Proceedings of the Linux Symposium

July 13th–16th, 2010
Ottawa, Ontario
Canada

Contents

Boosting up Embedded Linux device: experience on Linux-based Smartphone

Kunhoon Baik
Samsung Electronics Co., Ltd.
knhoon.baik@samsung.com

Saena Kim
Samsung Electronics Co., Ltd.
saina.kim@samsung.com

Suchang Woo
Samsung Electronics Co., Ltd.
suchang.woo@samsung.com

Jinhee Choi
Samsung Electronics Co., Ltd.
jh106.choi@samsung.com

Abstract

Modern smartphones have extensive capabilities and connectivities, comparable to those of personal computers (PCs). As the number of smartphone features increases, smartphone boot time also increases, since all features must be initialized during the boot time. Many fast boot techniques have focused on optimizing the booting sequence. However, it is difficult to obtain quick boot time (under 5 seconds) using the fast boot techniques, and many parts of the software platform require additional optimization. An intuitive way to obtain instant boot times, while avoiding these issues, is to boot directly from hibernation. We apply hibernation-based techniques to a Linux-based smartphone, and thereby overcome two major obstacles: long loading times for snapshot image and maintenance costs related to hardware change.

We propose two mechanisms, based on hibernation, to obtain outstanding reductions in boot time. First, minimize the size of snapshot image via page reclamation, which reduces the load time of image. Snapshot is split into two major segments: *essential-snapshot-image* and *supplementary-snapshot-image*. The essential snapshot image is a minimally-sized image used to run the Linux kernel and idle screen, and the supplementary-snapshot-image contains the remained that could be restored on demand. Second, we add additional device information to the essential-snapshot-image, which is used when the the device is reactivated upon booting up. As a result, our mechanism omits some time-consuming jobs related to device re-initialization and software state recovery. In addition to quick boot times, our solution is low maintenance. That is, while the snapshot boot[3] is implemented in the bootloader, our solution utilizes the kernel infrastructure because it is implemented in the kernel. Therefore, there is little effort required, even when the target hardware is changed. We prototyped our quick boot solution using a S5PC110[17]-based smartphone. The results of our experiments indicate that we can obtain get dramatic gain in performance in a practical manner using this quick boot solution.

1 Introduction

Smartphones generally require long boot times. As the number of smartphone functions increases, the initialization times required for the corresponding software modules also increase. In addition, as smartphones are equipped with more and more peripheral devices such as sensors, cameras, Bluetooth and WiFi, these devices require their own initialization times, which further increases boot time.

To obtain instant boot times, "boot optimization" or "hibernation-based boot" techniques can be used. In the case of "boot optimization", each module must be optimized and the initialization flow must be modified after a profiling step. This can be difficult to accomplish if there are many software modules involved, or if the initialization process is complex. However, in the case of "hibernation-based boot" techniques, we can obtain instant boot times quite easily. In this paper, we apply hibernation-based fast boot techniques to a Linux-based smartphone.

There remain some barriers to applying hibernation-based boot techniques.

1. Mobile software platforms, such as Android[14], hold about 100MB of RAM capacity, but Flash

memory offers only poor I/O speed. If the read performance is 20MB/s, then snapshot image alone require loading time of about 5 seconds. Therefore, we cannot obtain instant boot times via the hibernation-based boot technique alone.

2. Because swsusp's the device reactivation flow in the standard Linux kernel was developed for generic purposes[1], it has some additional steps to reactivate devices. The snapshot boot technique eliminates these steps by restoring snapshot image in the bootloader, but also requires additional implementations in the bootloader.

3. If the same snapshot image is used every time the device boots up, information inconsistency problems will occur in the file system and database.

In this paper, we introduce new methods to obtain instant boot times by solving these issues. We focus on the following two methods. The first method optimizes the size of snapshot image to be less than 15MB without compression, to reduce snapshot image loading time. The second method improves the device reactivation flow to obtain similar performance to the snapshot boot technique, without tinkering with the bootloader. We also briefly discuss related issues such as information inconsistency problems.

This paper is organized as follows. In section 2, we summarize fast boot techniques already developed in Embedded Linux systems, and compare them with our approach. In section 3, we analyze smartphone boot times and investigate points where improvements can be made. In section 4, we introduce our approach, which optimizes snapshot image loading times with on-demand-paging and early device reactivation. Section 5 describes the experimental environment and provides experimental results. Finally, in sections 6 and 7, we suggest directions for future work and summarize the paper.

2 Related studies

Until recently, Linux development has focused on the desktop and server markets, in which boot time is not an important issue. However, boot time has become an important feature as more and more embedded systems are adopting Linux due to benefits such as low cost and the ability to be utilized across a variety of hardware platorms.

Boot time optimization techniques include profiling, reduction and optimizing techniques. These techniques were well summarized by the Bootup Time Working Group of the CE Linux Forum[5]. This section introduces some of these techniques, which can be used with our approach.

At the bootloader level, uncompressed kernel[6] or fast kernel decompression[7] techniques can be used. In the case of uncompressed kernel techniques, the kernel image loading time is longer but decompression time is not required. Fast kernel decompression improves kernel decompression performance using fast decompress mechanisms such as UCL[18]

At the kernel level, disable console[8], preset loops per jiffy(LPJ)[9] and deferred initcalls[10] techniques may be used. The disable console technique minimizes kernel printk messages during boot time to reduce serial console accessing time. The preset LPJ uses a constant delay value instead of the `calibrate_delay()` function that is commonly used for calibrating delay time in the kernel. The deferred initcalls technique forces some initcalls to run later if they do not need to be initialized early.

Hibernation-based techniques also reduce boot time. Hibernation is a feature used for power management in Linux. As a power saving mode, hibernation backs up the running state of the system into the disk space as a snapshot image, and powers down the system. When the power comes back up, the system is restored to the running state based on the snapshot image. Hibernation can be implemented by several techniques. Among these, the most common techniques are the swsusp technique that is included in the standard Linux kernel, and the TuxOnIce(suspend2)[11] technique that is provided as a patch of the kernel. The fundamentals are almost the same in these two techniques, but TuxOnIce offers more useful options compared to swsusp. However, the TuxOnIce patch requires many changes for the kernel, and therefore also incurs additional maintenance costs according to the kernel revision. The snapshot boot technique is a fast boot technique based on swsusp. In this technique, every time a device boots up, the snapshot

[1]swsusp (Software Suspend) is a suspend-to-disk implementation in the 2.6 series Linux kernel. It is the Linux equivalent of Windows hibernate functionality.

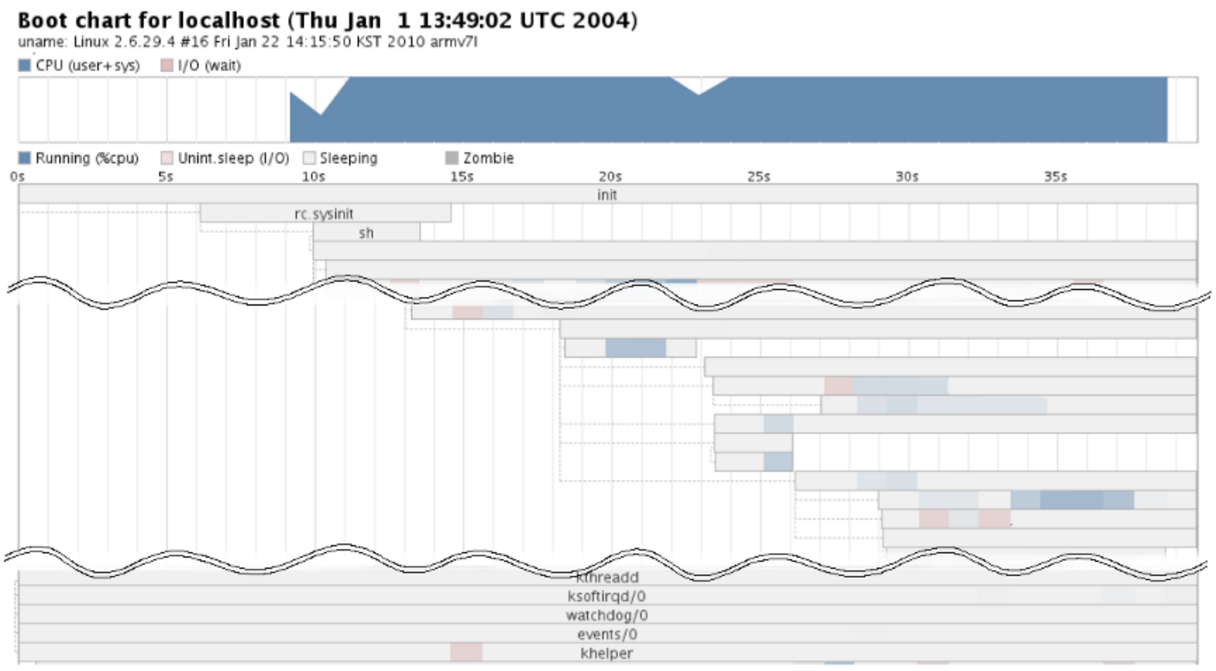


Figure 1: Bootchart – normal boot sequence of the smartphone used in this study

image is loaded in the bootloader instead of the original kernel image. Device initialization tasks are also performed at the bootloader level to improve the device reactivation flow in swsusp. However, most of the changes in the bootloader are heavily dependent on hardware, and for this reason, the associated maintenance costs are increased due to required changes of hardware design. This shortcoming makes the application of snapshot boot techniques less practical. Even if the snapshot boot technique is applied to a system, instant boot may not be achieved without further optimizing the size of the snapshot image. Recent smartphones require more memory space than older models, because of their extensive functionalities, and for this reason optimizing snapshot image must be considered a necessity. In a previous case study examining the use of the snapshot boot technique for digital TV systems[4] many parts of the software platform were modified to minimize the size of the snapshot image. However, such an approach increases maintenance costs due to the necessity of hardware and software platform revisions.

3 Smartphone Boot Time

Figure 1 is a the bootchart[12] of the smartphone model used in our experiments. More than 30 seconds of boot time are required to initialize the user area. This indicates that it will be difficult to reduce boot time to less than 5 seconds by optimizing the boot sequence. Even if we implement hibernation-based fast boot techniques, we cannot achieve 5 second boot times due to the barriers described in section 1.

To solve these problems, we must analyze each element of hibernation-based boot time. We calculate hibernation-based boot time(t_b) using the following formula.

$$t_b = t_p + t_l + \sum t_r$$

t_p includes the block device setup time for loading snapshot image and the cpu/clock/timer/power setup times for minimal operation. These constitute the necessary initialization events for booting from hibernation, and therefore the time required for these steps cannot be reduced. t_l is the time required to load the image from disk to the original memory location, and t_r is the time required to restore the cpu/device to the same state as when the snapshot image was made. These two factors can be optimized by improving the implementation of hibernation. t_l can be calculated using the following formula.

$$t_l = \frac{\textit{size of snapshot image}}{\textit{disk read performance}} + t_c$$

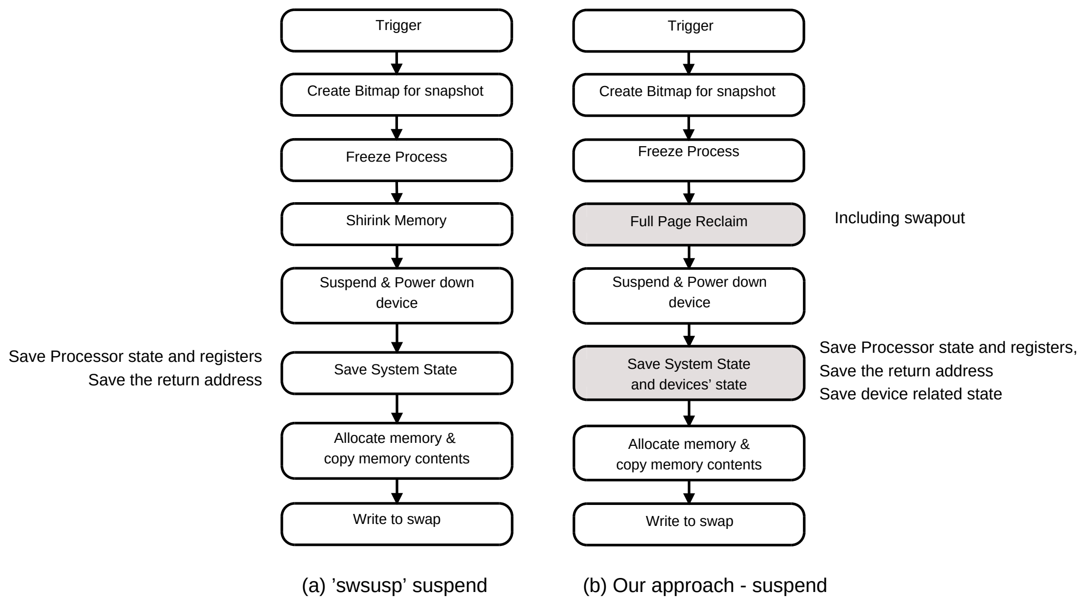

Trigger
Create Bitmap for snapshot
Freeze Process
Shrink Memory
Suspend & Power down device
Save Processor state and registers
Save the return address
Save System State
Allocate memory & copy memory contents
Write to swap
(a) 'swsusp' suspend
Trigger
Create Bitmap for snapshot
Freeze Process
Full Page Reclaim
Including swapout
Suspend & Power down device
Save System State and devices' state
Save Processor state and registers,
Save the return address
Save device related state
Allocate memory & copy memory contents
Write to swap
(b) Our approach - suspend

Figure 2: Comparision between the *swsusp* suspend method and our suspend approach

t_c is the time required to copy the loaded snapshot image, stored in in temporal memory, to the assigned memory location. As mentioned above, as the size of snapshot image get bigger, t_l becomes longer. When this happens, we can simply use a compression method such as TuxOnIce to reduce the size of the snapshot image. However, this method requires additional decompression time which increases t_c.

Another factor that influences hibernation-based boot time is t_r which is heavily dependent on the method used to restore the device. In the case of swsusp, all peripherial devices are initialized and then suspended to place them in a resumable state, meaning the same state as when the snapshot image was made. Therefore, if the number of peripherial devices is increased, t_r and t_c are also increased because the memory required for those device drivers is occupied. The snapshot boot technique places peripherial devices into the resumable state and loads snapshot image at the bootloader level. In this way, t_r is much reduced and t_c is eliminated. However, the snapshot boot technique requires additional maintenance costs associated with necessary changes of hardware.

In this paper, we suggest the following two mechanisms to speed up boot times. The first is to minimize the size of the snapshot image in order to reduce snapshot image loading time (t_l) which is the most influential factor hibernation-based boot time. To implement this mechanism, we store snapshot image separately as essential-snapshot-image, which will be loaded at boot time, and supplementary-snapshot-image, which will be restored on demand. The other mechanism is to place the peripherial devices in resumable states using information stored in snapshot image, to reduce $\sum t_r$. The details of these mechanisms are described in the next section.

4 Minimizing Boot Times: Our Approach

4.1 Overall Architecture

In this section, we analyze the suspend/resume flow in swsusp and introduce our improved suspend/resume flow.

As shown in Figure 2, we modify the "shrink memory"[2] stage to "full page frame reclamation," which involves minimizing the size of snapshot image by reclaiming almost of all of the memory required except for essential code and data required for hibernation. At the "save system state" stage, we save information about device related states as well as processor related states to resume the device stage after power up.

As shown in Figure 3-(a), the swsusp resume is started after all devices are initialized. And at the "Suspend device" stage, all of them are suspended – in other words,

[2]A stage in the swsusp suspend flow that ensures enough memory space is allocated to create the snapshot image in memory space.

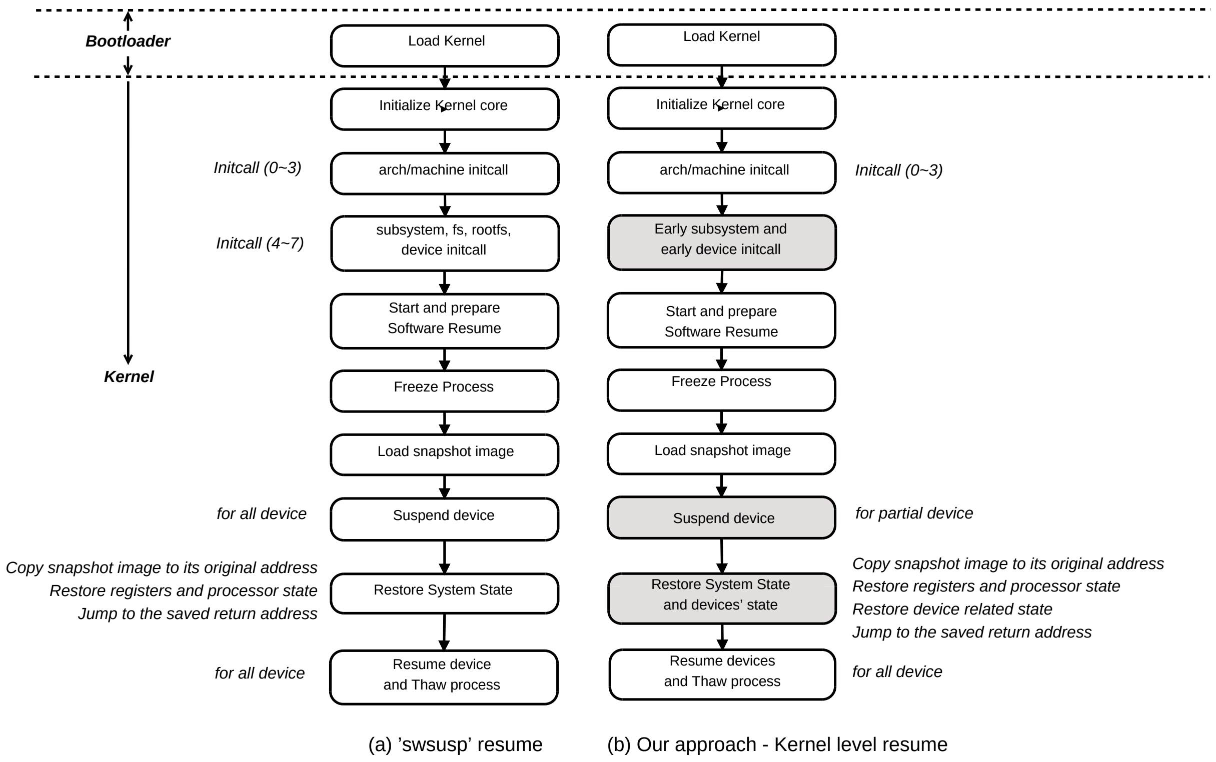

Bootloader
Kernel
Load Kernel
Initialize Kernel core
Initcall (0~3)
arch/machine initcall
Initcall (4~7)
subsystem, fs, rootfs, device initcall
Early subsystem and early device initcall
Start and prepare Software Resume
Freeze Process
Load snapshot image
for all device
Suspend device
for partial device
Copy snapshot image to its original address
Restore registers and processor state
Jump to the saved return address
Restore System State
Restore System State and devices' state
Restore device related state
for all device
Resume device and Thaw process
Resume devices and Thaw process
(a) 'swsusp' resume
(b) Our approach - Kernel level resume

Figure 3: Comparision between the *swsusp* resume approach and our resume approach(kernel level)

this stage place them into the resumable state. Therefore, if there are more devices, or device complexity increases, the time required to initialize and suspend devices will be increased. On the other hand, in our approach as shown in Figure 3-(b), the device initialization and suspend stages are removed and the "restore devices state" is added to the "restore system state" stage. At the "restore devices state" stage, we can place the devices into resumable states based on information that is saved in the snapshot image.

4.2 Full Page Reclamation

Figure 4 outlines a logical view for "full page frame reclamation." Using the "swap out" mechanism in Linux, all application code and data can be reclaimed except locked memory and the caches can be dropped. The reclaimed memory can be restored on demand using the "on demand paging" mechanism in Linux. By taking advantage of these features, the snapshot image can be seperated into two parts, the essential-snapshot-image, which will be restored at boot time, and the supplementary-snapshot-image, which will be restored on demand while the system is running.

To implement the mechanism described above, we create a new swap device for the supplementary-snapshot-image and reclaim pages in the "shrink memory" stage until the number of reclaimable pages reaches zero. We define this mechanism as "full page frame reclamation." As shown in Figure 4, supplementary-snapshot-image, backed up to the file, or dropped.[3] Among the remaining parts, we can exclude the unnecessary parts such as the kernel code[4] using the `register_nosave_region()`. As a result, the essential-snapshot-image includes a minimal number of pages, and we can enter the running state simply by restoring the essential-snapshot-image. Other pages requested by users will be restored on demand by the Linux memory management mechanism.

Smartphones require many processes that must be restored right after boot up, such as idle screens and other service daemons, causing natural delays after boot up. At the moment of boot up, many pages are swapped in for initial running. To improve performance, the supplementary-snapshot-image can be split up and

[3]Before entering hibernation, all pages in the other swap partition must be swapped in

[4]Kernel code is already included in the original kernel image that is loaded by the bootloader.

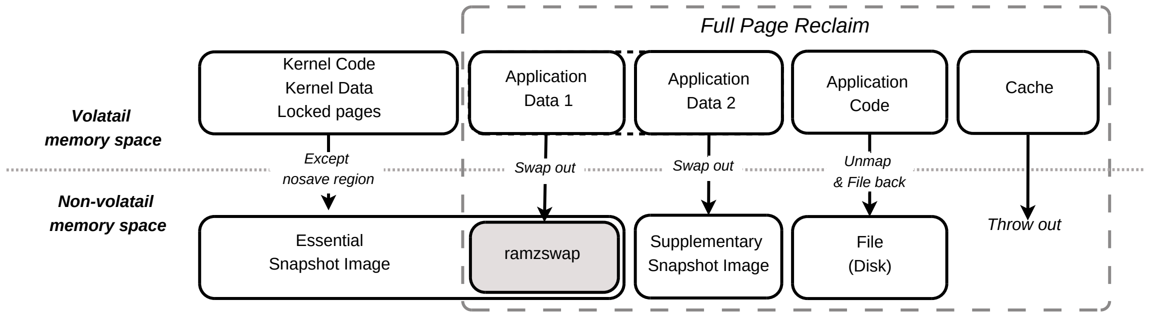


Figure 4: Making a snapshot image

stored in separate swap memory areas: ramzswap[13] and flash/disk swap. Because ramzswap is a RAM based block device, the read performance of ramzswap is better than other swap memory areas. If pages that must be restored right after boot up are stored in the ramzswap area, users only rarely perceive the latency. However, when making the snapshot image, the ramzswap partition is included in the essential-snapshot-image because it is a part of the kernel area. The method chosen to divide the supplementary-snapshot-image is important to improve performance after boot up. An easy method to achieve this is to make the flash/disk swap partition first and the ramzswap partition later. At the "full page reclaim" stage, inactive pages are reclaimed first and active pages are reclaimed later. Therefore, most, or all, inactive pages are stored in the flash/disk swap partition first, and the rest of the pages, including active pages, are stored in the ramzswap partition.

4.3 Fast Device Reactivation

In smartphones, sleep mode, or suspension to RAM (STR), is a necessary implementation because standby time is much longer than actual used time. When a smartphone goes into sleep mode, processes are frozen and devices are suspended to save power. The waking up process reverses the sleep process. A notable characteristic of this mechanism is that the suspended device information is backed up to memory before entering sleep mode, and then restored from memory when woken up by external stimuli.

In our approach, we store suspended device information in the essential-snapshot-image when entering hibernation. This is different from STR because the alive and non-alive block information for the processor are both stored in our approach, while STR stores only non-alive block information. When restoring from hibernation, we place the peripherial devices into resumable state based on the stored information instead of the initializing and suspending stages. However, some devices, including some block subsystems and some block devices that are used for loading snapshot image or devices that require special initializations, should be initialized.

To implement this mechanism, three new initcall sections are added: "early subsystem initcall," "early device initcall," and "resume initcall," as shown in Figure 3. "Early subsystem initcall" and "early device initcall" are required to initialize necessary devices that are used to restore devices from hibernation or to initialize devices that require special initialization. At the "resume initcall" section, the kernel performs the rest of the resume sequence in `software_resume()`.

In fact, the "initall 4∼7" sections[5] are the most time consuming parts of the normal booting sequence because it includes lots of delay routines. If there are many devices, or many kinds of devices, involved, then boot time will be increased. As a result, the time required for those sections is decreased with our mechanism.

Our mechanism operates via simple re-ordering of part of the subsystem/device initializing sequence, so it can continue to work in case of normal boot up, as shown in Figure 5. This mechanism can be used when restoring at the bootloader level, like the snapshot boot technique with a simple modification. When restoring at the kernel level, the technique is more generic and does not require additional management costs. The trade-offs between boot time and management cost can be minimized by manipulating the features of the system. More details about these trade-offs are discussed in section 6.

[5]initcall 4∼7 has "subsystem initcall," "device initcall," and "rootfs/fs initcall."

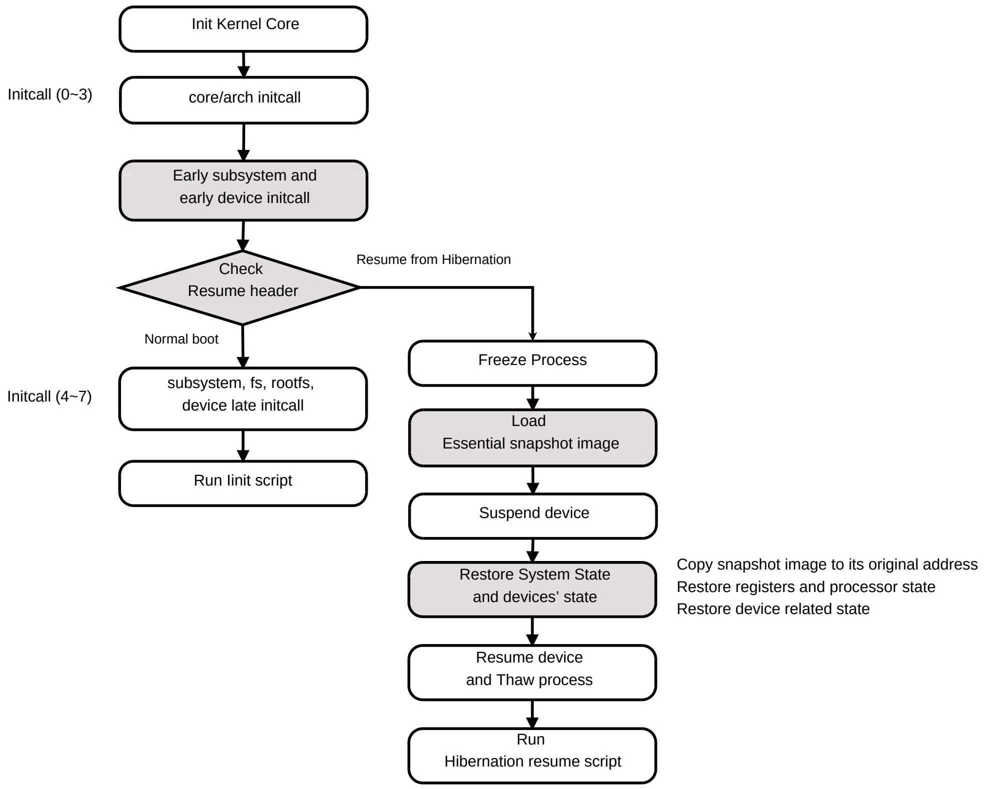

Init Kernel Core
Initcall (0~3)
core/arch initcall
Early subsystem and early device initcall
Check Resume header
Resume from Hibernation
Normal boot
Initcall (4~7)
subsystem, fs, rootfs, device late initcall
Run Iinit script
Freeze Process
Load Essential snapshot image
Suspend device
Restore System State and devices' state
Copy snapshot image to its original address
Restore registers and processor state
Restore device related state
Resume device and Thaw process
Run Hibernation resume script

Figure 5: Our mechanism - booting sequence flow

4.4 Resolving Inconsistency Problems

The original purpose of hibernation is not for booting but for restoring, and in such cases the snapshot image is used only once. However, for hibernation-based booting, if the snapshot image is created newly at every bootup the life of flash memory may decrease due to frequent I/O operation. In addition, it is difficult to produce a snapshot image representing the state of a system right after boot-up, and it takes a long time to do so. Situations of power failure situation must also be considered when entering hibernation.

However, the keep-image mode[6] results in inconsistency problems, because the information in storage can be changed anytime. TuxOnIce recommends following two methods to resolve inconsistency problems. The first is using a read-only file system. The second is to unmount the file system before entering hibernation and re-mount the file system after restoration from hibernation. However, in the real world, such constraints may be unacceptable. So, we tried the other way to resolve this problem. The way is updating superblock[7] and inodes[8] in memory when restoring from hibernation, forbidding to modify inodes included in a snapshot image after restoration from hibernation.

In keep-image mode, SIM[9] or database information inconsistency problems and changes of user configurations must also be considered for implementations. Modem devices use external storage such as SIM card, and information saved in a SIM card, or the SIM card itself, can be changed anytime. Some service daemons like alarm must be reinitialized according to configuration changed by user. Therefore, proper synchronization is required after boot up from hibernation.

5 Experiments

A smartphone based on Linux kernel 2.6.29 is used for this experiment. This smartphone has a Samsung S5PC110 CPU and a Cortex A8 processor, 512MB of Flash memory and 384MB of DRAM. A UBI[15] and

[6]Using the same snapshot image for every boot-up is referred to as keep-image mode in TuxOnIce.

[7]A structure representing the underlying filesystem

[8]The objects that represent the underlying files

[9]Subscriber Identity Module

three UBIFS[16] are used as the file system, and the LCD resolution is 400x800. The I/O performances of the Flash memory are 20MB/s for read and 3MB/s for write. Before the experiment, we implemented swsusp for ARM Cortex A8 because the kernel does not support the software suspend mechanism for ARM. The ACPI code lines were disabled because ARM does not support ACPI.

We make the snapshot image in IDLE screen view right after boot up, and keep it in the swap partition. Before making the snapshot image, we mark some regions as nosave_region to exclude them from the snapshot image, such as the kernel code, a portion of the frame buffer, the sound buffer, and the reserved region for the camera and 3D using `register_nosave_region()`. Every boot up, the same snapshot image is loaded to measure the performance.

5.1 Full Page Reclamation

Table 1: Boot time - Full Page Reclamation

Category		Time(ms)
	initialization	597
Bootloader	kernel image loading[a]	270
	go kernel	27
	Kernel core init[b]	214
	initcall 0 ∼ 3	37
	initcall 4 ∼ 7	3,749
	prepare resume	12
kernel	snapshot image loading[c]	741
	device suspend (all)	236
	copy memory to original	77
	resume device and thaw process	453
Total		**6,413**

[a] Size of kernel image = about 5.5MB
[b] Include 100ms of calibrating delay
[c] Size of snapshot image = 15MB

Before applying "full page frame reclamation", the size of snapshot image is 120MB, and loading time alone takes about 6 seconds. After applying "full page frame reclamation", we obtain a 15MB essential-snapshot-image and a 50MB supplementary-snapshot-image. As a result, we can reduce the size of the snapshot image about 87.4%, and the loading time is dramatically reduced to 0.75 second. We measure the time for each stage using a hardware (H/W) timer, and Table 1 shows the boot time when only "full page frame reclamation" is applied. Total boot time is 6.4 seconds, but there is some delay required in initial operation to load the supplementary-snapshot-image on demand. If 10MB of ramzswap partition is applied, it will require an additional 494ms of boot time to load 10MB of ramzswap partition, but the user will only rarely perceive the latency.

5.2 Fast Device Reactivation

Table 2: Boot time - Full Page Reclamation and Fast Device Reactivation

Category		Time(ms)
	initialization	597
Bootloader	kernel image loading[a]	270
	go kernel	27
	kernel core init[b]	214
	initcall 0 ∼ 3	37
	early subsystem initcall early module initcall	59
Kernel	prepare resume	7
	snapshot image loading[c]	741
	device suspend (partial)	35
	copy memory to original	61
	resume device and thaw process	492
Total		**2,540**

[a] Size of kernel image = about 5.5MB
[b] Include 100ms of calibrating delay
[c] Size of snapshot image = 15MB

According to Table 1, restoring from hibernation is started after 4.894 seconds. The most time-consuming task is initcall 4∼7, because the smartphone used in this experiment includes many peripheral devices. In the "fast device reactivation" technique, we add the "early system init" and "early device init" sections before resume. The "early system init" includes a memory technology device (MTD) and block I/O subsystem initialization, and the "early device init" includes flash device initialization. Initialization of the power management chip is added to the "early device init." As shown in Table 1, restoring from hibernation is started after 1.204 seconds with the "fast device reactivation" technique. As a result, we achieve boot up within 3 seconds when applying both "full page frame reclamation" and "fast device reactivation". The time required for the "device suspend" stage is reduced by about 85%. By extension, we can compare these results with results for restoring the bootloader level. If the snapshot image is loaded at the bootloader level, we can skip some tasks – kernel image loading (270ms), go kernel (27ms), kernel core init

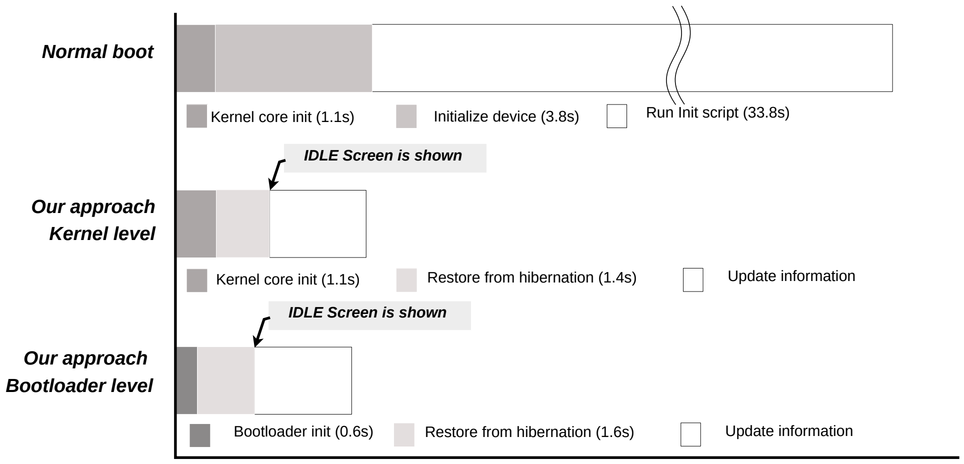


Figure 6: Estimated boot time for each technique

(214ms), initcall 0∼3(37ms), early subsystem/module initcall (59ms), prepare resume (7ms), device suspend (35ms), and copy memory to original (61ms). The total time requied for these tasks is 610ms, not including the time for calibrating the delay (100ms), but we must add 206ms because the loading kernel image includes kernel data as well as kernel code. In other words, the total reduction from loading the snapshot image in the bootloader is under 0.5 second. Further details about applying our approach at the bootloader level are discussed in the next section.

5.3 Resolving Inconsistency Problems

We add a file system recovery stage after boot from hibernation to solve the file system inconsistency problem. The file system recovery stage includes following operations: UBI re-scanning, updating UBIFS superblock in memory, updating inodes in memory. Our current implementation does not include forbidding modifications for inodes which are included in essential-snapshot-image yet, and it is left as our future work. As a result, 2.4 seconds[10] are added after boot from hibernation to recover the file system, but it can be improved by improving the UBI re-scanning method.

To resolve the modem service inconsistency problem, we simply stop the modem service daemon before making the snapshot image. After boot from hibernation, we execute the modem service daemon to synchronize with the modem device. For other service daemons like alarm daemon, we publish an update notification message using inotify to force them to update their information.

[10]The UBI re-scanning operation requires 1.8 seconds for 512MB memory and updating UBIFS superblock requires 0.6 seconds.

5.4 Estimation

Figure 6 shows a comparision of boot times between normal boot mechanisms and our improved mechanisms. While a normal boot takes about 40 seconds, we visualize the idle screen within 3 seconds and total boot time does not exceed 6 seconds with our mechanism. If the approach is applied at the bootloader level, we realize an additional reduction of 0.5 seconds.

6 Discussion

To apply our mechanism at the bootloader level, some functions must be implemented in the bootloader which are already implemented in the kernel: snapshot image loading, initializing some devices, and some other functions. As a result, applying these mechanisms at the bootloader level can eliminate another 0.5 seconds of boot time. Although the bootloader level approach require additional implementation and management, the required works are much less than the snapshot boot. This result suggests that there is a trade-off between boot time and management cost.

7 Conclusions and Future Work

This paper introduce two mechanisms: "full page frame reclamation," which minimizes the sizes of snapshot images, and "fast device reactivation," which improves device reactivation flow. As a result, we designed a platform independent mechanism that can be easily applied to Linux-based software platforms and eventually obtained instant boot time in a Linux based smartphone. We also considered some issues stemming from keep-image-modes. Obviously, some obstacles still remain to applying these mechanisms to commercial products, such as showing splash, and some other inconsistency problems. However, we believe that these issues may be overcome with proper user workflow and careful verification.

References

[1] Tim R. Bird, "Methods to Improve Bootup Time in Linux," In Proc. of the Linux Symposium, 2004.

[2] A. Leonard Brown, Rafael J. Wysocki, "Suspend-to-RAM in Linux," In Proc. of the Linux Symposium, 2008

[3] Hiroki Kaminaga, "Improving Linux Startup Time Using Software Resume," In Proc. of the Linux Symposium, 2006

[4] Heeseung Jo, Hwanju Kim, Hyun-Gul Roh, and Joonwon Lee, "Improving the Startup Time of Digital TV," IEEE Transactions on Consumer Electronics, Volume 52, Issue 2, May 2009.

[5] CELF - Boot Time, `http://eLinux.org/Boot\_Time`

[6] Uncompress Kernel, `http://elinux.org/Uncompressed\_kernel`

[7] Fast Kernel Decompression, `http://elinux.org/Fast\_Kernel\_Decompression`

[8] Disable console, `http://elinux.org/Disable\_Console`

[9] Preset LPJ, `http://elinux.org/Preset\_LPJ`

[10] Deferred Initcalls, `http://elinux.org/Deferred\_Initcalls`

[11] TuxOnIce (suspend2), `http://www.tuxonice.net/`

[12] Bootchart, `http://www.bootchart.org/`

[13] Ramzswap, `http://code.google.com/p/compcache/`

[14] Android, `http://www.android.com/`

[15] UBI, `http://www.linux-mtd.infradead.org/doc/ubi.html`

[16] UBIFS, `http://www.linux-mtd.infradead.org/doc/ubifs.html`

[17] Samsung, `http://www.samsung.com/global/business/semiconductor/`

[18] UCL, `http://www.oberhumer.com/opensource/ucl/`

Implementing an advanced access control model on Linux

Aneesh Kumar K.V
IBM Linux Technology Center
aneesh.kumar@linux.vnet.ibm.com

Andreas Grünbacher
SUSE Labs, Novell
agruen@suse.de

Greg Banks
gnb@fmeh.org

Abstract

Traditional UNIX-like operating systems use a very simple mechanism for determining which processes get access to which files, which is mainly based on the file mode permission bits. Beyond that, modern UNIX-like operating systems also implement access control models based on Access Control Lists (ACLs), the most common being POSIX ACLs.

