Detection for Embedded Computing Systems

Ekaterina Gorelkina
SRC Moscow, Samsung Electronics
e.gorelkina@samsung.com

Sergey Grekhov
SRC Moscow, Samsung Electronics
grekhov.s@samsung.com

Alexey Gerenkov
SRC Moscow, Samsung Electronics
a.gerenkov@samsung.com

Abstract

Nowadays applications on embedded systems become more and more complex and require more effective facilities for debugging, particularly, for detecting memory access errors. Existing tools usually have strong dependence on the architecture of processors that makes its usage difficult due to big variety of types of CPUs. In this paper an easy-portable solution of problem of heap memory overflow errors detection is suggested. The proposed technique uses substitution of standard allocation functions for creating additional memory regions (so called *red zones*) for detecting overflows and intercepting of page faulting mechanism for tracking memory accesses. Tests have shown that this approach allows detecting illegal memory access errors in heap with sufficient precision. Besides, it has a small processor-dependent part that makes this method easy-portable for embedded systems which have big variety of types of processors.

1 Introduction

Currently software programs running on modern embedded computing systems have quite complicated behavior in sense of memory manipulation that make the process of debugging very time consuming. In order to simplify debugging procedure, various helper utilities were designed. The most known are Valgrind [1] and using `MALLOC_CHECK_` environment variable for GNU C library [2].

Valgrind replaces the standard C memory allocator with a custom implementation and inserts extra instrumentation code around almost all instructions, which enables to detect read and write errors when a program access memory outside of an allocated block by a small amount. This approach has a strong dependency on the type of processor of computing system. This makes usage of Valgrind on different computing systems quite difficult due to relatively complicated porting procedure.

Recent versions of GNU C library are tunable via environment variables. Setting variable `MALLOC_CHECK_` to values 1, 2 or 3 allows detecting simple memory errors like double free or overrun of a single byte. The drawback of this approach is low precision of error detection and performance degradation which is caused by using less efficient implementations of allocating functions.

In contrast to [1] and [2], the proposed method solves the problem of memory errors detection with sufficient precision and has a small processor-dependent part which minimizes the time porting of this method on various processor architectures (that is especially valuable for embedded systems). It also does not require program re-compilation. As a result, this method can detect heap object overflow problem for wide range of processors of embedded computing systems. Testing results also have shown the low execution overhead of suggested technique in comparison to Valgrind tool.

2 General Description on Idea

Suggested method relates to detection of typical memory errors: heap object overflow, heap object underflow, access to un-allocated memory.

The basic idea of the method is to intercept memory allocation requests, add special region (red zone) to each heap object, protect allocated memory for reading and

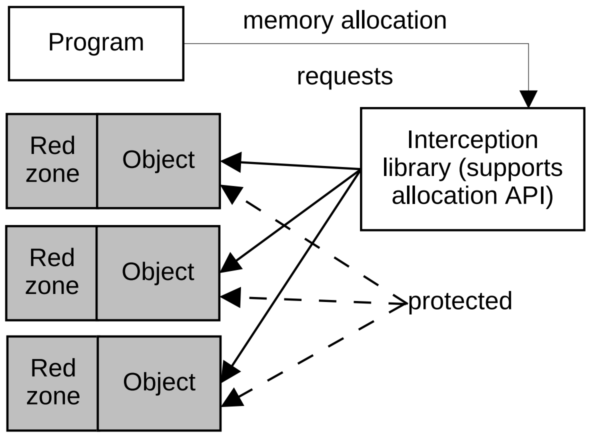


Figure 1: Allocation requests of considering program are intercepted by a special library which allocates each object with red zone and protect them both from reading and writing

writing and use mechanism of handling incorrect memory accesses in address space (here and after page faulting mechanism) in order to track accesses to the memory and detect the erroneous ones. The principal scheme of two main parts of this idea - intercepting allocation functions and intercepting page faulting mechanism - are illustrated in Figure 1 and Figure 2 respectively.

The description of Figure 1 is following. During program execution it requests and releases dynamically allocated memory. The special interception library handles these requests and processes them by making allocations, adding a special region (red zone) and protecting both of them from reading and writing. These red zones are used for detecting typical overflow errors (as it will be explained further in details).

The description of Figure 2 is following. After allocating necessary memory program tries to access some of created objects. Since all of them are protected from reading and writing, such attempt will generate page fault exception. It is a special event which occurs when specified memory can not be read or written. The parameters of this event include the address of access and the type of access (read/write). Thus, such accesses can be classified into valid and invalid by comparing the address of access with addresses of allocated heap objects.

After handling page fault exception the protection of the accessed object should be annulled and the program execution should be resumed (if memory access was valid) or stopped (if memory access was erroneous).

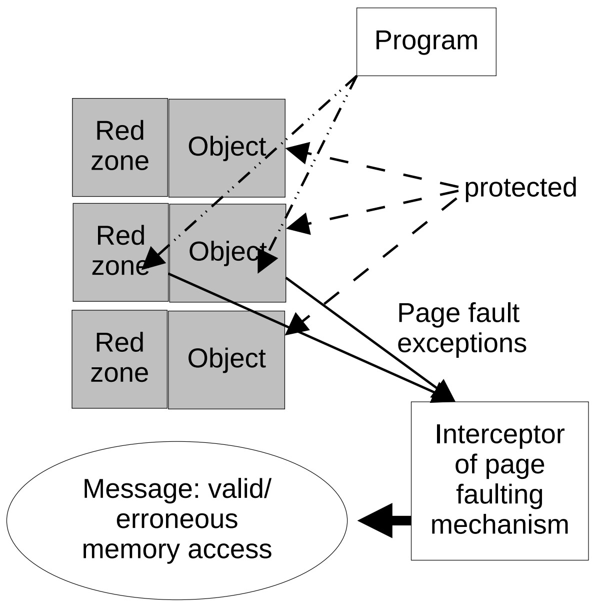


Figure 2: Considering program accesses protected objects or its red zone and generate page fault exception; analyzing parameters of this exception allows detecting overflow errors

2.1 Interception of Allocation and De-allocation Requests

As it was mentioned earlier, at run-time all memory allocation requests are intercepted by a special library. Herewith, it is allocated a memory region of requested size plus additional memory region (red zone).

Figure 3 demonstrates the difference between original and intercepted allocation request. In the first case the allocation request is processed by allocation library and object is created. In the second case the allocation request is processed by interception library. The result of processing is the requested object plus red zone (a special memory region for detecting errors). Herewith, both allocated object and red zone are protected from reading and writing.

If program uses, for example, malloc() and free() functions from standard GNU C library for allocating memory, then invocation of these function can be substituted in Linux by `special_malloc()` and `special_free()` using `LD_PRELOAD` environment variable. These special functions create/release heap objects with additionally allocated red zone and protect them from reading and writing using mprotect() function.

Since, in general case, memory can be protected only page by page, object will be allocated within one or

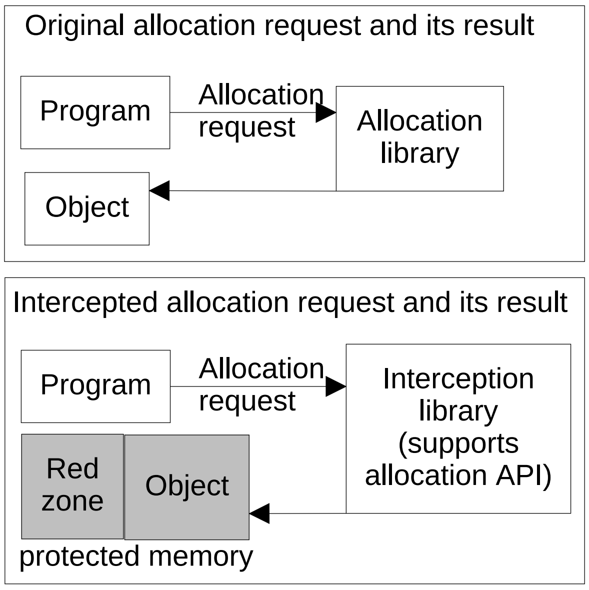

Original allocation request and its result
Program
Allocation request
Allocation library
Object
Intercepted allocation request and its result
Program
Allocation request
Interception library (supports allocation API)
Red zone
Object
protected memory

Figure 3: The difference between original allocation request and intercepted allocation request: in the second case the red zones are added to allocated objects and both regions are protected from reading and writing

more virtually continuous memory pages. It can be easily seen that when size of heap object is multiple of page size then the size of corresponding red zone will be zero (in this situation it will be impossible to detect illegal access). This issue can be resolved the following way. Let's denote the size of one memory page in computing system as P and the size of the requested allocation as S. Then N - the number of pages required for special allocation - should satisfy the two following conditions: $N*P \geq S+P/4$ and $(N-1)*P < S+P/4$. The additional component P/4 prevents the situation when S is a multiple of P and red zone size is zero. Thus, the final formula for calculating N is following:

$$N = (S+P/4)/P + sign[(S+P/4)modP] \quad (1)$$

where all mathematical operations are integer-value, *sign* means the function which return 0 is arguments is 0 or 1 if argument is greater than 0, and *modP* means function which return the residue of division of argument by P.

Another issue when processing allocation request is the alignment of returned memory. By default, the malloc() function from GNU C library returns the align address of allocated memory. This is caused by specific requirements of returned address to be suitable for any kind

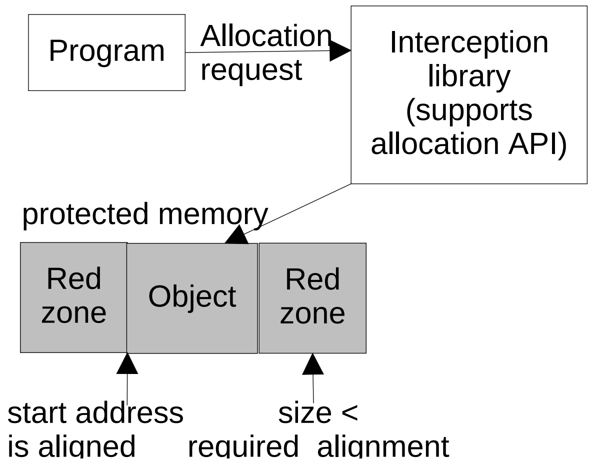


Figure 4: Due to restrictions for malloc() return value there can be two red zones: before and after specially allocated heap object

of variable. Thus, in fact, the address of each specially created heap object should be aligned (for example, to 4 bytes on ARM processor) and there will be two red zones: before and after the heap object. The size of second red zone (after heap object) varies on the size of heap object and required alignment (see Figure 4).

As it was mentioned in chapter II, heap object is unprotected after processing page fault exception and detecting a valid memory access. Moreover, the application execution is resumed. Thus, if application accessed unprotected red zone, then it is impossible to detect memory access by intercepting page faulting mechanism. The solution of this problem is following. After the allocation of red zones, they should be filled with identical values - stamps - which are used for detecting write accesses. If application wrote to red zone then most probably the stamps are changed. Checking the consistency of stamp before freeing the memory enhances the precision of overflow detection. This procedure can be implemented during the processing of intercepted of deallocation requests. Let us note that this solution can not detect read accesses to unprotected memory. This issue is unresolved within the scope of proposed method.

2.2 About Interception of Page Faulting Mechanism

Page faulting mechanism is a part of operating system. The typical approach for intercepting of its functions is using dynamic instrumentation tools. Well-known Systemtap dynamic instrumentation tool based on Kprobe [3] allows creating handlers for intercepted kernel functions, however it can not track the calls of `do_page_fault()` function because of restrictions of Kprobe. Another well-known tool LTTng [4] uses static instrumentation and, thus, requires kernel recompilation that can be quite inconvenient in case of embedded systems. Therefore authors used dynamic instrumentation tool SWAP [5] developed in Samsung Research Center in Moscow.

This tool allows at runtime obtaining arguments of interception kernel functions and performing custom actions before executing the body of intercepted function. Particularly, it is possible to intercept `do_page_fault()` function and obtain the address of accessed memory and the type of access (read o write).

Let us note that page fault exceptions occur in normal situations and are needed for valid system functioning. Therefore, the procedure of processing of accesses to protected heap objects should not damage the original handling of page faults in kernel.

The interception mechanism can be implemented as a kernel module. The information about allocated heap objects can be transferred from the interception library via /proc or /dev filesystem.

2.3 Handling Accesses to Protected Heap Objects

According to the general idea of the method, the page faulting mechanism should be intercepted for detecting access to allocated and protected heap objects, obtaining the parameters of this access (address and type) and classifying this access as valid or erroneous. As it was written in previous chapter, the `do_page_fault()` function can be intercepted and custom actions can be performed before executing its body. These actions relates to classifying of detected memory access and resuming or stopping execution of considering application depending on the classification results.

The algorithm of classification is following.

Step 1. If address of accessed memory is inside allocated heap object, then: previously unprotected object is protected (if it exists); protection of this object is annulled; program execution is resumed (see Figure **??**); otherwise go to step Step 2

Step 2. If address of accessed memory is inside red zone of allocated heap object, then protection of this object and unprotected remains unchanged, message about incorrect memory access is reported, and program execution is stopped (see Figure 6); otherwise go to step Step 3.

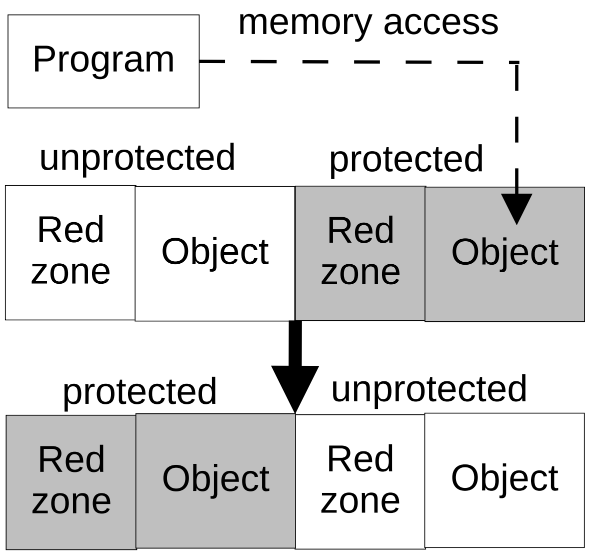


Figure 5: Step 1 of Algorithm 1: if address of accessed memory is inside a valid heap object then unprotect it and its red zones

Step 3. If address of accessed memory is outside of any allocated heap object and red zones but inside a valid allocated memory region (for example, stack, read-only data or executable code) then protection of heap objects remains the same and program execution is resumed (see Figure 7); otherwise go to step Step 4.