The ACL model implemented by the various versions of Windows is more powerful and complex than POSIX ACLs, and differs in several aspects. These differences create interoperability problems on both sides; in mixed-platform environments, this is perceived as a significant disadvantage for the UNIX side.

To address this issue, several UNIXes including Solaris and AIX started to support additional ACL models based on version 4 of the the Network File System (NFSv4) protocol specification. Apart from vendor-specific extensions on a limited number of file systems, Linux is lacking this support so far.

This paper discusses the rationale for and challenges involved in implementing a new ACL model for Linux which is designed to be compliant with the POSIX standard and compatible with POSIX ACLs, NFSv4 ACLs, and Windows ACLs. The authors' goal with this new model is to make Linux the better UNIX in modern, mixed-platform computing environments.

1 Introduction

File access control is concerned with determining which activities on file system objects (files, directories) a legitimate user is supposed to be permitted. It mediates attempts to access files, and allows or denies them based on administrative metadata attached to those files.

Linux has traditionally had a file access control model based on the traditional UNIX file mode model standardised by POSIX [7]. This model is proven, robust and simple. Beyond that, Linux implements the non-standard but widely deployed POSIX ACL model suitable for more complex permission scenarios, all within the bounds that the POSIX standard defines.

However, when a Linux system uses a remote filesystem access protocol like CIFS or NFSv4 to share files with a Microsoft Windows system, the mismatch between the Linux and Windows access control models poses a significant interoperability challenge. This is a very common and economically significant deployment scenario, and other UNIX-like systems (Solaris [10] and AIX [1]) have a solution to these problems.

In this paper, we discuss the challenges involved in implementing an advanced access control model on Linux which is designed to address these problems. The new model is based on the NFSv4 ACL model [6], with design elements from POSIX ACLs that ensure its compliance with the standard POSIX file permission model.

The NFSv4 ACL model is in turn based, somewhat more loosely, on the Windows ACL model [3]. This means the mapping between the new model and Windows ACLs, while not completely trivial, is at least predictable, understandable, and not lossy. This has the benefit of smoothing remote file access interoperability.

Another benefit is to provide Linux system administrators with an access control model for local filesystems which is finer-grained and more flexible than the traditional POSIX model. Some system administrators might also find the new model more familiar.

For compatibility and security, it is necessary to ensure that applications using the traditional POSIX file mode based security model still work when using a filesystem which implements the new ACL model. This results in a number of technical challenges whose solution we will describe.

2 File Permission Models

This section describes and compares the main file permission models in use today: the standard POSIX file permission model and the widely supported POSIX ACLs on the UNIX side, Windows ACLs on Windows, and NFSv4 ACLs, a hybrid between these two major approaches.

2.1 The POSIX File Permission Model

The traditional file permission model implemented by all UNIX-like operating systems including Linux follows the POSIX.1 standard [7]. The standard can be thought of as a "contract" between application programs and the operating system: POSIX.1 defines the mechanisms available to portable applications and specifies how compliant operating systems will react. Application programs can rely on the POSIX.1 behaviors; this is of major importance to system security.

POSIX.1 distinguishes between read (r), write (w), and execute/search (x) access. The read permission allows to read a file and directory, the write permission allows to write to a file and create and delete directory entries, and the search/execute permission allows to execute a file and access directory entries.

As explained in POSIX.1 Base Definitions, each file system object is associated with a user ID, group ID, and a file mode which includes three sets of file permission bits. A process that requests access to a file system object is classified into one of the three categories owner, group, and other depending on its effective user ID, effective group ID, and supplementary group IDs. This so-called file class determines which set of permissions determine if the requested access is granted. The access is granted if the set of permissions associated with the file class includes the permissions needed for the requested access, and otherwise denied. Figure 1 depicts this graphically.

To give an example, assume that a process tries to open a file for read access and the file permission bits as shown by the `ls` command are `rw-r-----`, granting read and write access to the owner class and read access to the group class and no access to the other class. The access is granted if the effective user ID of the process matches the user ID of the file, or if the effective group ID or any of the supplementary group IDs of the process match the group ID of the file; in all other cases, the access is denied.

The file permission bits are usually set so that the group class has the same or fewer permissions than the owner class, and the other class has the same or fewer permissions than the group class. However, other values like `-w-r-----`, which grants write access to the owner class and read access to the group class, can also be used. Because a process can only be in one file class at any one time, these file permission bits do not allow any process to get read and write access simultaneously.

While this model is flexible enough for a large number of real-world scenarios, the three permissions and three possible roles of processes can become a burden or be too limiting. For example, when people form an ad-hoc team which the operating system does not know about, they will have difficulties with sharing files in this team: the file permission model will not allow them to grant each other access to files without granting others outside this group access as well. The system administrator can help by creating a new group, but this administrative overhead is undesirable, and the number of groups can grow unreasonably large.

In awareness of these limitations, the POSIX.1 standard defines that the file group class may include other implementation-defined members, and allows additional and alternate file access control mechanisms:

- Additional file access control mechanisms may only further restrict the access permissions defined by the file permission bits.
- Alternate file access control mechanisms may restrict or extend the access permissions defined by the file permission bits. They must be enabled explicitly on a per-file basis (which implies that no alternate file access control mechanisms may be enabled for new files), and changing a file's permission bits with the `chmod` system call must disable them.

Many texts on the UNIX operating system describe the POSIX.1 file permission model in more detail including Advanced Programming in the UNIX(R) Environment [15].

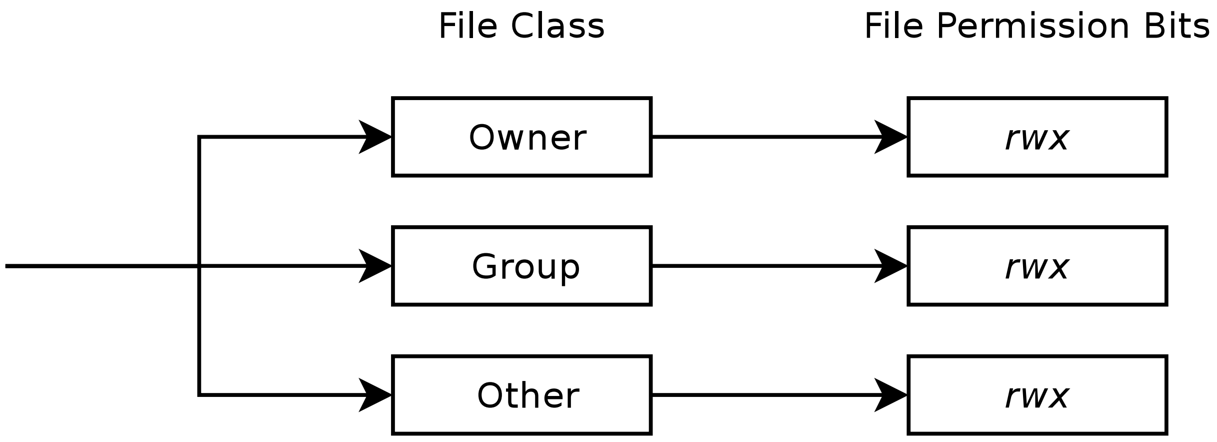


Figure 1: The POSIX.1 File Permission Model

2.2 POSIX.1e Access Control Lists

As we have seen in the previous section, the POSIX.1 file permission model only uses the file user ID and file group ID to distinguish between users; there is no way to grant permissions to additional users or groups. POSIX Access Control Lists (ACLs)[1] remove this restriction. Each file system object is associated with a list of Access Control Entries (ACEs), which define the permissions of the file owner ID, the file group ID, additional users and groups, and others.

In the usual POSIX ACL text form, the `user::` entry stands for the file user ID, the `group::` entry stands for the file group ID, and the `other::` entry stands for others. Further, `user:<name>:` entries stand for additional users and `group:<name>:` entries stand for additional groups with the specified names.

In POSIX.1 terms, POSIX ACLs are an additional file access control method. The Working Group has also made use of the provision that the file group class may include other implementation-defined members by assigning the additional user and group entries to this class. This raises the following questions:

1. As an additional file access control mechanism, POSIX ACLs may only further restrict the access permissions defined by the file permission bits. But since the additional user and group entries are members of the file group class, what if they grant permissions beyond the file group class permissions?

   The Working Group has answered this question by defining that the file group class permissions act as an upper bound or "mask" to the group class. An entry in the group class may include permissions which are not in the file group class permissions, but only permissions which are in the entry as well as in the file group class permissions are effective.

2. If the file group class permissions continue to define the permissions of the file group ID, how can additional users and groups be granted more permissions than the file group ID, since by definition of additional file access control mechanisms, the file group class cannot have permissions beyond the file group class permissions?

   This question has been answered by defining that the file group class permissions no longer define the file group ID permissions. Instead, in POSIX ACLs, the `group::` entry stands for the file group ID, and the new `mask::` entry stands for the file group class.[2]

   The file group ID entry remains a member of the file group class.

Figure 2 shows the relationship between file classes, ACEs, and the file permission bits: the file owner class contains exactly one ACE, the file group class contains

[1]POSIX.1e [8] was never ratified as a standard. The POSIX ACL implementations found on UNIX-like operating systems are based on drafts of the POSIX.1e Working Group.

[2]If an ACL contains no entries for additional users or groups, the group class only contains a single entry. In this case, the Working Group has defined that the `group::` entry shall continue to refer to the file group class permission bits and no `mask::` entry shall exist, resulting in the same behavior as without POSIX ACLs.

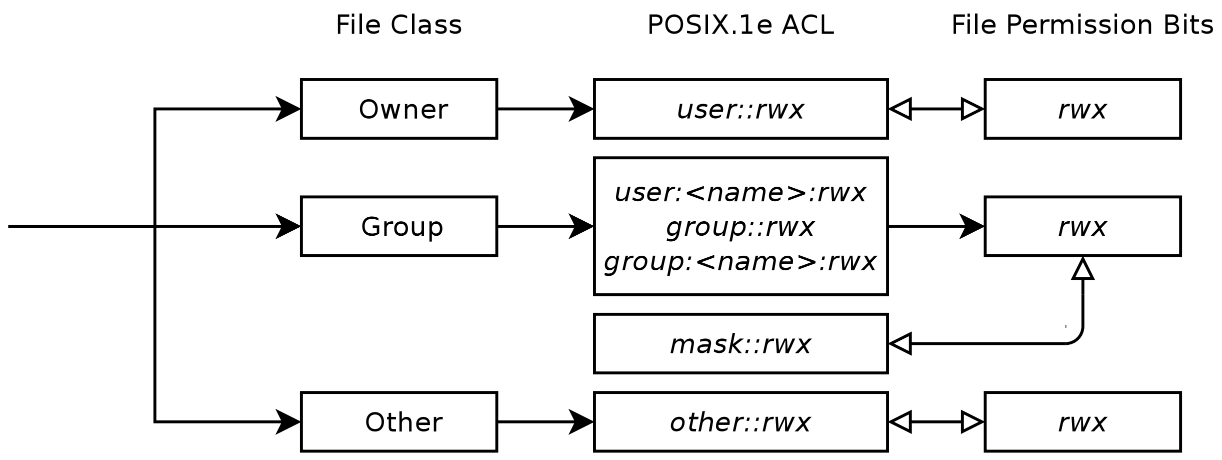


Figure 2: POSIX.1e Access Control Lists

one or more ACEs, and the file other class again contains exactly one ACE.

The open-headed arrows in Figure 2 show how ACEs and the file permission bits are kept in sync: per definition, the `user::` entry and owner class, `mask::` entry and group class, and `other::` entry and other class file permission bits are kept identical; changing the ACL changes the file permission bits and vice versa.

As an example, consider the following POSIX ACL as shown by the getfacl utility (line numbers added by hand):

```
1  # file: f
2  # owner: lisa
3  # group: users
4  user::rw-
5  user:joe:rwx       #effective:rw-
6  group::r-x         #effective:r--
7  mask::rw-
8  other::---
```

The first three lines indicate that the file is called `f`, the file user ID is `lisa`, and the file group ID is `users`. Line 4 shows that the owner, Lisa, has read and write access. Line 5 shows that Joe would have read, write, and execute access, but the file mask in line 7 forbids execute access, so Joe effectively only has read and write access. Line 6 shows that the group Users would have read and execute access, but effectively only has read access. Finally, line 8 shows that others have no access.

In addition to these “normal” ACLs, POSIX.1e also defines so-called default ACLs which have the same structure as “normal” ACLs. When a file system object is created in a directory which has a default ACL, the default ACL defines the initial value of the object’s “normal” ACL and, if the new object is a directory, also the object’s default ACL. Default ACLs have no effect after file creation.

2.3 Windows ACLs

Before Windows NT, Windows did not have an ACL model and permissions could only be attached to an entire exported directory tree (aka share). Other companies tried to fill this gap by inventing additional file permission schemes [14]. With the introduction of the NTFS filesystem in 1993, Microsoft introduced a new ACL based file permission model, commonly referred to as Windows ACLs or CIFS ACLs. Its key properties are:

- Windows ACLs are used to control access to a variety of OS objects in addition to filesystem objects, e.g. mutexes, semaphores, window system objects, and threads. Here we are concerned only with ACLs on filesystem objects.
- Controlled Windows objects have two ACLs, the DACL (Discretionary ACL) and SACL (System ACL). The SACL can only be edited by privileged users, and is used to implement logging of file accesses (or failures to access). Here we are concerned only with DACLs.
- Each filesystem object is owned by a Security Identifier (SID), a variable-length unique binary identifier which can refer to either a user or a group.

- Each Windows ACE contains a mask of 14 different permission bits, see Table 1 for a summary and File and Folder Permissions [2] on Microsoft TechNet for details.

- Windows ACEs are one of three types: Access-Denied (used in a DACL to explicitly deny access), Access-Allowed (used in a DACL to explicitly grant access), and System-Audit (used in a SACL to cause an entry to be made in the system security log). We shall refer to these by the short forms `DENY`, `ALLOW`, and `AUDIT` respectively.

- The order in which permissions are granted and denied in Windows ACLs matters. ACEs are processed from top to bottom until all requested permissions have been granted, or a requested permission has been explicitly denied.

- Each ACE contains a SID which identifies the user or group that the ACE refers to. There is no way to tell user from group ACEs. There are also some special SIDs with special semantics.

- The owner of a file can be explicitly mentioned in an ACE, and implicitly mentioned in an inherit-only ACE.[3] However, there is no way to construct an ACE that always applies to the file's current owner even if the owner is changed.[4]

- A Windows ACL entry can apply to the special SID `Everyone`, which includes the owner and all users and groups explicitly mentioned in other ACL entries. The POSIX.1 concept of file classes, where a process is classified into the owner, group, and other classes and cannot obtain permissions which go beyond the permissions of its class, does not exist, but `DENY` entries can be used to explicitly deny some permissions.

- Windows supports inheritance of permissions at file create time, and since Windows 2000, a feature called "Automatic Inheritance." With Automatic Inheritance, changes to the permissions of a directory can propagate to all files and directories below that directory. Create time inheritance and Automatic Inheritance use as a number of ACE flags (`INHERITED_ACE`, `INHERIT_ONLY_ACE`, `CONTAINER_INHERIT_ACE`, `OBJECT_INHERIT_ACE`, and `NO_PROPAGATE_INHERIT_ACE`).

[3]using the special *Creator Owner* SID

[4]not even using the special *Owner Rights* SID which appeared in Windows Vista [4]; such ACEs are actually disabled when the file's ownership changes.

2.4 NFSv4 ACLs

Up to version 3, the NFS protocol was mainly UNIX oriented, and its file permission model was limited to exposing POSIX file modes (in NFS terminology: the mode attribute) over the network. This created the implicit assumption of POSIX-like behavior.

Version 4 broke with the protocol's legacy and introduced a new ACL model based on Windows ACLs. The mode attribute was initially deprecated in favor of ACLs; more recent updates to the protocol re-endorsed the mode attribute and clarified some of the interactions between mode and ACLs (but some inconsistencies still remain).

The key properties of the NFSv4 ACL model are:

- NFSv4 [5] supports the same 14 permissions as Windows. NFSv4.1 [6] adds two additional permissions, `ACE4_WRITE_RETENTION` and `ACE4_WRITE_RETENTION_HOLD`, which have no equivalent in Windows or POSIX ACLs.

- The `ALLOW` and `DENY` ACE types are supported, and optionally also the `AUDIT` and `ALARM` types.

- The NFSv4 and Windows permission check algorithms are equivalent.

- NFSv4 uses principal strings modelled on Kerberos for identifying users and groups, e.g. `fred@example.com`. User and group ACEs are distinguised by an ACE flag.

- Each file system object is owned by a user, and has an owning group. The special `OWNER@` principal refers to the current owner, and the special `GROUP@` principal refers to the current owning group of a file, even when the owner or owning group changes.

- The special `EVERYONE@` principal refers to Everyone, which includes the owner and all users and groups explicitly mentioned in other ACL entries.

- NFSv4 recognizes that there is a relationship between the mode attribute and the ACL, but instead of connecting this to the POSIX file permission model and sticking to the same requirements, NFSv4.1 makes up its own special rules for updating the ACL when the mode changes, and vice versa. These rules are not fully compatible with POSIX.1 or POSIX.1e.
- NFSv4 supports inheritance of permissions at file create time. NFSv4.1 adds support for Automatic Inheritance.

2.5 ACL Model Differences

The various ACL models differ in a number of important details.

- POSIX1.e entries can only allow access, i.e. `ALLOW` semantics. Windows and NFSv4 entries can either `ALLOW` or `DENY` access. This provides considerably more expressive power (albeit at the cost of complexity).
- The order of evaluation of POSIX.1e entries is not significant, as all the entries are additive. In Windows and NFSv4 ACLs, entries can deny access, and thus their order is significant.
- The permission bits in POSIX.1e entries are quite simple, with only 3 bits defined. Windows has a total of 14 permission bits, most of which affect a smaller number of actions. NFSv4 follows Windows closely (even when the permission bits make no sense), but NFSv4.1 adds two more permission bits (`ACE4_WRITE_RETENTION_HOLD` and `ACE4_WRITE_RETENTION`) which have no equivalent anywhere else.
- POSIX.1e and NFSv4 ACEs have state which determine whether the ACE refers to a user or a group, because in POSIX these are strictly different namespaces. Windows ACEs contain only a SID, which might refer to either a user or a group.
- The POSIX.1e model reuses the POSIX.1 concept of process file classes, where a process is classified into the owner, group and other classes. The Windows model has no precise equivalent to any of these classes. A Windows ACE can apply to Everyone, but unlike the POSIX "other" class, that includes the owner and all users explicitly mentioned in other ACEs. The NFSv4 model compromises between the two, defining special `OWNER@` and `GROUP@` principals with the POSIX semantics, and `EVERYONE@` with the Windows semantics.
- The possible existance of `DENY` entries in Windows and NFSv4 models, and the differences in file class boundaries, also complicate the permission algorithm compared to POSIX.1e. The more complex algorithm must track both allowed and denied masks.
- The POSIX.1e model allows for inheritance of ACLs from a parent directory at the time when a filesystem object is created, using a separate "default ACL" stored on the parent directory. By contrast, the Windows, NFSv4 models combine the actual and default ACLs into one, with extra flags on each ACE to indicate whether it is to be inherited or not, and whether it is to be used only for inheritance.
- The POSIX.1e model has no explicit support for recursive modifications of ACLs on existing trees; like `chmod-R` this is expected to proceed entirely in a userspace utility which recurses over a tree and modifies ACLs. The Windows and NFSv4 models also rely on a userspace utility but have more complex ACE propagation algorithms (Automatic Inheritance) which need some extra bits stored on the ACL. The NFSv4 standard forgot these bits, and they only appear in NFSv4.1.
- In addition to the `ALLOW` and `DENY` types of entries, the Windows and NFSv4 models allow for `AUDIT` and `ALARM` entries. In Windows, these entries trigger system management side effects. In NFSv4, their meaning is undefined.
- The POSIX.1e model identifies users and groups using traditional UNIX user and group ID numbers. The Windows model uses SIDs. which are something like binary hierarchically scoped user and group IDs and provide a single namespace for both users and groups. The NFSv4 model uses principal strings modelled on Kerberos, e.g. `fred@example.com`.

3 Why We Need a New ACL Model

There are two main reasons why we consider new ACL model for Linux to be necessary.

Firstly, while POSIX.1e is the current default ACL model on Linux, and works well, its power and expressiveness is limited by its small set of permission bits and additive `ALLOW`-only semantics. Subtractive concepts like "grant read permission to all of the accounting department, but not to the trainees" are difficult to achieve and have to be painfully approximated. Windows can express these neatly and concisely, and we feel this will be useful to Linux system administrators.

Secondly, in a file serving scenario, there is a significant interface mismatch between POSIX ACLs and the ACL models expected or provided by Windows systems. This mismatch makes interoperability between Linux and Windows machines unnecessarily difficult, requiring the Linux-side software to perform complex, lossy, and potentially insecure mappings backwards and forwards between the models.

When files are available through different channels (e.g. Samba and NFS on the same Linux server) with different approaches to the ACL issue, there is the potential for users to be able to bypass intended access controls.

Even in the homogeneous case when a Linux client mounts a filesystem via NFSv4 from a Linux server, and both Linux systems understand POSIX ACLs, the ACL model enforced by the NFSv4 protocol requires two difficult mappings in order to transmit an ACL on the wire. The Linux NFS client presents ACLs using different formats and utilities than those used for the local filesystem on the Linux server, which leads to unnecessary confusion.

We need a solution which makes the ACL models of the client, the server and the protocol as similar as possible, so that mappings between them are much easier and safer. It should also provide a single point of ACL enforcement for all protocols and for local applications, and consistent management tools on the client and server.

4 Rich ACLs

To bring together the disparate models and address interoperability issues, we propose a new ACL mechanism for Linux called Rich-acl.

4.1 Design Principles

The proposed new ACL model uses NFSv4 ACLs at its core. Unlike NFSv4 and Windows ACLs, it identifies users and groups by their numeric UNIX IDs. This allows access decisions to be made for all Linux processes without having to translate identifiers to their local form first.

The `ALLOW` and `DENY` ACE types are supported; support for `AUDIT` and `ALARM` type ACEs might make sense to add in the future.

Rich-acl supports the same 14 permission bits as NFSv4 (three of which have a dual meaning and mnemonic for files and directories) plus the two additional write retention permissions of NFSv4.1. The permissions have the following meaning:

- `READ_DATA`, `WRITE_DATA`, `APPEND_DATA`: Read a file, modify a file, and modify a file by appending to it only.
- `LIST_DIRECTORY`, `ADD_FILE`, `ADD_SUBDIRECTORY`: List the contents of a directory, add files, and add subdirectories.
- `DELETE_CHILD`: Delete a file or subdirectory from a directory.
- `EXECUTE`: Execute a file, traverse a directory.
- `READ_ATTRIBUTES`: Read the stat information of a file or directory.
- `READ_ACL`: Read the ACL of a file or directory.
- `SYNCHRONIZE`: Synchronize with another thread by waiting on a file handle.
- `DELETE`: Delete a file or directory even without the `DELETE_CHILD` permission on the parent directory.
- `WRITE_ATTRIBUTES`: Set the access and modification times of a file or directory.
- `WRITE_ACL`: Set the ACL and POSIX file mode of a file or directory.
- `WRITE_OWNER`: Take ownership of a file or directory. Set the owning group of a file or directory to

the effective group ID or one of the supplementary group IDs.[5]

- `READ_NAMED_ATTRS`, `WRITE_NAMED_ATTRS`: Read and write Named Attributes. Named Attributes neither refer to Windows Alternate Data Streams nor to Linux Extended Attributes. These permissions will be stored, but have no further effect.

- `WRITE_RETENTION`, `WRITE_RETENTION_HOLD`: Set NFSv4.1 specific retention attributes. These permissions will be stored, but have no further effect.

Some of the Rich-acl permissions are a subset of a POSIX permission; others go beyond what the file permission bits can grant. Table 1 shows a complete mapping between Rich-ACL and POSIX permissions:

- The `READ_DATA` and `LIST_DIRECTORY` permissions map to the POSIX Read permission, the `WRITE_DATA`, `APPEND_DATA`, `DELETE_CHILD`, `ADD_FILE`, and `ADD_SUBDIRECTORY` permissions map to the POSIX Write permission, and the `EXECUTE` permission maps to the POSIX Execute/Search permission. These permissions fit the concept of an additional file access control mechanism.

- The `READ_ATTRIBUTES`, `READ_ACL`, and `SYNCHRONIZE` permissions are permissions which cannot be denied under POSIX. Denying these operations could cause problems with POSIX applications, so we always grant these permissions no matter what the ACL says (see section 5.4).

- The `DELETE`, `WRITE_ATTRIBUTES`, `WRITE_ACL`, and `WRITE_OWNER` permissions denote rights which go beyond the POSIX permissions, and the `READ_NAMED_ATTRIBUTES`, `WRITE_NAMED_ATTRIBUTES`, `WRITE_RETENTION`, and `WRITE_RETENTION_HOLD` permissions denote rights which have no equivalent in Linux. These eight permissions can only be enabled as part of an alternate file access control mechanism.

[5]Also see the setfsuid(2) and setfsgid(2) Linux manual pages.

In addition to defining how the permissions of the two models map onto each other, we need to define how processes are classified into the owner, group, and other classes; this determines which file permission bits affect which processes. We use the following rules analogously to POSIX ACLs:

1. Processes are in the owner class if their effective user ID matches the user ID of the file.

2. Processes are in the group class if they are not in the owner class and their effective group ID or one of the supplementary group IDs matches the group ID of the file, the effective user ID matches the user ID of an ACE, or the effective group ID or one of the supplementary group IDs matches the group ID of an ACE.

3. Processes are in the other class if they are not in the owner or group class.

Finally, POSIX requires that after creating a new file or changing a file's permission bits with the `chmod` system call, processes are not granted any permissions beyond the file permission bits of their file class. This requirement can be implemented in different ways:

1. The ACL can be replaced by an ACL which grants the equivalent of the file permission bits to the owner, the owning group, and others.[6]

2. The ACL can be changed so that it does not grant any permissions beyond the file permission bits. This may require removing permissions from ACEs. In addition, if the owner class has fewer permissions than the group class or the group class has fewer permissions than the other class, additional `DENY` ACEs may be needed.[7]

3. The ACL can be left unchanged; in this case, the access check algorithm must take both the ACL and the file permission bits into account, and only grant permissions which are granted by *both* mechanisms.

[6]NFSv4 ACLs on IBM GPFS and JFS2 do this, Sun/Oracle ZFS offers this as an option.

[7]SUN/Oracle ZFS tries to do this by default, but the documented behavior [10] does not always lead to the correct result.

Permission Bit	**POSIX Mapping**
`READ_DATA` (= `LIST_DIRECTORY`)	Read
`WRITE_DATA` (= `ADD_FILE`)	Write
`APPEND_DATA` (= `ADD_SUBDIRECTORY`)	Write
`DELETE_CHILD`	Write
`EXECUTE`	Execute/Search
`READ_ATTRIBUTES`	Always Allowed
`READ_ACL`	Always Allowed
`SYNCHRONIZE`	Always Allowed
`DELETE`	Alternate
`WRITE_ATTRIBUTES`	Alternate
`WRITE_ACL`	Alternate
`WRITE_OWNER`	Alternate
`READ_NAMED_ATTRS`	Alternate (No Effect)
`WRITE_NAMED_ATTRS`	Alternate (No Effect)
`WRITE_RETENTION`	Alternate (No Effect)
`WRITE_RETENTION_HOLD`	Alternate (No Effect)

Table 1: Mapping Between Rich-ACL and POSIX Permissions

We have chosen a variation of approach 3 for Rich-acls because it does not require complicated ACL manipulations, and is a consequent adaptation of the masking mechanism already found in POSIX ACLs, which has already proven itself.

The need for a variation to approach 3 becomes obvious when considering that the file permission bits are limited to the Read, Write, and Execute/Search permissions, and we would end up without a way to explicitly enable any of the alternate rich-acl permissions.

To get around this restriction, we introduce the new concept of file masks:

- Each file class (owner, group, and other) is associated with a file mask which contains a set of rich-acl permissions.
- When the file permission bits are changed, each file mask is set to the rich-acl permissions which correspond to the file permission bits of its class.
- The file masks can be changed explicitly to include alternate rich-acl permissions. Changing the file masks will also change the file permission bits.
- The access check algorithm grants an access if the rich-acl grants the access, and the file mask matching the process also includes the reuested rich-acl permisssions.

Figure 3 shows the relationship between file classes, the ACL, the file masks, and the file permission bits in rich-acls.

4.2 Specific Changes

To achieve the above principles, we made the following code changes.

- Modify the kernel VFS interface to allow filesystems optionally to exert more control over whether a process is allowed to create or delete filesystem objects. The rich-acl permission algorithm requires more information in these two cases than either the POSIX or POSIX.1e models do.
- Define a machine-independent binary encoding of a rich-acl ACL, and use the Linux extended attribute (xattr) mechanism to store encoded rich-acl ACLs on filesystem objects. The same encoding is used in the filesystem and on both the NFS client and server (which is not true of the current Linux NFS ACL code). The attribute used is `system.richacl`.
- Provide an in-kernel library for manipulating ACLs. This includes creating and destroying ACLs, performing permission checks, calculating a file mode from an ACL and applying a new mode

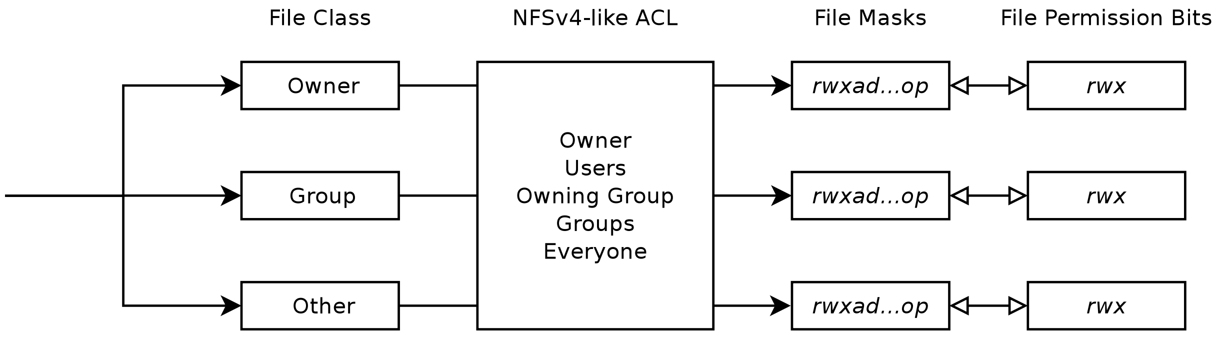


Figure 3: Rich Access Control Lists

to an ACL. and encoding an ACL to an xattr and decoding an ACL from a xattr.

- Use the kernel library to enhance the ext4 filesystem to store, retrieve and enforce the new permission model. A new ext4 superblock option `richacl`, settable with the `tune2fs` utility, is defined to control whether rich-acls are enabled.
- Use the kernel library to enhance the NFS client and server to store and retrieve rich-acl ACLs (enforcement is done in the server-side backing filesystem, not in NFS code). The server-side conversion between the rich-acl kernel in-memory format and the NFS wire format is much simpler than with POSIX.1e ACLs.
- Provide a userspace library for manipulating ACLs. It is similar to the kernel library, except that it does not provide a permission check algorithm.
- Use the userspace library to provide a command-line utility `setrichacl` to allow users to store, retrieve, and manipulate ACLs on files and directories (loosely equivalent to `chmod` and `ls`).

One of the advantages of this approach is consistency of use: the same tools can be used to examine and manipulate rich-acl ACLs on the NFS client and server, as well as for local applications.

Furthermore, with the rich-acl model ACLs are stored and enforced consistently and in one place: the server-side backing filesystem. This prevents users being able to evade access control by using different access techniques, such as logging into the server or using NFS instead of CIFS.

The code is available in two git repositories, kernel [12] and userspace [13]. There is also a patch for `tune2fs` [11].

5 Further Considerations

5.1 Standards

The text of the NFSv4 standard, as it applies to ACLs, has undergone several revisions and clarifications. The initial version appears to have been the result of a purely theoretical design exercise and not of implementation experience. Subsequent versions have had progressively fewer flaws and ambiguities, but some difficulties remain.

The behaviour of Windows ACLs is well documented by Microsoft. Sometimes the documentation is accurate; sometimes experiment is required to determine the true behaviour.

5.2 Multiple group entries

POSIX allows a user to be a member of multiple groups. The POSIX.1e model allows access if any one of the groups that the user belongs to, is allowed access. In contrast, rich-acl ACEs are processed in order. If the requested access mask bit matches the access mask bit present in a `DENY` ACE, then the access is denied. Each ACE is processed until all the bits of the requested access mask are allowed. This implies that when mapping a POSIX ACL to rich-acl we need to impose an ordering constraint on ACEs, such that ACEs which `ALLOW` access to any group must preceed ACEs which `DENY` access to any group.

5.3 OTHER vs EVERYONE@ ACEs

One of the major differences between the POSIX.1e model and the rich-acl model is that the rich-acl `EVERYONE@` includes both user and group classes, whereas the POSIX.1e `OTHER` class excludes user and group classes. When mapping a POSIX.1e ACL to a rich-acl, the `OTHER` ACE will be mapped to a trailing `ALLOW EVERYONE@` ACE, but to limit it's effect that ACE may need to be preceeded with one or more `DENY` ACEs which deny some access to specific groups.

5.4 Permissions which are always enabled

The NFSv4 ACL model has some permission bits which control actions which are always allowed under POSIX, such as `ACE4_READ_ATTRIBUTES` for reading file attributes `ACE4_READ_ACL` for reading the ACL itself. To limit impact on the Linux code (for example, by introducing new error cases in complex and critical code paths) and on existing POSIX applications we have chosen not to enforce these permission bits in the rich-acl model. In line with our design goals, ACEs which mention these permission bits will be accepted and stored, but the permission bits will have no effect.

One effect of this choice is that the NFS server will successfully complete a `SETATTR` operation which sets an ACL containing an ACE intended to `DENY` these permissions, despite not being able to accurately enforce the intended effect of the ACE. Such behaviour is prohibited by language in the NFSv4.1 RFC [6], which specifies that the NFS server must return the `NFS4ERR_ATTRNOTSUPP` error in this case.

Our experiments show that it is very easy for a user using the Windows permission editor GUI to set such an ACE, as an unintended side effect of the common idiom of denying another user the ability to read file data. This is due to the editor having "basic" and "advanced" modes; in the basic mode the user is presented with abstracted permissions like Read which are amalgams of multiple underlying permissions. So if our implementation were to strictly obey the RFC then Windows users would be unnecessarily inconvenienced and possibly Windows applications might be broken.

5.5 Sticky bit and capabilities

When describing the POSIX model above, we did not mention some of the more complex corners of the model. To meet our design goal of preserving expected POSIX behaviour, the rich-acl permission algorithm needs to take these into account.

The POSIX sticky(t) bit is used in the file mode of a directory to change the permission check for deleting files in the directory. It is usually employed to allow multiple users to share the `/tmp` directory in such a way that each user can delete only her own files. Such behaviour could be approximated with a well designed ACL on the `/tmp` directory; however our design goal meant that we need to preserve the behaviour of the sticky bit regardless of the presence of ACLs. A further reason is the security risk involved with perturbing the behaviour of `/tmp`.

POSIX defines capabilities `CAP_DAC_OVERRIDE` and `CAP_DAC_READ_SEARCH` which allow privileged processes to gain access regardless of the results of an access control check (with some limits). Another capability, `CAP_FOWNER`, allows privileged processes to gain access normally allowed only to the owner of an object and not subject to the POSIX DAC controls. Of course, `CAP_FOWNER` interacts with the implementation of the sticky bit.

5.6 Migration

Many filesystems using POSIX ACLs are already deployed. Therefore, in order to enable rich-acl on an existing Linux file system, we need to provide a mechanism for migrating the filesystem from existing POSIX.1e ACLs to rich-acl ACLs.

The current design has a two-step conversion process:

- In the first step, the kernel filesystem code constructs a temporary rich-acl ACL on the fly when an ACL on a filesystem object is required (for a permissions check or an xattr fetch), and the filesystem has the `richacl` option enabled with `tunefs`, and the object has no rich-acl ACL stored, and the object has a POSIX.1e ACL stored. Once the `richacl` option is enabled, objects in the filesystem appear to have both a POSIX.1e ACL and a functionally equivalent rich-acl ACL.
- The converted ACL is discarded after use, and not written back to the filesystem object. Hence we need a second step to complete the conversion: the

`setrichacl` utility reads the temporary rich-acl from the xattr and writes the same bytes back to the xattr, causing the filesystem to make the rich-acl permanent on disk. As a side effect of setting the rich-acl xattr, the kernel deletes the POSIX.1e ACL xattr.

The advantage of this technique is that the filesystem is immediately available with functioning rich-acls after the `richacl` option is enabled without requiring any modification of the on-disk metadata. This also allows experimentally enabling rich-acls on a filesystem in order to test application compatibility, enabling rich-acls on a readonly filesystem, and more easily fine-tuning the conversion algorithm used in the user-space utility used if needed.

Note that migrating a filesystem back from rich-acls to POSIX.1e ACLs is not supported once the rich-acls are made permanent on disk, as converting in that direction is usually lossy.

6 Open Issues / Future Work

Rich-acls are usable today by kernel developers and early adopters, but there is work remaining to be done.

- Use the userspace rich-acl library to enhance the Samba server to store and retrieve rich-acl ACLs (enforcement is done in the server-side backing filesystem, not in Samba code). The conversion between the rich-acl userspace in-memory format and the CIFS wire format is much simpler than with POSIX.1e ACLs. This could be based on the existing SGI [9] patch.
- Use the kernel rich-acl library to enhance the smbfs client to store and retrieve rich-acl ACLs.
- The `ls` program should indicate those filesystem objects which have rich-acls, for example by showing a + sign.
- The `find` program should be aware of ACLs, at the very least by providing a predicate which tests whether an object has a rich-acl. Even better would be predicates to test more subtle effects of rich-acls.
- The `setrichacl` utility should be updated to provide a convenient one-line command to perform the second step in the process of migrating a filesystem from POSIX.1e ACLs to rich-acl ACLs. In this mode it would traverse the filesystem tree, fetching the rich-acl xattr and setting it back again, causing the rich-acl ACL to become permanent on disk. Currently the program must be invoked twice per file or directory.
- The GNOME and KDE desktops need GUI applications written to allow users to display and edit rich-acls on filesystem objects without resorting to the command-line interface.
- The issue of user identity is not completely resolved. Linux, NFSv4 and Windows all use different unique identifiers for users. Rich-acls use Linux user ids, which require mapping to and from Windows SIDs or NFSv4 principals on demand. The mechanisms for such mappings can be awkward and slow. It may be useful to investigate storing SIDs and principals in the filesystem.
- It might be useful to implement Windows/NFSv4 SACLs and the `AUDIT` and `ALARM` ACE types.
- The Linux kernel NFSv4 server places a smaller limit on the maximum size of NFSv4 ACLs than does either the rich-acl implementation or the NFSv4 standard. Fixing this properly requires significant surgery to the NFSv4 XDR code.
- The `setrichacl` utility does not yet perform Automatic Inheritance.
- The POSIX API does not provide an interface for an application to atomically create a file or other filesystem object with a given set of extended attributes; this needs to be performed as two separate actions. Both the CIFS and NFSv4 protocols do however provide such an ability. This creates a race condition where a filesystem object may briefly have an unintended ACL and be less secure than the application expected. Such an interface could be provided to allow the Samba server to avoid the race.
- The POSIX `access` function and the NFSv4 `ACCESS` operation could be enhanced to allow a Rich-acl-aware application to test for more fine-grained access types.

- Windows Vista introduced a feature that allows to deny file owners the *Read Permissions* and *Change Permissions* permissions which they are otherwise implicitly granted [4]; support for this has not been implemented, yet.

7 Conclusion

The demand for improved interoperability in modern, mixed-platform environments is increasing over the years. On UNIX-like systems, the most widely available kind of ACLs is POSIX ACLs, and they will remain in that role for some time to come. Still, POSIX ACLs have proven unsuitable for addressing the interoperability challenges we are facing today.

In this paper we have discussed the many goals that a better, more interoperable ACL model should meet. The proposed new model meets those goals. A lot of work still remains to be done at all levels until end users will be able to reap the benefits, but all the key building blocks are there already.

8 Acknowledgements

The authors would like to thank IBM, Novell, and SGI for supporting this work and its predecessors at various times. We're also grateful for technical support from various members of the NFS Working Group, and the NFS and CIFS communities. David Disseldorp has reviewed this paper.

Legal Statement

References

[1] IBM Corp. Working with filesystems using NFSV4 ACLs. `http://www.ibm.com/developerworks/aix/library/au-filesys_NFSv4ACL/index%.html`.

[2] Microsoft Corp. File and folder permissions. `http://technet.microsoft.com/en-us/library/cc732880.aspx`.

[3] Microsoft Corp. How security descriptors and access control lists work. `http://technet.microsoft.com/en-us/library/cc781716.aspx`.

[4] Microsoft Corp. Security Identifiers (SIDs) new for Windows Vista. `http://technet.microsoft.com/en-us/library/cc749445%28WS.10%29.aspx`.

[5] IETF Network Working Group. RFC 3530: Network File System (NFS) version 4 protocol. `http://tools.ietf.org/html/rfc3530`.

[6] IETF Network Working Group. RFC 5661: Network File System (NFS) version 4 minor version 1 protocol. `http://tools.ietf.org/html/rfc5661`.

[7] The Open Group. Portable Operating System Interface. `http://www.unix.org/version3/`.

[8] The Open Group. What could have been IEEE 1003.1e/2c. `http://wt.tuxomania.net/publications/posix.1e/download.html`.

[9] Silicon Graphics Inc. Native NFSv4 ACLs for Linux XFS & NFS. `http://oss.sgi.com/projects/nfs/nfs4acl/`.

[10] Sun Microsystems Inc. Solaris ZFS administration guide. `http://dlc.sun.com/pdf/819-5461/819-5461.pdf`.

[11] Aneesh Kumar K.V. Rich-acl e2fsprogs patch. `http://kernel.org/pub/linux/kernel/people/kvaneesh/richaclv1/e2fsprogs/%`.

[12] Aneesh Kumar K.V. Rich-acl kernel git repo. `http://git.kernel.org/?p=linux/kernel/git/kvaneesh/linux-richacl.git;a=%summary`.

[13] Aneesh Kumar K.V. Rich-acl userspace git repo. `http://git.kernel.org/?p=fs/acl/kvaneesh/acl.git;a=summary`.

[14] Novell, Inc. Netware 6 trustee rights: How they work and what to do when it all goes wrong. `http://support.novell.com/techcenter/articles/ana20030202.html`.

[15] W. Richard Stevens. *Advanced Programming in the UNIX(R) Environment*. Addison-Wesley, June 1991.

Consistently Codifying Your Code: Taking Software Development to the Next

Keith Bergelt
Your affiliation
`your-address@example.com`

Abstract

Keith Bergelt (kbergelt@openinventionnetwork.com)

Consistently Codifying Your Code: Taking Software Development to the Next Level

Over the years, sophisticated systems have been put in place to monitor software development and ensure that code integrity is maintained. Software developers expect to regularly use a revision control system such Git, CVS or SVN as part of their endeavors.

One step that has historically been missing as a routine part of the development process is the codification of invention. Software developers continuously innovate. Due to a number of factors, these new innovations unfortunately have often failed to be published in a way that facilitates the ongoing protection of individual and community rights to these inventions.

In order to improve the documentation of invention and lessen the ability of companies and patent trolls to leverage intellectual property against open source companies, as a community we must begin to capture invention regularly and in real time.

Keith Bergelt, CEO of Open Invention Network, a company formed by IBM, NEC, Novell, Philips, Red Hat and Sony to enable and defend Linux, will share his insights into ways that companies can capture and codify invention at the time of development, ensuring that innovation is documented and leveraged in a manner so that the entire open source community will benefit.

Developing Out-of-Tree Drivers alongside In-Kernel Drivers

Jesse Brandeburg
LAN Access Division, Intel Corporation
`jesse.brandeburg@intel.com`

Abstract

Getting your driver released into the kernel with a GPL license is promoted as the holy grail of Linux hardware enabling, and I agree. That said, producing a quality GPL driver for use in the entire Linux ecosystem is not a task for the faint of heart. Releasing an Ethernet driver through kernel.org is one delivery method, but many users still want a driver that will support the newest hardware on older kernels.

To meet our users' requirements for more than just hardware support in the latest kernel.org kernel, we in Intel's LAN Access Division (LAD) developed a set of coping strategies, processes, code, tools, and testing methods that are worth sharing. These learnings help us reuse code, maintain quality, and maximize our testing resources in order to get the best quality product in the shortest amount of time to the most customers. While not the most popular topic with core kernel developers, out-of-tree drivers are a necessary business solution for hardware vendors with many users. Our Open Source drivers generally work with all kernel releases 2.4 and later, and I'll explain many of the details about how we get there.

1 Introduction

This paper's goal is to lay out a roadmap for others to use in order to streamline the out-of-tree development process. Since many developers are able to live solely in the kernel, a secondary goal is to expose some of the business realities that our product group has to cope with, and the solutions we have developed.

Our business goal is simple: Sell hardware. This hard reality guides many of our decisions. To do this, we enable drivers for as many users and operating systems as possible. In an ideal world we would have unlimited resources, and a fully staffed development and testing team with plenty of idle time, but instead we have to make do with busy developers, constrained testing resources, and of course business and customer needs. As such, we've developed tricks and common practice within our code and development process in order to maximize the number of Operating Systems supported.

Intel® Wired Ethernet developers actively maintain multiple (eight) drivers in the kernel, and strive to be good open-source contributors and supporters while still creating an out-of-tree driver, i.e. we are not the enemy.

2 Reasons You Might Need an Out-of-tree Driver

Business Need

Our software support opens new business opportunities for network hardware sales by leveraging the (awesome) environment of the Open Source community and vendors. We sample silicon and boards months before we ship, and need something to deliver to customers to allow them to test.

We avoid a lot of thrash and introduction of last minute OS support requirements from hardware vendors by having a tested and ready "out-of-tree" driver that OEM system integrators and vendors can use to ship our hardware. We also make our out-of-tree drivers available via `e1000.sourceforge.net` in the e1000 project, and on intel.com.

Provide More Complete Hardware Support

Customers are happier if our hardware works in every kernel they might use right out of the box.

No Pre-announcing Hardware

We are unable to ship driver support to the kernel for hardware that either hasn't shipped or won't ship "real soon now." We try not to give up any competitive advantage we might achieve by not informing our competitors of our plans. The out-of-tree driver allows for a testable and feature rich launch of hardware with supporting drivers, even if the driver hasn't made it all the way through net-next into the upstream kernel. It also means we have something to ship on the software CD "in the box."

Users on Old Kernels

Some customers are using 2.4 kernels (still) in production. This creates a bit of a headache for drivers like ixgbe that have many new features that depend on newer kernels. For operating systems and kernels this old we have a policy of "just make it work" which is generally all that is required. Some customers upgrade to the latest and greatest system and network hardware but will not or cannot upgrade their OS for their own business reasons. For example, we have met enterprise customers who have an application that **will not run** on any OS newer than RedHat Enterprise Linux 3. However, RedHat is still supporting RHEL3, at least for now. In our case the right thing to do here is follow the lead of the OS Vendor and try to provide basic support. In some cases, only new hardware is available to replace existing failing hardware, forcing users to upgrade hardware while keeping their existing infrastructure and certifications in place.

Silicon Validation

One of the tasks we have as driver developers is to validate that the silicon we ship will work with our driver code as well as validating new silicon features. This often needs a lot of "non-production" code to be developed and we do that development in our out-of-tree driver, usually on a branch.

Dirty Laundry

The out-of-tree driver source contains many comments and even some code that is never published, that allow us to reference internal bug tracking databases, investigation notes, and debug code (possibly for silicon validation.) There is no value in shipping this code to the open source community and the process of building source allows us to maintain higher quality code, while still having the functionality in the code that we need.

3 Implementation

Shared Code

Our definition of "shared code" is code that is usable under multiple operating systems, with a non-encumbering license. In our environment we have factored out the code that supports functions/features common to all driver hardware tasks like initialization, reset, link management, etc.

The shared code makes up almost 10,000 lines of code (out of 24,961) in our current ixgbe driver. It is shared across multiple OS drivers, including Linux, FreeBSD, Windows drivers (all versions), Windows Testing tools (control panel), Manufacturing/Test tools, as well as customers.

In order to do such a thing without GPL violations, we **maintain exclusive copyright** to the shared code files, allowing us to release the code with any license we (as the exclusive copyright holder) need. Another option would be to dual license the code, but this has some legal implications that we weren't interested in dealing with, and that are beyond this paper's scope.

When we design this code for a family of silicon, we use function pointers to cover the initialization and setup sections that might have differing implementations for each release of silicon.

This code has #defines that are typically negatively defined to allow drivers that don't want to compile in support for a given piece of hardware to strip all the unused code from their driver build. The code typically looks like:

```
#ifndef NO_NNN_HARDWARE_SUPPORT

/* some code specific to NNN */

#endif
```

We also use #defines to mark pre-release sections of code for new hardware or feature support, allowing us to control the driver/features and hardware implemented for a particular driver build. Often for cleanliness of the code those #defines are removed after hardware or the feature first ships.

Some of the other advantages to sharing pieces of the driver initialization code are: More consumers of the code means more developers available to work bugs; more testing coverage because the shared code gets used repeatedly. This maximizes limited testing and development resources to achieve the most productivity.

Build the Code

Our drivers' code base is actually built via a Makefile. The Makefile takes several passes over the code. The major innovation is using unifdef.c from the kernel to clean out #defines and code that we don't want included. This is done via a list of #defines in the Makefile that declare which code we want to keep or strip. Using this method we implement our new hardware support, allowing us the flexibility to add/remove new hardware to a particular driver build right up to the ship date. After the hardware and software support ships for a given release, we typically leave in #defines for hardware support that we might want to discard to reduce code size (as above), but remove #defines that we might have used for new hardware support in the base driver portion of the code. The assumption here is that once the driver supports a given piece of hardware it always will. Another advantage of this build process is that the driver source can be branched and stabilized with a particular set of hardware and features supported, without the code forking from the mainline development.

Create New Drivers

One of the lessons we've learned is that a driver should not have endless hardware revisions added to it. It creates too much regression testing load, and new hardware support too often breaks existing functionality. While it is immensely seductive to reuse all the code in a driver, experience has shown us that driver code is typically brittle. Our conclusion is that whenever possible create a new driver for "the next generation" of silicon. This of course creates more work and more code to maintain. This is an issue that we are continuing to struggle with, but we believe is the correct way forward.

Coding Style

We allow ONLY Linux kernel style for the "shared code" files. In the kernel, there is a Documentation/CodingStyle file and we have implemented an internal process that requires the shared code (and our drivers' core code) to conform to the that document. This causes some discussion among the differing software camps, but saves many headaches in the long run. This is especially useful when keeping code in the out-of-tree and in-kernel drivers the same.

Internal Maintainers

We have implemented mailing lists and automated check-in notification emails that encourage and ease peer review of code. In particular for our shared code we have a single committer that is the only user allowed to commit changes. This guarantees code goes through a minimum level of review by the maintainer before commit to ensure process is followed and that code meets requirements. This has prevented many hours of pain and suffering of developers having to fix bugs or quality issues other users introduce to the shared code.

Consistent External Maintainer Interface

The internal development of the out-of-tree driver is typically followed by changes for the in-kernel driver, which are all pushed through our primary maintainer. Over the past several years we've developed and refined a relationship with the maintainers of the networking stack and networking drivers. Having a single person that is our contact with the maintainers guarantees consistent communication, process, and dramatically increases our chances of getting patches accepted. Jeff Kirsher's paper in other proceedings of the 2010 Linux Symposium explains this in greater detail.

Patch All the Time

We've consistently been asked by the upstream maintainers to not "patch bomb" the lists every 6-12 weeks. We've also found that during development the best way to do kernel (upstream) patches is to immediately introduce any change made to our out-of-tree driver to our internal kernel patch process. We have eased this process by mimicking the kernel development process internally. We use internal mailing lists, an internal patchwork server, and internal git servers. A developer who has just created a patch for the out-of-tree code is in the perfect position (just the right knowledge) to create the kernel patch for the same change. The developer creates the patch, typically uses stgit to email it to the list, and then the patch is tracked in patchwork through testing and then eventual submittal via email to the networking maintainer.

Kernel Compatibility Layer

We follow a similar model that is used by libata to provide backport compatibility to some distribution kernels. Our Ethernet driver kcompat.h and kcompat.c files allow for "upstream" looking core driver code which works on older kernels (yes it's GPL, so you can use it too.) When combined with strategically placed #defines in our driver core code, our drivers can compile and load on almost all 2.4 and 2.6 kernel versions. Of course #defines for certain OS capabilities are unavoidable and end up breaking up mainline source with #ifdefs, but using flags in the driver to advertise driver/hardware features and capabilities can minimize the "#ifdef thrash."

4 Version Control

Our current infrastructure uses CVS but we could easily switch to any other version control system that has a sufficient ecosystem to allow easy cross platform (aka Windows) development. The large features that we rely on version control to provide are branching, tagging, and change tracking. We have taken great pains to enforce adherence to committing only a single change at a time.

Nightly Labels - auto-builds

We have recently started the nightly process of automatic labeling and building of certain components of our software. The shared code is built in both a DOS and UNIX linefeed version. The drivers then consume that "built" version in our build tool when the "checkout" is prepared before a driver source build. Finally the source is built on a Linux machine via make, and compile tested on several different distributions.

Reproducible Build Process

One of the benchmarks for our process is reproducible builds. We take steps to make sure that any given build can be rebuilt in the future should something go wrong. We periodically make practice runs to prove it is working.

5 Pitfalls

Kernel standards can conflict with internal requirements

One of the issues we ran into is that the kernel community requested e1000e use C99 initializers for function pointers. We made that change, but it required our in-kernel driver to fork from the internal shared code because C99 syntax doesn't work with DOS compilers.

GPL concerns

We must maintain exclusive copyright in order to multi-license the shared code. We can't take in code changes to our shared code, when submitted against GPL source, unless we transfer copyright or rewrite code and counter-propose to maintain authorship/copyright. We don't expect everyone to understand our licensing concerns, but we do try to offer changes and alternatives to patches on the list that allow us to maintain our copyright and still not allow too much drift between our out-of-tree shared code and the kernel version.

Distributions and Backports

Backports typically come from upstream changes only, which means that the distribution engineers are often reinventing the wheel we've already created in our standalone driver.

In all fairness Novell has a great KMP (kernel module package) model that allows us to provide them a driver from our out-of-tree code that they build and provide to users.

Bug fixes often go into distributions but sometimes don't make it upstream. Even when they do make it upstream, it is difficult to track what is required to be changed in the out-of-tree driver.

What to Test?

Pre-production

Pre-production testing tests mostly the out-of-tree driver, looking for hardware bugs and verifying driver functionality. If we had unlimited time and resources we would ideally make the driver backport for the distribution, and then test. Often distribution code submittal windows are closed before we have final silicon, and yet the silicon will ship before the distribution in question.

Early Release

Early (before release of the hardware) kernel submittal often has just basic functionality. Testing this gives you a warm fuzzy feeling and can be the basis of the initial kernel submittal, and so is a worthwhile effort.

Distributions

Testing our driver that is included in the distribution is the hardest because they likely don't even have hardware support yet for the device you're testing. The distribution is assumed to have basically the same driver that is upstream, but in practice because of the backporting the distribution has to do, the driver is an actual fork. The confusion due to differing version numbers in drivers must be managed in some way or another. In our case we add -kN to our version numbers for drivers submitted to the kernel. In addition we also ask the distributions if they make any changes to our driver to update the -kN to the next odd value, with the goal of having in-kernel drivers have all even numbers for N, i.e. 2,4,6, and distribution backports would hopefully have odd values of N, i.e. 1,3,5.

Limitations to This Path

Double work

Most driver changes must be made once to the internal version and once to the kernel version. After that, changes to the distribution drivers need to be initiated and tracked through the relevant methods.

Upstream vs Out-of-tree

Keeping the drivers in sync is a significant effort. Our solution is diligence, patience, and a significant time and resource commitment. The whole team participates in the open source community via monitoring mailing lists and submitting patches.

6 Common Questions

Why not release code earlier?

We have: 82599 driver released 4 weeks before general hardware availability. 82580 driver released November 7 2009, at least 8 weeks before general hardware availability.

Why not develop in the open?

By this, I believe the question to be "why don't we have a public git tree?" We push patches upstream as soon as they are ready. Developing in a public git repository only allows us to target one kernel version, and our requirements include other delivery vehicles besides the upstream kernel.

How come you don't just develop for the upstream kernel?

Our customers demand support in older kernels, not just the upstream kernel driver.

Why don't you use a "real" version control system?

We'd like to use GIT, but training Windows, BSD developers, technical marketing engineers, and software configuration management engineers to use GIT is a big effort. In short, we're working on it but don't expect quick changes.

7 Conclusion

Developing in-kernel and out-of-tree drivers can be done with a common source base and a minimum of work. This paper shows some of the methods and practices Intel Wired Ethernet developers utilize. We welcome any follow-up questions and discussion either directly to the author or on our public email list.

`e1000-devel@lists.sourceforge.net`

Open Source Governance: An Approach

Art Cannon
Your affiliation
your-address@example.com

Abstract

Database on Linux in a virtualized environments over NFS

Bikash Roy Choudhury
Your affiliation
your-address@example.com

Abstract

KVM for ARM

Christoffer Dall and Jason Nieh
Columbia University
`{cdall,nieh}@cs.columbia.edu`

Abstract

As ARM CPUs grow in performance and ubiquity across phones, netbooks, and embedded computers, providing virtualization support for ARM-based devices is increasingly important. We present KVM/ARM, a KVM-based virtualization solution for ARM-based devices that can run virtual machines with nearly unmodified operating systems. Because ARM is not virtualizable, KVM/ARM uses lightweight paravirtualization, a script-based method to automatically modify the source code of an operating system kernel to allow it to run in a virtual machine. Lightweight paravirtualization is architecture specific, but operating system independent. It is minimally intrusive, completely automated, and requires no knowledge or understanding of the guest operating system kernel code. By leveraging KVM, which is an intrinsic part of the Linux kernel, KVM/ARM's code base can be always kept in line with new kernel releases without additional maintenance costs, and can be easily included in most Linux distributions. We have implemented a KVM/ARM prototype based on the Linux kernel used in Google Android, and demonstrated its ability to successfully run nearly unmodified Linux guest operating systems.

1 Introduction

To provide the benefits of virtualization to Linux users, Kernel Virtual Machine (KVM) has been included in Linux starting with kernel version 2.6.20. Its tremendous success is in large part due to its open-source distribution and its relative simplicity compared to other approaches. This simplicity is achieved by leveraging the functionality already provided by the Linux kernel, and relying on some level of hardware virtualization support. KVM runs on a wide range of architectures, currently providing full support for x86 and PowerPC, and experimental support for Itanium (ia64) and s390. The x86 and ia64 implementations rely on hardware virtualization extensions and the PowerPC and s390 architectures are virtualizable.

Unfortunately, KVM does not support the ARM architecture, which is increasingly ubiquitous. While ARM is known for excellent power consumption, small die size, and compact code, recent CPUs based on the ARM architecture are also quite powerful, and are being incorporated in growing numbers into a wide range of products. Mobile phones are almost exclusively based on ARM, and ARM-based tablets and laptops with 3G connections are increasing in popularity. Users increasingly expect these devices to be able to perform a multitude of tasks, including browse the Internet, play games, and run thousands of other applications from an online application store. ARM Linux is becoming more important with the introduction of several Linux-based distributions targeting mobile and embedded ARM-based devices, Google Android being one of them.

The key challenge in providing virtualization on ARM is that the ARM architecture is not virtualizable. A virtualizable architecture would allow a virtual machine to directly execute on the real hardware while guaranteeing that the virtual machine monitor (VMM) retains control of the CPU. This is done by running the operating system in the virtual machine, the guest operating system, in non-privileged mode while the VMM runs in privileged mode. ARM is not virtualizable because there are a number of sensitive instructions, used by operating systems, which do not generate a trap when executed in non-privileged mode. Their behavior is either unpredictable or they behave differently, causing an operating system that uses these sensitive instructions to not run correctly if run in a virtual machine directly executed on the real hardware. There is also no hardware virtualization support on ARM. The result is that ARM CPU and memory virtualization are difficult.

We present KVM for ARM (KVM/ARM), a KVM-based virtualization solution for ARM that runs nearly unmodified operating system instances in virtual machines. KVM/ARM retains the simplicity of the KVM architecture in the absence of ARM hardware virtualization support by introducing *lightweight paravirtualization*. Lightweight paravirtualization is a script-based method to automatically modify the source code of the guest operating system kernel to issue calls to KVM instead of issuing sensitive instructions to enable a *trap-and-emulate* virtualization solution. Lightweight paravirtualization is architecture specific, but operating system independent. It is completely automated and requires no knowledge or understanding of the guest operating system kernel code. This is in stark contrast to traditional paravirtualization, which is both architecture and operating system dependent, requires detailed understanding of the guest operating system kernel to know how to modify its source code, and then requires ongoing maintenance and development to maintain heavily modified versions of operating systems that can be run in virtual machines.

This paper presents the design and implementation of KVM/ARM. Section 3 describes KVM/ARM CPU virtualization using lightweight paravirtualization. Section 4 describes KVM/ARM memory virtualization. Section 5 describes the current implementation status of KVM/ARM and ideas for improvement. Finally, we present some concluding remarks.

2 Related work

Virtualization has been around since the of the 1970s [6], but re-emerged in the 1990s as commodity x86 hardware became fast enough to run multiple operating systems simultaneously. The x86 architecture was previously not virtualizable since many sensitive instructions did not trap when executed in non-privileged mode [1]. Emulation could be used where each guest instruction is interpreted in software, but this is too slow for practical use. To overcome this problem, VMware [13] introduced efficient dynamic binary translation mechanisms to translate sensitive instructions to other instructions to enable x86 virtualization with low performance overhead. However, the dynamic binary translation mechanisms are not easy to implement, resulting in a complex solution that is likely to be too heavyweight to use for more resource-constrained mobile devices such as smartphones. Xen [5] used paravirtualization [17] to provide x86 virtualization, in which guest operating systems are extensively modified by hand to use a rich set of hypercalls in lieu of sensitive instructions. However, paravirtualization cannot run existing unmodified operating systems, and the modifications are extensive enough that supported guest operating systems lag significantly behind the latest available unmodified operating system versions.

Intel and AMD have recently begun equipping x86 CPUs with native hardware virtualization support, Intel VT [9] and AMD-V [2], respectively. A new CPU guest mode is provided for running virtual machines such that sensitive instructions automatically trap so they can be handled by a VMM, and nested page tables provide hardware translation between physical memory addresses perceived by the guest operating system and host physical addresses on the real hardware through a data structure managed by the VMM. KVM leverages hardware virtualization support for x86 CPUs together with existing Linux kernel functionality to provide a relatively simple virtualization solution compared to VMware and Xen. KVM implements a simple kernel module, which provides full native virtualization supporting completely unmodified guest operating systems. Unfortunately, no such hardware virtualization support exists for ARM. KVM/ARM is designed to preserve the simplicity of KVM as much as possible while enabling virtualization support on a non-virtualizable architecture.

The growing ubiquity of ARM CPUs and continued advances in their performance have spurred various efforts to provide virtualization on ARM. Several commercial solutions are being developed, including VLX for ARM by VirtualLogix [15], OKL4 Microvisor by OK Labs [11], MVP by VMware [16], and INTEGRITY secure virtualization by Green Hills [7]. None of these solutions are open-source and all of them require paravirtualization.

Xen ARM [8] is the only other open-source ARM virtualization approach available. Xen paravirtualization requires access to guest operating system source code and maintenance of changes to each version of the source tree. The price of paravirtualization is increased maintenance cost and more limited availability in terms of supported guest operating system versions. For example, Xen ARM requires modifying by hand approximately 4500 lines of code in the guest operating system [14]. The most recent kernel version it can support in a guest operating system is a modified Linux 2.6.11 kernel, a relatively old version of Linux. In contrast, KVM/ARM's lightweight paravirtualization requires minimal modifications to guest operating systems, and those modifications are simple enough that they can be completely automated by a script. This makes it relatively easy for KVM/ARM to support more recent versions of guest operating systems.

3 CPU virtualization

Virtual machines must not be allowed to access the privileged state of the physical CPU and thereby gain unwanted control of hardware resources. Therefore, guest operating systems must always run in a non-privileged mode. The non-privileged mode on ARM is called *user mode*.

Popek and Goldberg [12] define sensitive instructions as the group of instructions where the effect of their execution depends on the mode of the processor or the location of the instruction in physical memory. A sensitive instruction is also privileged if it always generates a trap, when executed in user mode. The VMM can only guarantee correct guest execution without the use of dynamic translation if all sensitive instructions are also privileged. In other words, an architecture is virtualizable if and only if the set of sensitive instructions is a subset of the set of privileged instructions. If that is the case, the VMM can be implemented using a classic trap-and-emulate solution. Unfortunately, ARM is not virtualizable as the architecture defines both sensitive privileged instructions and sensitive non-privileged instructions.

The sensitive privileged instructions defined by the ARM architecture are the *coprocessor access instructions* which are used to access the *coprocessor interface*. There is no such thing as a physical coprocessor, but the semantics are used merely to extend the instruction set by transferring data between general purpose registers and registers belonging to one of the sixteen possible coprocessors. The architecture always defines coprocessor number 15 which is called the *system control coprocessor* and controls the virtual memory system. Specific implementations of the ARM architecture can define other coprocessors to allow software to access special hardware or otherwise leverage additional hardware logic. For instance, coprocessor 14 is often used to access floating point hardware. The coprocessor access instructions are: `CDP`, `LDC`, `MCR`, `MCRR`, `MRC`, `MRRC`, and `STC`. These instructions do not have to be handled specially as they trap when they are executed in user mode. When that happens, KVM/ARM catches the trap and emulates the sensitive privileged instruction in software.

The ARM architecture also defines sensitive non-privileged instructions which cannot be handled using just trap and emulate because they do not trap. These instructions deal with processor modes, status registers, and memory accesses that depend on CPU mode.

Processor mode instructions relate to ARM's 7 processor modes: user mode and 6 privileged modes.[1] Each mode has a number of *banked registers*, which means that, for instance, register 13 points to a different physical register in *supervisor mode* than in user mode.[2] Specific versions of *load/store multiple instructions* access user mode registers even when the processor is in a privileged mode. When executed in user mode, these instructions do not trap and are therefore sensitive and non-privileged. These instructions are the `LDM(2)` and `STM(2)` instructions.

Status register instructions relate to special ARM status registers. ARM processors have a special register called the *Current Program Status Register* (CPSR), which specifies the current mode of the CPU and other state information. Some of the bits in the CPSR are privileged, such as the mode bits, and some are accessible in user mode. Five of the privileged modes also have a banked *Saved Program Status Register* (SPSR), which contains a copy of the user mode CPSR as it was when the processor entered the privileged mode.[3]. The *status register access instructions* `CPS,MRS,MSR,RFE,` `SRS` read and write the CPSR and the SPSR. Writes to privileged bits are ignored when the CPU is in user mode and access to the SPSR is unpredictable in user mode. Further, almost all data processing instructions exist in a special mode, which replaces the content of the CPSR with that of the SPSR. These instructions are denoted by appending an S to the instruction name: `ADCS`, `ADDS`, `ANDS`, `BICS`, `EORS`, `MOVS`, `MVNS`, `ORRS`, `RSBS`, `RSCS`, `SBCS`, and `SUBS`. Likewise, the `LDM(3)` instruction replaces the content of

[1]See pages A2-3 to A2-5 in the ARM Architecture Reference Manual [3] for more information

[2]The differences between the privileged modes only concern the banked registers and can be ignored throughout this paper.

[3]Actually, the CPSR is only copied to the SPSR when entering privileged mode through exceptions and not when manually switching modes

the CPSR with that of the SPSR in addition to loading multiple registers. The behavior of these instructions is unpredictable when executed in user mode and the instructions are therefore all sensitive and non-privileged.

Memory access instructions that depend on CPU mode relate to access protection. The virtual memory system on ARM processors uses access protection bits to limit access to memory depending on the CPU mode. Regular memory accesses are *not* sensitive according to Goldberg and Popek, as they will trap when executed in a less privileged mode (reduced memory access rights). However, the architecture defines a number of instructions that access memory using user mode access permissions even though the CPU is in a privileged mode. These instructions are called *Load/Store with translation* and there are four of them: `LDRBT`, `LDRT`, `STRBT`, and `STRT`. When executed in user mode, these instructions behave as regular memory access instructions. Thus the effect of executing these instructions depends on the mode of the processor and the instructions do not trap due to memory access violations. They are therefore sensitive and non-privileged.

3.1 Lightweight paravirtualization

To avoid the problems with sensitive non-privileged instructions, we modify the guest kernel source code slightly. We do not have to worry about user space software as user space applications will execute in the same CPU mode as if they were executing directly on a physical machine. Sensitive instructions are not generated by standard C-compilers and are therefore only present in assembler files and inline assembly.

We modify the guest kernel source code using an automated scripting method. The script is based on regular expressions and has been tested on a number of kernel versions with success. The script supports inline assembler syntax, assembler as part of preprocessor macros, and, assembler macros.

It works by replacing sensitive non-privileged instructions with trap instructions and emulating the sensitive instruction in software when handling the trap. However, KVM/ARM must be able to retrieve the original sensitive instruction including its operands to be able to emulate the sensitive instruction when handling a trap. We experimented with inserting the trap instruction immediately *before* the sensitive instruction, but this caused problems with PC-relative addressing and *fix-up tables*.[4] To avoid the need to manually fix the patched code, we defined an encoding of all the sensitive non-privileged instructions and their operands into trap instructions.

The `SWI` instruction on ARM always traps and is normally used for making system calls. The instruction contains a 24-bit immediate field (the payload), which we can use to encode sensitive instructions. Unfortunately, the 24 bits are not quite enough to encode all the possible sensitive non-privileged instructions and their operands. However, all coprocessor access instructions trap if they access an undefined coprocessor. If we specify coprocessors zero through seven, which are not defined by the ARM architecture, all the coprocessor access functions will trap regardless of their operands. The coprocessor access instructions use 24 bits for their operands, which we can also leverage to encode the sensitive non-privileged instructions.

The VMM needs to be able to distinguish between guest system calls and traps for sensitive instructions. We make the assumption that the guest kernel does not make system calls to itself. Under this assumption, we simply interpret the payload if the virtual CPU is in privileged mode and emulate the encoded instruction. If the virtual CPU is in user mode, we consider the `SWI` instruction a system call made by guest user space to the guest kernel.

The ARM architecture defines 24 sensitive non-privileged instructions in total. We encode the instructions by grouping them in 15 groups; some groups contain many instructions and some only contain a single instruction. The upper 4 bits in the `SWI` payload indexes which group the encoded instruction belongs to (see Table 1). This leaves us 20 bits to encode each type of instruction. Since there are 5 status register access functions and they need at most 17 bits to encode their operands, they can be indexed to the same type and be sub-indexed using additional 3 bits. There are 12 sensitive data processing instructions and they all use register 15 as the destination register and they all always have the S bit set (otherwise they are not sensitive). We index them in two groups: one where the I bit is set and one where it's clear. In this way, the data processing instructions need only 16 bits to encode their operands

[4]Fix-up tables is a method used by the kernel to verify access on copy to and from user space operations. It uses offsets to the PC linked at a special section and added to the PC on memory access violations.

leaving us 4 bits to sub-index the specific instruction out of the 12 possible. The sensitive load/store multiple and load/store with translation instructions are using 12 of the remaining 13 index values as can be seen in Table 1.

Index	Group / Instruction
0	Status register access instructions
1	LDM (2), P-bit clear
2	LDM (2), P-bit set
3	LDM (3), P-bit clear and W-bit clear
4	LDM (3), P-bit set and W-bit clear
5	LDM (3), P-bit clear and W-bit set
6	LDM (3), P-bit set and W-bit set
7	STM (2), P-bit set
8	STM (2), P-bit clear
9	LDRBT, I-bit clear
10	LDRT, I-bit clear
11	STRBT, I-bit clear
12	STRT, I-bit clear
13	
14	Data processing instructions, I-bit clear
15	Data processing instructions, I-bit set

Table 1: Sensitive instruction encoding types

In Table 1 only the versions of the load/store instructions with the I-bit clear are defined. This is due to a lack of available bits in the `SWI` payload. We encode the versions with the I-bit set using the coprocessor access instruction. When the I-bit is set, the load/store address is specified using an immediate value which requires more bits than when the I-bit is clear. Since the operands for coprocessor access instructions use 24 bits, we can use 2 bits to distinguish between the 4 sensitive load/store instructions. That gives us 22 bits to encode the instructions with the I-bit set, which is exactly what is needed.

We illustrate the implementation of our solution by an example. Consider this code in `arch/arm/boot/compressed/head.S`:

```
mrs     r2, cpsr   @ get current mode
tst     r2, #3     @ not user?
bne     not_angel
```

The `MRS` instruction in line one is sensitive, since when executed as part of booting a guest, it will simply return the hardware CPSR. However, we must make sure that it returns the virtual CPSR instead. Thus, we replace it with a `SWI` instruction as follows:

```
swi     0x022000   @ get current mode
tst     r2, #3     @ not user?
bne     not_angel
```

When the `SWI` instruction in line one above generates a trap, KVM/ARM loads the instruction from memory, decodes it, emulates it, and finally returns to line two.

The approach differs from Xen's paravirtualization solution in that it requires no knowledge of how the guest is engineered and can be applied automatically on any OS source tree compiled by GCC. For instance, Xen defines a whole new file in `arch/arm/mm/pgtbl-xen.c`, which contains functions based on other Xen macros to issue hypercalls regarding memory management. Calls to these functions are placed instead of existing kernel code through the use of preprocessor conditionals many places in the kernel code. The presented solution completely maintains the original kernel logic, which drastically reduces the engineering cost and makes the solution more suitable for test and development of existing kernel code.

3.2 Exceptions

An exception to an ARM processor is a common term for traps and interrupts. Traps are caused by software issuing an instruction that traps and interrupts are generated externally by hardware events. Common for all of them are that when they occur, the processor changes to a privileged mode and jumps to a predefined virtual memory address. The ARM architecture defines the exceptions shown in Table 2. It is configurable at run-time whether the low or high address shown in the table are used.

The reset exception occurs when the physical reset pin on the processor is asserted. Undefined exceptions happen when an unknown op-code is used for an instruction or when software in user mode try to access privileged coprocessor registers or when software access non-existing coprocessors. Software interrupt exceptions happen when `SWI` instructions are executed. Prefetch and data abort exceptions happen when the processor cannot fetch the next instruction or complete load/store instructions, respectively. These exceptions

Exception	Low addr.	High addr.
Reset	0x00000000	0xffff0000
Undefined	0x00000004	0xffff0004
Software interrupt	0x00000008	0xffff0008
Prefetch abort	0x0000000c	0xffff000c
Data abort	0x00000010	0xffff0010
Interrupt	0x00000018	0xffff0018
Fast-interrupt	0x0000001c	0xffff001c

Table 2: ARM exceptions overview

are either caused by missing page table entries or by memory protection violations. Interrupt and fast interrupts are caused by external hardware asserting a pin on the processor.

ARM operating systems must configure the exception environment before any exceptions occur. ARM processors have interrupts disabled at power on. During the boot process, the operating system makes sure not to generate any traps as these would cause unpredictable behavior. After the OS sets up page tables and enables virtual memory, it makes sure to map an *exception vector page* into 0xffff0000 (if high vectors are used). Only then it enables interrupts and starts generating traps. The instructions at the specific addresses in Table 2 are usually branch instructions to more or less complex handler functions.

When the guest runs, it is likely going to use almost the entire virtual address space. Especially if we are running the same guest and host OS, there is clearly going to be a conflict. Therefore, VMs execute in their own separate address space. The only host pages mapped in the VM address space are the exception vector page and the *shared page*. We explain the shared page in more details in Section 4.2.

When the CPU is executing guest code, the only way for KVM/ARM to regain control is through an exception. Unfortunately, we cannot use the host kernel exception handlers to handle exceptions when running the guest, but we have to write our own handlers. The KVM/ARM exception handlers are mapped at the exception vector page address in the VM's address space and are designed to re-enter the host kernel address space and return to host kernel code when an exception occurs. KVM/ARM then examines the guest exit reason and performs required emulation before resuming guest execution.