Step 4. If address of accessed memory is outside of any allocated heap object and corresponding red zone and any valid memory region, then error message about access to unallocated memory is created, and program execution is stopped (see Figure 8)

The error message created on Steps 2 and 4 of Algorithm 1 consists of the address of illegally accessed memory and the type of access, the title of the binary object that made illegal access, the address of instruction which caused the error (within the binary). This information can be used for creating a user-friendly report about occurred memory access problem: debug information together with mentioned content of error message allows displaying the line of the source code which produced error.

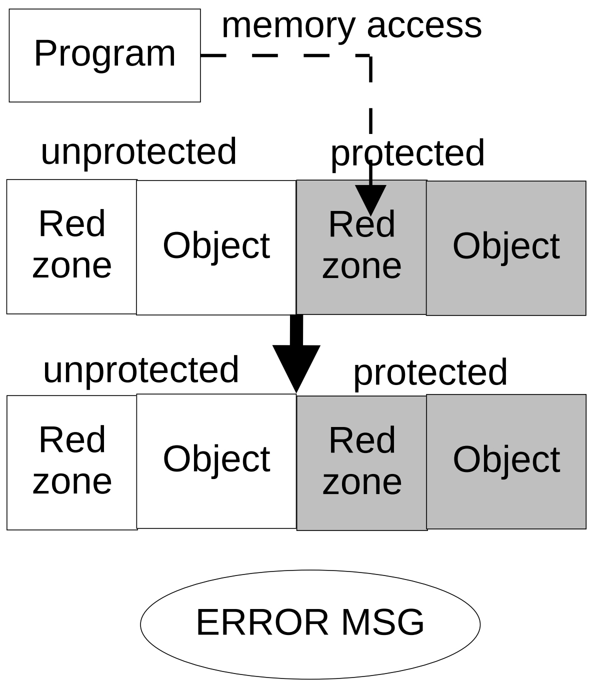


Figure 6: Step 2 of Algorithm 1: if address of accessed memory is inside a red zone then keep current objects' protection, report about error and stop execution of the program

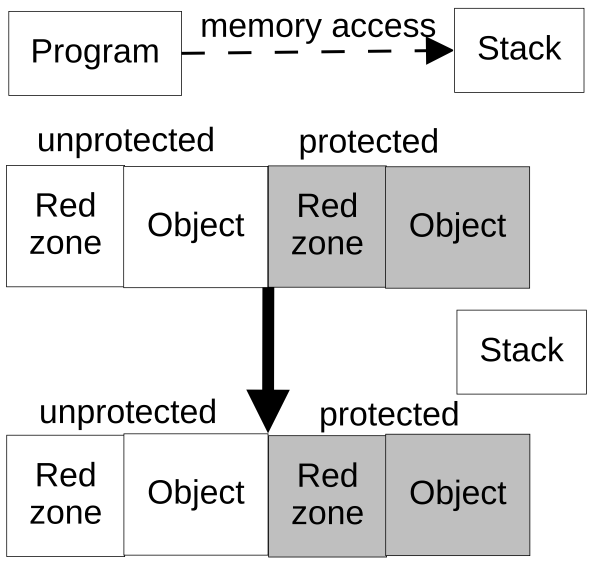


Figure 7: Step 3 of Algorithm 1: if address of accessed memory is outside of heap, but inside a valid memory region (e.g. stack) then keep current heap objects' protection and resume program execution

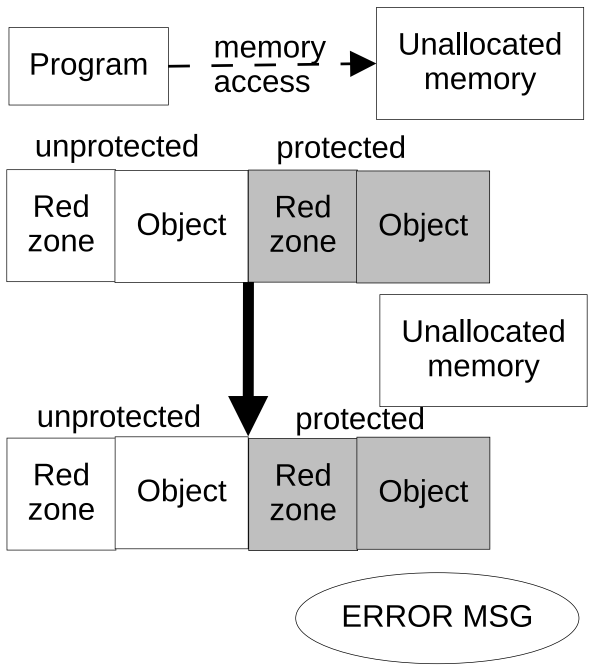


Figure 8: Step 4 of Algorithm 1: if address of accessed memory is outside of heap and any valid memory region then keep current heap objects' protection, report about error and stop program execution

3 Method Advantages and Drawbacks

The first main advantages of suggested method of memory overflow errors detection is its fast portability to various processor architectures. As it can be seen from previous chapters, the processor-dependent part of the method includes only obtaining of address of accessed memory by interception of page faulting mechanism. Thus, in comparison to the methods which use interpreting of binary instructions, the suggested technique can be easily moved across various processors. This is especially valuable for developer of embedded systems.

Since the suggested method uses dynamic instrumentation engine SWAP, the two following advantages are obtained:

- developer should not recompile erroneous application before starting the search of memory errors
- suggested method does not need debug information for the program running on embedded system.

As it can be seen from the method description, the program which produces memory errors runs on embedded system and is not changed during memory errors detection.

However, there exist several drawbacks. Firstly, method can not track accesses to the unprotected red zone with help of page faulting mechanism. Thus, read operation which accesses the unprotected red zone will be never detected. The write operation which accesses the unprotected red zone can be partially detected by using stamps. At the same time, this disadvantage has a positive aspect. Since memory accesses made within the unprotected object are not tracked, the execution of the program is going faster. This is also valuable for embedded systems.

Two more drawbacks are common for most of tools of overflow detection:

- it is impossible to detect the out-of-bounds situation when application accesses a valid memory region.
- it is impossible to detect the out-of-bounds situation within the allocated structure or class

4 Testing Results

The environment of all tests was following:

- Nvidia Tegra board, ARM-based, kernel 2.6.29
- Beagle board, ARM-based, kernel 2.6.33

During testing it was discovered that both target boards have similar behavior and produce identical results for test applications. Thus, authors provide testing results without specifying the particular environment implying that they are equal for both embedded devices.

The proposed method was tested on artificial test applications. The test applications have similar structure: three heap objects of size 4092, 4096 and 4100 respectively are allocated. During each separate test some particular out-of-bounds situations are created. For example, access the first object, and after that access the red zone of the next object. The following Table 1 provides the results of testing in comparison to Valgrind.

As it can be seen, the accuracy of suggested method is worse in comparison to Valgrind. However, it is still quite high and method can be used for debugging.

Tool	Total number of tests	Passed tests (accuracy)
Valgrind	192	144 (75%)
Proposed method	192	114 (59%)

Table 1: The accuracy of detecting errors for artificial test applications

Test environment	Time of execution (seconds/degradation degree)
Test application without memory checking tool	0.0025 / Not available
Test application with memory checking tool based on proposed method	0.466 / 186.4
Test application with memory checking tool Valgrind	2.1283 / 851.32

Table 2: Time of execution of test application with and without memory checking tools

The goal of the next test was to measure the overall degradation in performance when executing test application with memory checking tools. The test application allocates 5 arrays of 256 bytes and accesses sequentially each of them in the cycle. The average degradation of performance of test application is shown in Table 2.

It can be seen from the table, the method proposed in this article shows better performance. This advantage compensates the worse accuracy of error detection.

The following Table 3 provides information about architecture-dependent code of implementation of proposed method in comparison to Valgrind.

Tool	Total number of lines of code	Total number of architecture-dependent code (fraction)
Valgrind	> 900000	> 170000 (19 %)
Proposed method	> 180000	> 5000 (3 %)

Table 3: The amount of code dependent on the architecture of CPU.

Tool	Result of detection
Valgrind	Detected, up to specifying the line of source code which produced the error
Proposed method	Detected, up to specifying the line of source code which produced the error

Table 4: Detecting memory access error for bug in glib-2.15

It can be seen that Valgrind has bigger processor-dependent part that makes the procedure of its porting more difficult. In comparison the method proposed in the article has smaller part which depends on type of CPU, thus it is easier to be ported on embedded systems with different processors.

The last test consisted of detecting an error for the bug from glib-2.15 library bug tracking system (see [6] for more details). The erroneous behavior of library was produced when insufficient amount of memory was allocated for saving binary data encoded to base64 text form. As a result, accessing beyond the end of array caused segmentation fault exception. The error was fully detected by the proposed method with indication the line of the source code which produced the error. Table 4 shows the results of testing for Valgrind and proposed method.

Both tools produced identical results by detecting the error and specifying the line of code which produced this error.

5 Conclusion

In this paper authors proposed a method for heap overflow memory errors detection which is easy-portable on various processors' architectures of computing systems. The method shows sufficiently good results in memory overflow errors detection. Despite of some of its drawbacks, it can be efficiently used on embedded systems.

6 Acknowledgments

Authors would like to thank Jaehoon Jeong, senior engineer of System S/W Group in Samsung Electronics, and Hyunju Ahn, senior engineer of System S/W Group in Samsung Electronics, for fruitful and extensive discussion on the proposed approach and Mikhail P. Levin, the head of Advanced Software Group of Samsung Research Center in Moscow, for assistance in preparation of the article.

References

[1] Valgrind project (`http://valgrind.org/`)

[2] Malloc manual page (`http://www.kernel.org/doc/man-pages/online/pages/man3/malloc.3.html`)

[3] Systemtap tool (`http://sourceware.org/systemtap/`)

[4] Alexey A. Gerenkov, Ekaterina A. Gorelkina, Sergey S. Grekhov, Sergey Yu. Dianov, Jaehoon Jeong, Oleksiy Kokachev, Leonid V. Komkov, Sang Bae Lee, Mikhail P. Levin, System-wide analyzer of performance: Performance analysis of multi-core computing systems with limited resources, IEEE EUROCON 2009, pp. 1299-1304

[5] Mathieu Desnoyers, Michel R. Dagenais, LTTng: Filling the Gap Between Kernel Instrumentation and a Widely Usable Kernel Tracer, LFCS'2009

[6] Red Hat bug tracking system (`https://bugzilla.redhat.com/show_bug.cgi?id=474770`)

Recovering System Metrics from Kernel Trace

Francis Giraldeau
École Polytechnique de Montréal
francis.giraldeau@polymtl.ca

Julien Desfossez
École Polytechnique de Montréal
julien.desfossez@polymtl.ca

David Goulet
École Polytechnique de Montréal
david.goulet@polymtl.ca

Michel Dagenais
École Polytechnique de Montréal
michel.dagenais@polymtl.ca

Mathieu Desnoyers
EfficiOS Inc.
mathieu.desnoyers@efficios.com

Abstract

Important Linux kernel subsystems are statically instrumented with tracepoints, which enables the gathering of detailed information about a running system, such as process scheduling, system calls and memory management. Each time a tracepoint is encountered, an event is generated and can be recorded to disk for offline analysis. Kernel tracing provides system-wide instrumentation that has low performance impact, suitable for tracing online systems in order to debug hard-to-reproduce errors or analyze the performance.

Despite these benefits, a kernel trace may be difficult to analyze due to the large number of events. Moreover, trace events expose low-level behavior of the kernel that requires deep understanding of kernel internals to analyze. In many cases, the meaning of an event may depend on previous events. To get valuable information from a kernel trace, fast and reliable analysis tools are required.

In this paper, we present required trace analysis to provide familiar and meaningful metrics to system administrators and software developers, including CPU, disk, file and network usage. We present an open source prototype implementation that performs these analysis with the LTTng tracer. It leverages kernel traces for performance optimization and debugging.

1 Introduction

Tracing addresses the problem of runtime software observation. A trace is an execution log of a software, that consists essentially of an ordered list of events. An event is generated when a certain path of the code is executed, commonly called tracepoint. Each event consists of a timestamp, a type and some arbitrary payload.

Tracepoints can be embedded statically in software or dynamically inserted. Dynamic tracing allows custom tracepoints to be defined without source code modification. While this approach is flexible, static tracepoints are generally faster. In addition, tracing can be performed at the kernel and user-space level. In this paper, we focus on the static instrumentation of the Linux kernel provided by the Linux Trace Toolkit next generation (LTTng) [1]. Unlike a debugger, tracing a program does not interrupt it. As such, performance of the tracer is critical to minimize disturbance of the running software. LTTng offers this level of performance, allowing to trace the kernel very efficiently.

Runtime information on a system can also be obtained by recording metrics periodically from files under the `/proc` directory. Utilities like `top` and `ps` use this interface, parse their content and format them for display. This technique provides statistics about the system at a sampling frequency based determined by the interface. In contrast, kernel tracing records all events according to time. Instead of pooling metric values, they can be recovered at arbitrary resolution afterwards from the trace.

This paper is organized as follows. The kernel tracing infrastructure is presented in section 2. This presentation applies to the latest stable release of LTTng 0.249, which is used throughout the paper. In section 3, we present available tracepoints in the Linux kernel, their meaning and how we can recover metrics by processing them. A prototype that performs such analysis is presented in section 4. Finally, future work for LTTng 2.0 is discussed in section 5.

2 LTTng kernel tracer

The tracer is based on static tracepoints in the kernel source code. Each time a tracepoint is encountered and is enabled, an event is added to an in-memory ring buffer. There are three operating modes for subsequent data processing.

The *normal mode* is suitable for offline analysis. When a buffer is full, a signal is sent to a transport daemon, which then syncs buffers to disk before they get overwritten. On average, disk throughput must be higher than event output. If all buffers are full, which can happen if disk bandwidth is lower than event generation throughput, then events are dropped. The number of such lost events is kept in the trace. Lost events can compromise further trace analysis. To avoid lost events, buffer size can be increased at trace start, but not while tracing, because buffers are allocated at trace setup. Hence, enough space must be reserved according to disk speed and maximum expected event throughput.

In cases where high throughput is expected and only the most recent data is desired, the *flight recorder* mode is well suited. In this mode, the ring buffer is overwritten until a condition occurs, and then the most recent events are written to disk. No event will be lost in this mode, but the actual trace duration depends on the buffer size.

The final supported mode is *live reading*. In this mode, buffers are flushed to disk regularly, before each read, to ensure consistency between different trace streams within a bounded delay (e.g. 1 second). This applies even for buffers that are only partially filled. The flush guarantees the consistency of the trace, avoiding the possibility of reception of older events out-of-order, which would otherwise appear to the analysis module out of chronological order.

LTTng uses per-CPU buffers to avoid data access synchronization between CPUs in multi-core architectures, and allows scalable tracing for large number of cores. Events are grouped in channels. Each channel represents an event stream and has its own buffers. Hence, the total number of allocated buffers is $(P \times C)$, where P is the number of processors and C the number of channels.

The Linux kernel is instrumented with over 150 tracepoints at key sites within each subsystem. Each tracepoint is compiled conditionally by the `CONFIG_MARKERS` configuration option. This option depends on `CONFIG_TRACEPOINTS`, which enables other kernel built-in instrumentation. Once tracepoints are compiled, they can be later activated to record a trace. A tracepoint compiled in, but not activated, reduces to a `no-op` instruction, hence the performance impact is undetectable. Kernel tracing is highly optimized, but the overall performance impact is proportional to the number of events generated per unit of time. Benchmarks show that each event requires 119 ns to process on 2 GHz Intel Xeon processor in cache-hot condition [3].

Another aspect to observe is the impact of compiled tracepoints on the kernel size. For each tracepoint, a new function and static data is added, and thus increases the kernel size. For kernel 2.6.38, compiling tracepoints results in an increase of about 122 kB of the vmlinuz image, or 1%. This includes LTTng tracepoints and other built-in kernel tracepoints.

2.1 Trace format

An event is composed of a timestamp, an event type and an arbitrary payload. The timestamp is mandatory to sort events according to time. The event type is used to determine the format of the payload.

The timestamp uses the hardware cycle counter and is converted to nanoseconds since the boot of the system. Special care is taken to guarantee that the time always increases monotonically between cores. The time is represented with 27 bit time delta, while the event id is five bits wide, for a total event header size of 32 bits. If no event occurs and the time delta overflows, which occur in the order of 100 ms on recent CPUs, an extended header is written. The timestamp is extended to 64 bits, while 16 bits are reserved for the event type.

The payload is an ordered set of fields, where the size and format are defined by the event type. Fields can be any standard C basic type, as well as variable size strings. The size of a field may differ from the actual type used in the code in order to compress the data. For example, an integer enum value can be recorded in the trace as a byte if the actual value is always less than 255, thus saving space.

2.2 Trace reading

The library `libltttvtraceread` is provided to read events from a trace for further processing. The library opens all files from the trace at the same time and returns events in total order. It provides a merged view of the trace, which abstracts the complexity of handling per-CPU and per-channel files. The library provides convenient functions to seek at a particular time in the trace performed by a binary search. The library handles endianness transformations automatically if necessary. It parses each event in the trace and returns the timestamp, event type and an array of parsed fields.

3 Recovering metrics

System metrics include CPU usage, memory allocation, file operations and I/O operations, such as network and block devices. The trace contains a comprehensive set of data from which system metrics can be measured and accounted to a process. This section presents events and algorithms used to perform these computations.

3.1 Trace metadata

The Trace metadata channel is used to declare the trace event types. It contains two pre-defined events types, `core_marker_format` and `core_marker_id` that are implicit and fixed. The purpose of `core_marker_format` is to list available channels and event types in the trace, along with the format of each field in their payload. Meanwhile `core_marker_id` lists event IDs and their corresponding channel and event name. This event inventory enables selection of the correct format to parse the payload of a particular event.

3.2 System state dump

The state dump consists of information about the system as it was at the beginning of the trace. The available information in the state dump is listed in Table 1. Events in the state dump can be split in two categories, static and variable.

Static information is invariant for the duration of the trace. This information may be referenced by other events. For example, the system call table enables system call IDs to be resolved to a meaningful name. This is necessary because system call IDs may differ from one system to another.

Variable information is the initial inventory of resources, which may be modified by later events. For example, the state dump contains the list of active processes at the beginning of the trace. This list is modified by fork and exit events, that add and remove processes respectively.

In addition, the special event `statedump_end` indicates the end of the state dump and metadata.

3.3 Process recovery

As presented in Section 3.2, the state dump includes the initial process list. This list includes the process id, thread id, thread group id, parent process id and a flag to indicate a kernel thread. While tracing, the event `kernel.process_fork` indicates a new process, while `kernel.process_exit` means a process terminated. The executable name can be deduced from the field `filename` of the `fs.exec` event. Until this event occurs, the executable name is the same as the parent process or the previously known name.

An additional step is required to link an event to a process. Since the process id is not saved on a per-event basis, this information must be recovered from the trace itself. The corresponding CPU on which an event occurs is known from the trace file id. The relation with the process id running on a given CPU can be established from the `kernel.sched_schedule` events. This event type contains two integers, `prev_pid` and `next_pid`. Each event occurring on a given CPU following a scheduling event can be related to the process `next_pid`, assuming no scheduling event is dropped.

3.4 CPU usage

CPU usage is a basic system metric for understanding the behavior of a system. To recover CPU usage per process on the system, we use scheduling events. A process has only two states, which are *scheduled* or *idle*. If no process runs on a given CPU, the kernel schedules the special thread `swapper` and we consider the CPU as idle. The total CPU time used by a process is the sum of intervals for which it was scheduled.

To chart the CPU usage according to time, we first divide the trace into fixed intervals to match the desired

State channel	Type	Description
`fd_state.file_descriptor`	Variable	File name and PID of opened fd
`irq_state.idt_table`	Static	Interrupts descriptor table for processor exceptions
`irq_state.interrupt`	Variable	Hardware IRQ ids, names and addresses
`module_state.list_module`	Variable	List of loaded kernel modules
`netif_state.network_ipv4_interface`	Variable	List of IPv4 interface names and IP addresses
`netif_state.network_ipv6_interface`	Variable	Same as IPv4, but for IPv6 network interfaces
`softirq_state.softirq_vec`	Static	List of SoftIRQ ids, names and addresses
`swap_state.statedump_swap_files`	Variable	Swap block devices
`syscall_state.sys_call_table`	Static	List of system call ids, names and addresses
`task_state.process_state`	Variable	List of active processes information
`vm_state.vm_map`	Variable	List of all memory mappings by process

Table 1: Available events in state dump

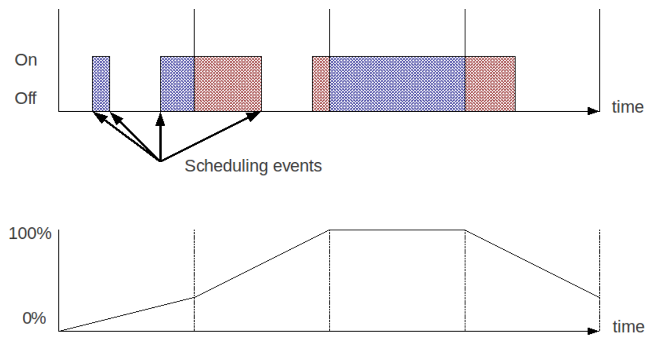


Figure 1: Average CPU usage recovery

resolution. Then the CPU usage of one interval is obtained by summing all overlapping scheduled intervals, as shown in Figure 1.

3.5 Memory usage

Tracepoints related to memory management provide information at the page level. The events `page_alloc` and `page_free` are encountered when a memory page related to a process is allocated or freed at the kernel level. The system memory usage at the beginning of the trace is given by the event `vm_state.vm_map` in metadata. Separate tracepoints report huge page operations. Thus the system memory usage on the system at a given time can be obtained by replaying subsequent page allocation and free events.

Note that the physical memory usage observed at the kernel level can differ from memory operations performed by the application. This behavior is intended to reduce the number of system calls that shrink or grow the heap and the stack of a process. The GNU libc may not release memory to the operating system after a call to `free()` to speedup subsequent allocations.

3.6 File operations

File operations are performed, for instance, through `open`, `read`, `write` and `close` system calls by user-space applications. These system calls are instrumented to compute file usage statistics, such as the number of bytes read or written, and to track opened files by a process. These events occur only if the system call succeeds. File operation events are recorded for files accessed through a file system as well as via network sockets.

The `fs.open` event contains the filename requested by the application and the associated file descriptor. All subsequent operations performed on this file descriptor by `fs.read` and `fs.write` contain the file descriptor and the byte count of the operation.

Tracing network operations on sockets is similar to files on a file system. Socket file descriptors are created upon `net.socket` and `net.accept` events, and byte transfer count is reported by `net.socket_sendmsg` and `net.socket_recvmsg` or `fs.write` and `fs.read`, depending on the system call involved.

3.7 Low-level network events

Low-level network events provide TCP and UDP packets fields, like ports and addresses involved in the communication. These extended network events are not enabled by default. To enable them, `ltt-armall` must be called with the switch `-n`.

Various metrics can be recovered directly from the payload of these events. The information provided by extended network events is a subset of what is available with tcpdump. Recording this information at the kernel level has the advantage of precise timestamps relative to system events. A network packet is always sent before being received, thus providing a convenient way to correlate traces from multiple systems without a common clock source. Average trace synchronization accuracy of 68.8 μs with TCP on standard 100 Mbit/s Ethernet has been achieved by using the Convex Hull algorithm [5].

3.8 Block Input/Output

In addition to file operations, underlying block device activity can be traced. Actions of block I/O scheduler, such as front and back merge requests, are recorded. Once the queue is ready, the event `fs_issue_rq` is emitted with the related sector involved. Each issue event is followed by the event `fs_complete_rq` when the request is completed. The delay between the issue and complete event indicates the latency of the disk operation.

Global disk offset can be observed with block level events. Disk offset is the difference between two consecutive sector requests to the device. For mechanical disks, high average offset between requests degrade throughput. Such bad performance is sometimes difficult to understand when it results from multiple processes performing independant operations on physically distant areas of the disk. The disk offset can be computed from `fs_issue_rq` event. The sector number from this event corresponds to the hardware sector. To relate this request to an inode, the `bio_remap` event is required. It contains the inode, the target device, the block requested at the file system level and the result of block remap, which is the hardware sector global to the device. Disk offset can thus be accounted on per inode basis.

4 LTTng kernel trace analyzer

We implemented a prototype of CPU usage computation to demonstrate the usefulness of the analysis. The software is coded in Java and uses the `libltttvtraceread` JNI interface. The system traced has two cores and runs Linux 2.6.36 with LTTng 0.249. The system is loaded with three `cpuburn` processes, started with one second interval between each. Figure 2 shows the trace loaded in the graphical interface.

The interface is divided in two parts. The top half contains the chart view of CPU usage according to time. The bottom half lists processes sorted according to total CPU usage. The chart has interactive features. Selecting an interval on the chart updates the process list statistics, while selecting a specific process displays its CPU usage. In Figure 2, the first `cpuburn` process is selected, and we can observe the CPU saturation occurring in this workload once these processes are started.

5 Future work

The goal of the LTTng project is to provide the community with the best tracing environment for Linux. We first present upcoming enhancements to the tracing infrastructure and future development of analysis tools.

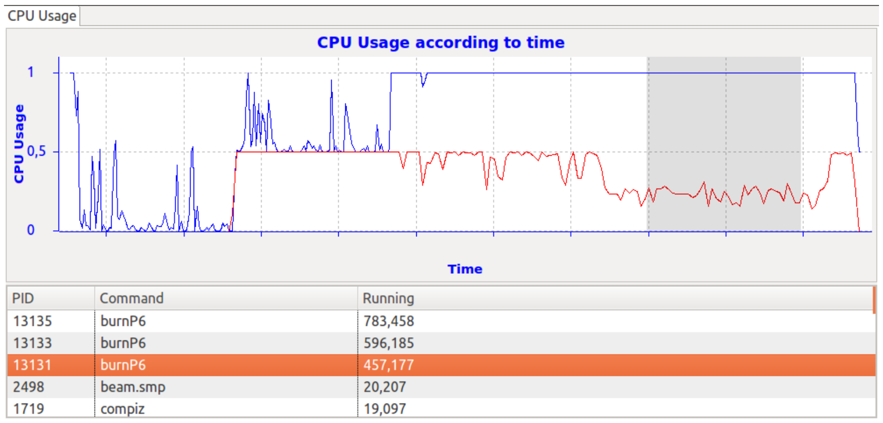
5.1 Tracing infrastructure

The current stable version of LTTng requires the kernel to be patched. Those patches are being converted to modules to work with vanilla kernel. The mainline kernel is already well instrumented. By default, those tracepoints will be available. Using stock kernels reduces the complexity of distributing LTTng, and helps towards making tracing ubiquitous.

The Common Trace Format (CTF) will be used to record the trace [2]. CTF has the ability to describe arbitrary sequence of binary objects. This new format will make it easier to define custom tracepoints in an extensible fashion.

Kernel tracing requires root privileges for security reasons. Enhancements to tracing tools will allow kernel tracing as normal user based on group membership, simplifying system administration. In addition, the unified git-like utility command-line tool `lttng` will control both kernel and user-space tracing on per-session basis. The live tracing mode will support reading data directly in memory, without flushing it to disk.

As presented in section 3.3, recovery of the PID of an event requires all scheduling events to be recorded. Omitting the PID from recorded events reduces the trace size, but increases analysis complexity. In some situations, appending a context to each event may be desirable to simplify event analysis, tolerate missing scheduling events, or to get values of performance counters



PID	Command	Running
13135	burnP6	783,458
13133	burnP6	596,185
13131	burnP6	457,177
2498	beam.smp	20,207
1719	compiz	19,097

Figure 2: LTTng trace analyzer prototype

when specific events occur. The next release will allow such event context to be added as a tracing option. Available context information includes, among others, the PID, the nice value, and CPU Performance Monitoring Unit (PMU) counters.