The above approach differs significantly from KVM on x86, which is based on hardware virtualization support. With hardware virtualization support, the exit path from guest execution is not through normal exceptions. Instead, the hardware automatically changes segment registers and thereby changes the address space back to the host kernel and resumes execution after the original world switch instruction. Software can then simply read a special register to determine the guest exit reason.

When KVM/ARM handles a hardware interrupt, it needs to run the host kernel hardware interrupt handler. If the host kernel handler is not run, the host kernel may miss important hardware events such as timer ticks or network packets. The host kernel interrupt handler queries the interrupt controller to find out what happened and manages the device that caused the interrupt as necessary. However, since we want to run the host kernel handler *after* the actual hardware interrupt went off, we are entering the host handler differently than usual. Unfortunately, it's not just a matter of a simple function call as the host kernel handler expects a certain state of the stack and CPU registers when the handler is called. KVM/ARM sets up a state exactly as it would have been if the host kernel interrupt handler had handled the interrupt directly, and executes the host kernel handler.

We note that in the typical use of KVM the guest kernel never needs to handle interrupts as a result of hardware interrupts. The guest kernel will only be exposed to emulated devices. Interrupts from emulated devices are generated by software and artificially injected into the guest. When this happens, or when we need to inject a trap to the guest (e.g. guest user space issues a system call to the guest kernel) an address space conflict can occur.

Suppose the guest is using the high address for the exception vector page and the host is also using the high address to handle actual hardware exceptions. The two physical frames can obviously not be mapped at the same virtual address during guest execution, and the KVM/ARM exception vector page must always be mapped, since otherwise there is no way for the host kernel to regain control of the system. When the guest kernel is about to handle an exception, it will trap on the instruction memory access permissions, since the guest does not have access to the KVM/ARM exception vector page. In this case we simply tell the hardware to use the low vector addresses and map the KVM/ARM exception page at that address instead. If later the guest needs to access the low vector address for other purposes, we simply tell the hardware to use the high vector addresses again. Switching the vector location is a very simple operation and involves only writing to a coprocessor register and invalidating a few TLB entries and cache lines.

The exception handlers used by KVM/ARM are compiled as a part of KVM and get linked at an unknown address. When a VM is created in KVM/ARM, the exception handlers are relocated from the address they were linked at to a newly allocated exception vector page belonging to that VM. Due to limited width of the immediate fields in ARM instructions even assembler code can generate binary code, which cannot simply be copied to a new location as it may reference data at specific addresses. Therefore the exception handler code is written in location-independent assembly. For instance, the following instruction would generate a load from a PC-relative address, which would not be easy to detect when relocating the code:

```
code_start:
   ldr     lr, =0xffff1000
   ...
code_end:
```

Instead, we write the code like the following, which loads the 32-bit immediate value from a local label relative to the PC. When we relocate the code segment we copy the code until the `code_end` label, which will include the data value:

```
code_start:
   ldr     lr, 1f
   ...
1: .word   0xffff1000
code_end:
```

4 Memory virtualization

Virtual machines need access to any part of the virtual address space they desire. But we cannot simply allow the guest OS to manage the physical memory or the MMU, as the host OS must retain control over physical memory and be protected from the guest at all times. Therefore, the memory must be virtualized.

The memory system on ARM exists in two flavors: one without an MMU (replaced by a *Memory Protection Unit* (MPU)) and one with an MMU. The latter translates virtual to physical addresses in hardware using a rather flexible two-level page table layout. The presented solution is developed for systems with an MMU. MMU-less cores are usually used in extremely simple embedded systems where virtualization may be of less importance anyway.

Memory virtualization introduces a new address space: *guest physical addresses*. Guest physical addresses are the addresses that the guest thinks represent physical memory addresses. However, since the memory is virtualized they are simply offsets into the memory region allocated to the guest. Guest page tables, which are managed by the guest kernel, translate from guest virtual addresses to guest physical addresses and can therefore not be used for address translation by the MMU. The guest physical addresses must first be translated to *host physical addresses* (also called machine addresses)[5]. See Figure 1 for an illustration of the address spaces.

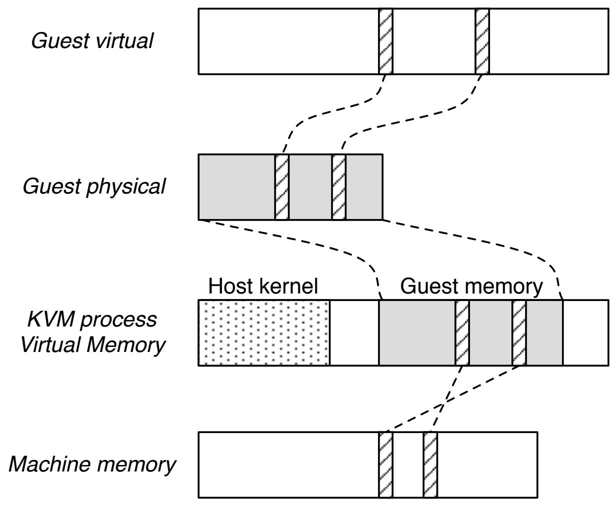


Figure 1: KVM address spaces

4.1 Shadow page tables

Shadow page tables are data structures managed by KVM/ARM, which are used by the hardware to translate from guest virtual addresses to machine addresses during guest execution.

[5]Recent hardware support for virtualization include technologies called Nested Page Tables or Extended Page Tables. These technologies add an extra translation table, which translates guest physical addresses to machine addresses in hardware.

When a new shadow page table is allocated, two special entries are always created and the rest of the table is left blank. The two special pages are the *shared page* explained in Section 4.2 and the exception vector page discussed in Section 3.2. If the guest tries to access any page other than the two just mentioned, a page fault occurs.

Guest pages are mapped into shadow page tables on demand, when KVM/ARM handles page faults occurring in VMs. The custom KVM/ARM exception handler will determine the virtual address which caused the fault and create an appropriate mapping in the shadow page table. Such a mapping must translate from the fault address to a machine address. KVM/ARM translates the fault address into a guest physical address by walking the guest page tables in software. The guest physical address is then translated to a host virtual address through architecture independent KVM functionality and finally the host virtual address is translated to a machine address by using standard kernel virtual memory translation functions.

ARM level-1 page table entries can be either *section descriptors*, pointers to *coarse page tables* or pointers to *fine page tables*. Coarse and fine page tables are commonly referred to as level-2 tables. Sections is a way to map a 1MB virtual memory region to a corresponding 1MB of contiguous physical memory. Coarse page table entries map pages of either 64KB (large pages), 4KB (small pages) or 1KB (tiny pages). Linux uses almost exclusively coarse page tables with pointers to 4KB small pages.

The entries in the shadow page tables should generally be of the same type as the entries in the guest page tables. However, this may not be possible when the guest uses section descriptors. Since Linux operates with a 4KB page size and KVM/ARM requests physical memory pages from Linux user pages, we cannot guarantee 1MB contiguous free physical memory on the host. Therefore, all shadow page table entries use 4KB small page mappings. This approach causes no loss in functionality, but it may affect performance negatively. The reason is that section descriptors only occupy a single entry in an ARM TLB, where a similar mapping of a 1MB area based on 4KB small pages occupy 256 TLB entries.

When the guest modifies page tables, KVM/ARM must also update the shadow page table. For instance, if the guest kernel uses copy-on-write, it will change the mapping on the first write to the COW section. Fortunately, the guest must invalidate the TLB when changing page tables. TLB invalidation is a privileged operation so KVM/ARM will catch the operation. When it happens, KVM/ARM simply re-initializes the shadow page table to only contain the shared page and the exception page and maps in the updated entries on demand.

4.2 Shared page and world switches

The shared page mentioned above is, as the name suggests, a page which is shared between the host and the guest. It is always mapped at the same address in the host kernel as in the guest kernel and it is used to perform world switches. The reason why we need a shared page is that guests execute in completely separate address spaces from the host and the ARM architecture requires that a change of address spaces is done from a page mapped at the same virtual address in both the previous and new address space - otherwise the behavior is unpredictable.

The shared page is mapped such that the guest cannot access the page and any reads or writes from or to the page will generate a page permission fault. If that happens, KVM/ARM must map the shared page at a different virtual address in both the host and the guest and map a guest page at the fault address instead. By doing so, the existence of the shared page is hidden from the guest OS. For Linux guests we take advantage of the reserved memory region in ARM Linux (see `Documentation/arm/memory.txt` for more info).

The shared page is structured as shown in Figure 2. The code contains two entry points: `__vcpu_run` and `__exception_return`. The first is used to switch to the guest and the second is called from the custom exception handlers to restore the host environment after an exception occurs. The data in the top of the page is needed to complete world switches. For instance, when switching to the guest, the code in the shared page reads the base physical address of the active level-1 page table from a special register called the *Translation Table Base Register* (TTBR) and stores the value at the top of the shared page. The data section also contains the physical base address of the shadow page tables and the world switch code writes this value to the hardware TTBR. After the page tables have been switched, the stack pointer is no longer valid as the kernel stack location is no longer mapped to the right physical memory. Therefore we reserve 2K from the top of the shared page for the stack.

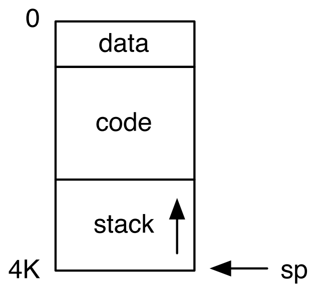


Figure 2: Shared page layout

Because KVM is compiled and distributed as a module, and because we want to be able to run multiple guests simultaneously[6], and because we may not know the shared virtual address at compile time, the code on the shared page can not be linked directly by the kernel. Instead, a shared page is allocated per VM, and the world switch code is relocated to the corresponding shared page. Like the code for the exception vector page, the code for the shared page is written in position-independent assembly to make it relocatable.

4.3 Memory protection

Memory protection on ARM works through two concepts: Domains and Access Permissions. These features are used by traditional ARM operating systems to protect for instance kernel memory from user space applications and to implement features such as copy-on-write.

There are 16 domains on ARM - domain 0 to 15. Each level-1 page table entry describes a 1MB virtual address region. The level-1 entries contain a 4-bit value denoting which domain the 1MB address region belongs to. Each domain can be in one of the following three modes:

- No access
- Client
- Manager

[6]Since the data on the shared page belongs to the specific VM, the shared page cannot be shared across VMs

If a page belongs to the no access domain, all accesses to that page will generate faults. If a page belongs to an administrator domain, all accesses will succeed. Finally, if a page belongs to a client domain, the access permissions on page table entries are checked.

The access permissions define permissions depending on the privilege level of the CPU. In Table 3 we show the possible access permissions for normal memory on ARMv5.

AP[1:0]	**Privileged**	**User**
0b00	No access	No access
0b01	Read/write	No access
0b10	Read/write	Read only
0b11	Read/write	Read/write

Table 3: Access permission settings

Since the guest will always run in user mode, KVM/ARM must translate the guest page access permissions to a value resulting in the same level of protection on the shadow page tables. For example, if the guest page tables allow read/write access in privileged mode but no access in user mode, and the VM is in privileged mode, the shadow page table access permissions must use read/write for user mode. When the VM changes CPU mode, KVM/ARM updates access permissions on shadow page table entries correspondingly. See Table 4 for the translation scheme used to translate between guest access permissions and shadow access permissions.

Domains are essentially easy to handle in shadow page tables, since their settings can simply be copied from the guest page tables and used on the shadow page tables. However, the shared page and the exception vector page must always have read/write privileged access and no user mode access, in order to protect the host from the guest. Consequently these pages must be mapped using a client domain. Since the domain is specified for a 1MB virtual address space range, a single level-1 page table entry can specify the domain for both pages belonging to the guest and for the exception vector page or the shared page or both. In this case, KVM/ARM changes the level-1 page table entry domain to a client domain and modifies access permissions on the shadow page table entries in the same 1MB address range to correspond to the guest domain setting.

	Guest page table permissions			Shadow page table permissions		
Guest mode	Priv.	User	AP	Priv.	User	AP
User	NA	NA	00	RW	NA	01
Priv.					NA	01
User	RW	NA	01	RW	NA	01
Priv.					RW	11
User	RW	RO	10	RW	RO	10
Priv.					RW	11
User	RW	RW	11	RW	RW	11
Priv.					RW	11

Table 4: Access permission translation from guest to shadow page tables

5 Implementation status

The presented work is based on the 2.6.27 kernel running on the Google Android emulator, which emulates the ARMv5 architecture (specifically an arm926E core) on a custom "Goldfish" platform. The emulator provided us with a very convenient development environment as it supports GDB debugging of kernel code.

The solution successfully boots a Linux kernel with a simple user space init environment. We can run small programs although with poor performance. This was not surprising, as the implementation has not been optimized for performance. Instead, the work has been focused on correct functionality and full support for all architecture features. As a consequence many features have been implemented naively, which eased debugging and code clarity and performance optimizations have been postponed for future work.

The VM supports the devices on a standard ARM integrator development platform. Theoretically, since all devices are memory-mapped on ARM and thereby communicate with QEMU using the same functionality, all devices should work once a single device works. However, there may be timing constraints which requires the use of paravirtualized drivers, coalesced MMIO or something completely different.

The MMU emulation does not support tiny pages. The reason is simply that we have not come across a guest using them yet. Further, the use of tiny pages is deprecated from ARMv6 and forward.

We are currently working on ARMv6 and ARMv7 support. The ARMv6 work is done on the HTC Dream G1 developer phone which is equipped with a ARM1136EJ-S core. The ARMv7 work is done on BeagleBoards which feature Cortex-A8 cores. The majority of this work consists of supporting the new page table formats introduced in ARMv6, supporting processor-specific cache and TLB manipulation functions, ensuring cache coherency on shared data, and adding emulation code to support a few instructions added in ARMv6 such as `SRS` and `RFE`.

Additionally we are also taking steps to improve the performance of the system. Specifically, we are taking advantage of new ARMv6 features to reduce the world switch costs. ARMv6 and newer supports physically tagged caches, which in part avoids the need to flush caches on world switches. Further, ARMv6 supports tagging of TLB entries with *Application Space Identifiers* (ASIDs), which allows several TLB entries with the same virtual address, but belonging to different address spaces, to reside in the TLB at the same time. By using ASIDs we can avoid TLB invalidations on world switches. Finally, we are also experimenting with other performance improvements such as removing unnecessary memcopies, caching shadow page tables and more.

ARM TrustZone[4] is a security technology available in a number of recent processors from ARM and is the closest thing to hardware support for virtualization on ARM. In contrast to the latest versions of x86 hardware virtualization extensions, TrustZone does not support memory virtualization and TrustZone does not provide functionality making it easier to decode sensitive instructions for emulation. We plan to further investigate the benefits and options for using TrustZone with KVM/ARM. PikeOS by SYSGO is an ARM hypervisor targeted towards security critical real-time systems.

PikeOS is based on TrustZone [7] and thereby limited to processors with programmable TrustZone support. Unfortunately there are no technical details available on how their solution is engineered in details.

We hope to get the solution included as part of the mainline kernel and QEMU source trees. The solution hardly modifies any code outside the KVM module and even inside KVM there are very few changes to the architecture independent code.

6 Conclusions

We have presented KVM/ARM, the first working open-source ARM virtualization solution based on KVM. Although ARM is not virtualizable, we show how lightweight paravirtualization can be used to automatically enable commodity operating systems to run in a KVM/ARM virtual machine. The approach is minimally intrusive and requires no knowledge or understanding of commodity operating system code, making it relatively easy for KVM/ARM to support more recent versions of commodity operating systems. By building on KVM, KVM/ARM can enjoy the same benefits of KVM, including having its code base kept in line with new kernel releases without additional maintenance costs and being easily included in most Linux distributions. We have implemented a KVM/ARM prototype based on the Linux kernel used in Google Android and have demonstrated its ability to successfully run nearly unmodified Linux guest operating systems.

Our KVM/ARM implementation work has benefited from community support and improvements are ongoing. Newer versions of the ARM architecture provide features to improve KVM/ARM and bring virtualization to embedded devices, smartbooks, and mobile phones in the future. For more information about the project or to get involved, please refer to the project wiki at: `https://wiki.ncl.cs.columbia.edu/wiki/index.php/AndroidVirt:MainPage`.

7 Acknowledgments

Hollis Blanchard, Alexander Graf, and Oren Laadan provided many helpful suggestions in implementing and debugging KVM/ARM. This work was supported in part by NSF grants CNS-0914845 and CNS-0905246, and AFOSR MURI grant FA9550-07-1-0527.

References

[1] Keith Adams and Ole Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *ASPLOS-XII: Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–13, 2006.

[2] AMD Virtualization (AMD-V$^{\text{TM}}$) Technology. `http://sites.amd.com/us/business/it-solutions/virtualization/Pages/amd-%v.aspx`.

[3] ARM Ltd. ARM Architecture Reference Manual (ARM DDI 0100I), 2005.

[4] ARM Ltd. TrustZone Security White Paper, 2010. `http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD%29-GENC-009492C_trustzone_security_whitepaper.pdf`.

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[6] Robert P. Goldberg. A Survey of Virtual Machine Research. *IEEE Computer*, 7(6):34–45, 1974.

[7] Green Hills Software Inc. White paper: Integrity Secure Virtualization for ARM, 2010. `http://www.ghs.com/ds/index.php?ds=integrity_virt_ARM`.

[8] J-Y. Hwang, S-B. Suh, S-K. Heo, C-J. Park, J-M. Ryu, S-Y. Park, and C-R. Kim. Xen on ARM: System Virtualization using Xen hypervisor for ARM-based Secure Mobile Phones. In *Fifth IEEE Consumer Communications and Networking Conference*, pages 257–261, 2008.

[9] Intel®Virtualization Technology (Intel®VT). `http://www.intel.com/technology/virtualization/technology.htm`.

[10] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. **`kvm`**: The Linux Virtual

Machine Monitor. In *OLS 2007: Proceedings of the Linux Symposium*, pages 225–230, 2007.

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

[12] Gerald J. Popek and Robert P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[13] Mendel Rosenblum and Tal Garfinkel. Virtual Machine Monitors: Current Technology and Future Trends. *IEEE Computer*, pages 39–47, 2005.

[14] Sang-bum Suh. Presentation: Secure Architecture and Implementation of Xen on ARM for Mobile Devices, 2007. `http://www.xen.org/files/xensummit_4/Secure_Xen_ARM_xen-summit-04_07_Su%h.pdf`.

[15] VirtualLogix. Real-time Virtualization for Conencted Devices. `http://www.virtuallogix.com/solutions/product/arm.html`.

[16] VMware. VMware Mobile Virtualization Platform, Virtual Appliances for Mobile phones. `http://www.vmware.com/products/mobile/`.

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

UBI with Logging

Brijesh Singh
Samsung, India
`brij.singh@samsung.com`

Rohit Vijay Dongre
Samsung, India
`rohit.dongre@samsung.com`

Abstract

Flash memory is widely adopted as a novel non-volatile storage medium because of its characteristics: fastaccess speed, shock resistance, and low power consumption. UBI - Unsorted Block Images, uses mechanisms like wear leveling and bad block management to overcome flash limitations such as “erase before write”. This simplifies file systems like UBIFS, which depend on UBI for flash management. However, UBI design imposes mount time to scale linearly with respect to flash size. With increasing flash sizes, it is very important to ensure that UBI mount time is not a linear function of flash size. This paper presents the design of UBIL: a UBI layer with logging. UBIL is designed to solve UBI issues namely mount time scalability & efficient user data mapping. UBIL achieves more than 50% mount time reduction for 1GB NAND flash. With optimizations, we expect attach time to reduce up to 70%. The read-write performance of UBIL introduces no degradation; a more elaborate comparison of results and merits of UBIL with respect to UBI are outlined in the conclusion of the paper.

1 Introduction

Flash memories are extensively used in embedded systems for several remarkable characteristics: low power consumption, high performance and vibration tolerance. However flash storage has certain limitations namely “erase before write”, write endurance, bad blocks. The block of a flash memory must be erased before writing again. Besides, each block has limited erase endurance; the block can be erased for a limited number of times. Traditional applications need software assistance to overcome these limitations.

There are two common approaches to deal with the flash limitations. Firstly, a flash translation layer (FTL) that does transparent flash management. It gives a generic disk interface. The traditional file systems like ext2, FAT work unchanged. This approach limits optimizations as file systems are not flash aware.

Second approach uses flash file system. Flash file systems, like JFFS [1], YAFFS [2], are designed to handle flash limitations. In this approach, every flash file system address flash limitations. It is ideal to address them in separate flash layer. This leads us to the third approach. A flash aware file system that can co-operate with a software layer for optimum flash usage. UBI [3] is a software layer designed to follow this approach.

UBI is a flash management layer which also provides volume management. A UBI volume can be a static volume or a dynamic volume. For flash management, UBI provides following functionalities.

- Bad block management
- Wear leveling across device
- Logical to Physical block mapping
- Volume information storage
- Device information

2 Related Work

UBI was developed in 2007. UBI gives logical block interface to the user; each logical erase block (LEB) is internally associated with a physical erase block (PEB). This association is called “Erase Block Association (EBA)”. EBA information of each PEB is stored in VID header. VID header of a physical block resides in the same block. Apart from this, UBI also stores EC header in each physical block; EC header stores erase count of the block. Typical UBI block structure is shown in Figure 1. Initialization of UBI demands processing of both headers from every block. UBI scans complete flash in order to build in-RAM block associations. This introduces a scalability problem. UBI's initialization time scales linearly with respect to flash size; increase in flash size increases mount time of UBI. With flash sizes increasing up to several GB's, it is very important to ensure that UBI mount time is not a linear function of flash size.

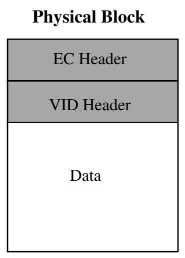


Figure 1: UBI Physical Block Structure

Lei et al. [4] proposed Journal-based Block Images (JBI) which focuses on reducing number of write operations and flash space requirement. To achieve this, JBI uses fragmented mapping table and journal system. Limited work has been carried out to reduce mount time of UBI. To address mount time scalability issue, it is important to avoid scan of complete flash. Possible solution to this problem is to store mapping information in fixed group of blocks on flash.

3 UBIL: UBI with Log

In this paper we present UBIL: "UBI with Log", to address mount time scalability issue. In order to reduce initialization time, UBIL stores block mapping information to the flash. This design consists of super block, commit block and EBA log. Super block which stores location information for commit block and EBA Log, is stored at fixed physical location. Commit block is a snapshot of valid UBI block mapping. EBA Log is a difference between present state and last commit. Commit and EBA Log can move anywhere in flash. Hence these blocks are wear-leveled.

3.1 Super Block (SB)

Super block is stored at two erase blocks in flash. First super block instance is present in first good erase block and second instance is present in last good erase block. The two instances of super block are not mirror of each other. Instead, only one of them contains valid super block entry. Every super block entry occupies page size of flash. To update super block, instead of erasing and writing the block, we log the super block. It means, any update to super block is written in one of the physical blocks.

Super block is written alternatively to one of the two copies (like ping-pong table). As shown in figure, first super block entry 'Entry0' is written on first block, SB0. Next entry 'Entry1' is written to second super block SB1. Subsequent entries Entry2, Entry3... are written alternatively in each block. This gives advantage over mirroring as space is not wasted. Also this improves lifetime of physical blocks reserved for super block.

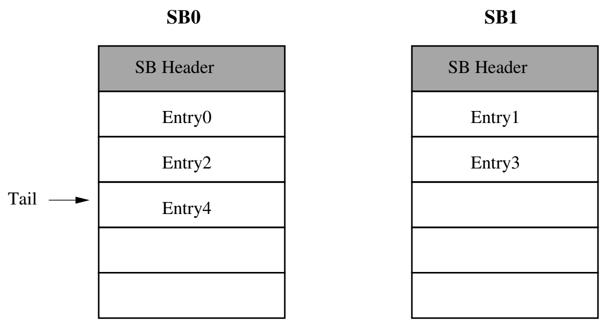


Figure 2: Super Block Update Sequence

While reading super block, we scan through super blocks and find latest written entry, which is a valid super block entry. In Figure 2, valid super block entry is pointed as tails. Writing super block entry may fail. In such situations, other instance of super block contains valid entry.

3.2 Commit block (CMT)

Commit contains mapping information. Size of commit is decided at the time of Ubinize. Depending on partition size, commit may span up to multiple PEBs. Commit information is crucial. Hence two mirror copies of commit are maintained. Even if one of the copies is correct, it is possible to recover the commit. For clean detach, UBI uses commit information during subsequent attach. In case of failure replay of EL is done to restore latest state. Super block contains two map information of commit; present commit and future commit. During commit process, list of future commit blocks in super block is updated first. Then commit is written to these blocks. On successful completion, super block is updated replacing present commit by new commit. Hence commit operation is atomic and tolerant to power failure. If commit is incomplete during detach, all the failed

commit blocks are recovered and given for garbage collection. EL becomes invalid after commit. New empty log is initialized during commit.
Note: UBIL gives option of compressing CMT. This decreases average read/write time of CMT.

3.3 EBA Log (EL)

EBA log contains mapping information of each physical erase block updated after last commit. Hence EL is difference between last commit and present UBI state. Each EL entry contains "EC and VID header" of a physical erase block. EL may contain valid and invalid entries. When EL gets full, only valid entries are written to the commit. After successful commit, old EL is invalid and fresh log is created. This operation is done by reserving new PEBs for EL and handing over old PEBs for garbage collection.
Note: It is possible to configure number of blocks allocated to EL at compile time.

4 UBIL: Initialization

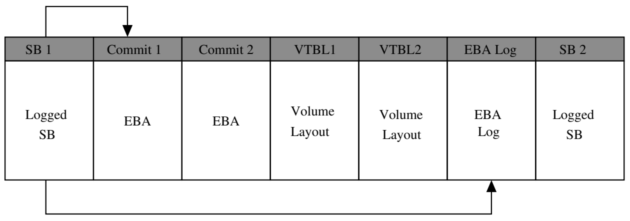


Figure 3: UBIL Flash Layout

UBIL flash layout is shown in Figure 3. UBIL initialization starts with reading super block and finding latest super block entry. Super block locates CMT and EL. For good detach, initialization involves reading CMT. For bad detach, some of mapping information may be present in EL. Hence, initialization involves reading CMT and replaying EL. After successfully reading CMT and EL, other sub-systems of UBIL are initialized. This includes volume initialization, wear-leveling initialization and EBA initialization. During initialization if one of CMT, SB or EL shows recoverable read errors, UBIL initialization proceeds. In this case, after successful initialization of all sub-systems, commit process is called. This guarantees that, CMT is moved to safer erase block, less vulnerable for corruption. Due to removal of scanning, UBIL initialization time is very less as compared to UBI. Steps followed in UBIL initialization are outlined below.

1. Find latest super block by finding tail of super block.
   (a) If the tail is bad (power cut happened while writing super block) the other super block PEB contains valid super block entry.
2. Locate CMT, EL blocks from super block.
3. Generate latest snapshot of UBI.
   (a) Read CMT.
   (b) Apply EBA Log.
4. If previous commit has failed, recover reserved blocks for commit.
5. Initialize Volumes.
6. Initialize Wear leveling.
7. Initialize EBA information.

5 Performance Measurement

We have compared performance of UBIL against UBI on SLC NAND flash. Mount time performance and read-write performance tests were conducted. Tests were performed on Apollon board with OMAP 2420 chipset having 64 MB RAM. We tested UBIL with Linux kernel 2.6.33.

5.1 Mount time performance

UBI attach time increases linearly to partition size. This is due to scanning of complete flash. In case of UBIL, commit size increases with increase in flash size. Causing UBIL attach time to increase marginally. But this increase is very minimal in comparison to UBI. Mount time performance comparison is shown in Figure 4. It is evident that UBIL performs far better than UBI. As partition size increases, UBIL performs better than UBI in terms of attach time. UBIL achieves more than 50% attach time reduction for 1 GB NAND flash.

As partition size increases, UBIL performance better than UBI in terms of attach time. UBIL achieves more than 50% attach time reduction for 1GB NAND flash.

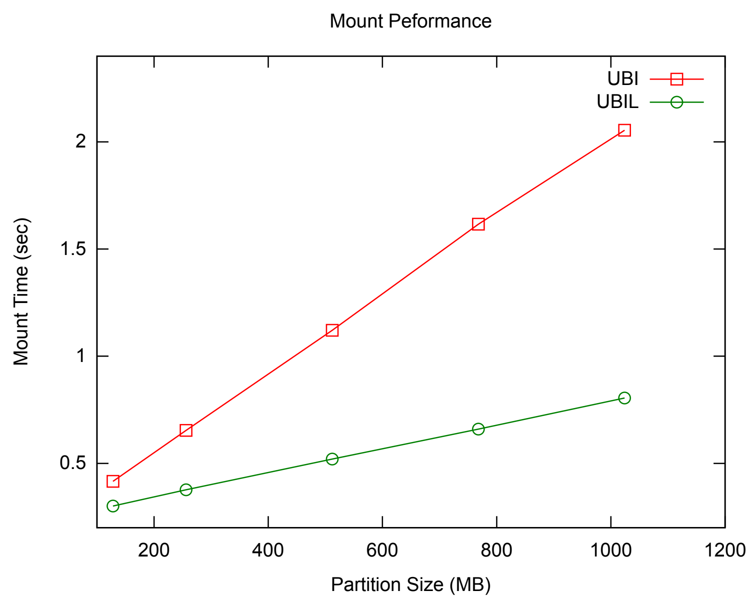


Figure 4: Mount Performance : UBIL vs UBI

5.2 Read-Write Performance

This test measures actual file system read-write performance. For performing this test we used Iozone running on partition mounted with UBIFS. In read-write test we performed sequential and random read-write tests. Performance measurements are given in Table 5.2. It can be inferred from table that there is no significant effect on read-write performance. This is because, UBIL EL writing frequency is same as meta data write frequency of UBI.

Table 1: IO Performance

Operation	UBI (MB/s)	UBIL (MB/s)
Read	6.33	6.33
Write	3.49	3.71
Re-read	6.33	6.33
Re-write	3.39	3.64

6 Conclusion and Future Work

In this paper we presented UBIL to effectively deal with mount time scalability issue of UBI. While UBI stores mapping information across flash, we maintained mapping information at one place. This significantly reduce mount time by avoiding full flash scan. Bedsides UBIL, do not perform any extra read-write operation, causing read-write performance comparable to UBI. As discussed in results, Our approach reduces mount time by 50% without affecting read-write performance.

Commit process can be optimized in future by writing EBA mappings directly to the flash. As per present UBIL design, super block is written at fixed location. These blocks are not wear-leveled. Super block handling can be improved by using block chaining scheme as discussed in JFFS3 [6] design.

References

[1] D. Woodhouse, *JFFS: The Journaling Flash File System*, In Proceedings of 2001 Linux Symposium, Ottawa, Canada, July 25-28, 2001

[2] *YAFFS: Yet Another Flash File System*, http://www.yaffs.net

[3] T. Gleixner, F. Haverkamp, A. Bityutskiy, *UBI-Unsorted Block Images*, http://www.linux-mtd.infradead.org/doc/ubidesign/ubidesign.pdf

[4] Lei Jiao, Y. Zhang, W. Lin*Journal-based Block Images for Flash Memory Storage Systems*, The 9th International Conference for Young Computer Scientists, pp. 1331-1336, 2008

[5] D. Woodhouse, *Memory Technology Device (MTD) Subsystem for Linux*, http://www.linux-mtd.infradead.org/doc/general.html, Feb 2010

[6] A. Bityutskiy, *JFFS3 Design Issues*, Version 0.25, http://www.linux-mtd.infradead.org/doc/JFFS3design.pdf, Oct 2005

[7] Brijesh Singh, Rohit Dongre, *UBIL Performance Log for NAND*, http://git.infradead.org/users/brijesh/ubil_results /blob/HEAD:/nand_mount_ti me.pdf

[8] Brijesh Singh, Rohit Dongre, *UBIL- UBI with Log*, Source code, http://git.infradead.org/users/brijesh/ubi-2.6.git

Looking Inside Memory

Tooling for tracing memory reference patterns

Ankita Garg, Balbir Singh, Vaidyanathan Srinivasan
IBM Linux Technology Centre, Bangalore
`{ankita, balbir, svaidy}@in.ibm.com`

Abstract

Memory is a critical resource that is non-renewable and is time consuming to regenerate by reclaim. While there are several tools available to understand the amount of memory utilized by an application, there is presently little infrastructure to capture the physical memory reference pattern of an application on a live system. This knowledge would enable the software developers and hardware designers to not only understand the amount of memory used, but also the way the references are laid out across RAM. The temporal and spatial reference patterns can provide new insights into the benchmark characteristics, which would enable memory related optimizations. Additional tools could be developed on top to extract useful data from the reference information. For example, a tool to understand the working set size of an application, and how it varies with time. The data could also be used to optimize the application for NUMA systems. Kernel developers could use the data to check fragmentation and generic data placement issues.

In this paper, we introduce a memory reference instrumentation infrastructure in the Linux kernel that is built as a kernel module, on top of the trace framework. It works by collecting memory reference samples from page table entries at regular intervals. The data obtained is then post processed to plot various graphs for visualization. In this paper, we would provide information on the design and implementation of this instrumentation, along with the challenges faced by such a generic memory instrumentation infrastructure. We will demonstrate few additional tools built on this infrastructure to obtain interesting data collected from several benchmarks. The target audience are people interested in kernel based instrumentation, application developers and performance tuning enthusiasts.

1 Introduction

Typically, application developers are abstracted from the physical view of the memory by the memory management subsystem of the operating system. An application would request for memory using several well-defined APIs, which is then serviced by the operating system. The OS uses sophisticated algorithms to ensure that the memory allocation for a particular application takes place from the most appropriate location, for example, on a NUMA system, memory requests should be satisfied from a local node to reduce overhead in accessing remote memory. However, a developer looking for optimizations, can greatly benefit by having a good understanding of the way the memory is being used by the application. Finding a program's memory usage on Linux is complex. However, utilities like `ps`,`free`, `vmstat`, and proc filesystem interfaces like `/proc/meminfo` etc, provide a goof estimate of the various memory usage statistics.

Most of the existing tools, however, generally only provide information about the memory usage statistics of an application and/or system. There is very little information available on the way the memory is being referenced or the classification of accesses into kernel, user and buffers. Also, it is difficult to obtain read/write access patterns. All this information can however be obtained from hardware simulation but at a much slower speed. The infrastructure we propose here enables gathering the above missing pieces of information on a live system, running real-world applications.

2 Memory Organization

Memory is typically organized in a hierarchical manner in modern systems to have the right balance of access speed and cost. Accesses to memory addresses or pages are first looked up in TLBs (or page tables) to get the physical address (which is needed for physically tagged cache) and then searched in several levels of caches (L1, L2 and/or L3). If the addresses are not found, then the right cache line is looked up in the main memory and loaded into the cache and registers for use. A hierarchy of page tables or similar address translation mechanisms are used by the hardware memory management unit (MMU) in the processor to map a virtual address to a physical address in memory. On most architectures, a page table entry (PTE) consists of mainly the page's physical address, and other attributes like access permissions and a **referenced** or **accessed** bit. The processor sets this bit when the page is accessed. Once set, the processor itself does not clear the bit. Software is expected to clear this bit to record new accesses. PTE also typically contains another bit called the **dirty** or **changed** bit, which indicates whether a page has been written to or not.

Memory pages can be broadly classified into :

- Kernel pages – This corresponds to the kernel code including device drivers and its data
- User pages – The application pages (both code and data)
- Page cache or buffer pages – Indirectly used by applications (unless mapped)

3 Memory Reference Pattern Instrumentation Infrastructure

The objective of the instrumentation is to track references to every page of memory over a given time interval. As explained before, memory is referenced using the page tables. Using these page table entries, memory references could be tracked in a number of ways.

3.1 Tracking Page Faults

One of the approaches to capture references is to modify the PTE entries such that a page fault is generated at every access. The fault could then be handled in a custom page fault handler routine. This could be achieved fairly easily for user pages. One could also use this data to find the exact task/routine referencing the memory. The reference data obtained thus would be very accurate. However, the disadvantages of the approach out-weigh the benefits:

- Page fault for every memory reference would slow down the system tremendously, resulting in difficulty in running long-running, real-world benchmarks and also interfere with the benchmark execution itself.
- Reference data obtained would be voluminous and cumbersome to post process unless this level of accuracy is needed
- Complex to capture kernel page reference pattern with this approach, as all the kernel pages are present in memory and not demand paged in or out. Most parts of the kernel will expect the translation to be available and do not expect a page fault.

Alternatively, to get an estimate of the reference pattern and to reduce the overhead, we could periodically reclaim or unmap several pages and follow what gets faulted in. Also, by setting up a trace event in the page fault handler (to be triggered occasionally) and forcing reclaim to reduce the RSS, we can get information on the address range of pages that are being referenced by an application. However, this technique would not be useful in getting system-wide memory reference data. While this approach can estimate the RSS and virtual address range, it cannot estimate the per-page reference rate over a short sampling interval compared to just sampling the page reference bit as explained below in section 3.3

3.2 Performance Counters

Performance counters are special hardware registers available on most modern platforms. These registers keep a track of the count of certain types of hardware events, like, instructions executed, cache misses suffered, or branches mis-predicted. Since the counting is done in the hardware, it does not slow down the kernel or applications. The Linux Performance Counter subsystem provides an abstraction of these hardware capabilities, which would work depending on the support in the underlying hardware platform.

We could use the hardware cache miss counter to approximate the memory accesses, since a cache miss implies a memory reference. However, the absolute number of cache misses does not reflect the distribution of physical memory accesses, as a memory reference could happen without a miss as well (when the page is in cache). Besides, as noted before, it might not be possible to use this approach on platforms with no such counter. Further, it might not be possible to classify accesses into read/write.

3.3 Sampling Using PTE Reference bit

As explained in section 2 the referenced bit of the PTE tracks references to pages. In this method we adopt a sampling based approach, in which, at every sampling interval, we scan all the PTEs, looking for all the reference bits that were set to one since the previous sampling interval, indicating that those pages were accessed. We make a note of all such PTEs, along with their physical addresses. The reference bit is then cleared[1], to enable data capture for the next sampling interval. This process is repeated at every sampling interval. The particular page tables that are scanned define the type of pages that were referenced. For user space page references, the page tables of either a given task or of all the tasks are sequentially scanned for the reference bits To capture kernel page reference pattern, different approaches are needed for different platforms. On x86 systems, the `pgd` field in `struct mm` of the `init_task` points to the kernel page table directory. Using this pointer, the kernel page tables can be walked in a manner similar to user space pages. However, on Power platform, the page tables are handled differently. The hypervisor maintains a hash table of the page table entries. The entries corresponding to the kernel are bolted in the hash table. For every address, a key is generated, called `vsid`, that is used to hash into the table to obtain the PTE.