5.2 Trace analyzer

Analysis tools will require updates according to these changes to the tracing infrastructure. One purpose of the existing LTTng kernel patch is to extend the kernel instrumentation. Those tracepoints may not be available anymore with a vanilla kernel and the modules-based LTTng. We plan to propose additional tracepoints upstream to perform useful analysis for system administrators and software developers. For example, system calls are instrumented with entry and exit tracepoints, but arguments are not interpreted. Some of them may be pointers to structures, but saving an address is not useful for system metric recovery. Dereferencing the pointer and saving appropriate fields is required for many tracepoints.

Our current prototype implementation only includes the CPU usage. Our goal is to implement other presented metrics. System metrics computation can be integrated to the Eclipse LTTng plugin, part of the Eclipse Linux Tools project [6]. For this purpose, an open source native CTF Java reading library is being developed. Also, a command line tool similar to `top` is being developed to provide lightweight and live system metrics display from tracing data.

Further trace analysis are being developed to understand global performance behavior. Previous work has been done on critical path analysis of an application [4]. Our goal is to extend this analysis in two ways. First, computing the resource usage allows to get the cost of the critical path of an application. Secondly, analyzing communication paths between processes allows to recover links between distributed process. By combining those two analysis, we could provide a global view of the processing path of a client request made in a distributed application.

6 Conclusion

We showed that tracing can provide highly valuable data on a running system. This data can help to understand system-wide performance behavior. We presented the tracing infrastructure provided by LTTng and techniques to extract system metrics from raw events. Future developments to LTTng demonstrate the commitment to provide a state of the art tracing environment for the Linux community.

References

[1] Linux trace toolkit next generation. `http://lttng.org`.

[2] Mathieu Desnoyers. Common trace format specification v1.7. `git://git.efficios.com/ctf.git`.

[3] Mathieu Desnoyers. *Low-Impact Operating System Tracing*. 2009.

[4] P.M. Fournier and M.R. Dagenais. Analyzing blocking to debug performance problems on multi-core systems. *ACM SIGOPS Operating Systems Review*, 44(2):77–87, 2010.

[5] B. Poirier, R. Roy, and M. Dagenais. Accurate offline synchronization of distributed traces using kernel-level events. *ACM SIGOPS Operating Systems Review*, 44(3):75–87, 2010.

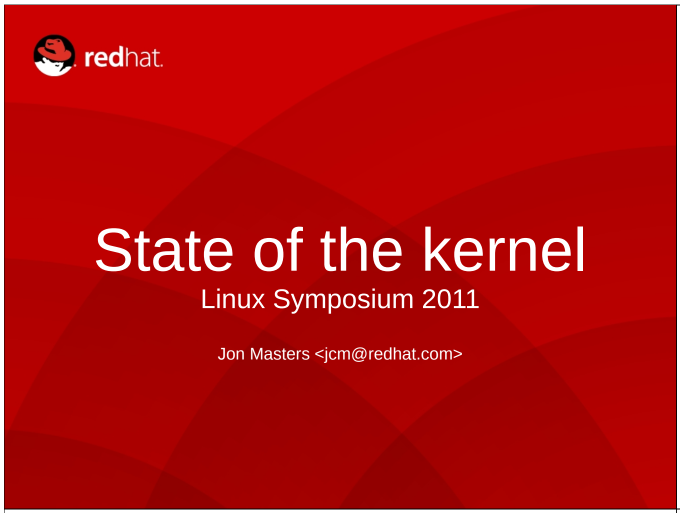
[6] D. Toupin. Using tracing to diagnose or monitor systems. *Software, IEEE*, 28(1):87–91, 2011.

State of the kernel

John C. Masters
Red Hat Inc.
jcm@redhat.com

Abstract

Slides from the talk follow.

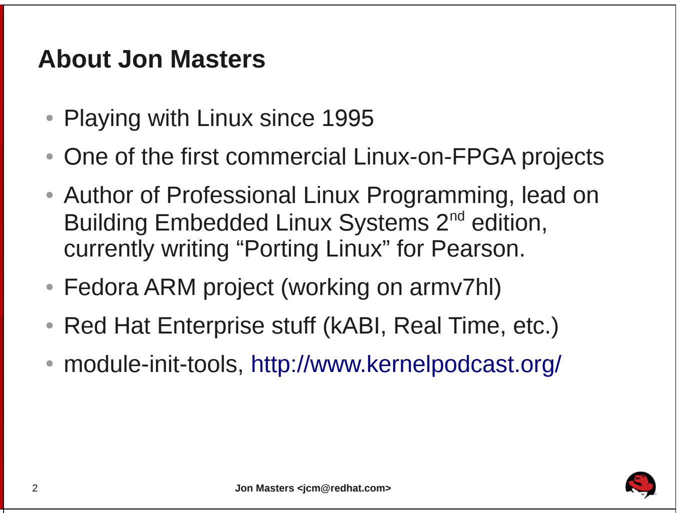


redhat.

State of the kernel

Linux Symposium 2011

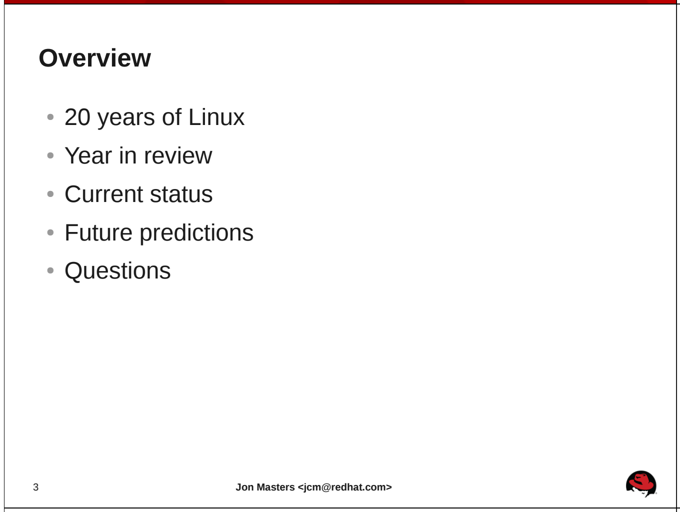
Jon Masters <jcm@redhat.com>



About Jon Masters

- Playing with Linux since 1995
- One of the first commercial Linux-on-FPGA projects
- Author of Professional Linux Programming, lead on Building Embedded Linux Systems 2nd edition, currently writing “Porting Linux” for Pearson.
- Fedora ARM project (working on armv7hl)
- Red Hat Enterprise stuff (kABI, Real Time, etc.)
- module-init-tools, http://www.kernelpodcast.org/

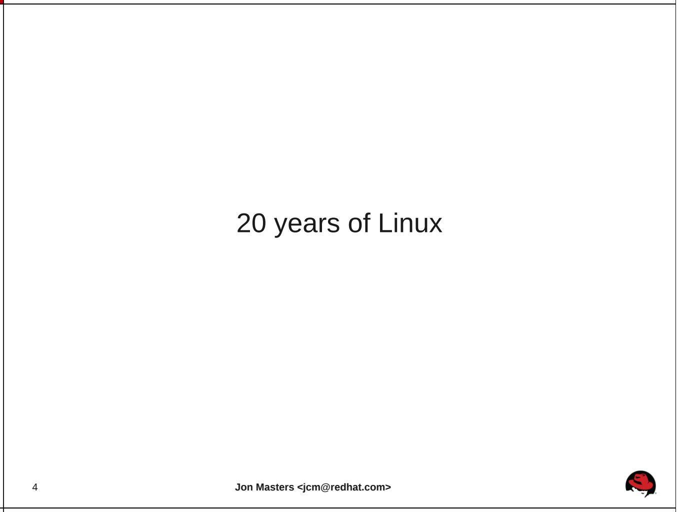
2 Jon Masters <jcm@redhat.com>



Overview

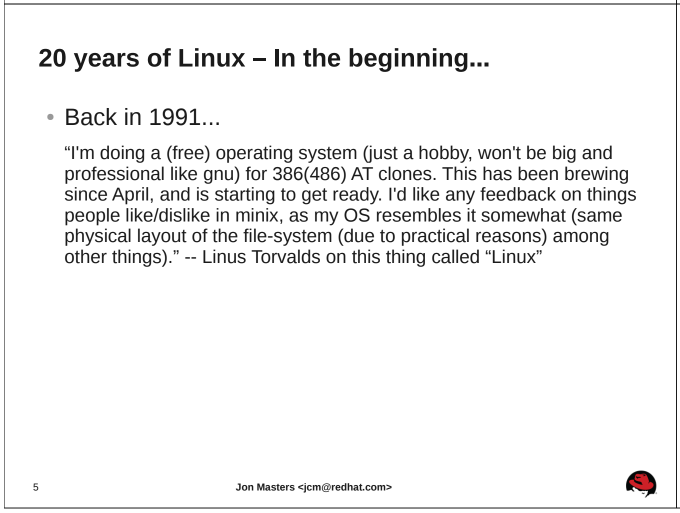
- 20 years of Linux
- Year in review
- Current status
- Future predictions
- Questions

3 Jon Masters <jcm@redhat.com>



20 years of Linux

4 Jon Masters <jcm@redhat.com>

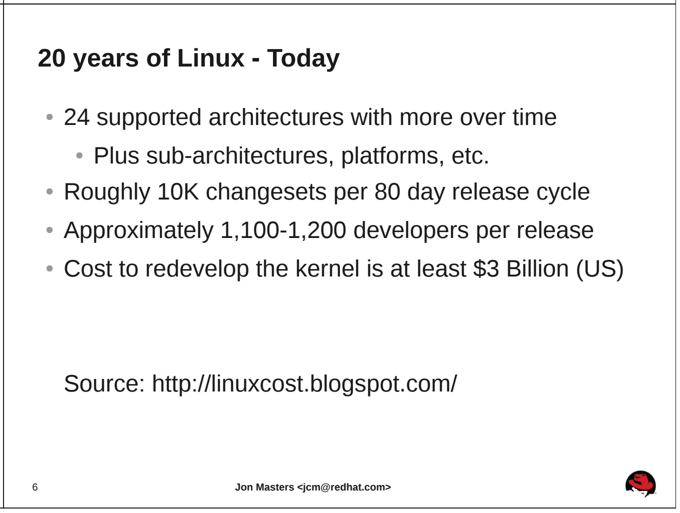


20 years of Linux – In the beginning...

- Back in 1991...

  “I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since April, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).” -- Linus Torvalds on this thing called “Linux”

5 Jon Masters <jcm@redhat.com>

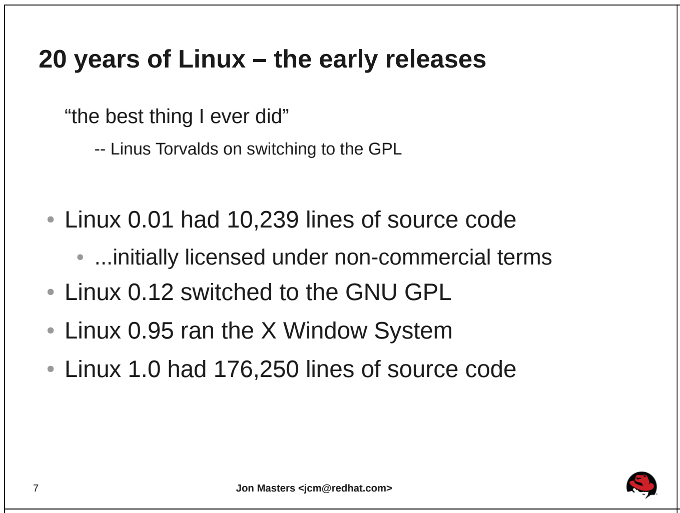


20 years of Linux - Today

- 24 supported architectures with more over time
  - Plus sub-architectures, platforms, etc.
- Roughly 10K changesets per 80 day release cycle
- Approximately 1,100-1,200 developers per release
- Cost to redevelop the kernel is at least $3 Billion (US)

Source: http://linuxcost.blogspot.com/

6 Jon Masters <jcm@redhat.com>



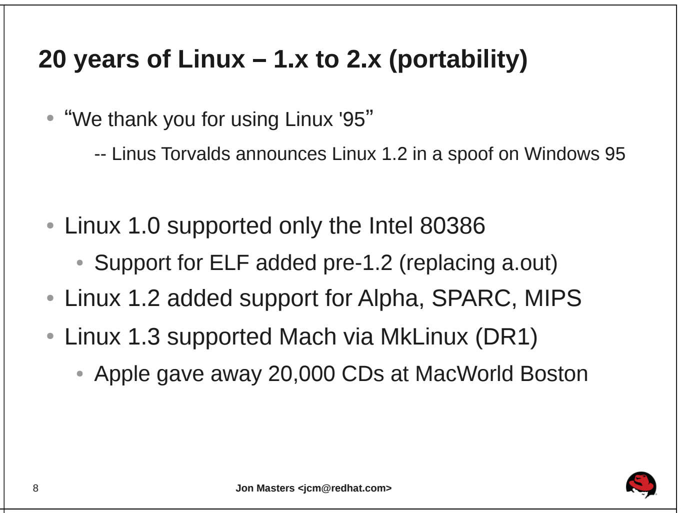
20 years of Linux – the early releases

“the best thing I ever did”

-- Linus Torvalds on switching to the GPL

- Linux 0.01 had 10,239 lines of source code
  - ...initially licensed under non-commercial terms
- Linux 0.12 switched to the GNU GPL
- Linux 0.95 ran the X Window System
- Linux 1.0 had 176,250 lines of source code

7 Jon Masters <jcm@redhat.com>



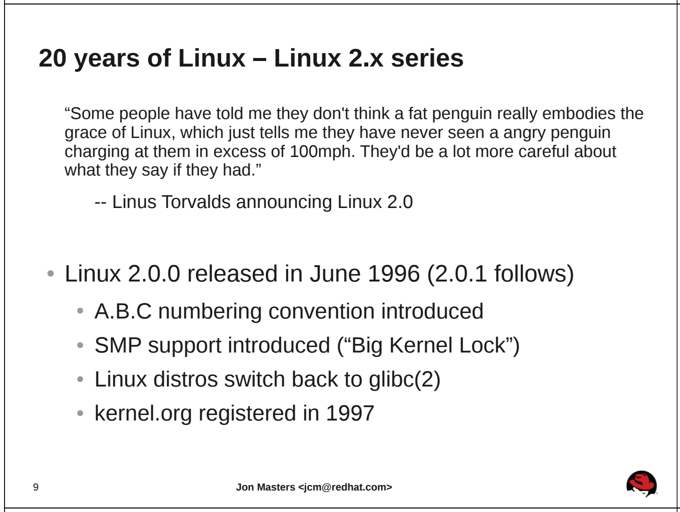
20 years of Linux – 1.x to 2.x (portability)