It is important to note here that if page access is served from the hardware cache, the reference bit in the PTE would still be set by the hardware and not strictly based on cache miss and memory access.

3.4 Design Overview

We have adopted the approach described in section 3.3 in our implementation. It can be easily seen that if the data is captured for every single page in the system, a lot of memory would be consumed in just capturing the information and also would pose difficulties in post-processing. Thus, instead of gathering data for every page, we group the pages in chunks and collect data for the group instead of individual pages. If any page within a group was accessed, the entire group of pages is marked as having been accessed in a particular sampling interval. On Power systems, we use the logical memory blocks (LMBs) as a means to group pages. LMBs are groups of contiguous memory, usually of 64MB or 128MB (tunable), that the hyperviser uses to give out memory to the partitions. On x86 systems however, there is presently work in progress [5] to create a notion of LMBs, in the absence of which, contiguous pages are internally grouped for the purpose of collecting data. The granularity of this group determines the accuracy of the reference pattern captured. The smaller the group size, the more accurate the data.

Seq ID	Phys Addr	Kernel	User	Page Cache	R/W

Figure 1: Output Data Format

At every sampling interval, the data that is captured is illustrated in Figure 1

The `sequence id` is a unique identifier associated with a sampling interval. The `physical address` corresponds to the physical address of the first page in a given group. The next three fields, `kernel, user and page cache`, indicate the count of the number of pages referenced in each types of page within the group. This helps in estimating the working set size of an application and also how it varies with time. We can derive more information by classifying the references into kernel, page cache or user pages. The last fields, `read and write`, indicate the number of pages in the group that had reads and writes, which is detected using the dirty or the changed bit in the PTE.

By default, the page tables of all the processes in the system are scanned for references, in addition to the kernel page tables. The user can also specify the pid of a task for restricting the PTE scanned.

3.5 perf Integration

To make it easy to use the memory reference pattern tracing infrastructure, we have integrated it with the `perf(1)` [2] framework. A trace event called `memref`

[1]The disadvantage of this approach is that it could interfere with the LRU algorithm that the reclaim subsystem uses. We therefore run these tests in a system that has sufficient memory to not enforce reclaim of pages, while we run our tracing framework

```
memref-2680  [005]  3549.112460: memref_log_data: 1230 268435456 12 0 0 1
memref-2680  [005]  3549.112461: memref_log_data: 1230 301989888 20 0 0 0
memref-2680  [005]  3549.112461: memref_log_data: 1230 335544320 0 0 0 3
memref-2680  [005]  3549.112462: memref_log_data: 1230 369098752 0 56 0 0
memref-2680  [005]  3549.112462: memref_log_data: 1230 402653184 0 10 0 1
memref-2680  [005]  3549.112463: memref_log_data: 1230 436207616 19 34 0 0
memref-2680  [005]  3549.112463: memref_log_data: 1230 469762048 0 0 0 3
memref-2680  [005]  3549.112464: memref_log_data: 1230 503316480 0 45 0 0
```

Figure 2: memref, integration with perf

is defined, which when enabled, starts capturing the reference pattern data. The `pid` is specified by echoing a value in the `memref_pid` file, created under the `tracing` directory. The binary data obtained can then be post-processed to obtain useful information, as discussed in the next subsection. Figure 2 shows a sample output from the `memref` trace event.

The first 3 columns indicate the process running, CPU number and time when the trace was captured. `memref_log_data` is the name of the trace event. Fifth column indicates the sequence number associated with the particular sampling interval. The next column indicates the physical address (corresponds to the start of the page group or a LMB). The next three columns are for kernel, user and page cache respectively. The value of the column indicates the number of pages of a particular type referenced with the group. The last column provides information on whether the access was read (0), write(1) or none (2). Also, within a group, there would be both reads and writes. However, here we trade off accuracy for the ease of data capture, by marking the whole group as having write reference.

3.6 Data Representation

The data obtained, either in raw binary or ASCII format, needs to be post-processed to obtain useful information about the reference pattern. A few useful representations of the data are as follows:

- Temporal Reference Pattern – With sampling time plotted on the X-axis and the total number of LMBs referenced on the Y-axis, we get the temporal reference pattern plot, which indicates the total amount of memory referenced at a particular time.

- Spatial Reference Pattern – The spatial reference plot indicates the number of times a given LMB was referenced over the period for which the data was collected. The LMBs are plotted on the X-axis and the reference count on the Y-axis. The plot also helps in understanding how the memory accesses are laid out in the RAM.

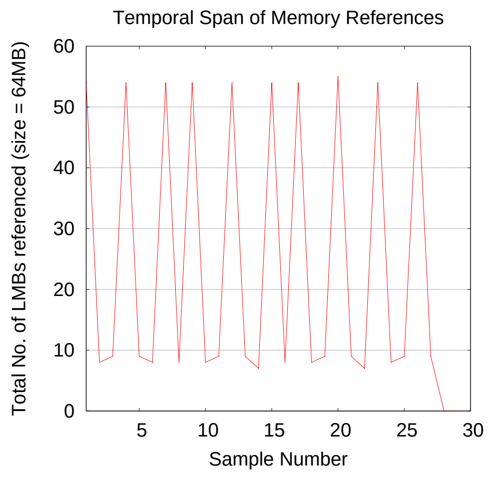


Figure 3: Temporal Reference Pattern for pagetest

- Working Set Size – The working set size of any process is defined as the set of all pages referenced by it in a given time interval. From the instrumentation data, we can compute the working set size of a process for a given sampling interval, by summing up the total number of pages referenced in each group of pages within that sampling interval, for a particular type of reference. Assuming that the page references are stable across sampling interval, one could extend it further by grouping samples to form a large time interval for computing working set size. It is important to note here that if a page is being referred to by both user and the kernel space, the kernel and user references would be accounted for separately for that page.

4 Data Verification

In order to verify the functionality of the framework, we obtain the memory reference pattern for the `pagetest` program. `Pagetest` allocates a chunk of memory, as specified by the user, using either malloc, anonymous mmap, file mmap or shared memory. It then performs either a `write` or a `read` operation on every single page that was allocated, for a specified number of iterations. To verify that the instrumentation indicates all the memory touched by the program as having been referenced, we tweak the test program to make it sleep for a

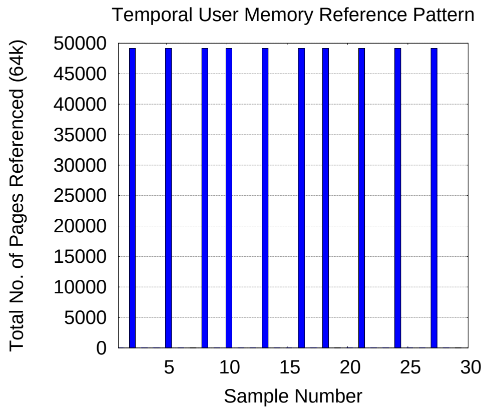


Figure 4: Temporal User Memory Reference Pattern for pagetest

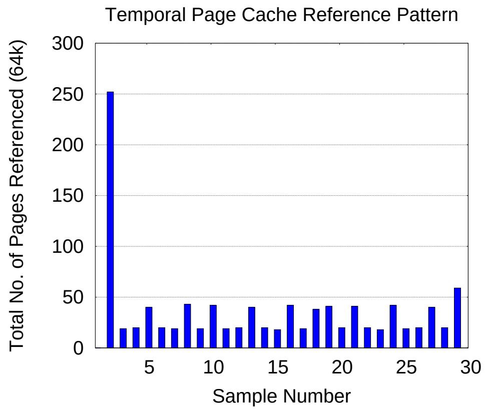


Figure 6: Temporal Page Cache Memory Reference Pattern for pagetest

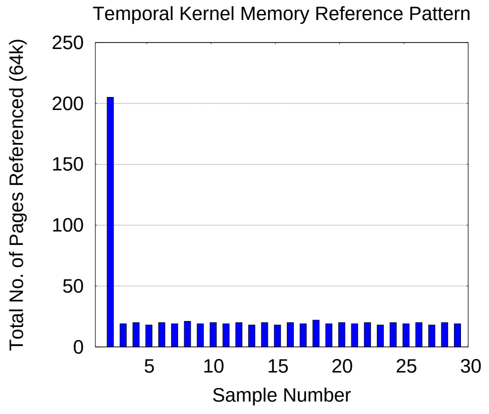


Figure 5: Temporal Kernel Memory Reference Pattern for pagetest

fixed amount of time after every iteration of touching all the pages. We run the instrumentation in parallel, at an interval that ensures that the reference data is captured after an iteration of `pagetest` is complete or in between iterations. This ensures that the instrumentation does not interfere with the reference information of the benchmark (solely for the purpose of verification). We run the benchmark to allocate 3GB of memory using `malloc`. We let the program run for 10 iterations. The temporal reference pattern span is as shown in Figure 3.

From the graph we can see that there are 10 spikes, with a height of about 52 LMBs, corresponding to the 10 iterations and usage of 3GB in each iteration. Flat lines at the bottom indicate periods when the benchmark was sleeping and no references were generated to its pages. We also verify the data obtained regarding classification of accesses into user, kernel and page cache. Figure 4 indicates the number of user pages accessed in a given sampling interval. About 49165 pages (each of size 64k) were marked as being referenced. This equals 3GB. Similarly, Figure 5 and Figure 6 indicate the number of kernel and page cache pages that were referenced respectively. Both the kernel and page cache account for less than 2MB of references each.

5 Errors & Approximations

The proposed method of aggregating memory reference patterns has known approximations and errors as detailed below:

- Effect of cache hierarchy – The primary mechanism through which the reference pattern is aggregated is the reference bit in the hardware memory management unit's page table entry. This bit is updated (or set) whenever there is a memory access to the region translated by this page. That region of memory could have been in the caches as well. Basically the reference bit is updated independent of whether the actual access was a cache hit or a cache miss. Hence if we have been looking to model bus traffic or traffic at memory controller level, then this metric needs to be correlated with last level cache miss counts to get a reasonable estimate.

- Effect of sampling – As described earlier, this method is based on periodic sampling and hence the resolution of information obtained depends on

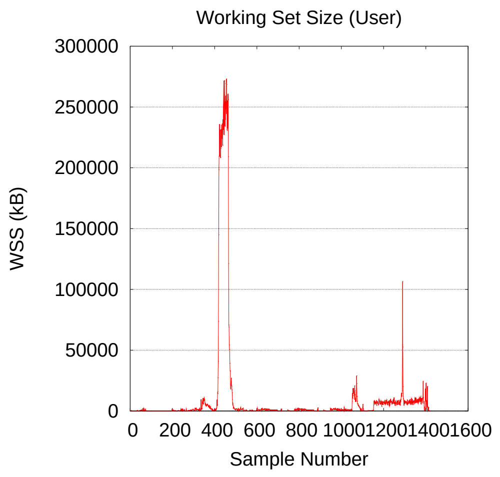


Figure 7: User Memory Working Set Size of kernbench with 64 threads

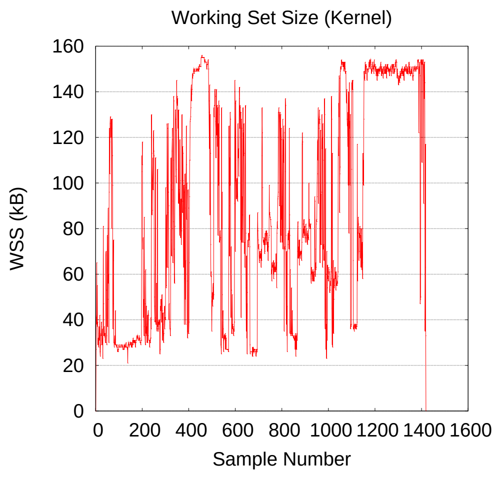


Figure 8: Kernel Memory Working Set Size of kernbench with 64 threads

the sampling interval. There are severe constraints in the lower bound of the sampling interval. Based on the size of the system and amount of data to be collected, the sampling interval may have to be larger leading to further approximation of the reference data. However one can assume that references with high temporal and spatial locality will most likely be cached and hence will not actually hit the bus and memory chips, leaving scope for optimization.

- Missing reference information – The memory regions used to store the page tables themselves may not be marked as referenced by the MMU hardware, though they may have significant reference rate. Similarly, IO DMA activities and other direct memory manipulation that does not traverse the processor MMU unit may not have been captured.

6 Sample Use Cases

The reference pattern information can be useful in several scenarios. Below we present information on some sample use cases for the different type of data obtained.

6.1 Working Set Size of an Application

Working Set Size (WSS) of an application is the amount of memory that is required to be present in the main memory at any time during its execution. The working set size could vary at different times of execution, depending on the application design and the amount of data being worked upon. If the WSS for an application is much bigger than the memory present on the system, some of the application pages would be swapped out to accommodate newer pages. This could lead to reduced performance as the swapping activity increases.

Accurately finding out the total memory that is being used by a process is complex. There is very little tooling that exists which can indicate the WSS of an application. An estimate of the instantenous WSS of an application could be obtained using the memory reference pattern data. The WSS information can be derived from the data about the number of pages accessed within a sampling interval. Ideally, the sum of the total number of kernel, user and page cache pages would be the WSS of the application for a given period. However, we present the data for each category separately, to highlight the capability of the framework. Figure 7, Figure 8 and Figure 9 show the WSS we obtained for the `kernbench` benchmark, running 64 threads on a machine with 16GB RAM and 8 processors. The WSS for user memory remains below 1MB for most times, however, hits a maximum of about 250MB. On the other hand, the maximum amount of kernel memory referenced is around 150-160KB, throughout the benchmark execution time. Page Cache memory references constitute to most of the memory references, the maximum instantaneous WSS being slightly over 500MB. This in-

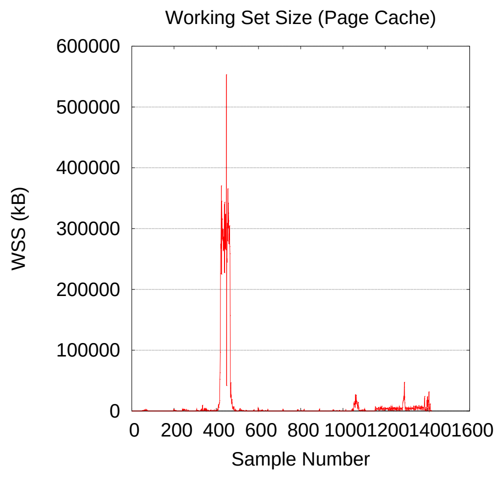


Figure 9: Page Cache Memory Working Set Size of kernbench with 64 threads

dicates that if the performance of kernbench has to be improved from memory perspective, its the optimizations in the page cache layer that would have the biggest impact. Besides, this information also indicates that if 64 threads of kernbench are to be run on this platform, maximum memory requirement would be atleast 500MB, for minimal performance impact.

Contrast this observation with the graphs in Figure 10, Figure 11 and Figure 12, which corresponds to kernbench run with only 16 threads on the same machine. It can be seen that the time taken for the benchmark is almost 50% more than the previous run. The amount of kernel memory referenced remains the same. User and page cache memory referenced are below 40MB.

6.2 Usage of Large pages

A virtual address in the virtual memory is translated into the physical address by a combination of hardware and software operations. A page is the smallest entity of address translation. Page tables store the mapping of virtual addresses to physical page addresses. There is some overhead involved in a single page translation. The total number of translations required for a program depends on the number of pages that are accessed by it. The overhead increases as the number of pages accessed are increased. Translation Lookaside Buffers (TLB) are used to reduce the overhead, by caching the frequently used page table entries. Depending on the workload, the TLB may not be sufficient to cache all the translations

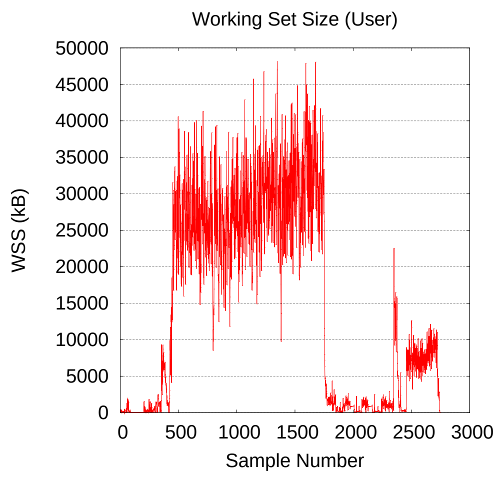


Figure 10: User Memory Working Set Size of kernbench with 16 threads

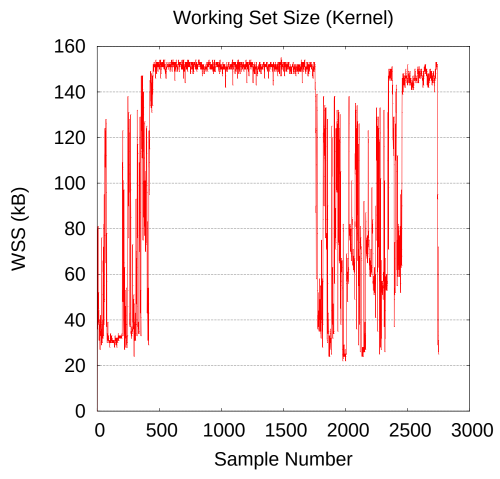


Figure 11: Kernel Memory Working Set Size of kernbench with 16 threads

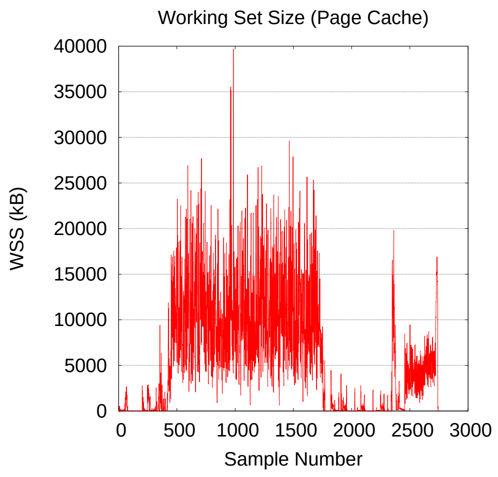


Figure 12: Page Cache Memory Working Set Size of kernbench with 16 threads

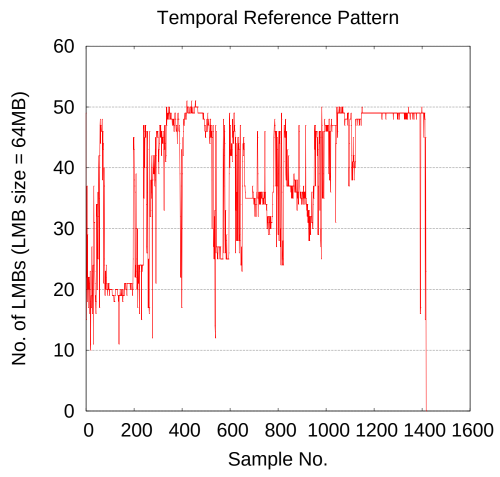


Figure 14: Temporal Reference Pattern for kernbench run with 64 threads

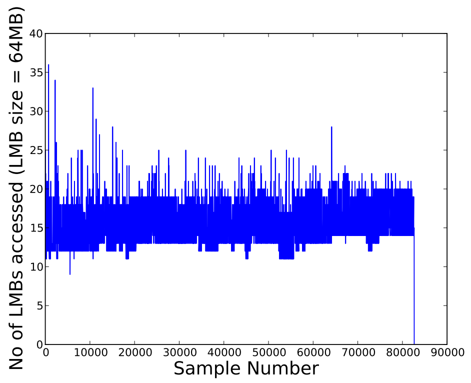


Figure 13: Temporal Reference Pattern for a JAVA Benchmark

needed by the application. Large page [4] support was thus developed to improve the performance for such applications. Large pages enable fewer TLBs to translate larger address ranges, thus allowing more entries to fit into the TLBs.

Use of large pages benefits applications that have a dense memory reference pattern. If the workload reference pattern is very sparse and making a small number of references, usage of large pages might in fact negatively impact performance, depending on the number of large page entries supported by the TLB and also due to memory fragmentation. Thus, the memory reference pattern would enable the application developer to estimate if the usage of large pages would yield performance improvement. Figure 13 indicates the temporal reference pattern of a JAVA benchmark. The graph shows a dense reference pattern and on an average, the memory reference span is about 10 LMBs, which equals about 640MB of memory. Thus, it can be inferred that this benchmark could benefit from usage of large pages.

As an example of an application that would not benefit from using large pages is kernbench. From the Figure 14, we see that its reference pattern is not dense and varies with time. Usage of large pages could potentially lead to memory fragmentation. This is a hypothetical analysis, but could serve as a good starting point when analyzing benchmark performance issues.

6.3 Understanding Memory Usage in NUMA Systems

On a NUMA system, memory allocation becomes slightly more complex due to the presence of local and remote node. There are a number of NUMA policies that determine where the memory for a particular process will be allocated from. For example, by default, memory is allocated on the node of the CPU that triggered the allocation. It is important to determine which NUMA allocation policy works best for a given application, since a wrong policy could lead to performance degradation as there is an additional overhead incurred when accessing memory from a remote node. The memory reference instrumentation framework can aid in understanding the way memory is utilized by an application on a NUMA system. Figure 15 indicates the spatial reference plot for `kernbench`, run on a JS22 blade

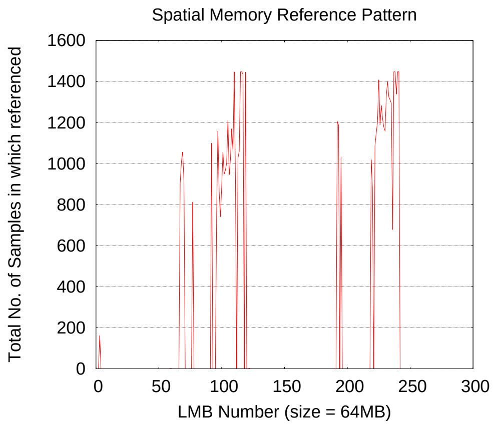


Figure 15: Without NUMA Biasing: Spatial Memory Reference Pattern of kernbench

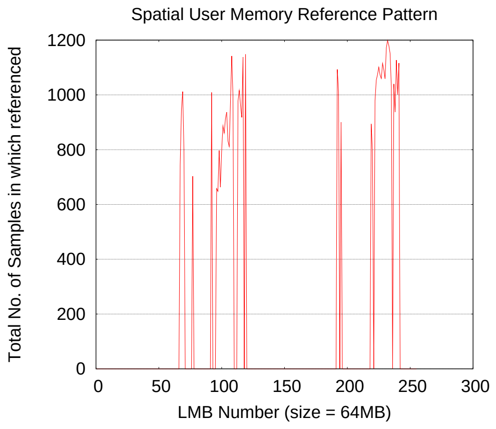


Figure 16: Without NUMA Biasing: Spatial User Memory Reference Pattern of kernbench

POWER6 blade, which had 16GB of RAM (8GB on each node). The LMB size used was 64MB. Thus, about 128 LMBs correspond to one NUMA node. It can be seen that the memory references come from both the NUMA nodes. Figure 16, Figure 17 and Figure 18 show the way references are spread across for user, kernel and page cache memory. However, when `kernbench` threads were tied to processors belonging to only a single NUMA node, we can see from Figure 20 that the user memory references originate from that node only. However, kernel and page cache references still spread across the nodes, as seen from Figure 21 and Figure 22, still giving an overall memory reference pattern as in Figure 19, similar to the one in Figure 15.

Today, one could use `numastat`[1] to obtain information on NUMA access statistics that are obtaied from hardware and maintained by the kernel, like `numa_hit`, `numa_miss`, `local_node`, `other_node`, etc. When an application is run in isolation, these counters would be a representative of the application itself. Also, `/proc/<pid>/numa_maps` gives information about how the process user pages are laid out across the NUMA nodes. However, with the help of the proposed instrumentation, we can also classify the NUMA accesses to kernel, user and page cache and also into read/write accesses. Such information would prove helpful to developers, chasing the cause of peculiar benchmark performance issues on NUMA platforms. This also serves as an effective tool to evaluate NUMA policy implementation in the Linux kernel.

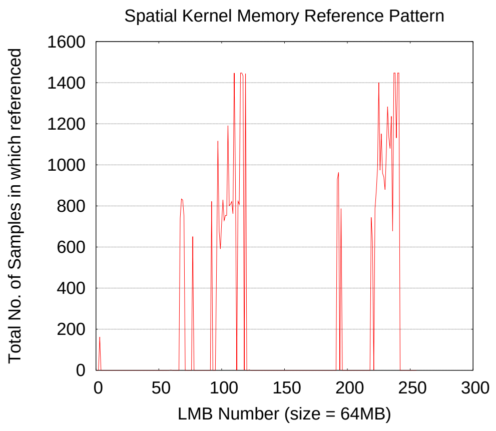


Figure 17: Without NUMA Biasing: Spatial Kernel Memory Reference Pattern of kernbench

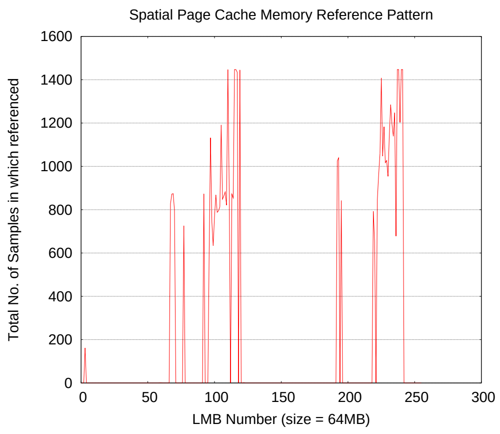


Figure 18: Without NUMA Biasing: Spatial Page Cache Memory Reference Pattern of kernbench

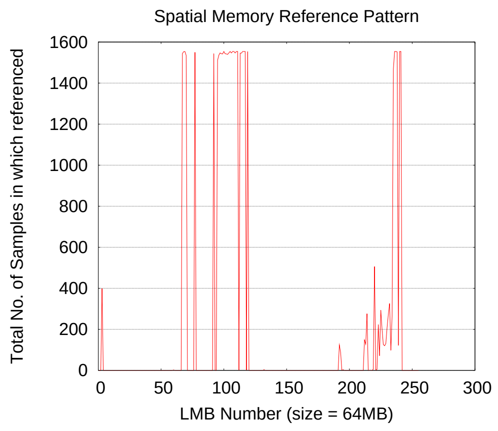


Figure 19: Biased to a NUMA Node: Spatial Memory Reference Pattern of kernbench biased to one NUMA node

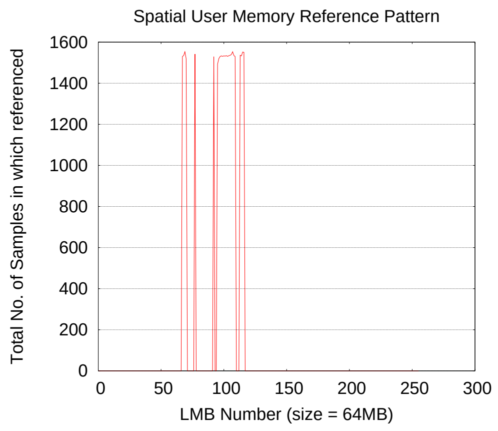


Figure 20: Biased to a NUMA Node: Spatial User Memory Reference Pattern of kernbench biased

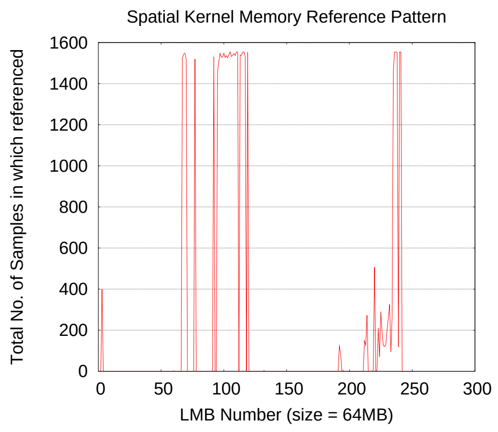


Figure 21: Biased to a NUMA Node: Spatial Kernel Memory Reference Pattern of kernbench

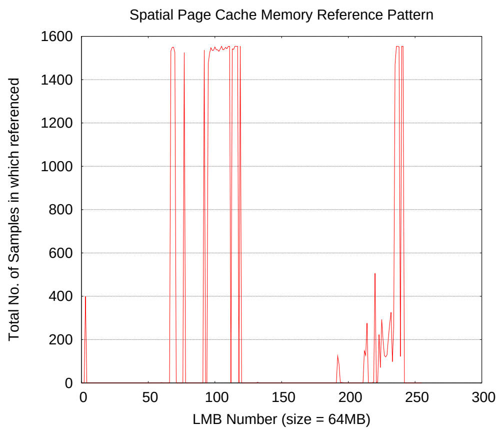


Figure 22: Biased to a NUMA Node: Spatial Page Cache Memory Reference Pattern of kernbench

7 Challenges

The goal of the instrumentation is to ensure that the data is captured with minimal impact on performance and no interference with the benchmark data. Some of the challenges that we faced are as follows:

- Missing information in Software – It is important to understand that memory references, either from caches or from main memory, are transparent and concurrent to the operating system. Due to this, the precise count of the number of times a particular page was referenced cannot be obtained from software. Hardware support in the form of counters that maintain a per-page reference data could improve the accuracy, as in System-Z [3]. Thus, from inside the software, we can only obtain information about whether a page was referenced or not within a sampling interval.

- Indeterminate run time of the instrumentation – The size of the kernel page table remains almost a constant during system runtime. Thus the time taken to scan it also remains a constant. However, the amount of time taken to scan the user page references is directly proportional to the number of processes active and their memory footprint. The larger the number of such processes, or larger their memory footprints, the greater the time required to collect the reference samples and vice versa. As a result, the run time of the scanning framework increases, leading to fewer samples being collected.

The data thus obtained may not be a true representation of the actual memory reference pattern.

- Concurrent execution of software and instrumentation – The reference data that is being obtained is simultaneously being updated or changed by software running on other CPUs. This also has an effect on the software where one of the software threads are delayed due to reference collection kernel thread thereby potentially delaying other software and affecting normal program execution like inducing lock contention.

8 Future Enhancements

Based on the limitations and challenges listed above, we have a good list of things to work on. Potential future improvements are:

- Compress or reduce data that is logged – Basically use simple encoding techniques to reduce the amount of memory used and data transferred to user space.
- Adaptively sample interesting areas of memory – Start with scanning full memory and page tables, but if we can quickly figure out the stale areas, i.e, memory areas that are rarely being accessed, we can make a note of them and scan them less frequently. We keep updating our statistics of such areas.

9 Conclusions

The rapidly developing Linux runtime tracing framework enables low overhead, complex and intrusive instrumentation to get interesting data and facilitate deeper insights into application behavior. Application memory reference tracing is one example where a combination of known techniques can be easily packaged and aptly used to get new insights into the way memory is accessed by an application, that could lead to optimizations.

10 Acknowledgements

The authors wish to thank their team at Linux Technology Centre, IBM and the management for their encouragement and support during the creation of the instrumentation framework and the paper. We would like to thank all colleagues who reviewed the patches and gave valuable feedback. Special thanks to Dipankar Sarma for his guidance all throughout.

The authors wish to thank the IBM management who generously provided an opportunity to work on this feature and paper, without which its presentation at the Linux Symposium 2010 wouldn't have been possible.

11 Legal Statement

References

[1] numastat. Linux kernel Documentation, src/linux/Documentation/numastat.txt.

[2] Performance counter frameowrk for linux. `http://perf.wiki.kernel.org/`.

[3] System z performance counters. .

[4] Large page support in the linux kernel, August 2002. `http://lwn.net/Articles/6969/`.

[5] Use lmb with x86, June 2010. `http://lkml.org/lkml/2010/6/16/32`.

Dynamic Binary Instrumentation Framework for CE Devices

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

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

Jaehoon Jeong
SAIT, Samsung Electronics
hoony_jeong@samsung.com

Abstract

Developers use various methods and approaches to find bugs and performance bottlenecks in their programs. One of the effective and widely used approach is application profiling by dynamic instrumentation. There are many various tools based on dynamic instrumentation. Each tool has its own benefits and limitations what often forces developers to use several of them for profiling. For example, in order to use `Kprobe`-based [1] Systemtap [2] tool developers need to write instrumentation script using special language. To use `Dyninst` [3] profiling library developers need to write instrumenting programs in C++. Thus each tool realizes its own profiling technology. Additionally various profiling tools produce output data in their own formats and those formats are incompatible. Thus two above problems significantly increase complexity of debugging.

In this paper we describe unique dynamic binary instrumentation engine concept which is used in our monitoring tool — System-Wide Analyzer of Performance (SWAP). This tool has modular open architecture and API which allow integrating various tools for providing powerful instrumentation and analysis framework for developers. `Dyninst` and `Kprobe`-based instrumentation engines are integrated into SWAP framework and used in a similar way. Modular structure of SWAP can be extended with other instrumentation and analysis methods by easy way. Also SWAP has several levels of API: instrumentation API, connection API, control API, user interface API and monitoring language framework API. This multilevel API architecture allows developers to re-use SWAP functionality and embed it into their own solutions. All above mentioned SWAP advantages essentially simplify debugging profiling process for embedded software.

1 Introduction

During the last decade computer market has been shifting its focus from PCs to consumer electronic devices that have made great steps in increasing their functionality. Now CE embedded systems can provide wide set of capabilities for user. Devices interact directly with consumers and quality of their work influence manufacturer's brand greatly. So bugs and ineffectiveness of software are very critical for CE products. It is agreeable that Linux has become popular as a platform for modern embedded systems. However, it still has issues to be solved due to limited resources (i.e. CPU power & memory size) and absence of network interface in the embedded environment. Linux developers use various methods and approaches to find bugs and performance bottlenecks in their programs. One of the effective and widely used approach is application profiling by dynamic instrumentation.

There are a number of tools based on dynamic instrumentation for *nix systems. Every tool has its own list of supported features that can be split into the following categories:

- Supported operating systems, kernel versions;
- Supported processor architectures;
- Instrumentation capability (functions entries/exits, functions bodies, certain types of instructions etc.);
- Type of collected information (variables, stack, user or kernel space data etc.);
- Instrumentation overhead;
- Format of collected profiling data;

Set of features supported by particular tool is limited, and the list of features provided by tool determines its advantages and disadvantages for developers. For example, well-known `Kprobe`-based tool `Systemtap` which provides powerful script language for dynamic instrumentation does not support MIPS architecture. Another well-known tool `LTTng` [4] supports great set of architectures (including MIPS), but its instrumentation capabilities are limited by hooks inserted into source code.

Since every tool has limited set of supported features developers should often make choice what tool to use in certain circumstances. The problem of choosing a suitable tool for dynamic instrumentation and profiling is widely discussed over the technical forums in the internet, Recently a comparison report [5] has been published for the tools mentioned above and `DTrace` [6], another instrumentation tool for Solaris, Mac OS X, BSD and QNX. All those discussions and comparison report were intended to provide developers with summarized view of available profiling functionality on different platforms.

As it was mentioned above features provided by single tool can be insufficient for developers, so it is reasonable to consolidate the advantages of multiple tools simultaneously to perform one experiment: typical case which usually requires the using of several tools is instrumenting both user and kernel space. But using several tools on CE devices is not quite convenient due to the following reasons:

- Some tools can not be executed on CE devices in standalone mode, due to device resource limitations, so experiments which need to run multiple tools simultaneously can not be performed;
- Not all tools support host-target architecture which is preferable for resource limited CE devices, so experiments which need to run multiple tools simultaneously can not be performed;
- Using multiple tool complicates instrumentation process, because instrumentation scripts/programs needs to be written for several tools;
- All tools have different format of collected data, so additional scripts/programs are needed to bring different output data to one format which can be used for analysis;
- Quite often data, collected using several tools, should be merged and synchronized in order to create ordered sequence of events which reflects the essence of experiment;

All above problems significantly increase complexity of profiling using multiple tools.

Let's consider the following example (see Figure 1) of profiling programs using multiple tools. Developer wants to analyze how many page faults are generated by memory accesses made by considered program and if it is possible to find access patterns which can be optimized. For this experiment developer needs to profile all memory access made by his program and all calls to `do_page_fault` kernel function. To instrument memory accesses he uses well-known tool — `Dyninst`, to instrument `do_page_fault` `Systemtap` is used. In order to use `Dyninst` engineer has to write special instrumentation program in C++, compile it and link with `Dyninst` library. For using `Systemtap` developer needs to write instrumentation script using special language. After gathering necessary data by both tools one needs to take care about merging data sets after experiment. This is not enough convenient for developer because of to perform data analysis developer needs to create a analyzing script which needs to parse data and represent it as objects for further processing. Finally, all these actions can be available if and only if CE device has enough resources for instrumenting the considered application in standalone mode. Otherwise, one should take care about remote instrumentation using typical host-target model.

Thus, using Universal Instrumentation Engine could significantly simplify using multiple dynamic instrumentation tools (see Figure 2):

- Instead of working with several separate tools developer uses universal instrumentation API which helps to avoid writing handlers with different languages and perform several linking/compiling procedures;
- Universal Instrumentation API encapsulates unique instrumentation technique for each tool and provides a unified method of using each DBI engine;

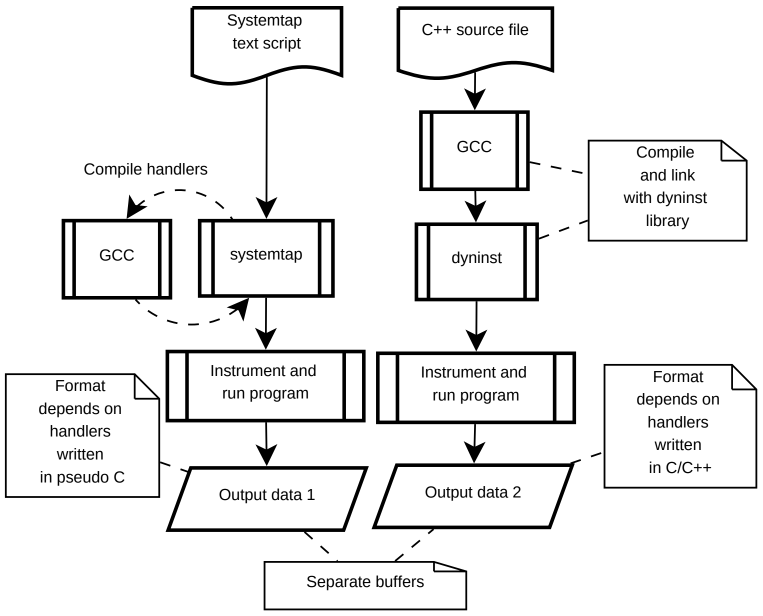

Systemtap text script
C++ source file
GCC
Compile and link with dyninst library
Compile handlers
GCC
systemtap
dyninst
Instrument and run program
Instrument and run program
Format depends on handlers written in pseudo C
Format depends on handlers written in C/C++
Output data 1
Output data 2
Separate buffers

Figure 1: Using Multiple Instrumentation Tools

- Universal engine provides requested dynamic instrumentation in standalone or remote mode using as less resources for multiple tools as possible;
- All handlers of multiple tools are written similarly in sense of output data format, so the collected trace of events will be, firstly, saved in unified data format and, secondly, provided in a merged state, thus avoiding developer from taking care of problems regarding parsing/merging different format data;

This approach eliminates disadvantages of process of using multiple instrumentation engines described above. Universal engine integrates several instrumentation methods in order to provide developer with ability to run his instrumentation scenarios using several methods through one universal interface. It also provides gathered data in unified format what simplifies post-mortem data analysis.