- “We thank you for using Linux '95”

  -- Linus Torvalds announces Linux 1.2 in a spoof on Windows 95

- Linux 1.0 supported only the Intel 80386
  - Support for ELF added pre-1.2 (replacing a.out)
- Linux 1.2 added support for Alpha, SPARC, MIPS
- Linux 1.3 supported Mach via MkLinux (DR1)
  - Apple gave away 20,000 CDs at MacWorld Boston

8 Jon Masters <jcm@redhat.com>



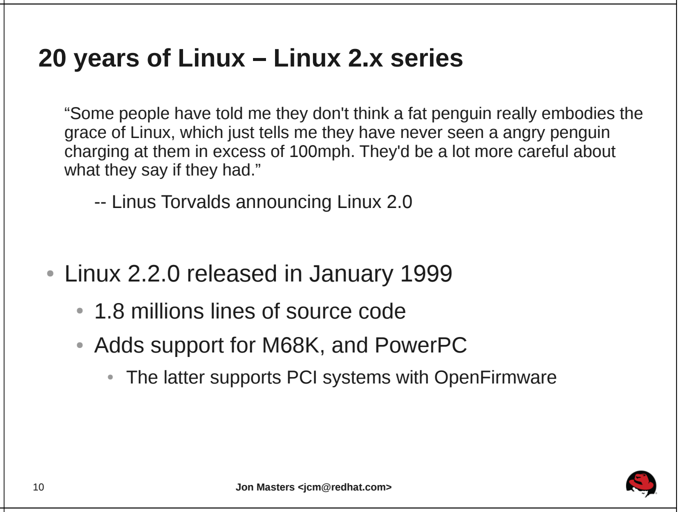
20 years of Linux – Linux 2.x series

“Some people have told me they don't think a fat penguin really embodies the grace of Linux, which just tells me they have never seen a angry penguin charging at them in excess of 100mph. They'd be a lot more careful about what they say if they had.”

-- Linus Torvalds announcing Linux 2.0

- Linux 2.0.0 released in June 1996 (2.0.1 follows)
  - A.B.C numbering convention introduced
  - SMP support introduced (“Big Kernel Lock”)
  - Linux distros switch back to glibc(2)
  - kernel.org registered in 1997

9 Jon Masters <jcm@redhat.com>



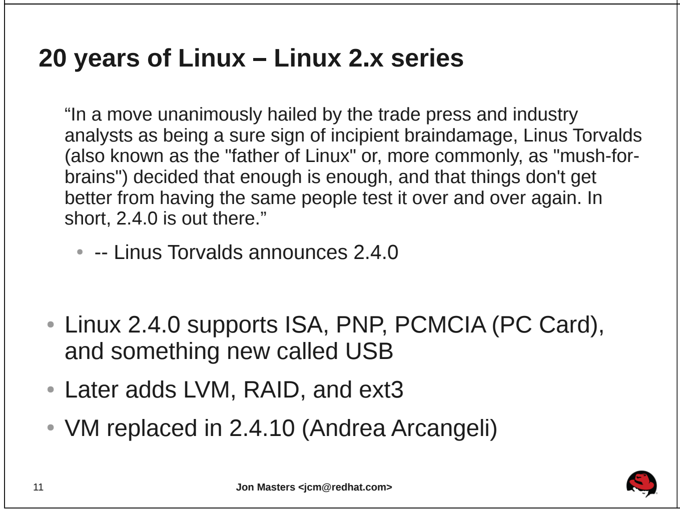
20 years of Linux – Linux 2.x series

“Some people have told me they don't think a fat penguin really embodies the grace of Linux, which just tells me they have never seen a angry penguin charging at them in excess of 100mph. They'd be a lot more careful about what they say if they had.”

-- Linus Torvalds announcing Linux 2.0

- Linux 2.2.0 released in January 1999
  - 1.8 millions lines of source code
  - Adds support for M68K, and PowerPC
    - The latter supports PCI systems with OpenFirmware

10 Jon Masters <jcm@redhat.com>



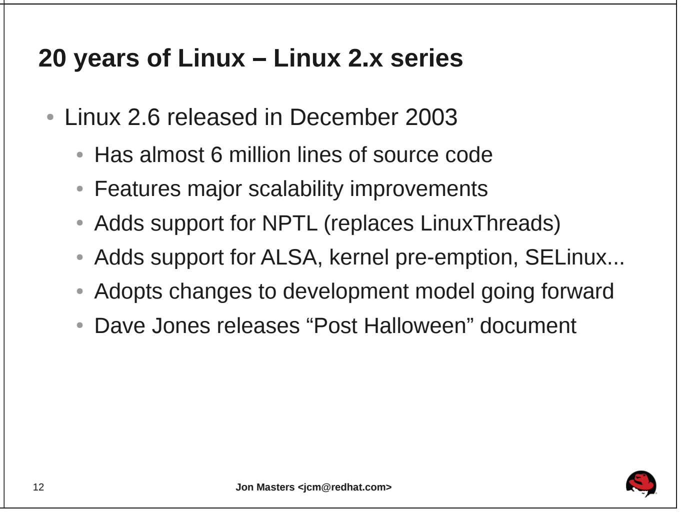
20 years of Linux – Linux 2.x series

“In a move unanimously hailed by the trade press and industry analysts as being a sure sign of incipient braindamage, Linus Torvalds (also known as the "father of Linux" or, more commonly, as "mush-for-brains") decided that enough is enough, and that things don't get better from having the same people test it over and over again. In short, 2.4.0 is out there.”

- -- Linus Torvalds announces 2.4.0

- Linux 2.4.0 supports ISA, PNP, PCMCIA (PC Card), and something new called USB
- Later adds LVM, RAID, and ext3
- VM replaced in 2.4.10 (Andrea Arcangeli)

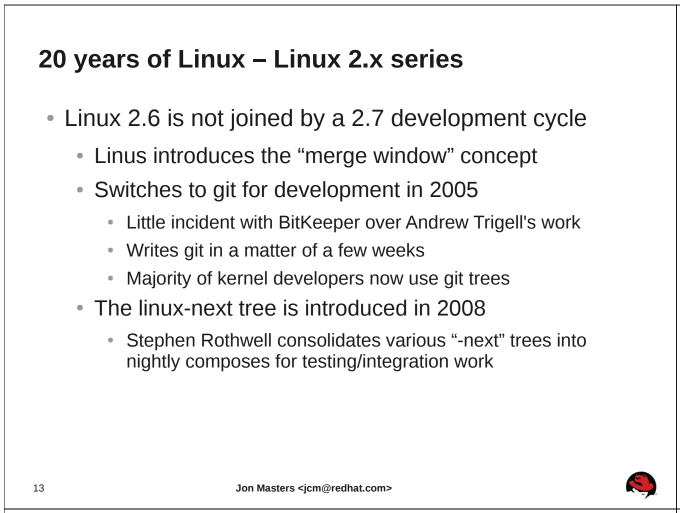
11 Jon Masters <jcm@redhat.com>



20 years of Linux – Linux 2.x series

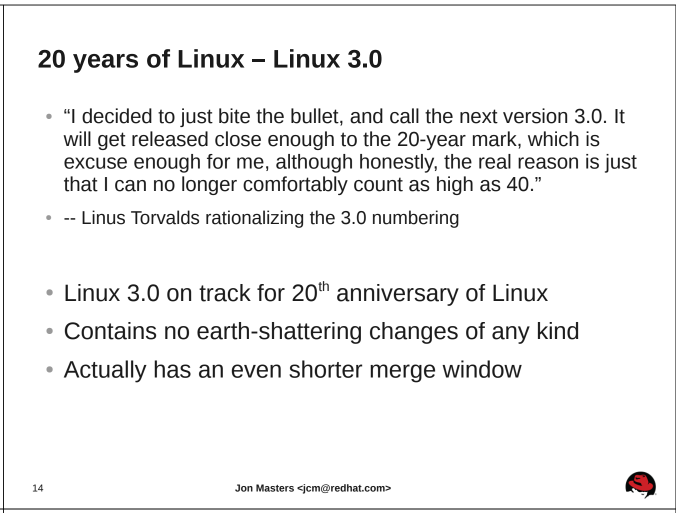
- Linux 2.6 released in December 2003
  - Has almost 6 million lines of source code
  - Features major scalability improvements
  - Adds support for NPTL (replaces LinuxThreads)
  - Adds support for ALSA, kernel pre-emption, SELinux...
  - Adopts changes to development model going forward
  - Dave Jones releases “Post Halloween” document

12 Jon Masters <jcm@redhat.com>



20 years of Linux – Linux 2.x series

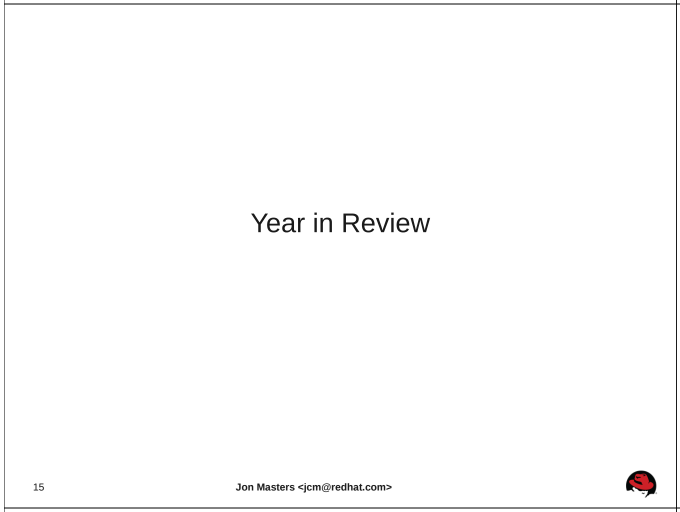
- Linux 2.6 is not joined by a 2.7 development cycle
  - Linus introduces the “merge window” concept
  - Switches to git for development in 2005
    - Little incident with BitKeeper over Andrew Trigell's work
    - Writes git in a matter of a few weeks
    - Majority of kernel developers now use git trees
  - The linux-next tree is introduced in 2008
    - Stephen Rothwell consolidates various “-next” trees into nightly composes for testing/integration work



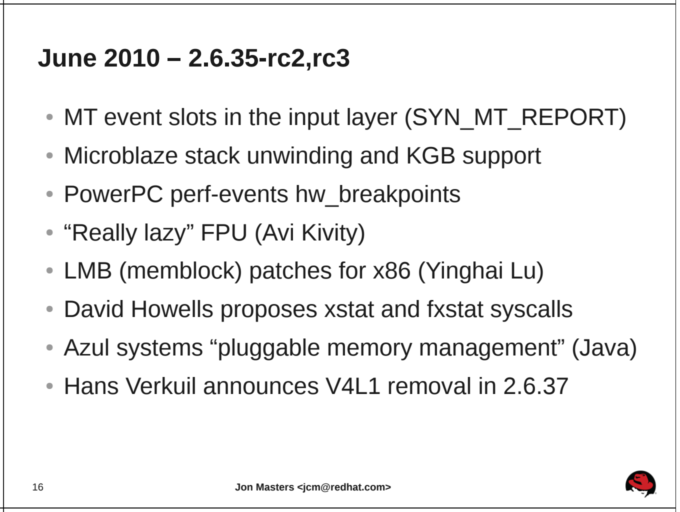
20 years of Linux – Linux 3.0

- “I decided to just bite the bullet, and call the next version 3.0. It will get released close enough to the 20-year mark, which is excuse enough for me, although honestly, the real reason is just that I can no longer comfortably count as high as 40.”
- -- Linus Torvalds rationalizing the 3.0 numbering

- Linux 3.0 on track for 20th anniversary of Linux
- Contains no earth-shattering changes of any kind
- Actually has an even shorter merge window

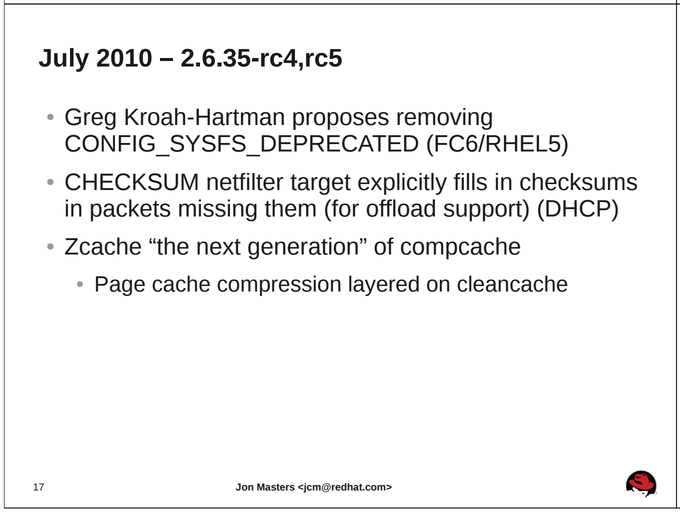


Year in Review



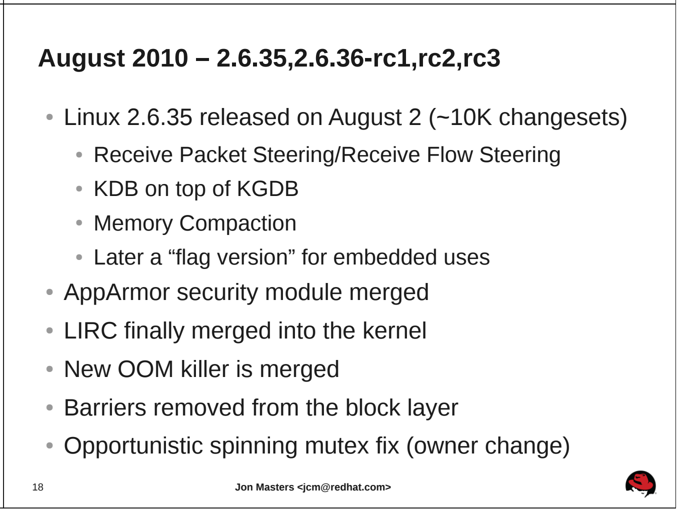
June 2010 – 2.6.35-rc2,rc3

- MT event slots in the input layer (SYN_MT_REPORT)
- Microblaze stack unwinding and KGB support
- PowerPC perf-events hw_breakpoints
- “Really lazy” FPU (Avi Kivity)
- LMB (memblock) patches for x86 (Yinghai Lu)
- David Howells proposes xstat and fxstat syscalls
- Azul systems “pluggable memory management” (Java)
- Hans Verkuil announces V4L1 removal in 2.6.37