2 Framework Description

In this paper we describe unique dynamic binary instrumentation engine concept which is used in our monitoring tool — System-Wide Analyzer of Performance (SWAP). This tool was designed for profiling of user and kernel space software on embedded Linux systems. It provides extensible framework for system profiling and analysis of gathered data. It uses dynamic instrumentation technique for collecting data about system behavior. SWAP integrates several instrumentation tools and provides developers with ability to use multiple tools for profiling applications on CE devices. It has open modular design which allows other tools (e.g. front-ends, GUIs) to re-use its functionality. To fit the needs of various embedded platforms SWAP also supports host-target communication concept and standalone mode of operation when there is no connection to host. Finally, it provides a simple and convenient toolkit for post-mortem data analysis based on Python scripts which can be easily extended on the needs of user.

2.1 Architecture

In general profiling of applications consists of the following steps:

1. Developer specifies probing points and handlers which should be attached to those points;
2. Instrumentation engine resolves probing points in application binary and prepares instrumentation code to be inserted into application;
3. Instrumentation code is transferred to target in case of host-target architecture;

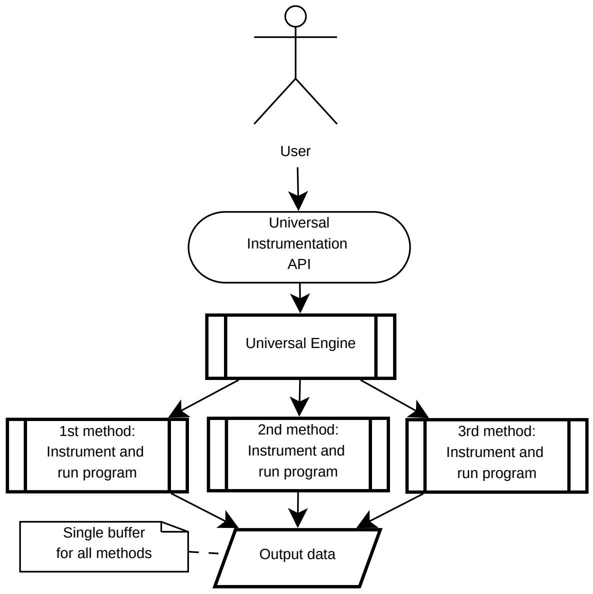

User
Universal Instrumentation API
Universal Engine
1st method: Instrument and run program
2nd method: Instrument and run program
3rd method: Instrument and run program
Single buffer for all methods
Output data

Figure 2: Universal Instrumentation Engine

4. Engine inserts instrumentation code into application and runs it if necessary;

5. Profiling data are collected by instrumentation handlers;

6. Engine removes instrumentation code;

7. Collected data is transferred from target to host in case of host-target architecture;

8. Collected data are stored in intermediate storage;

9. Developer analyzes collected data;

SWAP has open modular architecture (see Figure 3) which allows developers to re-use it for building their own tools. SWAP instrumentation framework consists of the following main components which implement above profiling scenario for embedded devices and provide the full set of capabilities for dynamic instrumentation and further data analysis:

- ***Instrumentation Engine.*** This is essential component of SWAP instrumentation framework which fully implements the functionality of described above Universal Instrumentation Engine. It provides API for instrumentation of user and kernel space functions using various instrumentation methods. It allows to instrument entry/exit and body of specified functions. Also it provides API to configure and control profile data collection. It consists of two parts: host and target. Host Instrumentation Engine performs step 2 from above scenario. It prepares data necessary for instrumentation, for example, resolves function addresses and prepares instrumentation code to be inserted. Target Instrumentation Engine performs steps 3 and 4 from above scenario. It does essential work: inserts instrumentation into application/kernel and collects data. Currently engine integrates two instrumentation methods: `Kprobe`-based and `Dyninst`-based. For kernel instrumentation `Kprobe`-based method is used. For user space `Kprobe` or `Dyninst` methods are used depending on what program points should be instrumented (e.g. functions or memory access instructions).

- ***Communication Agent.*** This component provides API for connecting to target and transferring commands and responses to target and obtaining collected data from target. It consist of two parts: host and target. The component encapsulates SWAP target communication protocol. This component is used by instrumentation engines to control instrumentation process and data transfer. In standalone mode this component is not used.

- ***Collected Data Storage.*** This is storage for collected data. It stores collected data and provides effective methods for retrieval of collected information when data are analyzed. SWAP supports several modes of data collection: normal mode when all data are collected on target and after experiment they are transferred to host and continuous mode when data are transferred to host continuously as they appear. These two modes allows to user to choose optimal data collection policy for his experiment. Normal mode has less overhead for target, but it is limited by size of buffer on target for collected data. So in normal mode amount of profiled data are limited by size of buffer on target. Continuous transfer mode has no such limitation. Buffer is used as intermediate storage before data are transferred to host. But this mode has greater overhead because of data transmission.

- ***Data Analyzing Framework.*** As a final step of resolving considered problem, SWAP provides to developers analyzing framework which allows easy data manipulating and analyzing. This component provides framework with API for loading, parsing and analyzing data kept in Collected Data Storage. Developers can easily extend this framework and reuse its components in their own analyzing scripts written in SWAP Python-based language.

The described components cover all basic functionality needed for monitoring with help of dynamic instrumentation: ability to instrument kernel space and user space, ability to use standalone mode or host-target model, ability to save collected data in unified format and, finally, perform data analysis for understanding considered behavior of the CE device. Let's consider general SWAP profiling scenario in details (see Figure 3):

- Upon user actions SWAP GUI configures instrumentation and starts it via instrumentation API provided by engine. Instrumentation configuration includes functions to be instrumented, size of buffer for collected data etc;

- Host instrumentation engine prepares information necessary for profiling, connects to target via host communication agent and sends data to target;

- Instrumentation agent on target inserts instrumentation code into application and starts it if necessary;

- Data collection started;

- When user stops data collection (via SWAP GUI) collected data are transferred back to host via communication agents and saved in data storage;

- Then when user wants to visualize results of profiling SWAP GUI uses analyzing scripts based on Data Analyzing Framework to calculate and represent various statistics;

- SWAP visualizes results;

2.2 Instrumentation API

As it was mentioned above SWAP instrumentation engine provides to developers universal instrumentation API. This API provides common way for profiling programs using different instrumentation methods. As it was said before currently there are two methods are supported by SWAP: `Kprobe`-based and `Dyninst`-based. This API provides general set of functions which does not depend on underlying instrumentation engine. API allows to:

- Add/remove probe for specified kernel function. Kernel function probes trace entry and exit from functions;

- Add/remove probe for specified user space application's function. User space function probes trace entry and exit from functions;

- Add/remove probe inside functions. Function body probes trace execution of instruction at specified address inside functions;

- Add/remove memory access probe inside functions. Memory access probes trace execution of memory access instructions inside functions;

- Configure target buffer size;

- Set filters by process IDs and names;

- Define conditions for data collection start and end;

- Start and stop profiling process;

- Retrieve list of libraries used by application;

- Retrieve list of application/library functions;

- Retrieve list of kernel functions;

- Configure data transfer mode for host-target model (one-time or continuous);

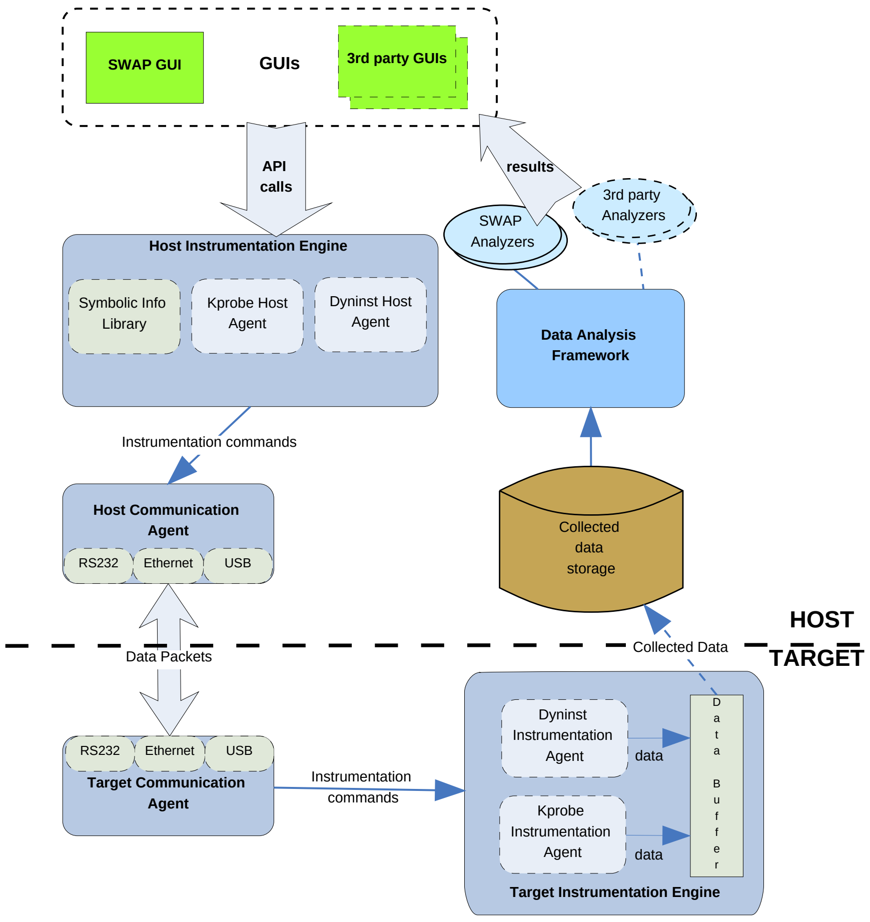


Figure 3: SWAP Architecture

3 Profiling With SWAP

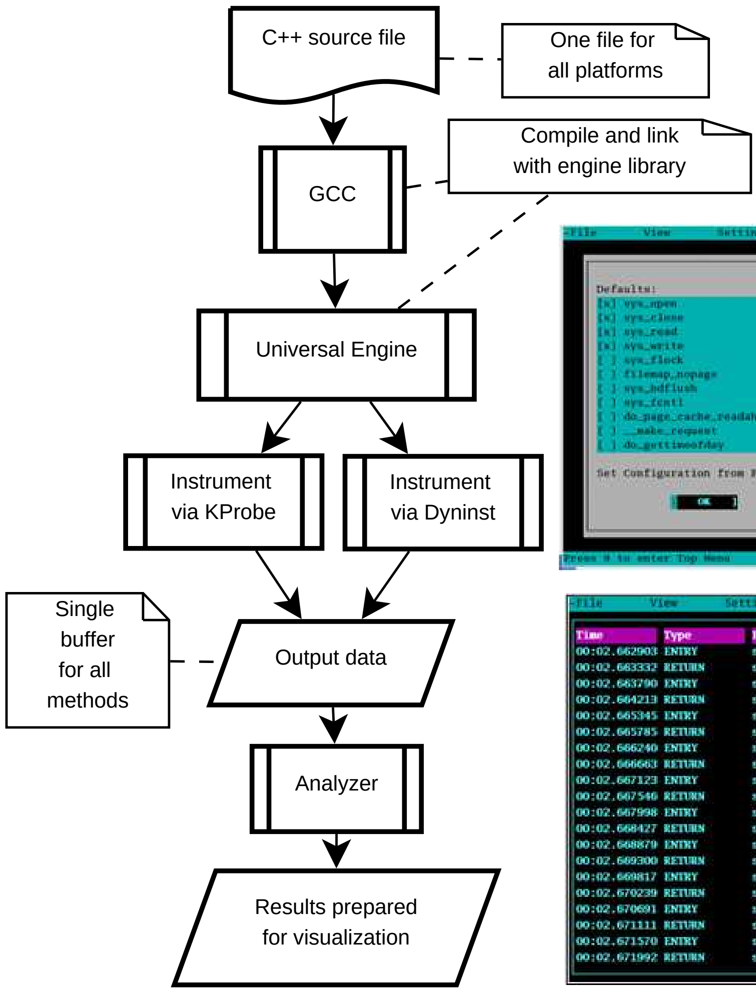
There are several ways in which developers can use SWAP instrumentation framework for profiling:

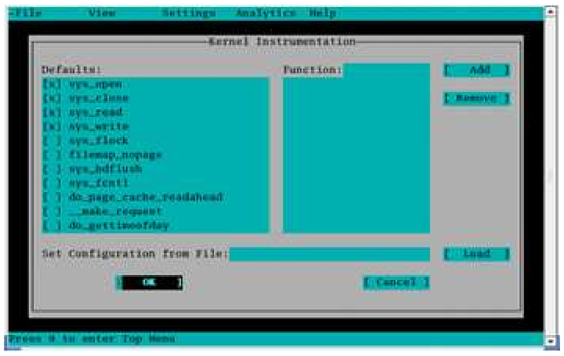
- They can use it as library in order to build their own tools on top of the engine;
- Developers can use SWAP GUI front-end itself for instrumentation, control and visualization of results;

As it was described earlier SWAP instrumentation engine is universal instrumentation engine which integrates several instrumentation tools (see Figure 4). SWAP GUI follows general approach of building tools on top of SWAP instrumentation engine. It uses it as shared library. Also it extends SWAP data analyzing framework with a set of own scripts which analyze collected data and produce data for visualization of various statistics. Let's see how SWAP solves problems raised by usage of multiple tools. We will consider example described in introduction where memory accesses and `do_page_fault` kernel function should be instrumented for the same application behavior (developer wants to analyze how many page faults are generated by memory accesses made by considered program and if it is possible to find access patterns which can be optimized).

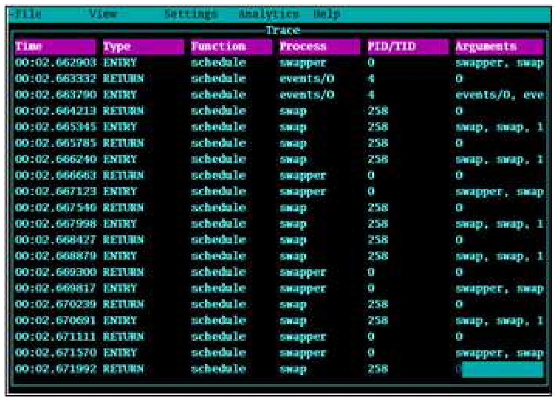
Steps which should be performed to instrument program using multiple tools `Systemtap` and `Dyninst`:

1. Write `Systemtap` script for kernel functions instrumentation using a built-in language;
2. Compile written handlers for kernel functions;
3. Instrument kernel functions and run considered program;
4. Collect and save essential data;


C++ source file
One file for all platforms
GCC
Compile and link with engine library
Universal Engine
Instrument via KProbe
Instrument via Dyninst
Single buffer for all methods
Output data
Analyzer
Results prepared for visualization



```
Kernel Instrumentation
Defaults:                          Function:
[x] sys_open
[x] sys_clone
[x] sys_read
[x] sys_write
[ ] sys_flock
[ ] filemap_nopage
[ ] sys_bdflush
[ ] sys_fcntl
[ ] do_page_cache_readahead
[ ] __make_request
[ ] do_gettimeofday
Set Configuration from File:
Press # to enter Top Menu
```



```
                                Trace
Time            Type     Function   Process    PID/TID   Arguments
00:02.662903 ENTRY    schedule   swapper    0         swapper, swap
00:02.663332 RETURN   schedule   events/0   4         0
00:02.663790 ENTRY    schedule   events/0   4         events/0, eve
00:02.664213 RETURN   schedule   swap       258       0
00:02.665345 ENTRY    schedule   swap       258       swap, swap, 1
00:02.665785 RETURN   schedule   swap       258       0
00:02.666240 ENTRY    schedule   swap       258       swap, swap, 1
00:02.666663 RETURN   schedule   swapper    0         0
00:02.667123 ENTRY    schedule   swapper    0         swapper, swap
00:02.667546 RETURN   schedule   swap       258       0
00:02.667998 ENTRY    schedule   swap       258       swap, swap, 1
00:02.668427 RETURN   schedule   swap       258       0
00:02.668879 ENTRY    schedule   swap       258       swap, swap, 1
00:02.669300 RETURN   schedule   swapper    0         0
00:02.669817 ENTRY    schedule   swapper    0         swapper, swap
00:02.670239 RETURN   schedule   swap       258       0
00:02.670691 ENTRY    schedule   swap       258       swap, swap, 1
00:02.671111 RETURN   schedule   swapper    0         0
00:02.671570 ENTRY    schedule   swapper    0         swapper, swap
00:02.671992 RETURN   schedule   swap       258       0
```

Figure 4: SWAP Profiling Scheme

5. Write C++ program which uses `Dyninst` for instrumentation of memory accesses;

6. Compile and link considered program with `Dyninst` library;

7. Run instrumented program second time;

8. Collect and save essential data;

9. Merge data collected by both tools;

10. Write program/script for data analysis, including data parser, data analysis itself and formatted output information;

In comparison with previous procedure, developer should perform much less steps to instrument program using SWAP:

1. Write C++ program which uses SWAP engine for instrumentation of memory accesses and kernel functions using two or more DBI engines simultaneously;

2. Compile and link considered program with engine library;

3. Run program only one time;

4. Collect and save essential data using standalone mode or host-target mode;

5. Write program/script for data analysis itself, avoiding time consuming development of data parsing and formatting of output information;

As it can be easily seen, in this example, SWAP instrumentation procedure:

- is more simple, universal and resource saving due to unified approach for instrumentation needs less compilations/linking of instrumentation handlers;
- produces synchronized data which do not need merging due to simultaneous use of two or more DBI engines;
- provides more correct conditions of experiment due to one-time launch of considered program;
- allows perform considered experiment under host-target mode or standalone mode, thus extending abilities under strict hardware conditions (lack of memory for saving data or unavailable network connection);
- avoids developer of implementing data parsing during data analysis, thus giving ability to concentrate on analysis itself;

In more complex instrumentation scenarios developer may spend significant time for writing `Systemtap` script and `Dyninst`-based tool.

4 Conclusion

There are a number of instrumentation tools which have their advantages and disadvantages. There is no ideal tool which will satisfy all developer's needs. So developers have to duplicate and maintain their instrumentation scenarios in various forms which are supported by every tool.

In this paper we presented our tool SWAP which is built on top of universal profiling framework which can be extended by users analyzing modules. SWAP framework makes use of universal instrumentation engine concept in order to integrate functionality of several instrumentation tools and provide developers with universal and effective way of profiling programs on various embedded platforms using different instrumentation methods and analyzing collected profiling data.

References

[1] William E. Cohen, Gaining insight into the Linux kernel with Kprobes, RedHat Magazine, Issue#5, March 2005

[2] Frank Ch. Eigler. Problem solving with systemtap. In Proceedings of the Ottawa Linux Symposium 2006, 2006.

[3] Giridhar Ravipati, Andrew R. Bernat, Nate Rosenblum, Barton P. Miller and Jeffrey K. Hollingsworth, Toward the Deconstruction of Dyninst, Technical Report, Computer Sciences Department, University of Wisconsin, Madison, 2007.

[4] Mathieu Desnoyers, and Michel R. Dagenais, The LTTng tracer: A Low Impact Performance and Behavior Monitor for GNU/Linux, Linux Symposium Proceedings, Volume 1, Ottawa, Ontario, 2006.

[5] Systemtap, Dtrace, LTTng Comparison. `http://sourceware.org/systemtap/wiki/SystemtapDtraceComparison`

[6] Richard McDougall, Jim Mauro, and Brendan Gregg, Solaris Performance and Tools: Dtrace and Mdb Techniques for Solaris 10 and Opensolaris, Prentice Hall, 2006.

Prediction of Optimal Readahead Parameter in Linux by Using Monitoring Tool

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

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

Jaehoon Jeong
SAIT, Samsung Electronics
hoony_jeong@samsung.com

Mikhail Levin
SRC Moscow, Samsung Electronics
m.levin@samsung.com

Abstract

Frequently, application developers face to the hidden performance problems that are provided by operating system internal behavior. For overriding such problems, Linux operation system has a lot of parameters that can be tuned by user-defined values for accelerating the system performance. However, in common case evaluating of the best system parameters requires a very time consuming investigation in Linux operating system area that usually is impossible because of strong time limitations for development process.

This paper describes a method that allows any application developer to find the optimal value for the Linux OS parameter: the optimal maximal read-ahead window size. This parameter can be tuned by optimal value in easy way that allows improving application performance in short time without getting any knowledge about Linux kernel internals and spending a lot of time for experimental search for the best system parameters.

Our method provides the prediction of optimal maximal read-ahead window size for Linux OS by using the monitoring tool for Linux kernel. Scenario of our method using is very simple that allows obtaining the optimal value for maximal read-ahead window size for the single application run. Our experiments for Linux 2.6 show that our method detects an optimal read-ahead window size for various real embedded applications with adequate accuracy and optimization effect can be about a few and even a few dozen percents in comparison to default case. The maximal observed optimization effect for accelerating the embedded application start-up time was 59% in comparison to default case.

Taking into account these facts the method proposed in this paper has a very good facilities to be widely and simply used for embedded applications optimization to increase their quality and effectiveness.

1 Introduction

In modern computing systems high performance disk drive systems very often have a serious problem related with its performance, effectiveness and speed. Usually the disk input-output (I/O) performance is related with time which needs to mechanical parts of the disk to move to a location of the data storing. This time usually defines the time delays in the data extraction. Operating systems incorporate disk read-ahead cache to minimize the time delays. Typically, the data from the disk are buffered in a memory with a relatively fast access time. If requested data is already reside in the cache memory, the data can be transferred directly from the cache memory to the requester. In result the performance is increased because the access to data from the cache memory is substantially faster than the access from the disk drive [1].

Very often such cache can be sufficiently effective. But sometimes it can produce the low system performance. This relates with the sensitivity of the read-ahead cache to cache hit statistics [2]. A read-ahead cache having a low hit rate may perform more poorly than an uncached disk due to caching overhead and queuing delays, among others. This problem is especially important to be solved for embedded systems, because they usually have less memory and slow CPUs than servers and workstations.

Performance of read-ahead subsystem can be controlled by its main parameter - maximal read-ahead window size: the maximum amount of data which can be read-ahead from block device. The performance of each application that is running under management OS depends on the performance of read-ahead cache. Thus, using optimal maximal read-ahead window size value for the current application can improve performance of this application significantly. Typically, the finding the optimal maximal read ahead window size for target application consists of the following steps: (1) tune operating system by current maximal read-ahead window; (2) reboot the system; (3) run the target application; (4) measure the time of execution of target application. These steps should be repeated several times for various maximal read-ahead window sizes until the optimal value of read-ahead window size will be found. The main disadvantage of such method is too much time that should be spent for finding the optimal value of maximal read-ahead window size. Thus, finding the optimal value of maximal read-ahead window size during single application run could significantly improve the performance of target application and save a lot of machine and human resources.

2 Method Overview

The suggested technique consists of the following steps: (1) user should collect data from the application and OS during application execution by monitoring tool for Linux kernel one time; (2) monitoring tool analyzes collected data and evaluates optimal maximal read-ahead window size automatically; (3) user can access evaluated optimal read-ahead window size value via monitoring tool user interface. Thus, in comparison to manual method of finding optimal read-ahead window size and other existing methods (see [11], [12], [13] for details), the suggested method has the following advantages: (1) determines optimal maximal read-ahead window size value during single run of application that is to be optimized; (2) uses existing read-ahead subsystem without any changes.

This method is designed to be used as a module of a monitoring tool such as SWAP - System-Wide Analyzer of Performance developed by Samsung Research Center (SRC) in Moscow. The initial version of SWAP was developed in 2006 and uses functional interfaces of Kprobe for providing dynamic instrumentation of Linux kernel for ARM and MIPS architecture. Next revision of SWAP tool was developed in SRC in 2007. This revision allowed collecting traces from predefined functions in Linux kernel that contains general information of system characterization. The current SWAP revision can monitor both kernel and application levels of the Linux system. It provides evaluation of a set of important system characteristics for main Linux subsystems (such as Memory Management, Process Management, File System and Network). Additional, current revision of SWAP has some automatic performance analysis features such as trace comparison, automatic bottleneck region localization, etc.

We have integrated our method into SWAP framework because SWAP satisfies to all necessary requirements of our method.

2.1 General Idea

We suggest that optimal solution can be obtained by minimization of the following two characteristics: the number of requests to the block device and the amount of pages that were forced to read by target application, but were not used.

The big value of maximal read-ahead window size provides minimization of the first characteristic – number of requests to the block device. However, in common situation the big value of maximal read-ahead window size provides reading a lot of pages from block device that are not used by application. In the other words the big value of maximal read-ahead window size makes OS performs a lot of unnecessary job that is very time consuming. It is obvious that application performance degrades because of such OS behavior.

The small value of maximal read-ahead window size provides minimization of the second characteristic – the amount of pages that were forced to read by target application, but were not used. However, in this case optimization effect of read-ahead cache is decreased because of a lot of application requests for memory pages invoke the requests to the block device instead of requests to the read-ahead cache.

Thus, to reach the optimal OS behavior for current application we need to obtain such maximal read-ahead window size value that provides minimization of both characteristics simultaneously.

2.2 Collected Data

The first step of our method requires collecting the important information about application behavior during execution for future analyzing and evaluating the result. For these purposes, it is necessary to monitor the accesses to the memory that has been performed by application. SWAP has ability for tracking memory accesses and organizing them in trace that allows estimating the following aspects: 1) the total number of accessed memory pages, loaded from block device; 2) the addresses of accessed memory pages and their source (loaded from block device or not); 3) the order of accesses.

Notice, that our method requires the single application run only for gathering the monitoring information for evaluation of the optimal values of maximal read-ahead window size.

2.3 Analysis of Collected Data

According to the basic idea, two following values should be minimized simultaneously:

- F - number of requests to the block device;
- G - number of pages that were forced to read by target application, but were not used.

Both values - F and G - can be evaluated according to the information about memory pages that have been accessed by application.

Our method performs the emulation of read-ahead procedure for obtaining the required parameters for resolving the minimization problem. Linux 2.6 read-ahead cache has a complex behavior: after each request to the page from block device, some additional following pages are loaded into read-ahead cache. Number of additional pages is less or equal to the maximal read-ahead window size value. During emulation of read-ahead cache behavior we simplify the real OS behavior: we consider that the number of additional pages that are loaded into read-ahead cache is equal to the size of read-ahead window (see Figure 1).

This emulation procedure allows determine those pages can be potentially loaded into read-ahead cache for any predefined maximal read-ahead window size value. According to this information it is possible to evaluate the number of requests to the block device F. Thus, the number of requests to the block devices F can be described as a function:

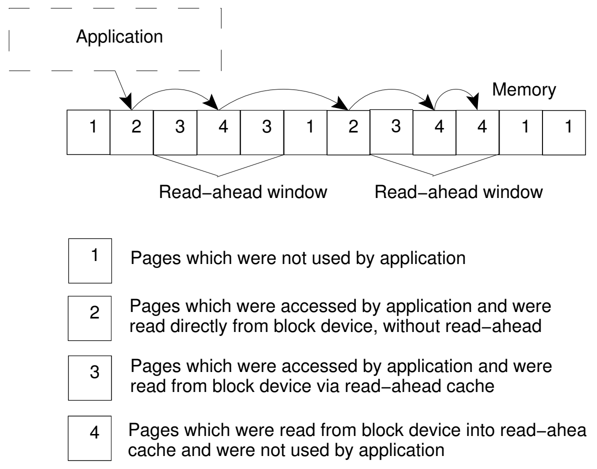

Application
Memory
1
2
3
4
3
1
2
3
4
4
1
1
Read–ahead window
Read–ahead window
1
Pages which were not used by application
2
Pages which were accessed by application and were read directly from block device, without read–ahead
3
Pages which were accessed by application and were read from block device via read–ahead cache
4
Pages which were read from block device into read–ahea cache and were not used by application

Figure 1: Read-ahead emulation procedure

$$F = Rrn(Ma, Rws) \quad (1)$$

where Rrn is the number of requests to the block device, Ma is number of memory accesses that requires mapping pages from block device and Rws is the maximal read-ahead window size.

The value of G that describes the number of pages that were forced to read by target application, but were not used, can be evaluated by taking into account the following peculiarities of Linux 2.6 read-ahead cache behavior:

- each read-ahead request produces a read operation for page from block device, which will not be placed in read-ahead cache;
- each read-ahead request additionally produces Rws read operations for pages from block device

Thus, the total amount of pages that were read from block devices within the performed read-ahead emulation can be expressed as a following function g:

$$g = Rrn * Rws + Rrn \quad (2)$$

Since the collected information on memory accesses provides information about total amount of pages requested by application, it is suitable to define the value of G (see Fig. 1) as follows:

$$G = Rrn * Rws + Rrn - Tap(Ma) \quad (3)$$

Here Tap is a total amount of pages from block device requested by application. Thus, we define G as a difference between the number of pages that have been really read from the block device as a result of the read-ahead cache behavior and the number of requested pages.

Notice that both functions F and G depend on two arguments, namely on the collected information on memory accesses (Ma) and on the read-ahead window size (Rws). The first argument Ma can be considered as a constant since the problem of performance optimization is solved for some fixed behavior of target application. Hence, both characteristics are the functions of a single argument - maximal read-ahead window size (Rws). It is possible to obtain different emulated values for both characteristics by varying Rws value.

According to the basic idea of the method the minimization problem for functions F and G should be solved. Both functions F and G reach their minimum value in extreme points satisfying to the following necessary conditions:

$$\begin{cases} \frac{\partial F}{\partial Rws} = 0 \\ \\ \frac{\partial G}{\partial Rws} = 0 \end{cases} \quad (4)$$

Since functions F and G depend on one variable Rws, we have a redefine system of two equations. To solve this incorrect problem [3], we change searching of exact solution of the redefine system of two equations by searching of its quasi-solution satisfying to solution of the following minimization problem:

$$min_{Rws}\{F^2(Rws) + G^2(Rws)\} \quad (5)$$

This minimization problem can be rewritten also as follows:

$$min_{Rws}\{Rrn^2 + [Rrn * (Rws + 1) - Tap]^2\} \quad (6)$$

Here we call an expression in figure brackets as a $coefficient of optimality$:

$$C_{opt} = \{Rrn^2 + [Rrn * (Rws + 1) - Tap]^2\} \quad (7)$$

It is easily to see that there are two limiting values for Rws which gives trivial solution exact for G:

1. $Rws = 0$, in that case each page, accessed by application is read from block device and none of the pages are read into read-ahead cache. In this case Rrn is equal to Tap. Thus, G is always equal to 0.

And for F:

2. $Rws =$ *total amount of memory in computing system*. In this case, after first access to the page with minimal address, all pages will be read into read-ahead cache.

Notice, when function F has the minimal value, then function G has the maximal value and vice versa. Both mentioned limited values of Rws allow restricting the set of possible values of Rws.

The minimization problem can be solved by the exhaustive search. However, during test period it is was noted that minimized expression is a function with one turning point - the point of minimum. We use the iterative method for speeding up the search: we emulate the read-ahead cache behavior for different values of maximal read-ahead window size Rws and choose such Rws value that makes a coefficient of optimality (7) becomes minimal.

3 Experimental Data

Using SWAP monitoring tool with our optimization method we collected several sample traces for embedded boards with running applications considered below. The appropriate results obtained in our experiments are also presented and discussed. For measuring inaccuracy of our method we use the following technique: the optimization percentage of optimal maximal read-ahead window size (found by hands) and quasi-optimal maximal read-ahead window size (proposed by our method) are measured (with comparison to default maximal read-ahead window size); the difference between optimal percentage and quasi-optimal percentage gives us the "amount" of missed optimizations. For example, the start-up of application for default $Rws = 255$ equals to 34.9 seconds; for quasi-optimal $Rws = 2$ equals to 14,5 seconds; for optimal $Rws = 1$ equals to 14 seconds. The optimization percentage of quasi-optimal Rws in comparison with default Rws is 58.45%. The optimization percentage of optimal Rws in comparison with default Rws is 59.88%. The difference between optimal and quasi-optimal solutions is $(59.88\% - 58.45\%) = 1.43\%$.

Environment	Details
Board	DTV_x260, MIPS-based
Kernel version	2.6.10
Application	Digital TV management software (exeDSP)
Initial read-ahead window size	255 pages
Performance measurement tool	SWAP
Base for memory access event information	`do_page_fault()` SWAP event

Table 1: Characteristics of DTV_x260 board and exeDSP application

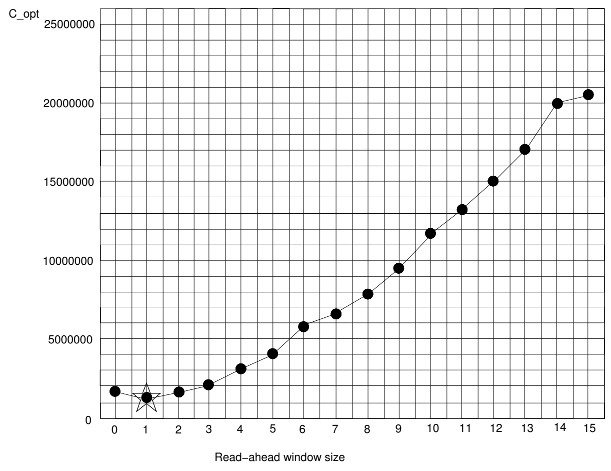


Figure 2: C_{opt} for application exeDSP. Quasi-optimal solution (point of minimum of C_{opt}) $Rws = 1$ is marked with star

3.1 Digital TV Management Application

The experiment consisted of optimizing the start-up procedure of application textitexeDSP (digital TV management software) which supervises digital TV board (DTV x260): the time of full initialization should be decreased as much as possible.

The diagnosis of our method (implemented in SWAP) was the following: quasi-optimal size of maximal read-ahead window *Rws* is equal to is 1 (see Figure 2).

The performance was measured manually for the following values of read-ahead window size enumerated in Table 2.

We can see that minimum is reached for $Rws = 1$ page

Read-ahead Window Size (Rws), pages	Time of startup, seconds
0	14.2
1	14
2	14.5
3	16
4	18.4
5	23
15	26.5
255 (default)	34.9

Table 2: Runs of exeDSP application on DTV_x260 board

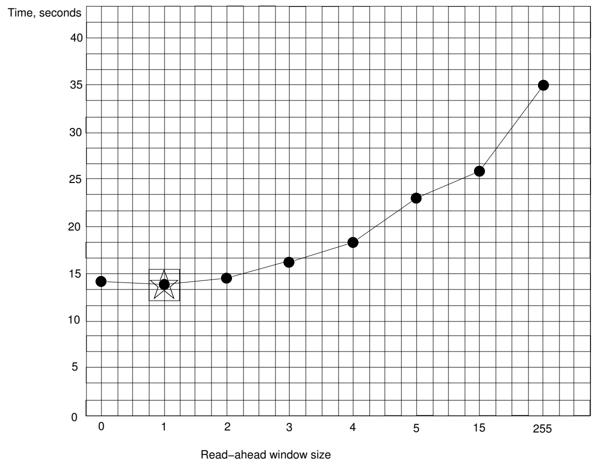


Figure 3: Start-up time for application exeDSP. Quasi-optimal solution is marked with star. Optimal solution $Rws = 1$ is marked with square

(see Figure 3). Optimal read-ahead window size is $Rws = 1$.

The difference of start-up time between default read-ahead (255 pages) and optimal read-ahead suggested by our method (1 page) is about 20.9 seconds or about 59% from default start-up time. The difference between optimal and quasi-optimal solutions is 0%, because quasi-optimal and optimal solutions are coincided.

3.2 Web Browser

The experiment consisted of optimizing the start-up procedure of FireFox (web browser) on multimedia board MVL2443: the time of full initialization should be decreased as much as possible.

Board	MVL2443, ARM based
Kernel version	2.6.16
Application	Firefox
Initial read-ahead window size	7

Table 3: Characteristics of MV2443 board and Firefox application

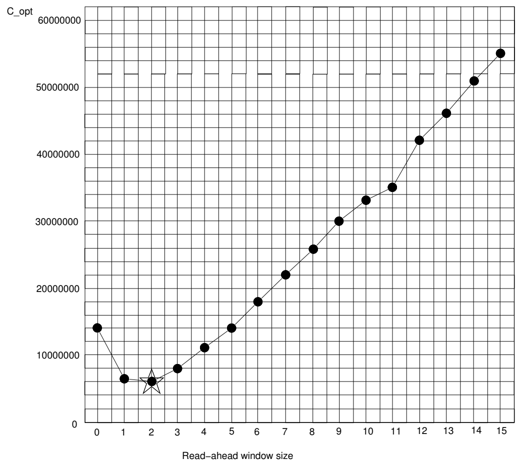


Figure 4: C_{opt} for application FireFox. Quasi-optimal solution (point of minimum of C_{opt}) $Rws = 2$ is marked with star

The diagnosis of our method (implemented in SWAP) was the following: quasi-optimal size of read-ahead window Rws is equal to 2 (see Figure 4).

The performance was measured manually for the following values of read-ahead window size enumerated in Table 4.

We can see that minimum is reached $Rws = 3$ pages (see Figure 5). Optimal read-ahead window size is $Rws = 3$. The difference of start-up time between default read-ahead (15 pages) and quasi-optimal read-ahead suggested by our method (2 pages) is about 5.8 seconds or about 7.34% from default start-up time. The difference of start-up time between default read-ahead (15 pages) and optimal read-ahead found by hands (3 pages) is about 6.3 seconds or about 7.97% from default start-up time. The difference between optimal and quasi-optimal solutions is $(7.97\% - 7.34\%) = 0.65\%$.

3.3 MP3 player

The experiment consisted of optimizing the file loading procedure of MadPlay MP3 player on the board

Read-ahead Window Size (Rws), pages	Time of startup, seconds
0	75
1	73.1
2	73.2
3	72.7
4	73.5
5	74.5
7	79
15 (default)	79
31	81

Table 4: Runs of Firefox application on MV2443 board

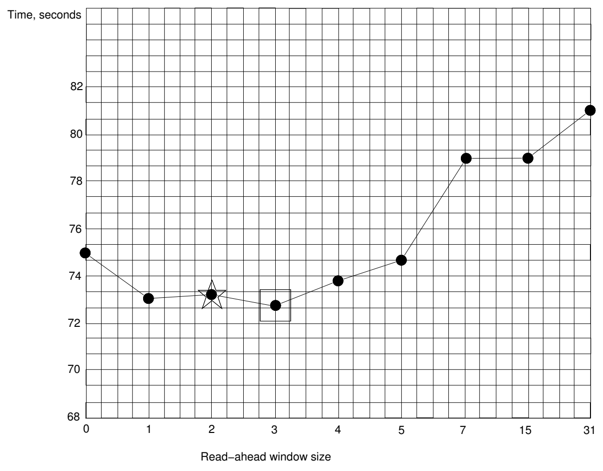


Figure 5: Start-up time for application FireFox. Quasi-optimal solution is marked with star. Optimal solution $Rws = 3$ is marked with square

OMAP5912OSK: the time of playback with null output device should be decreased as much as possible.

The diagnosis of our method (implemented in SWAP) was the following: quasi-optimal size of read-ahead window Rws is equal to 2 (see Figure 6).

The performance was measured manually for the following values of read-ahead window size enumerated in Table 6.

We can see that minimum is reached for value $Rws = 3$ (see Figure 7). Optimal read-ahead window size is $Rws = 3$. The difference of start-up time between default read-ahead (15 pages) and quasi-optimal read-ahead suggested by our method (2 pages) is 0.5 seconds or about 4.70% from default start-up time. The difference of start-up time between default read-ahead (15

Board	OMAP2912OSK ARM based
Kernel version	2.6.10
Application	MP3 Player (Madplay)
Initial read-ahead window size	15

Table 5: Characteristics of OMAP5912 OSK board and MadPLay application

Read-ahead Window Size (Rws), pages	Time of startup, seconds
0	10.66
2	10.13
3	10.09
4	10.17
5	10.17
6	10.23
7	10.31
8	10.31
15 (default)	10.63
19	10.73
31	10.95

Table 6: Runs of MadPLay application on OMAP5912 OSK board

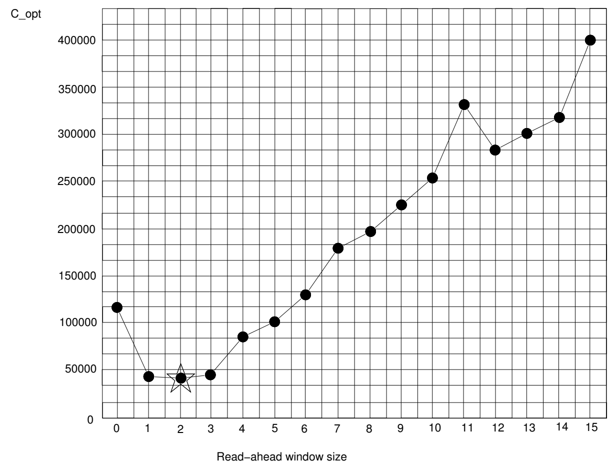


Figure 6: C_{opt} for application MadPlay. Quasi-optimal solution (point of minimum of C_{opt}) $Rws = 2$ is marked with star

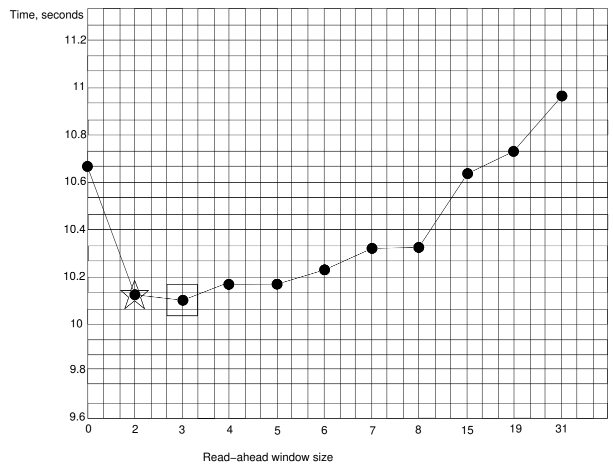


Figure 7: Time of input file loading for application MadPlay. Quasi-optimal solution is marked with star. Optimal solution $Rws = 3$ is marked with square

pages) and optimal read-ahead found by hands (3 pages) is 0.54 seconds or about 5.07% from default start-up time. The difference between optimal and quasi-optimal solutions is $(5.07\% - 4.70\%) = 0.37\%$.

4 Conclusion

We have presented a new method for automatic evaluation of optimal value for single Linux tunable parameter: maximal read-ahead window size. Tuning of this Linux parameter by obtained value can improve performance of embedded application significantly. Our method is simple to use and requires single run of application for evaluation of all necessary characteristics. It does not require any additional knowledge in system or optimization areas and can be used by any application developer who wants to increase performance of his application by reducing hidden performance bottlenecks in Linux operating system. Due to its advantageous characteristics, this method can be widely used for optimization of embedded systems to increase their quality and effectiveness.

References

[1] Daniel P. Bovet, Marco Cesati, Understanding the Linux Kernel, Third Edition, O'Reilly Media, 2006.

[2] Linux kernel source files (`http://kernel.org`).

[3] V. A. Morozov, Regular Solution Methods of Non-Correct problems, Nauka, Moscow, 1987.

[4] James R. Larus and Eric Schnarr, Computer Sciences Department, University of Wisconsin, Madison, EEL: Machine-Independent Executable Editing. (`ftp://ftp.cs.wisc.edu/wwt/pldi95_eel.ps`).

[5] Amitabh Srivastava, Alan Eustace, ATOM: A System for Building Customized Program Analysis Tools.

[6] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, and Brian Bershad, University of Washington, Brad Chen, Harvard University, Instrumentation and Optimization of Win32/Intel Executables Using Etch.(`http://etch.cs.washington.edu/etch-usenixnt/etch-usenixnt.html`).

[7] Ananth Mavinakayanahalli, Prasanna Panchamukhi, Jim Keniston, Anil Keshavamurthy, Masami Hiramatsu, Probing the guts of Kprobes, Ottawa Linux Symposium, pp.101-114, July 2006.

[8] Frank Ch. Eigler, RedHat, Problem Solving With Systemtap. (`http://sourceware.org/systemtap/wiki/OLS2006Talks?action=AttachFile&do=get&target=problem-solving-with-stap.pdf`).

[9] David J. Pearce, Paul H.J. Kelly, Tony Field, Uli Harder, Imperial College of Science, Technology and Medicine, London, GILK: A dynamic instrumentation tool for the Linux Kernel.

[10] Giridhar Ravipati, Andrew R. Bernat, Nate Rosenblum, Barton P. Miller and Jeffrey K. Hollingsworth, Toward the Deconstruction of Dyninst, Technical Report, Computer Sciences Department, University of Wisconsin, Madison (`ftp://ftp.cs.wisc.edu/paradyn/papers/Ravipati07/SystemtabAPI.pdf`).

[11] US5809560, Adaptive read-ahead disk cache.

[12] US2005154825A1, Adaptive file read-ahead based on multiple factors.

[13] US2003115410A1, Method and apparatus for improving file system response time.

Unprivileged login daemons in Linux

Serge Hallyn
IBM
serge@hallyn.com

Jonathan T. Beard
GeekNet / Sourceforge
jbeard@geek.net

Abstract

Login daemons require the ability to switch to the userid of any user who may legitimately log in. Linux provides neither a fine-grained setuid privilege which can be targeted at a particular userid, nor the ability for one privileged task to grant another task the setuid privilege. A login service must therefore always run with the ability to switch to any userid.

Plan 9 is a distributed operating system designed at Bell Labs to be a next generation improvement over Unix. While it is most famous for its central design principle - everything is a file - it is also known for simpler userid handling. It provides the ability to pass a setuid capability - a token which may be used by a task owned by one userid to switch to a particular new userid only once - through the /dev/caphash and /dev/capuse files. Ashwin Ganti has previously implemented these files in Linux. His p9auth device driver was available for a time as a staging driver. We have modified the concepts explored in his initial driver to better match Linux userid and groups semantics. We provide sample code for a p9auth server and a fully unprivileged login daemon. We also present a biased view of the pros and cons of the p9auth filesystem.

1 Introduction

For the past fifteen years, the Computer Security Institute has surveyed security practitioners and government and private institutions regarding security breaches and cybercrime [14]. In 2008, they surveyed 522 respondents with an average annual loss report just under $300,000. From 2004 through 2008, the number of respondents reporting unauthorized access (including privilege escalation) as a component of their reported attacks remained approximately a third. Similarly, in the mobile computing and console gaming arenas, jailbreaks through privilege escalation remain one of the leading security concerns for these platforms. As Linux forms the core operating system for a growing number of these devices [5], a solution that greatly reduces or eliminates opportunities for privilege escalation would represent a potential savings worth tens or hundreds of millions – at least from a comparison of potential losses due privilege escalation as described above.

One avenue to privilege escalation is the exploitation of flaws in privileged programs. For instance, a buffer overflow in the ping program might allow a regular unprivileged user to pass specific data as an argument to ping, invoking a shellcode resulting in a root shell. Therefore, it is a laudable goal to reduce the number of programs which need to run privileged.

We begin by reviewing the current design of login servers in Linux. We present two ways in which Linux could be extended to support making these login servers unprivileged. For the first, we provide sample code for both the privileged and unprivileged servers. We discuss the pros and cons of both approaches, in the hopes of raising a discussion on the merits of supporting one of the designs upstream.

2 Login services in Linux

Throughout this paper, familiarity with the user [11] and privilege [7] mechanisms of Linux is assumed. Tasks are owned by a numerical user ID and one or more numerical group IDs. These numerical IDs are known in userspace by textual names. Tasks also carry three sets of privileges which for historical reasons are called "POSIX capabilities" or just "capabilities". We usually say a task has some privilege if that privilege is in the task's "effective" set, in which case the task is allowed to escape certain access checks or perform sensitive tasks. For instance, the `CAP_DAC_OVERRIDE` privilege allows a task to read other users' private files and `CAP_SYS_BOOT` allows a task to reboot the system. Usually, user IDs and privileges are linked, and user ID

0 is special in that its privilege sets are full, while privilege sets for other user IDs are by default empty.

Tasks in Linux can switch user IDs in two ways. One is to execute a file with the "setuid bit" set. This bit indicates that when the file is executed, the "effective" user ID should be set to that of the file owner. The "real" user ID remains unchanged. The other way is to use the `setuid` family of system calls. This can be used by unprivileged tasks to switch values between the effective and real (and another, called "saved") user IDs. To set a user ID to an entirely new value requires the `CAP_SETUID` privilege. The changing of primary group IDs is analogous. Changing the set of auxiliary group IDs (for which there is no analogy with user IDs) always requires the `CAP_SETGID` privilege.

A simple login service in Linux can be structured as in Figure 1. The login daemon, running with process ID (PID) 300 waits for a username to be entered on its terminal, `/dev/tty`. Generally it then asks for a password on the same terminal. It can verify the validity of the password by checking hashed entries in `/etc/passwd` or `/etc/shadow`. If the password is valid, then the login task finds the user's default auxiliary groups, primary group ID, and user ID, switches to these credentials, and executes the user's default shell. The user now has a shell running with his credentials on `/dev/tty`.

Figure 1 also shows communication between login and PAM [15]. PAM is a set of libraries which offer authentication and session management services and ease system-wide configuration through a single set of configuration files. While these are great advantages over the alternative of each service implementing their own authentication and session management, library functions execute with the credentials of their caller. Therefore PAM does not help with reducing the amount of privilege required for login daemons to perform their authentication and to switch user IDs.

The principle of privilege separation [13] advises that a privileged program be structured so that privileged operations are pushed out into small, easier to verify, privileged helpers, each serving precisely one purpose. Then the bulk of the program can run without privilege. Figure 2 shows how the privilege-separate OpenSSH server manages to prevent the user from directly interacting, at any time, with a privileged process. A privileged parent process communicates with an unprivileged child running as an unused user ID (usually "ssh"). The child passes authentication communications to the parent and, if they are correct, then the parent ends the first unprivileged child and forks a new child which calls setresgid() and setresuid() to drop privilege and assume the authenticated user's identity.

While this can make successful privilege escalation attacks far less likely by reducing the types of messages which an attacker can cause to be sent to the privileged task, it does not completely eliminate attacks. While privilege is kept further away from the unprivileged user, the number of programs running with privilege are not reduced. The implementation of each login service is also greatly complicated.

3 Credentials passing

Unix domain sockets in Linux support the ancillary messages `SCM_RIGHTS` and `SCM_CREDENTIALS`. The first allows passing an open file descriptor to a target task. The second allows passing the sending task's credentials (user ID, group ID and PID) so the receiving task can verify the sender's identity. Alan Cox has proposed a new ancillary message type, which we call `SCM_AUTH`, which any task may use to convey the right for the recipient to switch to its own credentials. Such a feature could provide interesting new features, such as one unprivileged user granting another unprivileged user the temporary credentials needed to debug an environment problem for the first. Even a system service which is partially privileged (but without `CAP_SETUID`) or unprivileged could make use of this feature to act on behalf of its clients.

This proposal also elicited discussion about a related, less dangerous ancillary message for passing only audit credentials. These could be used to augment audit messages with the credentials of a client on whose behalf a back-end server was acting. This would make the audit messages more informative than if they carry only the server's credentials.

3.1 Unprivileged login based upon SCM_AUTH

The `SCM_AUTH` ancillary message type would facilitate the implementation of unprivileged login clients in Linux. A simple architecture for such a system is shown in Figure 3. A single privileged daemon can serve all unprivileged login servers. The login servers

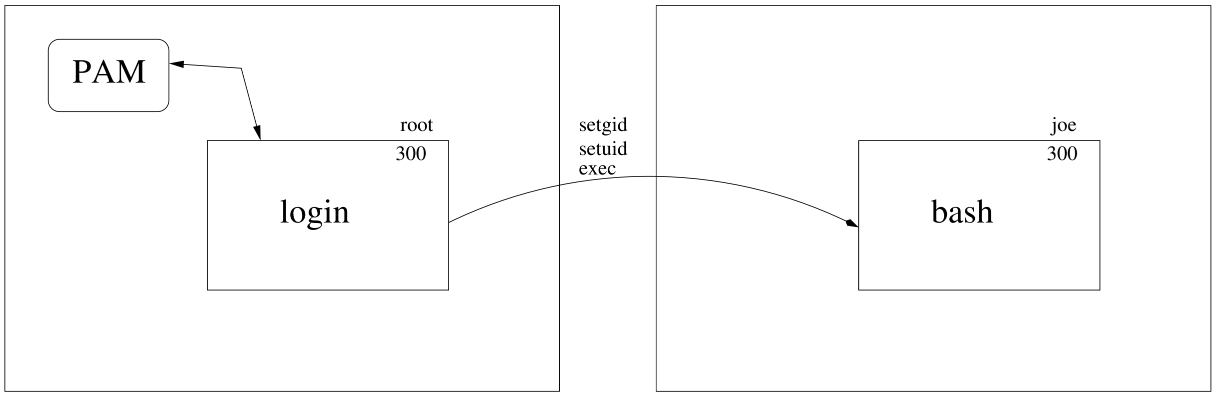


Figure 1: Simple login design using setuid.

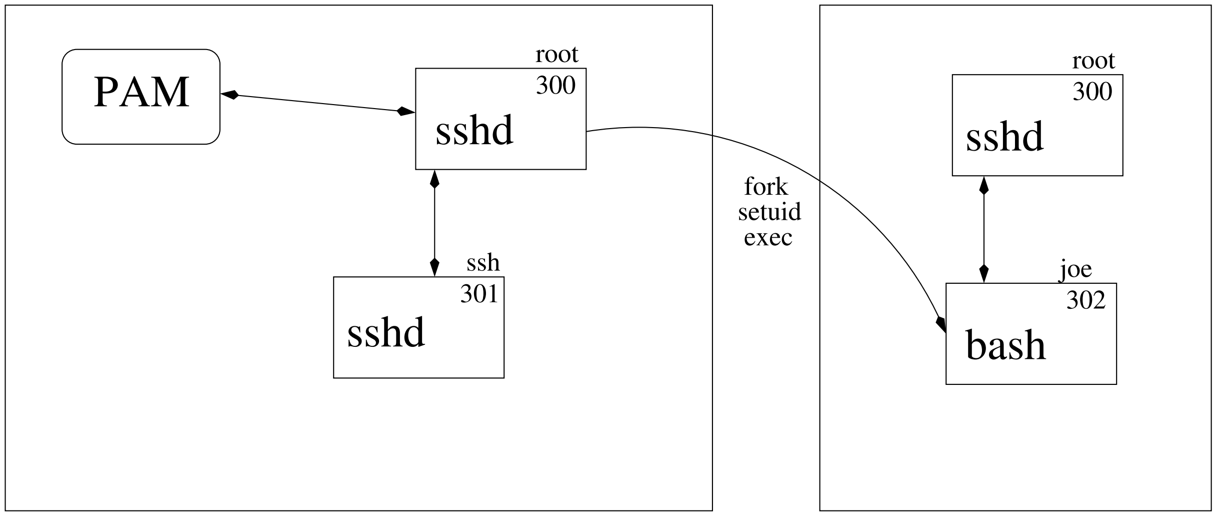


Figure 2: Login using privilege-separated OpenSSH.

act as a proxy to facilitate the communication allowing a user to authenticate his identity to the privileged server. Once satisfied of the user's identity, the privileged server forks a new task which establishes for itself the user's desired credentials using the standard `setresgid`, `setgroups` and `setresuid` system calls. It then passes a `SCM_AUTH` message to the login daemon, which uses a new `accept_id` system call to assume these received credentials. The login process can now proceed calling the user's login shell as it normally would after having explicitly called `setuid` if it had been running with privilege.

3.2 Security Considerations

When POSIX capabilities were first introduced, file capabilities remained unimplemented. In a POSIX capability system, file execution causes a task's permitted capability set to be recalculated. Any capabilities which end up in that set must be active in one of the executable file's capability sets. Therefore, without file capabilities, no task can have permitted capabilities after executing a file. As a way to achieve non-root partially privileged tasks, `capsetp` was implemented as a Linux-specific way to grant capabilities to another task [9]. Doing so required the `CAP_SETPCAP` privilege. However, the ability to grant privileges to another task was deemed so unsafe that the `CAP_SETPCAP` capability was placed in the system-wide capability bounding set, so that effectively no one was ever able to employ it.

Likewise, passing credentials over a UNIX socket could be seen as too dangerous a way to spread privileges and access rights. It particularly spreads the risk of any pro-

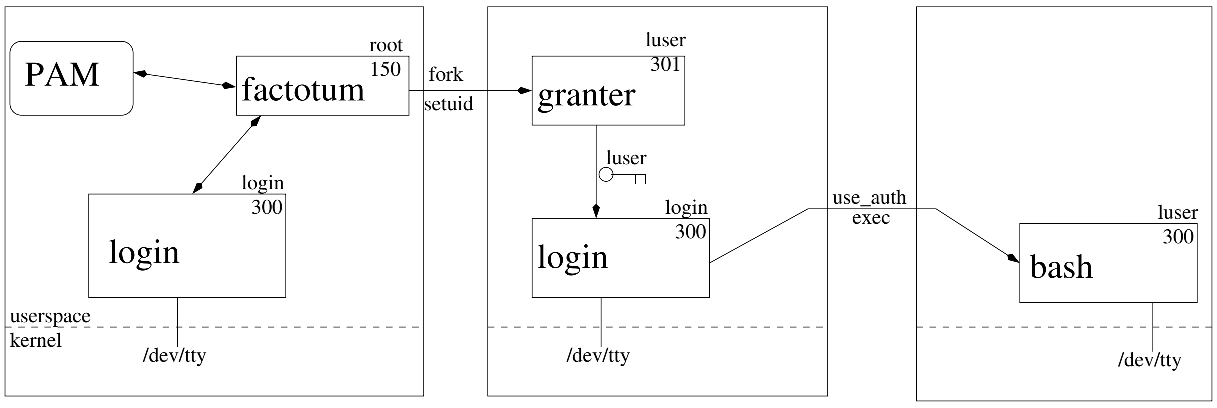


Figure 3: Login design using SCM_AUTH.

gram running with `CAP_SETUID`, since any task which may arbitrarily set its user ID can then pass the resulting credentials along to any other task. It also makes a Trojan even more dangerous. The Trojan now does not need to immediately do its damage, open a back door, release the information it targets, or hide a task which it creates with the victim's credentials. Instead, it can simply transfer the conquered credentials to the attacker who can stash them away for later. This could be undetectable.

One part of a defense against such a situation is notion of a timeout on the passed credentials. This could help prevent the tokens being too liberally passed around, accidentally inherited by an untrusted child task, or simply stashed away as a long term privilege token for later use. Analysis of used and stashed tokens could also help mitigate the dangers. Both the sending and the use of tokens should obviously be audited. Additionally, a list of sent but unused tokens and details on the sending and receiving tasks should be available, possibly through a `/proc` interface.

4 The p9auth filesystem

Ashwin Ganti implemented [6] a device driver for Linux which implemented the Plan 9 "capability [1] device" [2]. Briefly, an unprivileged login client persuades a privileged daemon that it is authorized to change to a particular user ID. The privileged daemon writes to the `/dev/caphash` device a token consisting of the client's current user ID and the authorized new user ID joined together with a special separating character ('@') and hashed with a random string. This token is then appended by the kernel to a list of such hashes. The daemon communicates the random string to the client, which can then use the token by writing the unhashed values (old user ID, new user ID, and random value, separated by '@') to `/dev/capuse`. This causes the client's real and effective user IDs to be changed. Since this was a proof of concept, some shortcuts were taken. For instance, the saved user ID is not updated [2], the primary and auxiliary groups remain unchanged, and the filesystem user ID (fsuid), which is used for filesystem accesses and by convention mirrors all effective user ID changes, is not updated.

We have changed the Linux p9auth semantics to better suit the Linux user and group behavior and simplify usage. The p9auth setuid capability now contains the original user ID, new user ID, a new primary group, and a list of auxiliary groups. Upon presenting a valid capability, a task's real, saved, effective, and filesystem user and group IDs are all assigned to the new values. While different effective and real user IDs can be useful for applications to get things done, a newly logged-in process needs a simple, sane, and useful initial set of credentials. We have also changed from a device interface to a more standard one using a custom filesystem. We have also implemented a timeout on setuid capabilities, placed a limit on the number of unused capabilities stored in the kernel, and made the implementation user namespace aware as will be discusssed later.

Figure 4 shows how login services can be structured

[1] Capability here is used in its classical sense, not as the unfortunately chosen pseudonym for a privilege.

[2] This may be intended as a feature to increase flexibility at the cost of more complicated users.

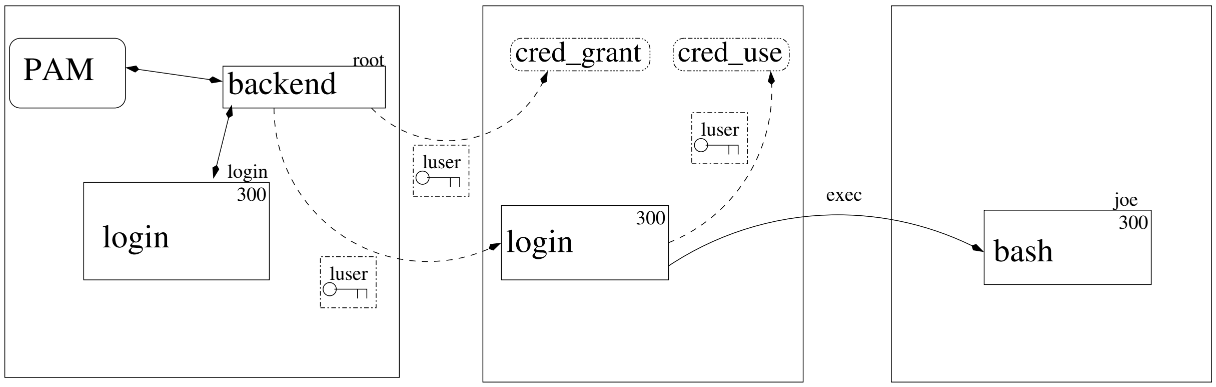


Figure 4: Login design using p9auth setuid tokens.

with the new p9auth filesystem. A single privileged backend service, called p9auth, can serve all login clients, and need never interact directly with users. An unprivileged login client, perhaps running with user ID "login", interacts with the user and passes the communication for an authentication protocol between the terminal and the privileged p9auth service. This service itself uses PAM for the actual authentication. Assuming authentication succeeds, p9auth looks up the initial credentials for the new user and creates a random string (`XYZ`) and a p9auth capability token which looks something like `121@1001@1001@0` In this example user ID 121 may switch to user ID 1001, primary group 1001, and no auxiliary groups. P9auth hashes the capability token with the random string and writes that to the `cred_grant` file in the p9auth filesystem. It also passes the token and the random string to the user. The user then passes `121@1001@1001@0@XYZ` to the `cred_use` file to effect the change of credentials.

From a higher level, we see that, as with `SCM_AUTH` ancillary messages, an unprivileged login client was made possible by providing a way for the ability to switch user IDs to be sent between tasks. In this case, one isolated privileged task can send to unprivileged console- or network-facing tasks tokens representing the ability to switch to a particular user ID.

4.1 A proof of concept p9auth service

Our goal is for a single p9auth service to serve all unprivileged login servers. For simplicity, it will communicate with them over a Unix file socket. Given that secrets like passwords will be sent over this socket, the communication protocol should begin with the p9auth service proving its own identity to the unprivileged login daemon. Details of such an algorithm are outside the scope of this paper, but could be based on a certificate or public key pair for the p9auth daemon.

A proof of concept (POC) implementation of a p9auth server and an unprivileged login client can be found at github [?] [3]. It is based upon the p9auth filesystem patchset which as of this writing was out of tree, and can be found at [?]. Since the userspace code is POC and does not implement server authentication, the client, called `frontend` and shown as pseudo code in Figure 5, simply connects to the Unix socket `/var/run/factotum` and assumes it is talking to the valid backend server. It writes its current user ID and desired new username, then serves as a PAM relay until it receives the desired setuid token in a messages beginning with "FINAL: ". It then writes that token to the `cred_use` file in the `p9auth` filesystem. That write triggers a callback in the kernel which effects the group and user ID changes. Finally, it executes the user's specified shell to complete the login.

The privileged server, called `backend` and shown in Figure 6, listens on the Unix socket for requests from login clients. It uses the `sock_conv()` conversation function (based on an example by Andrew Morgan [?]) to allow PAM, called with privilege by `backend` and guided by the system's PAM configuration file `/etc/pam.d/factotum`, to perform the actual authentication.

Assuming the PAM authentication succeeds, `backend`

[3]Note that this is purely a proof of concept, and not intended to be used as-is. For instance, the client does not currently clear the environment before executing the login shell.

```
void client(int sfd)
{
    sprintf(buf, "olduid %d\nusername %s\n", getuid(), username);
    write(sfd, buf, strlen(buf)+1);
    while (1) {
        read(sfd, buf, MAXLINE);
        if (strncmp(buf, "FINAL: ", 7) == 0) {
            break;
        ...
                } else if (strncmp(buf, "ECHO: ", 6) == 0 ||
                        strncmp(buf, "NOECHO: ", 8) == 0) {
                        /* get_input reads a user's response */
                        get_input(buf, buf[0] == ,E,);
                        write(sfd, buf, strlen(buf)+1);
                }
        }
        cfd = open("/mnt/p9auth/cred_use", O_WRONLY);
        p = buf + 7;
        write(cfd, p, strlen(p));

    shell = getpwnam(username)->pw_shell;
    execl(shell, shell, NULL);
}
```

Figure 5: Pseudo code for unprivileged p9auth login client.

generates a random string and a login token as described in Section 4, writing the encrypted hash of the login token with the random string to the `cred_grant` file and passing the concatenated token and random string to the client.

The p9auth filesystem is user-namespace-aware. Any setuid tokens granted by a particular privileged p9auth server are tied to that server's user namespace. Therefore, a p9auth server in a container cannot be used to bypass the host's p9auth server. At the same time, since each container will have its own `/var/run` directory and therefore its own `/var/run/factotum` socket, any properly configured container will be able to offer its own privileged p9auth server for use within the container.

4.2 Security Considerations

As with credentials passing, we must consider whether the ability to grant the ability to switch to new credentials with the p9auth concept should raise the same concerns as did transferring POSIX capabilities using `capsetp`. Just as with granting capabilities to third parties, we essentially have a privileged task which is allowed to grant to other tasks the privilege to switch to new user IDs. This is especially true since on most systems, by default, switching to the root user ID also raises all capabilities.

However, this concern is predicated on the risk of spreading privileges because p9auth can authorize a change of the user ID for an existing process. In fact, the opposite case is true and is described in the example below. First, assume that some unprivileged process, PID 3451, improperly manages to get p9auth to authorize its setuid to 0. This is no worse than if the same process tricks a privileged login service into forking a new process with user ID 0, running a binary specified by the malicious process. Additionally, since a single privileged p9auth service allows us to remove the `CAP_SETUID` capability from all privileged login services, like su, sudo, login, sshd, ftp, etc. By running those services completely unprivileged, this approach actually greatly decreases the amount of potentially vulnerable privileged code.

```
void handle_client(int clientfd)
{
    read(clientfd, buf, MAXLINE);
    sscanf(buf, "olduid %d\nusername %s\n", &olduid, username);
    /*
     * validate() performs PAM authentication and returns
     * the password entry (struct passwd *pe) for username.
     */
    struct passwd *pe = validate(clientfd, username);

    /*
     * readgroups() places all username's auxiliary groups
     * into gid_t *groups
     */
    numgroups = readgroups(username, pw->pw_gid, &groups);

    /*
     * make_token places the string
     *     olduid@newuid@newgrp@numgroups@grp1@...@grpn
     * into char *token
     */
    make_token(token, olduid, pe, numgroups, groups);

    /* generate a hash of the token and hand that to the kernel */
    len = generate_hash(token, hash, randstr);
    capfd = open("/mnt/p9auth/cred_grant", O_RDWR);
    write(capfd, hash, len);

    /* write the token and the random string to the client */
    sprintf(clientstr, "FINAL: %s@%s", token, randstr);
    write(clientfd, clientstr, strlen(clientstr)+1);
}
```

Figure 6: Pseudo code for p9auth backend.

Another advantage of the approach is that in a completely converted system we may have the init daemon fork off the p9auth service early and then drop `CAP_SETUID` from its capability bounding set, so that all other privileged daemons will not be able to expose the system to `CAP_SETUID` empowered rootkit exploits.

5 Other privilege needs

While the above designs, and the POC implementation of the p9auth-based unprivileged login daemon suffice for simple cases, some sites require additional setup at login. That setup itself may require privilege. For instance, in order to provide polyinstantiated directories, login session to an LSPP [3] system may require a private mounts namespace and custom directories to be mounted according to their privilege level. User joe logged in as "secret" sees a different `/tmp` than the same user logged in as "unclassified". Other sites may require a Smack [16] label to be specified at login, which requires the `CAP_MAC_ADMIN` privilege. Still other sites may wish to set up an initial inheritable capability set [7], which requires the login daemon to have `CAP_SETPCAP`.

These obstacles ought not be insurmountable. For instance, it it is hoped that both creating private namespaces and directory mounting will eventually be allowed for unprivileged users. The specification of Smack labels and initial capabilities could be added as extensions to the p9auth token specification, while the SCM_AUTH ancillary message could well represent full credentials, including security labels. Finally, while these designs remove the need for CAP_SETUID and CAP_SETGID from login daemons, other file capabilities can be added to the login daemon binaries if needed. Upon the execution of the login shell, the effective and permitted capabilities will be recalculated, so while it would be preferable to have the login daemon entirely unprivileged, some capabilities could be enabled if needed.

More generally, a reliable way must be found to set appropriate initial LSM (security) contexts upon login. Proper behavior will differ per LSM. TOMOYO contexts are meant to purely reflect the history of executed files, so a TOMOYO context should likely not be passable with SCM_AUTH or p9auth. SELinux contexts are the results of domain transitions initiated by execution of carefully designated "entry points". The login daemons can remain unprivileged, but their execution can trigger entry into a domain which can select an initial security context for the user's session. For POSIX capabilities, it would seem desirable to avoid the need to grant CAP_SETPCAP to an unprivileged login daemon by allowing the privileged backend daemon to specify an inheritable capability set. Passing permitted and effective capabilities may be found to have uses for non-login applications of SCM_AUTH, but is not useful for login daemons. We therefore may well want only the inheritable set to be specified. Smack contexts are generally set by userspace, but require the CAP_MAC_ADMIN capability to set, and therefore should not be trivially changeable, although allowing tasks with CAP_MAC_ADMIN to transfer their Smack label along with their credentials may be desirable.

Since LSMs may vary in how they treat the security context passed through SCM_AUTH or granted through p9auth, it seems prudent to provide a new LSM [**?**] hook which processes a set of initial and intended credentials, and produces the final LSM context for the login session.

6 Conclusion

Exploitation of bugs in privileged programs can allow an unprivileged user to illegitimately escalate privilege, and can serve as the second step in an outsider attack which begins with hijacking of a local unprivileged account. To reduce the potential for such attacks, it is desirable to reduce the amount of code which must run with privilege. We have presented two ways in which Linux can be extended to support unprivileged login daemons.

Both the SCM_AUTH and p9auth designs remove the need for login services to run with CAP_SETUID and CAP_SETGID privileges, concentrating privilege instead in a single system-wide authentication service. Without this support, each service needing to switch user IDs needs to be installed with the privilege to switch to any user ID. In that case, it requires constant vigilance and multiple redundancy to limit the privilege granted to the immediate user-facing terminal program – with every single instance being a potential oversight leading to exploit. With this support, the number of programs requiring privilege is reduced, and with it the gross odds of a successful privilege escalation attack.

7 Legal Statement

8 Acknowledgments

The authors would like to thank Ashwin Ganti for writing the original p9auth capability device, Greg Kroah-Hartman for the drivers staging tree and for hosting the p9auth driver, and Alan Cox for once again providing an interesting new insight into a proposed design.

References

[1] Alan Cox. *Regarding add p9auth driver,* `http://lkml.org/lkml/2010/4/21/88`, Apr 21 2010.

[2] Russ Cox, Eric Grosse, Rob Pike, Dave PResotto, Sean Quinlan, "Security in Plan 9", Proceedings of the 2002 Usenix Security Symposium, San Francisco.

[3] Janak Desai, George Wilson and Chad Sellers. *Extending SELinux to meet LSPP data import/export requirements,* Proceedings of the 2006 Security Enhanced Linux Symposium, March 2006.

[4] Chris Friedhoff. *POSIX Capabilities and File POSIX Capabilities,* `http://www.friedhoff.org/posixfilecaps.html`.

[5] C. Gallen. *Linux to Be the Fastest-Growing Smartphone OS over the Next 5 Years* ABI Research. August 2007. `http://www.abiresearch.com/press/922`.

[6] Ashwin Ganti. *Plan 9 authentication in Linux*, Operating Systems Review 42(5): 27-33 (2008).

[7] Serge E. Hallyn and Andrew G. Morgan. *Linux Capabilities: making them work,* Proceedings of the Ottawa Linux Symposium, July 2008.

[8] Linux man-pages project. *Capabilities.7 man page,* `http://linux.die.net/man/7/capabilities`.

[9] Andrew Morgan. *capsetp(3) manpage,* `http://linux.die.net/man/3/capsetp`.

[10] C. Ozancin. *Securing the linux environment: Programs that need root access.* pages 42âĂŞ45, March/April 2001.

[11] Dan Tsafrir and Dilma Da Silva and David Wagner. *The Murky Issue of Changing Process Identity: Revising "Setuid Demysitified.",* Linux Journal.

[12] J. Saltzer and M. Schroeder. *The protection of information in computer systems.* Fourth ACM Symposium on Operating System Principles (October 1973).

[13] N. Provos, M. Friedl, and P. Honeyman. *Preventing Privilege Escalation* Security '03 Paper, USENIX Security '03 Technical Program, 2003.

[14] R. Richardson. *2008 CSI Computer Crime & Security Survey*. Computer Security Institute. 2008.

[15] Vipin Samar, "unified Login with Pluggable Authentication Modules (PAM)," Proceedings of the Third ACM Conference on Computer Communications and Security, March 1996, New Delhi, India.

[16] Casey Schaufler. *Smack and the Application Ecosystem,* `http://linuxplumbersconf.org/2009/slides/Casey-SmackPlumbers2010.pdf`.

[17] Ylonen, T., "SSH-Secure Login Connections Over the Internet", 6th USENIX Security Symposium, pp 37-42. San Jose, CA, July 1996.

Twin-Linux: Running independent Linux Kernels simultaneously on separate cores of a multicore system

Adhiraj Joshi
LinSysSoft Technologies
adhiraj@linsyssoft.com

Swapnil Pimpale
LinSysSoft Technologies
swapnilp@linsyssoft.com

Mandar Naik
LinSysSoft Technologies
mandarn@linsyssoft.com

Swapnil Rathi
LinSysSoft Technologies
swapnilr@linsyssoft.com

Kiran Pawar
LinSysSoft Technologies
kiranp@linsyssoft.com

Abstract

There are three classes of common consumer and enterprise computing - Server, Interactive and Real-Time. These are characterized respectively by the need to obtain highest throughput, sustained responsiveness, and hard real-time guarantees. These are contradictory requirements hence it's not possible to implement an operating system to achieve all these goals. Most operating systems are designed towards serving only one of these classes and try to do justice to the other two classes to a reasonable extent.

We demonstrate a technique to overcome this limitation when a single hardware box is required to fulfill multiple of these computing classes. We propose to run different copies of kernels simultaneously on different cores of a multi-core system and provide synchronization between the kernels using IPIs (Inter Processor Interrupts) and common memory. Our solution enables users to run multiple operating systems each one the best for its class of computing. For ex., using our idea we can configure a quad core system with 2 cores dedicated for server class computing (database processing), 1 core for UI applications and remaining 1 core for real-time applications.

This idea has been used in the past, primarily on non-x86 processors and custom designed hardware. Our proposal opens the doors of this idea to the off-the shelf hardware resources. We present Twin-Linux, an implementation of this scenario for 2 processing units using Intel-Core-2-Duo system. This idea finds applications in - Filers,Intelligent Switches, Graphics Processing Engines, where different types of functions are performed in a pipelined manner.