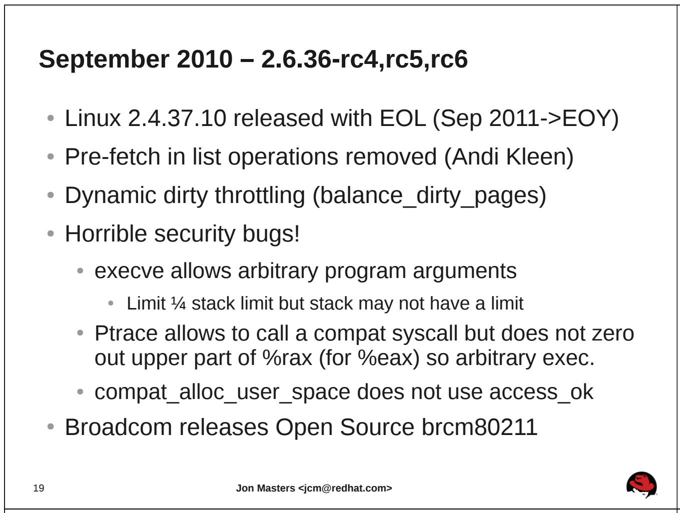
July 2010 – 2.6.35-rc4,rc5

- Greg Kroah-Hartman proposes removing CONFIG_SYSFS_DEPRECATED (FC6/RHEL5)
- CHECKSUM netfilter target explicitly fills in checksums in packets missing them (for offload support) (DHCP)
- Zcache “the next generation” of compcache
  - Page cache compression layered on cleancache



August 2010 – 2.6.35,2.6.36-rc1,rc2,rc3

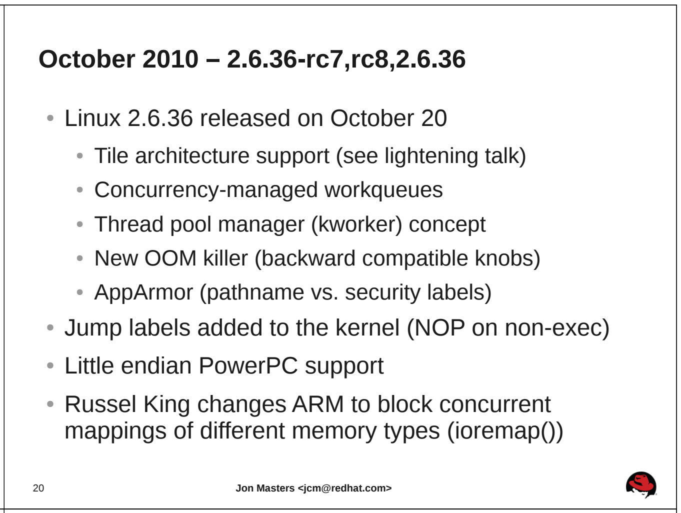
- Linux 2.6.35 released on August 2 (~10K changesets)
  - Receive Packet Steering/Receive Flow Steering
  - KDB on top of KGDB
  - Memory Compaction
  - Later a “flag version” for embedded uses
- AppArmor security module merged
- LIRC finally merged into the kernel
- New OOM killer is merged
- Barriers removed from the block layer
- Opportunistic spinning mutex fix (owner change)



September 2010 – 2.6.36-rc4,rc5,rc6

- Linux 2.4.37.10 released with EOL (Sep 2011->EOY)
- Pre-fetch in list operations removed (Andi Kleen)
- Dynamic dirty throttling (balance_dirty_pages)
- Horrible security bugs!
  - execve allows arbitrary program arguments
    - Limit ¼ stack limit but stack may not have a limit
  - Ptrace allows to call a compat syscall but does not zero out upper part of %rax (for %eax) so arbitrary exec.
  - compat_alloc_user_space does not use access_ok
- Broadcom releases Open Source brcm80211

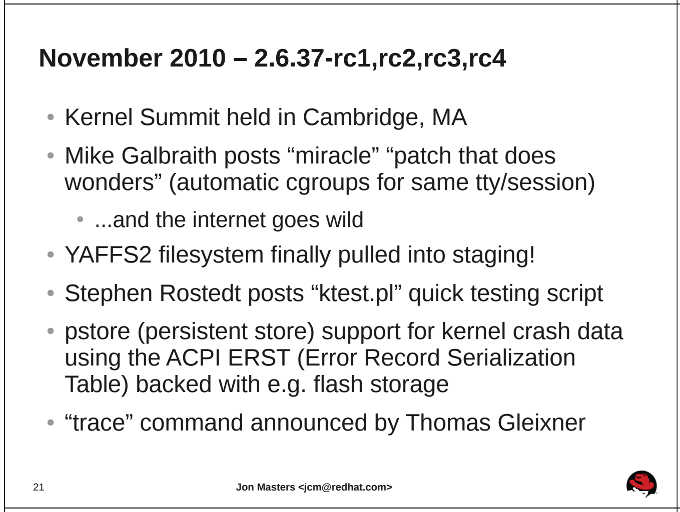
19 Jon Masters <jcm@redhat.com>



October 2010 – 2.6.36-rc7,rc8,2.6.36

- Linux 2.6.36 released on October 20
  - Tile architecture support (see lightening talk)
  - Concurrency-managed workqueues
  - Thread pool manager (kworker) concept
  - New OOM killer (backward compatible knobs)
  - AppArmor (pathname vs. security labels)
- Jump labels added to the kernel (NOP on non-exec)
- Little endian PowerPC support
- Russel King changes ARM to block concurrent mappings of different memory types (ioremap())

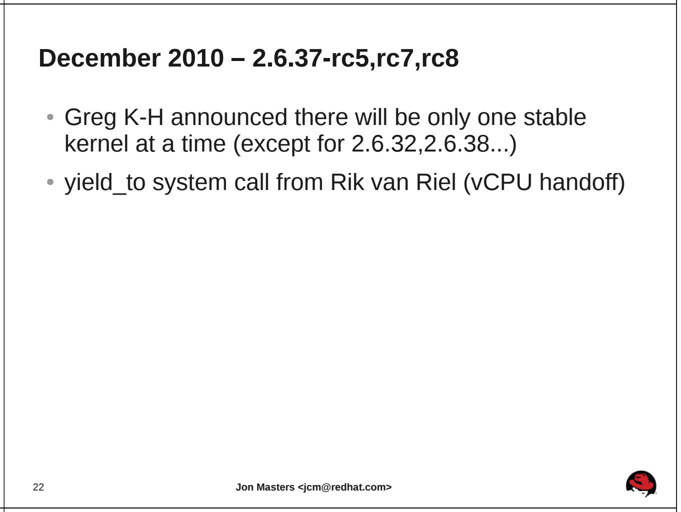
20 Jon Masters <jcm@redhat.com>



November 2010 – 2.6.37-rc1,rc2,rc3,rc4

- Kernel Summit held in Cambridge, MA
- Mike Galbraith posts “miracle” “patch that does wonders” (automatic cgroups for same tty/session)
  - ...and the internet goes wild
- YAFFS2 filesystem finally pulled into staging!
- Stephen Rostedt posts “ktest.pl” quick testing script
- pstore (persistent store) support for kernel crash data using the ACPI ERST (Error Record Serialization Table) backed with e.g. flash storage
- “trace” command announced by Thomas Gleixner

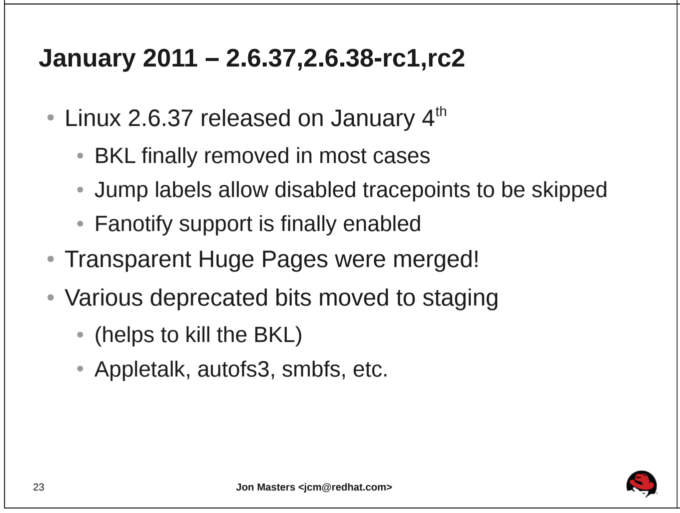
21 Jon Masters <jcm@redhat.com>



December 2010 – 2.6.37-rc5,rc7,rc8

- Greg K-H announced there will be only one stable kernel at a time (except for 2.6.32,2.6.38...)
- yield_to system call from Rik van Riel (vCPU handoff)

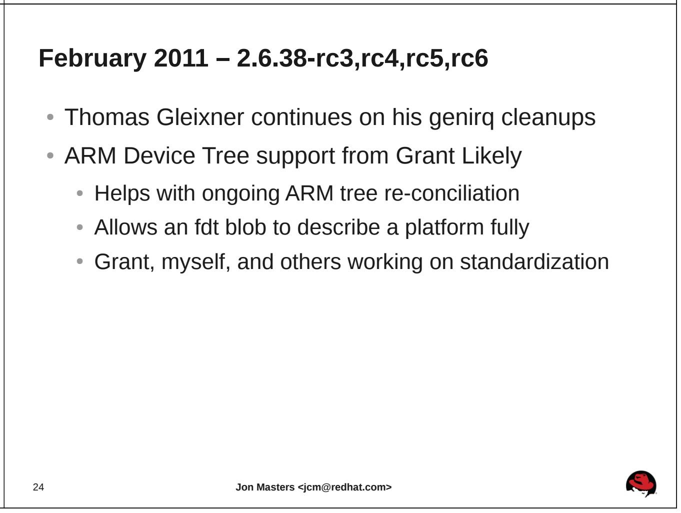
22 Jon Masters <jcm@redhat.com>



January 2011 – 2.6.37,2.6.38-rc1,rc2

- Linux 2.6.37 released on January 4th
  - BKL finally removed in most cases
  - Jump labels allow disabled tracepoints to be skipped
  - Fanotify support is finally enabled
- Transparent Huge Pages were merged!
- Various deprecated bits moved to staging
  - (helps to kill the BKL)
  - Appletalk, autofs3, smbfs, etc.

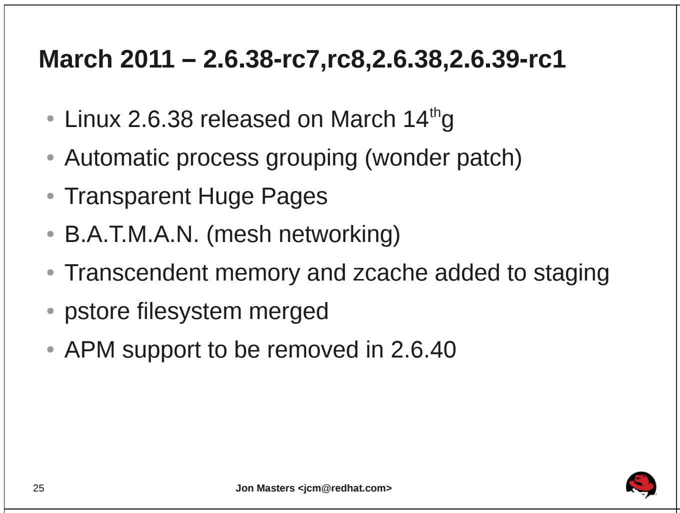
23 Jon Masters <jcm@redhat.com>



February 2011 – 2.6.38-rc3,rc4,rc5,rc6

- Thomas Gleixner continues on his genirq cleanups
- ARM Device Tree support from Grant Likely
  - Helps with ongoing ARM tree re-conciliation
  - Allows an fdt blob to describe a platform fully
  - Grant, myself, and others working on standardization

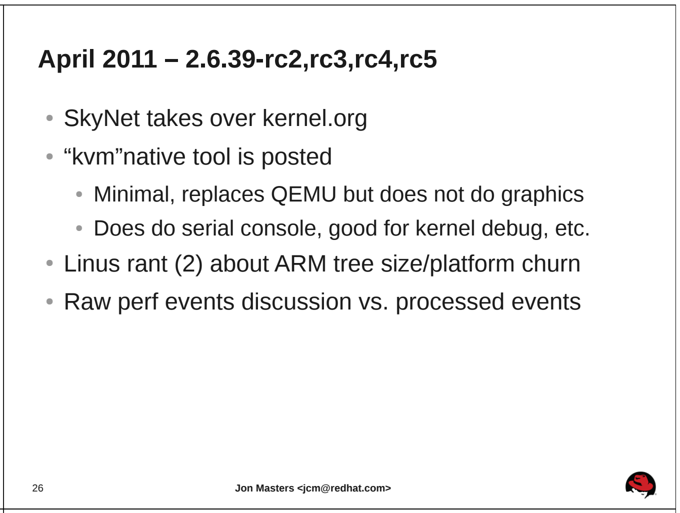
24 Jon Masters <jcm@redhat.com>



March 2011 – 2.6.38-rc7,rc8,2.6.38,2.6.39-rc1

- Linux 2.6.38 released on March 14thg
- Automatic process grouping (wonder patch)
- Transparent Huge Pages
- B.A.T.M.A.N. (mesh networking)
- Transcendent memory and zcache added to staging
- pstore filesystem merged
- APM support to be removed in 2.6.40

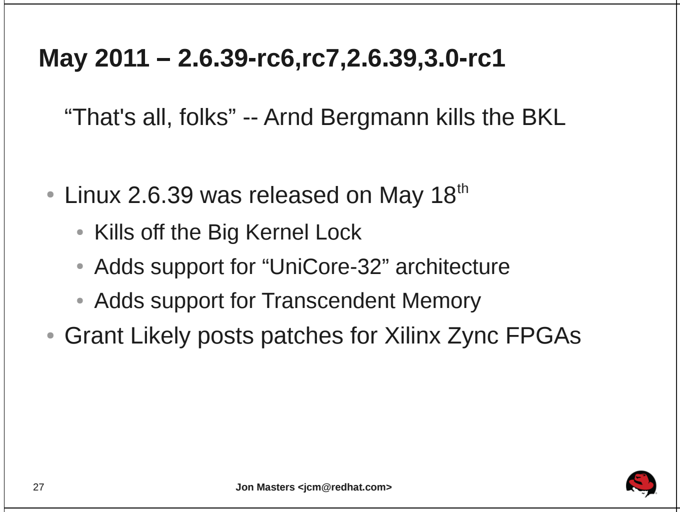
25 Jon Masters <jcm@redhat.com>



April 2011 – 2.6.39-rc2,rc3,rc4,rc5

- SkyNet takes over kernel.org
- “kvm”native tool is posted
  - Minimal, replaces QEMU but does not do graphics
  - Does do serial console, good for kernel debug, etc.
- Linus rant (2) about ARM tree size/platform churn
- Raw perf events discussion vs. processed events

26 Jon Masters <jcm@redhat.com>

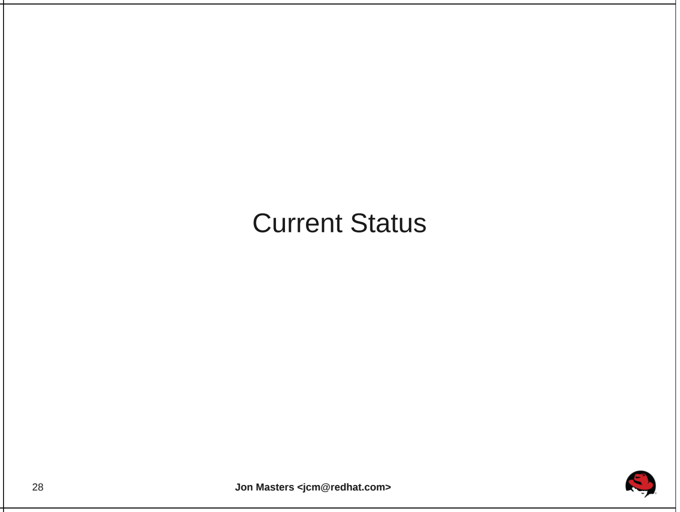


May 2011 – 2.6.39-rc6,rc7,2.6.39,3.0-rc1

“That's all, folks” -- Arnd Bergmann kills the BKL

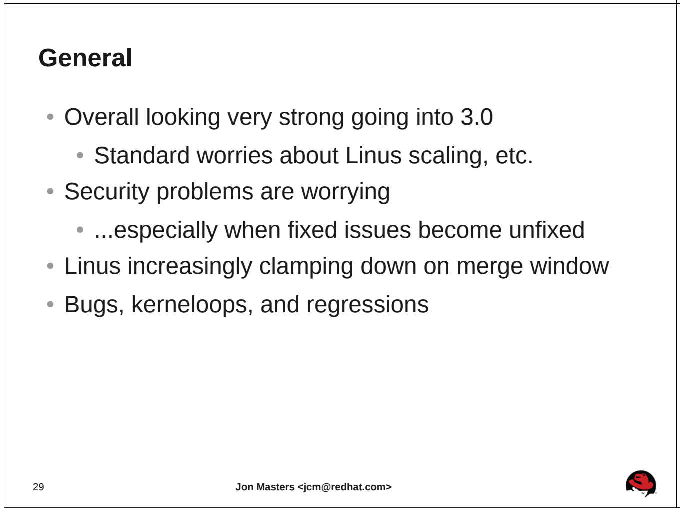
- Linux 2.6.39 was released on May 18th
  - Kills off the Big Kernel Lock
  - Adds support for “UniCore-32” architecture
  - Adds support for Transcendent Memory
- Grant Likely posts patches for Xilinx Zync FPGAs

27 Jon Masters <jcm@redhat.com>



Current Status

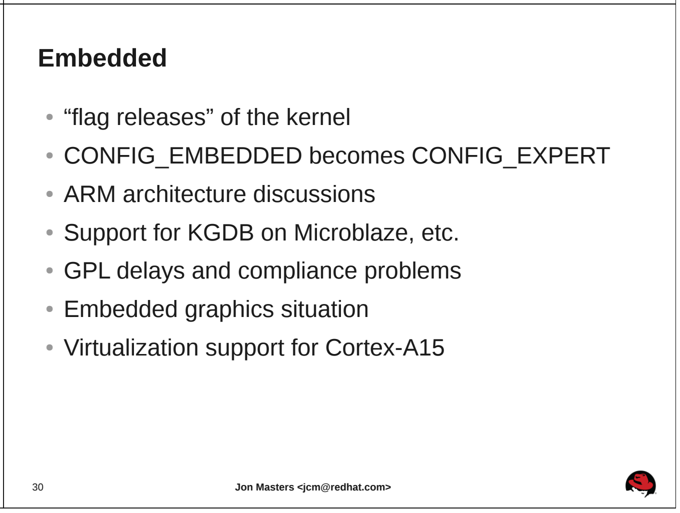
28 Jon Masters <jcm@redhat.com>



General

- Overall looking very strong going into 3.0
  - Standard worries about Linus scaling, etc.
- Security problems are worrying
  - ...especially when fixed issues become unfixed
- Linus increasingly clamping down on merge window
- Bugs, kerneloops, and regressions

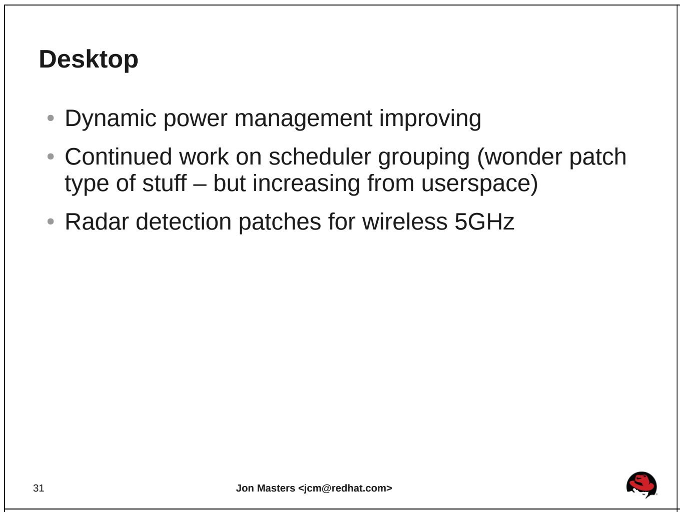
29 Jon Masters <jcm@redhat.com>



Embedded

- “flag releases” of the kernel
- CONFIG_EMBEDDED becomes CONFIG_EXPERT
- ARM architecture discussions
- Support for KGDB on Microblaze, etc.
- GPL delays and compliance problems
- Embedded graphics situation
- Virtualization support for Cortex-A15

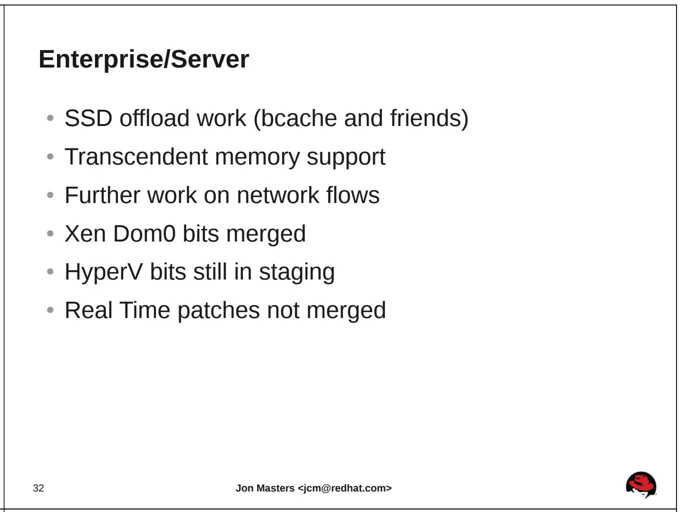
30 Jon Masters <jcm@redhat.com>



Desktop

- Dynamic power management improving
- Continued work on scheduler grouping (wonder patch type of stuff – but increasing from userspace)
- Radar detection patches for wireless 5GHz

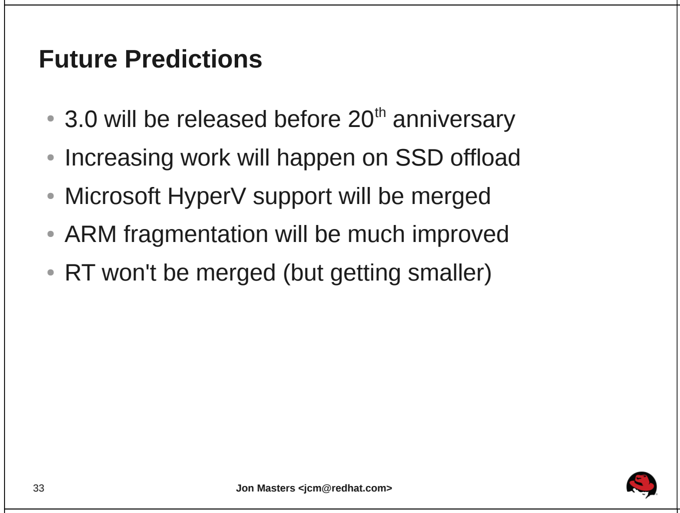
31 Jon Masters <jcm@redhat.com>



Enterprise/Server

- SSD offload work (bcache and friends)
- Transcendent memory support
- Further work on network flows
- Xen Dom0 bits merged
- HyperV bits still in staging
- Real Time patches not merged

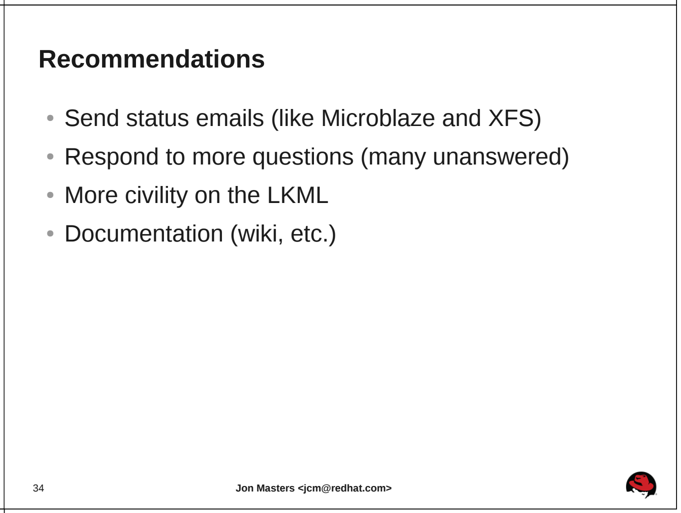
32 Jon Masters <jcm@redhat.com>



Future Predictions

- 3.0 will be released before 20th anniversary
- Increasing work will happen on SSD offload
- Microsoft HyperV support will be merged
- ARM fragmentation will be much improved
- RT won't be merged (but getting smaller)

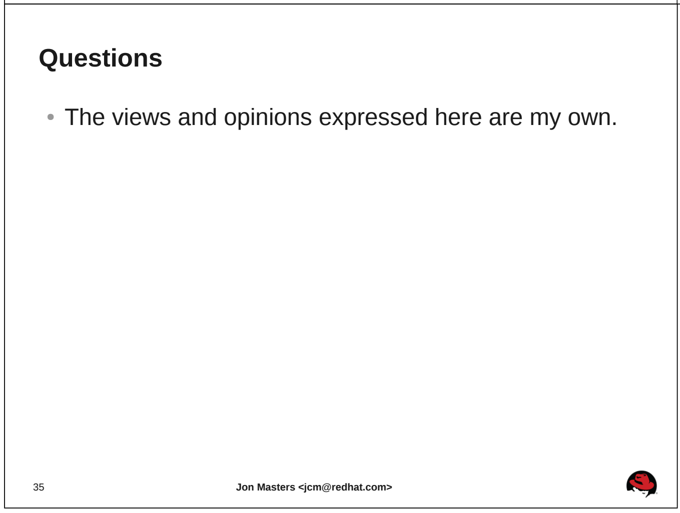
33 Jon Masters <jcm@redhat.com>



Recommendations

- Send status emails (like Microblaze and XFS)
- Respond to more questions (many unanswered)
- More civility on the LKML
- Documentation (wiki, etc.)

34 Jon Masters <jcm@redhat.com>



Questions

- The views and opinions expressed here are my own.

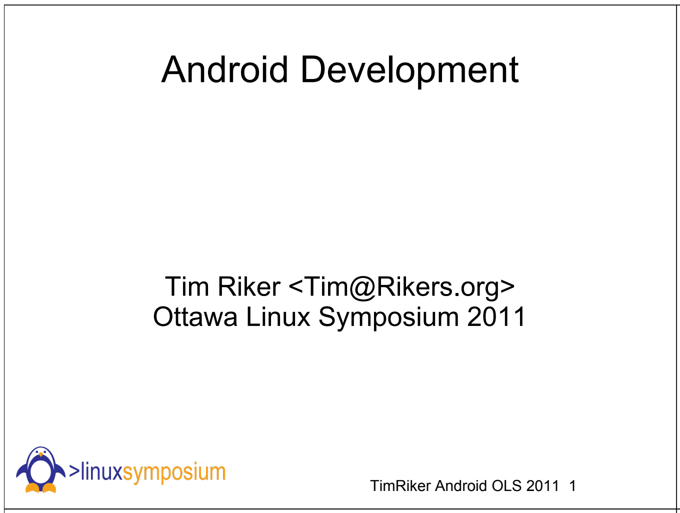
35 Jon Masters <jcm@redhat.com>

Android Development

Tim Riker
Tim@Rikers.org

Abstract

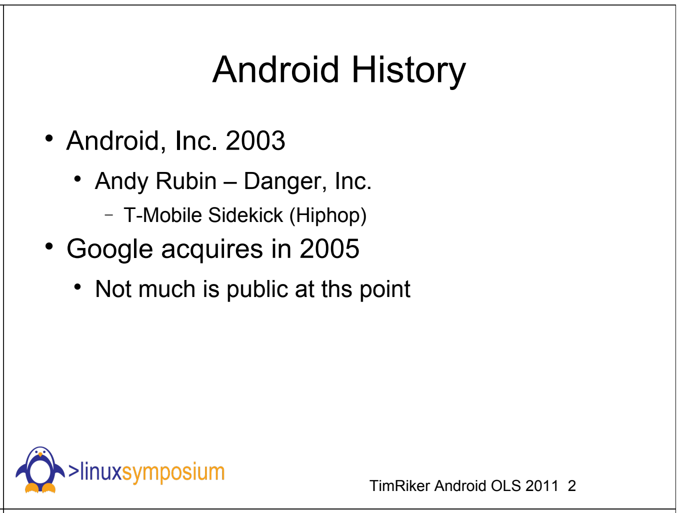
Slides from the talk follow.



Android Development

Tim Riker <Tim@Rikers.org>
Ottawa Linux Symposium 2011

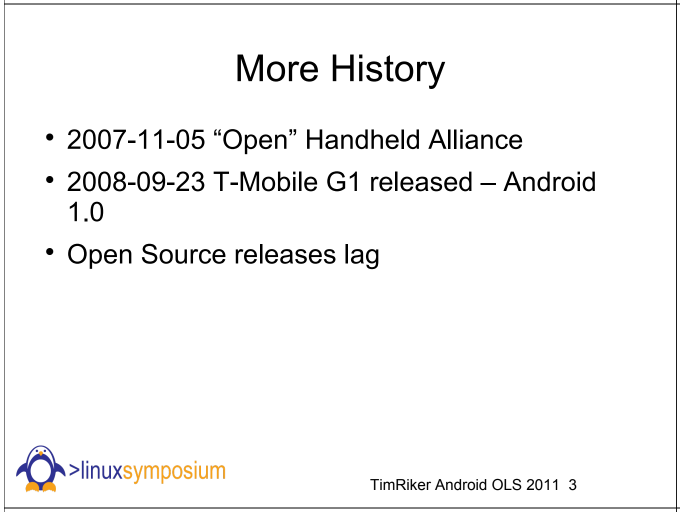
>linuxsymposium TimRiker Android OLS 2011 1



Android History

- Android, Inc. 2003
  - Andy Rubin – Danger, Inc.
    - T-Mobile Sidekick (Hiphop)
- Google acquires in 2005
  - Not much is public at ths point

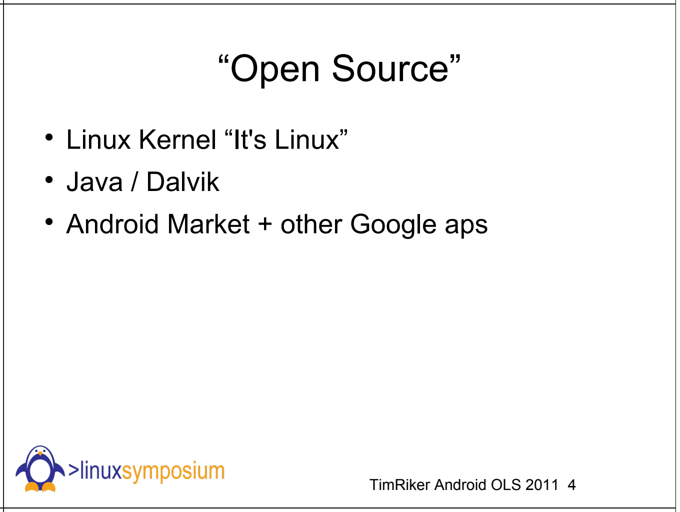
>linuxsymposium TimRiker Android OLS 2011 2



More History

- 2007-11-05 "Open" Handheld Alliance
- 2008-09-23 T-Mobile G1 released – Android 1.0
- Open Source releases lag

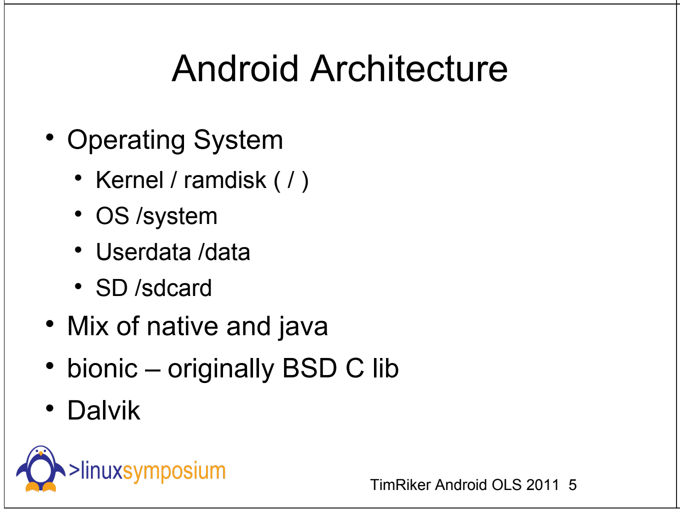
>linuxsymposium TimRiker Android OLS 2011 3



"Open Source"

- Linux Kernel "It's Linux"
- Java / Dalvik
- Android Market + other Google aps

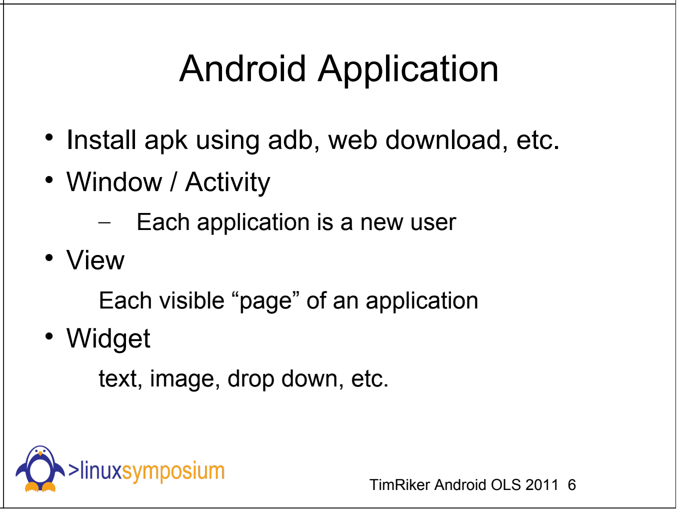
>linuxsymposium TimRiker Android OLS 2011 4



Android Architecture

- Operating System
  - Kernel / ramdisk (/)
  - OS /system
  - Userdata /data
  - SD /sdcard
- Mix of native and java
- bionic – originally BSD C lib
- Dalvik

>linuxsymposium TimRiker Android OLS 2011 5



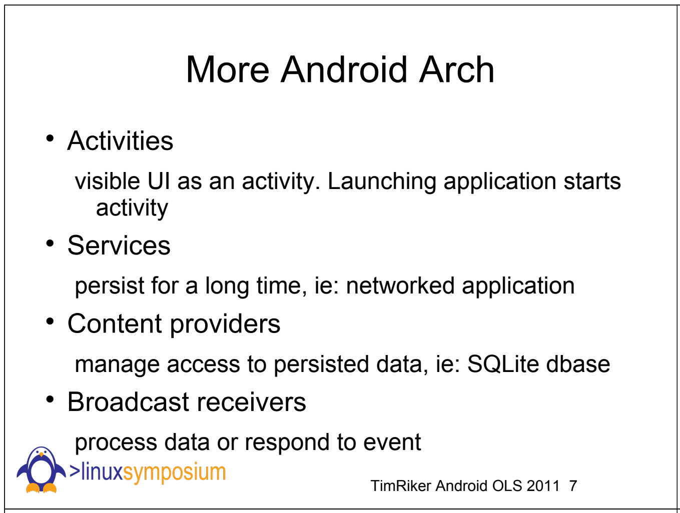
Android Application

- Install apk using adb, web download, etc.
- Window / Activity
  - Each application is a new user
- View

  Each visible "page" of an application
- Widget

  text, image, drop down, etc.

>linuxsymposium TimRiker Android OLS 2011 6



More Android Arch

- Activities

  visible UI as an activity. Launching application starts activity
- Services

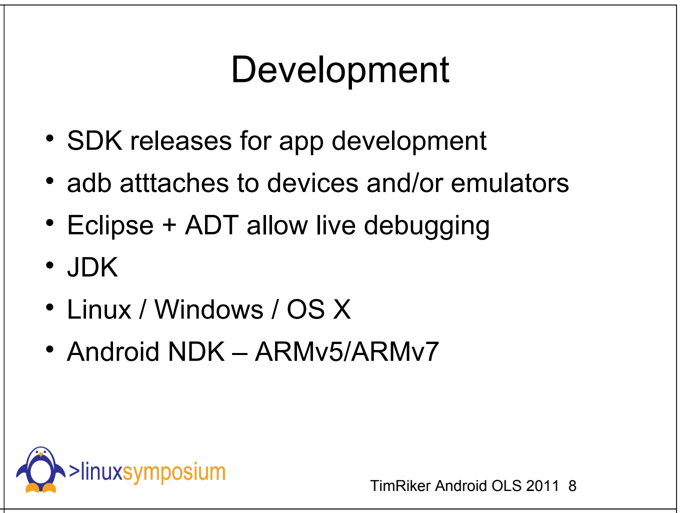
  persist for a long time, ie: networked application
- Content providers

  manage access to persisted data, ie: SQLite dbase
- Broadcast receivers

  process data or respond to event

>linuxsymposium

TimRiker Android OLS 2011 7

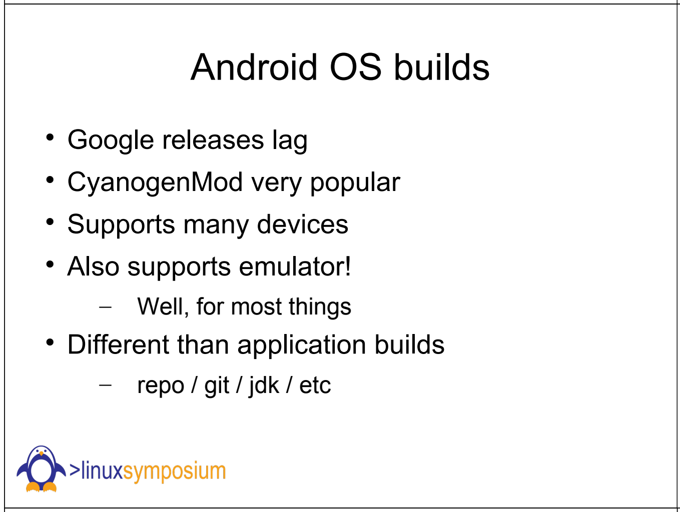


Development

- SDK releases for app development
- adb atttaches to devices and/or emulators
- Eclipse + ADT allow live debugging
- JDK
- Linux / Windows / OS X
- Android NDK – ARMv5/ARMv7

>linuxsymposium

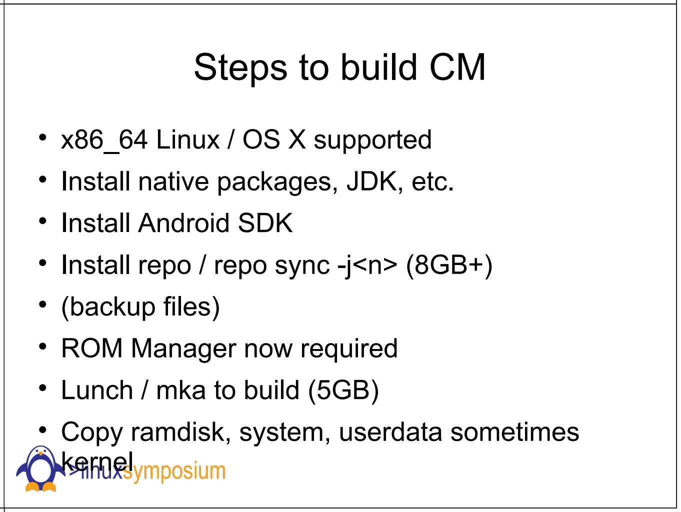
TimRiker Android OLS 2011 8



Android OS builds

- Google releases lag
- CyanogenMod very popular
- Supports many devices
- Also supports emulator!
  - Well, for most things
- Different than application builds
  - repo / git / jdk / etc

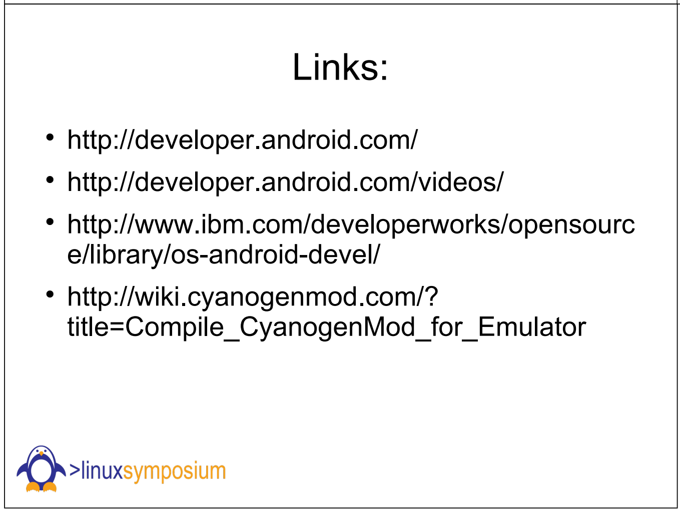
>linuxsymposium



Steps to build CM

- x86_64 Linux / OS X supported
- Install native packages, JDK, etc.
- Install Android SDK
- Install repo / repo sync -j<n> (8GB+)
- (backup files)
- ROM Manager now required
- Lunch / mka to build (5GB)
- Copy ramdisk, system, userdata sometimes kernel

>linuxsymposium



Links:

- http://developer.android.com/
- http://developer.android.com/videos/
- http://www.ibm.com/developerworks/opensourc e/library/os-android-devel/
- http://wiki.cyanogenmod.com/?title=Compile_CyanogenMod_for_Emulator

>linuxsymposium