1 Introduction

Consider an application that requires huge data computations and responsiveness simultaneously. In this case, an operating system with server kind of computing will only be able to handle data processing part and it lacks in responsiveness. while realtime system will be able to handle responsiveness but throughput will be very low. Hence there is a need to provide different operating system environment within the same hardware box.

We have implemented this concept on a Intel Core 2 Duo architecture. In current scenario (On a Intel Core 2 Duo system), there are two cores (processors) on a single chip, a SMP Kernel and a single copy of RAM. In normal case, a single copy of kernel boots up on core 2 duo machines in SMP (Symmetric Multi Processing) mode. At the boot time one of the cores boots the Linux Kernel and other keeps spinning till the Kernel boots up. The booting core is termed as the BSP(Bootstrap Processor) while the other cores are termed as the APs(Application Processor). After the booting process scheduler assigns the task to both cores in parallel so that there could be simultaneous execution of different threads on respective cores.

By replicating kernel code and by making performance boundaries explicit, Twin-Linux removes the kernel as a bottleneck to scaling up performance in large multiprocessor systems. Each kernel loaded on the cores is composed of a scheduler, a memory manager and code to coordinate communication between other kernels.

The rest of the paper is organized as follows: Section 2 provides background of SMP systems. Details of processor classification, ACPI tables and IPI messages are covered in section 2. Section 3 covers the motivation behind the project and the innovation of Twin-Linux. Section 4 covers the Twin-Linux design. It covers the implementation details and the algorithms used to locate the MADT and IPI communication. It provides an insight into the issues involved in loading the kernel on the respective cores. A concise description of the applications and marketability of the project is included in Section 5. Finally section 6 summarizes the conclusions of the project and the last section lists the References.

2 Features of SMP Systems

2.1 Processor Classification

The Intel Specification classifies Processors into two types: the bootstrap processor (BSP) and the application processors (AP). The BSP is chosen by the hardware or by the BIOS in conjunction with the hardware. The BSP is responsible for initializing the system and for booting the operating system.

The BSP executes the BIOS's boot-strap code to configure the APIC environment, sets up system-wide data structures, and starts and initializes the APs. When the BSP and APs are initialized, the BSP then begins executing the operating-system initialization code. Following a power-up or reset, the APs complete a minimal self-configuration, then wait for a startup signal (a SIPI message) from the BSP processor. Upon receiving a SIPI message, an AP executes the BIOS AP configuration code, which ends with the AP being placed in halt state. APs are activated only after the operating system is up and running. Once the MP operating system is up and running, the BSP functions as an AP.

2.2 ACPI Tables

For a multiprocessor system, the BIOS performs the following functions :

- Pass configuration information to the operating system that identifies all processors and other multiprocessing components of the system.
- Initialize all processors and the rest of the multiprocessing components to a known state.

The BIOS fills the ACPI system description tables and hands over the control to the Boot loader. In a multiprocessor system, multiple local and I/O APIC units operate together as a single entity, communicating with one another over the ICC bus. The APIC units are collectively responsible for delivering interrupts from interrupt sources to interrupt destinations throughout the multiprocessor system.

The local APIC units also provide interprocessor interrupts (IPIs), which allow any processor to interrupt any other processor or set of processors. There are several types of IPIs. Among them, the INIT IPI and the STARTUP IPI are specifically designed for system startup and shutdown. Each local APIC has a Local Unit ID Register and each I/O APIC has an I/O Unit ID Register. The ID serves as a physical name for each APIC unit. It is used by software to specify destination information for I/O interrupts and interprocessor interrupts, and is also used internally for accessing the ICC bus.

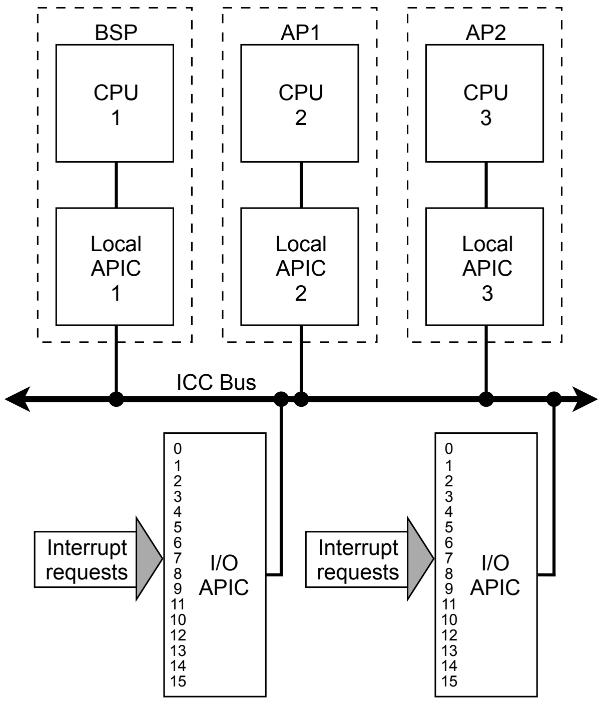


Figure 1: SMP System with APIC Configuration

2.3 Using INIT IPIs

The primary local APIC facility for issuing IPIs is the interrupt command register (ICR). INIT IPI is an Interprocessor Interrupt with trigger mode set to level and delivery mode set to "101" (bits 8 to 10 of the ICR).

The ICR consists of the following fields :

- Vector: The vector number of the Interrupt being sent.
- Delivery Mode: Specifies the type of IPI to be sent. This field is also know as the IPI message type field.

  101 – INIT IPI message to all the local APICs in the system to set their arbitration IDs to the values of their APIC Ids, the level flag must be set to 0 and trigger mode flag to 1.

  110 – Start Up IPI Sends a special "start-up" IPI (called a SIPI) to the target processor or processors. The vector typically points to a start-up routine that is part of the BIOS boot-strap code
- Destination Mode:
- Delivery Status (Read Only)
- Level
- Trigger Mode
- Destination Shorthand
- Destination

An INIT IPI is an IPI that has its delivery mode set to RESET. Upon receiving an INIT IPI, a local APIC causes an INIT at its processor. The processor resets its state, except that caches, floating point unit, and write buffers are not cleared. Then the processor starts executing from a fixed location, which is the reset vector location. To cause the processor to jump to a different location, the INIT IPI must be used as part of a warm-reset. By putting an appropriate pointer in the warm-reset vector, setting the shutdown code to 0Ah, then causing an INIT, the BIOS (or the operating system) can cause the current processor to jump immediately to any location.

2.4 Using Startup-IPIs

STARTUP IPIs are used with systems based on Intel processors with local APIC versions of 1.x or higher. These local APICs recognize the STARTUP IPI, which is an APIC Interprocessor Interrupt with trigger mode set to edge and delivery mode set to "110" (bits 8 through 10 of the ICR).

The STARTUP IPI causes the target processor to start executing in Real Mode from address 000VV000h, where VV is an 8-bit vector that is part of the IPI message. Startup vectors are limited to a 4-kilobyte page boundary in the first megabyte of the address space.

For an operating system to use a STARTUP IPI to wake up an AP, the address of the AP initialization routine (or of a branch to that routine) must be in the form of 000VV000h. Sending a STARTUP IPI with VV as its vector causes the AP to jump immediately to and begin executing the operating system's AP initialization routine.

3 Twin-Linux Applied

Consider a network services application. We describe the performance benefits that can be realized from the utilization of multiple processing cores for the application.

Functional pipelining is a technique that sub-divides application software into multiple sequential stages and assigns these stages to dedicated execution cores. Pipelining can increase locality of reference since each execution core runs a subset of the entire application, potentially increasing the cache hit rate.

In case of a 2-core system, we can have one core dealing with the network I/O requests that will process them partially and the other core will handle all the disk I/Os and remaining processing. Thus the first kernel will handle the networking part where it accepts requests from other systems, processes them partially for the other core and then the other kernel takes care of the storage part and processes those requests completely (as shown in the Figure 2). Distributing network processing to multiple cores can be achieved using multiple network interfaces (NICs) on a system, or affinitizing interrupts of NICs to different cores. This approach can be generalized to apply to other types of network connections between devices such as IPSec flows or IP header compression contexts. Since there is typically little or no locality of reference between different traffic flows like these, reducing the number of different traffic flows processed by each core can improve cache efficiency. Each execution core services a subset of the flows which confines memory accesses to a smaller set of memory addresses and potentially increases cache efficiency.

3.1 Innovation

Multi-core Processors have been adopted in various computer systems from commodity machines to embedded systems. To utilize these multi-core machines more efficiently, a promising way is to run multiple applications on one machine or share resources among different users. In such a situation, instead of using a single operating system, we can run multiple operating systems each one the best for its class of computing. This benefits the applications which demand different operating system environments.

This approach is the first of its kind and is most suitable for computation-intensive and responsive applications. This idea can be easily scaled for quad core and eight core architectures with minimal modification.

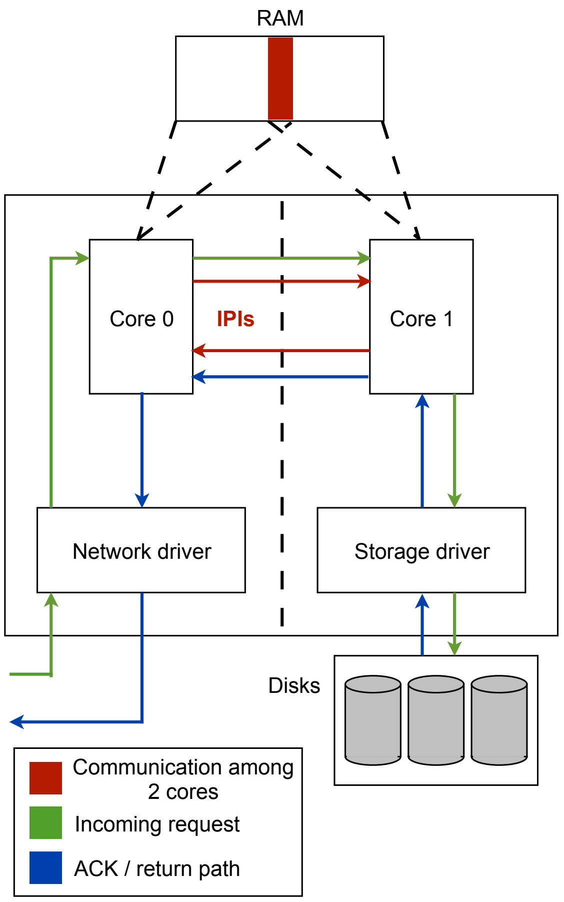


Figure 2: Twin-Linux applied to Filers

4 Design

4.1 Implementation Details

In the current scenario, The BIOS (Basic Input/Output System) selects the Bootstrap Processor (BSP) from a set of processors and the rest are Application Processors. The BIOS, initially, puts the APs in halted state, so that they do not try to execute the same BIOS code as the BSP. The BSP performs all the booting sequence and loads the Operating System after the POST (Power On Self Test). When the control is handed over to the boot loader, GRUB (GRand Unified Boot loader) in our case, only the BSP is active whereas the APs are in a halted condition with interrupts disabled. This means that the AP's local APICs are passively monitoring the APIC bus and will react only to INIT or STARTUP interprocessor interrupts (IPIs).

In our project, the GRUB code is modified in such a way that it will bring up all the APs unlike the normal scenario where only the BSP is up and the APs are in a halted state. Thus we provide SMP support to GRUB making it SMP compatible.

Our project involves the following steps::

4.1.1 Locating and Parsing the MADT (Multiple APIC Description Table)

The MADT is one of the ACPI (Advanced Configuration and Power Interface) tables which furnish information about Multiple Processors.

After locating the Root System Description Pointer (RSDP) structure, we locate the Root System Description Table (RSDT) or the Extended Root System Description Table (XSDT) using the physical system address supplied in the RSDP. The MADT is then located by walking through the RSDT. The MADT contains the

- Header
- Local APIC Address
- List of APIC Structures - This list contains all of the I/O APIC, Local APIC, Interrupt Source Override, Non-Maskable Interrupt Source, Local APIC NMI Source, Local APIC Address Override structures needed to support the platform. The APIC structures are searched for identifying the number of processors and their respective Local APIC IDs.

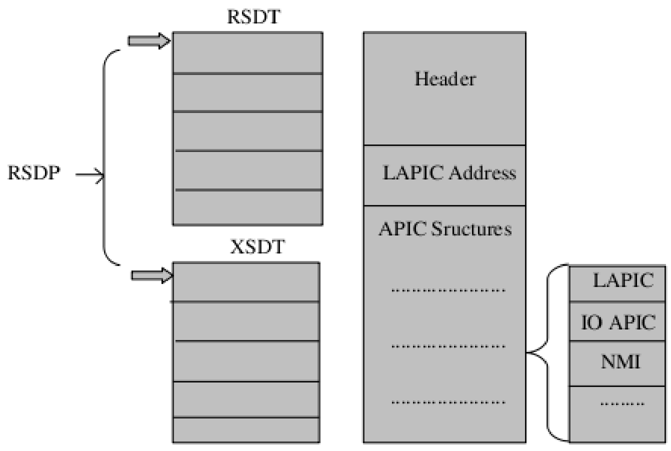
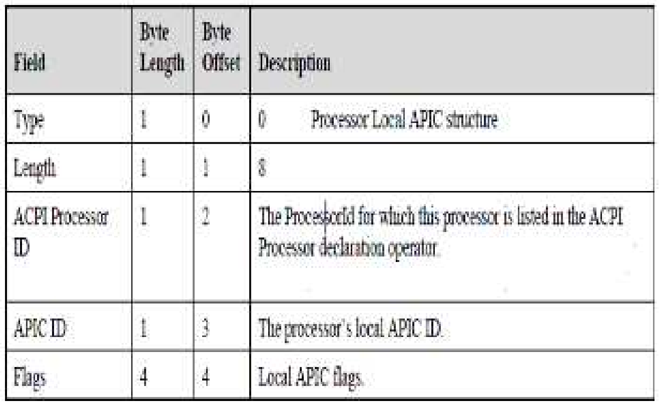


Figure 3: Overview of ACPI Tables

We obtain the Processor ID and the Processor Local APIC ID from the Processor Local APIC Structure which has the following format:

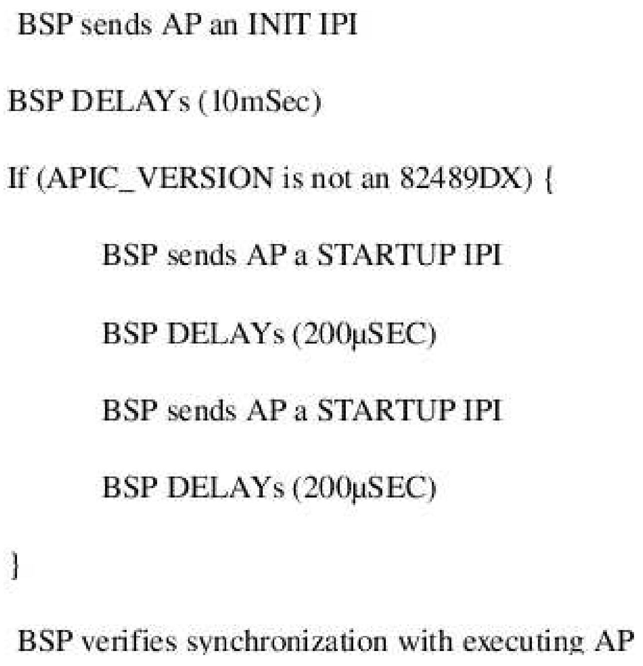


Field	Byte Length	Byte Offset	Description
Type	1	0	0 Processor Local APIC structure
Length	1	1	8
ACPI Processor ID	1	2	The ProcessorId for which this processor is listed in the ACPI Processor declaration operator.
APIC ID	1	3	The processor's local APIC ID.
Flags	4	4	Local APIC flags.

Figure 4: Processor Local APIC structure

4.1.2 Bringing up the APs

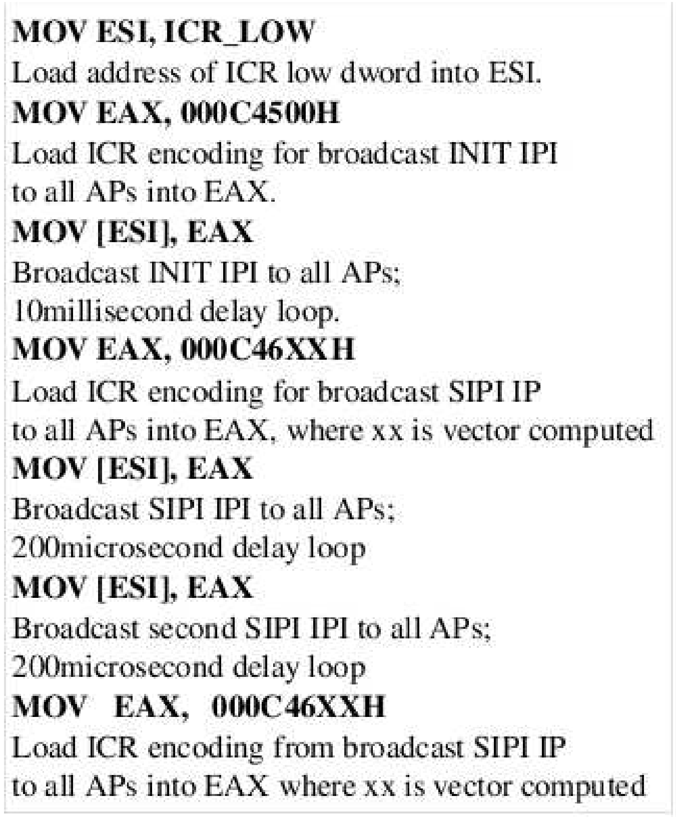
The APs will be activated by sending the IPIs using the local APIC IDs of the APs. We follow the "Universal Algorithm" for awaking the Application Processors mentioned in the Intel Multiprocessor Specification Version 1.4.



```
BSP sends AP an INIT IPI

BSP DELAYs (10mSec)

If (APIC_VERSION is not an 82489DX) {

        BSP sends AP a STARTUP IPI

        BSP DELAYs (200µSEC)

        BSP sends AP a STARTUP IPI

        BSP DELAYs (200µSEC)

}

BSP verifies synchronization with executing AP
```

Figure 5: Universal Algorithm

The following pseudo code Broadcasts an INIT-SIPI-SIPI IPI sequence to wake up and initialize the APs:



```
MOV ESI, ICR_LOW
Load address of ICR low dword into ESI.
MOV EAX, 000C4500H
Load ICR encoding for broadcast INIT IPI
to all APs into EAX.
MOV [ESI], EAX
Broadcast INIT IPI to all APs;
10millisecond delay loop.
MOV EAX, 000C46XXH
Load ICR encoding for broadcast SIPI IP
to all APs into EAX, where xx is vector computed
MOV [ESI], EAX
Broadcast SIPI IPI to all APs;
200microsecond delay loop
MOV [ESI], EAX
Broadcast second SIPI IPI to all APs;
200microsecond delay loop
MOV   EAX,   000C46XXH
Load ICR encoding from broadcast SIPI IP
to all APs into EAX where xx is vector computed
```

Figure 6: INIT-SIPI-SIPI IPI sequence

4.1.3 Loading Kernel on the APs

After bringing up all the APs, independent copy of the kernel will be loaded on each of them. Before loading of the kernel on the individual cores, following issues are taken care of:

1. Division of RAM:

   The RAM memory will be divided symmetrically according to the number of cores available, in our case, two. Thus, For a Intel Core 2 Duo processor running at 2.66 GHz with 2 GB RAM , Each kernel will be given 1 GB RAM memory and some predefined shared memory will be reserved for IPI communication. The similar approach can be followed for multi-core systems with some extensions.

2. Disabling SMP support:

   The SMP support needs to be disabled before loading of the kernels otherwise the kernel would also attempt to wake up the application processors.

3. Sharing the Hardware:

   One kernel will mask all PCI devices and other kernel will mask all the PCI-X devices. Both kernels will get one network adapter each (one PCI and other gets PCI-E cards) so that both are connected to the network.

4. User access to the OS:

   The video RAM will be divided into two halves so that the one kernel displays messages in the upper half and the other one uses the lower half for display.

5. Communication between Application processors:

   Communication between APs will be done through IPIs (Inter-Processor Interrupts) and some predefined common memory.

5 Applications

The project can be deployed in Linux environment for Intel core 2 duo architecture to efficiently utilize both the cores simultaneously.

Examples of systems where this method works are - filers (storage devices), SCSI targets, and graphics processing engines.

Intelligent switches: Intelligent switches do the job of a regular switch and filtering the data being transmitted. Filtering of data could be for identifying security threats or masquerading. This requires a combination of two classes of computing - real time computing for switch functionality (control plane) and server computing for filtering (data plane). Building an efficient switch using a single operating system kernel running on multiple cores could be a challenging task. The resultant switch may either waste computing resources or may not provide adequate responsiveness to connected devices. This situation can be improved with our proposal to run an operating system suitable for control plane on a single core while the rest of the cores run another operating system suitable for heavy server class computing in data plane.

6 Road Ahead

6.1 Enhancements

In future this idea can be implemented for architectures other than x-86 and for different boot loaders. This idea can be easily scaled for quad core and eight core architectures with minimal modification.

6.2 Conclusion

We suggested Twin-Linux an approach of running independent Linux kernels on multiple cores simultaneously. This approach opens the doors of mixed kind of computing to the x86 and open-source community. It enables users to run applications that require different operating system environments. It also provides separation of multiple environments for users. In addition, it reduces contention in operating systems kernels by reducing the synchronization costs.

It extensively takes advantage of the abundance of cores of multi-core systems. This project is a working prototype for systems which demand a combination of data-processing, real-time processing and UI processing. It also provides SMP support to GRUB.

References

[1] Intel's MP specification Version 1.4 [Online] Available `http://www.intel.com/design/archives/processors/pro/docs/242016.htm`

[2] Intel's ACPI specification Revision 4.0 [Online] Available `http://acpi.info/spec40.htm`

[3] Intel's 64 and IA-32 Architectures Software developer manual Volume 3A [Online] Available `http://www.intel.com/products/processor/manuals/`

[4] Intel Architecture Software Developers Manual Volume2: Instruction Set Reference [Online] Available `http://developer.intel.com/design/pentiumii/manuals/243191.htm`

[5] Prof. Godfrey C. Muganda, North Central College, *Intro to GNU Assembly Language on Intel Processors* [Online] Available `scr.csc.noctrl.edu/courses/csc220/asm/gasmanual.pdf`

[6] Danel P. Bovet, Marco Cesati, *Understanding The Linux Kernel*

[7] Robert Love, *Linux Kernel Development*

VirtFS—A virtualization aware File System pass-through

Venkateswararao Jujjuri
IBM Linux Technology Center
jvrao@us.ibm.com

Eric Van Hensbergen
IBM Research Austin
bergevan@us.ibm.com

Anthony Liguori
IBM Linux Technology Center
aliguori@us.ibm.com

Badari Pulavarty
IBM Linux Technology Center
badari@us.ibm.com

Abstract

This paper describes the design and implementation of a paravirtualized file system interface for Linux in the KVM environment. Today's solution of sharing host files on the guest through generic network file systems like NFS and CIFS suffer from major performance and feature deficiencies as these protocols are not designed or optimized for virtualization. To address the needs of the virtualization paradigm, in this paper we are introducing a new paravirtualized file system called VirtFS. This new file system is currently under development and is being built using QEMU, KVM, VirtIO technologies and 9P2000.L protocol.

1 Introduction

Much research has focused on improving virtualized disk and network device performance, and indeed, modern hypervisors like KVM have been able to obtain good performance as a result and are now a viable alternative to bare metal systems. Through the introduction of higher level interfaces for guests to interact with a hypervisor, we believe it is possible to obtain better performance than bare metal in consolidated workloads [8].

This paper explores an example of one such interface, VirtFS. VirtFS introduces a paravirtual file system driver based on the VirtIO [13] framework. This interface presents a number of unique advantages over the traditional virtual block device. The majority of applications (including most hypervisors) prefer to interact with an Operating System's storage API through the file system interfaces instead of dedicated disk storage APIs. By paravirtualizing a file system interface, we avoid a layer of indirection in converting guest application file system operations into block device operations and then again into host file system operations.

In addition to performance improvements over a traditional virtual block device, exposing guest file system activity to the hypervisor provides greater insight to the hypervisor about the workload the guest is running. This allows the hypervisor to make more intelligent decisions with respect to I/O caching and creates new opportunities for hypervisor-based services like de-duplification.

In Section 2 of this paper, we explore more details about the motivating factors for paravirtualizing the file system layer. In Section 3, we introduce the VirtFS design including an overview of the 9P protocol, which VirtFS is based on, along with a set of extensions introduced for greater Linux guest compatibility.

Section 4 describes the implementation of VirtFS within QEMU and within the Linux Kernel's v9fs file system. Section 5 presents our initial performance evaluation.

2 Motivation

Virtualization systems have historically focused on providing the illusion of access to underlying hardware devices (either by emulating the physical device interface or through a paravirtualization API). Within Linux, the de facto standard for paravirtual I/O communication is the VirtiIO subsystem. At the time of this writing, the mainstream Linux kernel had VirtIO devices for console, disk, and network as well as a PCI passthrough device and a memory ballooning device.

2.1 Paravirtual Application and System Services

What has been largely ignored within the mainstream virtualization community is the opportunity for raising the level of interface from the physical device layer to higher-level system and even application services. Such an approach has been used within academic and research communities in the implementation of microkernels [1], exokernels [4], multikernels [14], and satellite kernels [9]. Enabling paravirtualization of application and system services can provide a hybrid environment leveraging the security, isolation, and performance properties of microkernel-class systems within more general purpose operating systems and environments.

There are a number of reasons favoring the use of paravirtualized system services versus emulated or paravirtualized devices. One of the more compelling arguments is a higher degree of information available about what the guest system (or the guest system's user) is trying to do. For instance, within a desktop configuration, the hypervisor can provide a frame buffer device for graphics, but if we move the interface up a level we can provide paravirtualized access to the hosts windowing system (whether that windowing system is implemented by the operating system or an application infrastructure). This would allow applications within the guest to open new windows on the user's desktop versus within a frame buffer on the desktop. The difference is perhaps subtle in this example, but provide a dramatically different experience from the user's perspective.

Similarly, instead of virtualizing the network device, the hypervisor can provide a paravirtualized interface to the host IP stack. This eliminates complex configuration issues surrounding setting up network-address-translation, bridging devices, and so forth. Guest applications using sockets just use the host's TCP/IP stack directly. Such an approach works well for services which the host is already managing multiplexing for multiple applications (such as graphics, networking, audio, etc.).

An alternative to providing a paravirtualized server would be to use an existing distributed resource access protocol (such as X11 for graphics, PPTP for networking, etc) over a virtualized network device. Such an approach encounters a number of problems. First, it requires both the host and the guest have configured networks, and that the servers and clients be configured to connect to each other. Assuming that you aren't using a dedicated virtual network device for this communication you then incur a number of security concerns and potential sources for performance interference. Assuming one solves all of those problems, there is also the issue of the additional overhead of encapsulating service requests in TCP/IP which is completely unnecessary considering it is very unlikely that you will drop packets between guests and host, with much more effective flow-control being capable in such a tightly coupled system.

By contrast, a paravirtual interface provides a dedicated (performance and security isolated) channel, preconnected between guest and host (no configuration necessary), which doesn't incur any of the overheads of arbitrary and unnecessary encapsulation which going over a network (particularly a TCP/IP network) incurs. Additionally, the tighter binding of a paravirtual API may allow sharing optimizations and management simplification which is unavailable on either a device based or network based interface.

2.2 Paravirtual File Systems

File systems are a particularly good target as a paravirtual systems service. In addition to the points made above, the properties of how file systems are used, managed, and optimized by an operating system make them ideal candidates.

One of the principle problems with virtualized storage in the form of virtual disks is that data on the disk can not be concurrently accessed by multiple guests (or indeed even by the host and the guest) unless the disk is read-only. This is because of the large amount of in-memory state maintained by traditional disk file systems along with the aggressive nature of the Linux dcache and page cache. Read/write disk access can only be accomplished through exclusive access or when negotiated by a secondary protocol. The Linux storage caches introduce another level of inefficiency, caching independent copies of the disk block on both the host and the guest.

If we use a traditional distributed file system (such as NFS or CIFS) over a virtualized network device to access storage, we run into the configuration, management, and encapsulation overheads mentioned previously. We also encounter problems with two management domains for the purposes of user ids, group ids, ACLs, and so forth. The distributed file systems also incur the double-cache behavior of the virtual disks. The other problem is that many distributed file systems impose their own file system semantics on operations, which may be different from the behavior expected from a disk file system.

2.3 Application Use Cases

Perhaps the most straightforward application of a paravirtualized file system is to replace the virtual disk as the root file system. When the system boots, the bundled ram disk simply connects to an appropriately labeled VirtFS volume to retrieve the root volume. Such an approach would allow host-based stackable file systems to be used for rapid cloning, management, and update [10].

In addition to root file systems, the paravirtual file system can be used to access a shared file system coherently from several guests. It can also be used to provide guest to guest file system access. Additionally, it could be used to access synthetic file systems on the host or other guests in order to access management and control interfaces.

Another use case, which we utilized as part of the Libra [2] and PROSE [15] projects is to provide file system offload API for applications running within a LirbaryOS. In such instances, the application is only running on top of a very thin OS interface which doesn't itself contain system services such as a disk file system or network stack. These services are obtained remotely through forwarding I/O requests (at a file system or socket level) to a server running on the host.

Finally we would like to refer to the use cases of the cloud environment. In a cloud environment where multiple guests share resources on a host, portions of the host file systems can be exported and VirtFS mounted on guests giving a secure window of host file systems on the guest. This gives guests more like a local file system interface to portions of host file system. VirtFS also could be very useful to provide a secure way to provide storage services to different customers from a single host filesystem. This also helps to take advantage of various file system features like de-dup, snapshots etc.

3 Design

VirtFS provides functionality that is somewhat similar to a traditional network file systems (NFS/CIFS). The QEMU server elects to export a portion of its file system hierarchy, and the client on the guest mounts this using 9P2000.L protocol. Guest users see the mount point just like any of the local file systems, while the reads and writes are actually happening on the host file system.

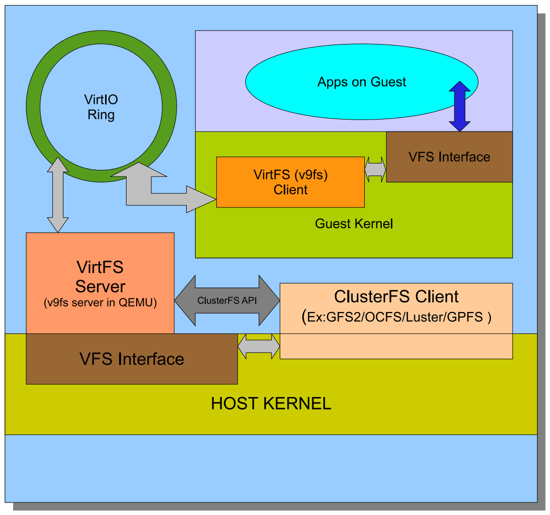


Figure 1: VirtFS block diagram

Figure 1 shows the high level VirtFS anatomy. Server is part of QEMU and client is part of the guests kernel. The protocol is exchanged between the client and the server over VirtIO transport (section 3.5). Server running in the user mode opens up potential for direct communication with the fileserver API (section 6.4).

The major difference between a traditional network file system and VirtFS is its simplicity and optimization to the KVM environment. Our approach is to leverage a light-weight distributed file system protocol directly on top of a paravirtualized transport in order to remote offload elements of the Linux VFS API to a server run on the virtualization host (or within another partition). In order to expedite implementation, we selected a preexisting distributed file system which closely matched our desired approach 9P, and provided a virtualization specific transport interface for it through VirtIO. We are extending the 9P protocol to better match the Linux VFS API. The 9P protocol [3], originally developed as part of the Plan 9 research operating system from Bell Labs [11], and incorporated as a network based distributed file system within the mainline Linux kernel during the 2.6.14 release [6].

3.1 Plan 9 Overview

Plan 9 was a new research operating system and associated applications suite developed by the Computing Science Research Center of AT&T Bell Laboratories (now a part of Lucent Technologies), the same group that developed UNIX , C, and C++. Their intent was to explore potential solutions to some of the shortcomings of UNIX in the face of the widespread use of high-speed networks to connect machines. In UNIX, networking was an afterthought and UNIX clusters became little more than a network of stand-alone systems. Plan 9 was designed from first principles as a seamless distributed system with integrated secure network resource sharing.

Plan 9s transparent distributed computing environment was the result of three core design principles which permeated all levels of the operating system and application infrastructure:

1. develop a single set of simple, well-defined interfaces to services
2. use a simple protocol to securely distribute the interfaces across any network
3. provide a dynamic hierarchical structure to organize these interfaces

In Plan 9, all system resources and interfaces are represented as files. UNIX pioneered the concept of treating devices as files, providing a simple, clear interface to system hardware. Plan 9 took the file system metaphor further, using file operations as the simple, well-defined interface to all system and application services. The benefits and details of this approach are covered in great detail in the existing Plan 9 papers [12].

3.2 9P Overview

9P represents the abstract interface used to access resources under Plan 9. It is somewhat analogous to the VFS layer in Linux. In Plan 9, the same protocol operations are used to access both local and remote resources, making the transition from local resources to cluster resources to cloud resources completely transparent from an implementation standpoint. Authentication is built into the protocol, and the protocol itself may be run over encrypted and digested transports. It is important to understand that all 9P operations can be associated with different active semantics in synthetic file systems. Traversal of a directory hierarchy may allocate resources, or set locks. Reading or writing data to a file interface may initiate actions on the server, such as when a file acts as a control interface. The dynamic nature of these synthetic file system semantics makes caching dangerous and in-order synchronous execution of file system operations a desirable default. For an example of the potential difficulties, think of interacting with the proc or sysfs file systems in Linux over a cached file system mount where some of the data can be dozens of seconds out of date with the actual state on the server.

The 9P protocol itself requires only a reliable, in-order transport mechanism to function. It is commonly used on top of TCP/IP, but has also been used over RUDP, PPP, and over raw reliable mechanisms such as the PCI bus, serial port connections, and shared memory.

The base 9P protocol is based on 12 paired protocol operations (each with a request and response version) and a special error response which is used to notify the client of problems with a request. They are made up of protocol accounting operations (version, authentication, attach, flush), file operations (lookup, create, open, read, write, remove, close), and meta-data operations (set attributes, get attributes). The nature of the operations is strict balanced RPC (each request gets a single response). The protocol is stateful, requiring the server to maintain certain aspects of the state of session and outstanding transactions. The protocol has provisions for supporting multiple outstanding transactions on the same transport and can be run in synchronous or asynchronous modes (depending on the implementation of the client and the server).

As mentioned previously, the 9P file system was added to the 2.6.14 mainline linux kernel with support for TCP/IP and named pipe transports (such that it could be used to access user-space file systems). Later, in 2.6.x the transport interfaces were modularized to allow for alternative transports to be plugged into the 9P client. In 2.6.x an RDMA interface was added by Tom

Tucker support infiniband, and in 2.6.x fscache support was added by Abhishek Kulkarni. It currently has the distinction of being the distributed file system implemented with the fewest lines of code in the Linux kernel.

3.3 9P Extensions

The 9P protocol was originally designed specifically for the Plan 9 operating system, and as such implements a subset of the rich functionality of the Linux VFS layer. Additionally, it has slightly different definitions for certain parameters (such as permissions, open-mode, etc.). It also takes a different approach to user and group management, using strings instead of ids.

In order to adapt 9P to better supporting POSIX during the Linux port, the protocol was extended with the 9P2000.u (for unix) version. This protocol version provided support for numeric user and group ids, provided additional mode and permission bits as well as an extended attributes structure in order to be POSIX compliant, and augmented existing operations to support symlinks, links, and creation of special files.

The .u protocol has been used for the past several years, but had a number of deficiencies. It did not include full support for Linux VFS operations. Notably absent was support for quotas, extended attributes, and locking. Furthermore, its overloading of existing operations to provided extended functions made developing servers and clients to support 9p2000.u problematic, and no partial support for 9p2000.u was possible due to differences in the size of operation protocol messages and ambiguous definitions of extension fields.

In response to these deficiencies, a new approach to offering extensions to the protocol was developed. While core protocol elements (such as size prefixed packets, tagged concurrent transactions, and so forth) remain the same, extended features will get their own operations in a complimentary op-code space to the existing prtoocol. The Linux binding for 9P, titled 9P2000.L, will be the first approach at such an extension and will be aimed at addressing the .u deficiencies while making the protocol address a larger subset of the functionality provided by Linux.

As stated previously, The new 9P2000.L extension will will exist in a complimentary op-code name space – that is to say operation identifiers will not overlap with existing operations, and all extensions will use new operations as opposed to changes to existing operations. Alternate extension op-codes spaces may be negotiated by optional postfixes during version negotiation – but every effort will be made to keep operations within the existing name space. It was also decided that all extensions should be treated as optional, servers which dont́ wish to implement them (or any subset of them) simply returns error – well behaved clients will fall back to the core 9P2000 operations.

3.4 KVM and QEMU

KVM is a set of Linux kernel modules that allows a userspace process to execute code in a special process mode. On x86, this is often referred to as compressed ring 0. Much like vm86 syscall, this mode allows a userspace program to trap interesting events such as I/O operations.

QEMU uses the interfaces provided by KVM to implement full system virtualization emulating standard PC hardware such as an IDE disk, VGA graphics, PCI networking, etc. In the case of KVM, any I/O requests a guest operating system makes are intercepted and routed to the user mode to be emulated by the QEMU process [7].

3.5 VirtIO Transport

VirtIO is a paravirtual IO bus based on a hypervisor neutral DMA API. With KVM on x86, which is the dominant platform targetted by this work, the underlying transport layer is implemented in terms of a PCI device.

A PCI device is used to enable support for the widest variety of guests since all modern x86 operating systems support PCI. The VirtIO PCI implementation makes extensive use of shared memory. This includes the use of lockless ring queues to establish a message passing interface and indirect reference to scatter/gather buffers to enable zero-copy bulk data transfer.

These properties of the VirtIO PCI transport allow VirtFS to be implemented in such a way that guest driven I/O operations can be zero-copy. This is a key advantage of VirtFS compared to using a network file system which would always require at least one (but usually more) data copies.

4 Implementation

4.1 Details of the Implementation of VirtFS server in QEMU

Making the VirtFS server part of QEMU is a very natural design decision. KVM+QEMU+VirtIO presents an ideal platform for the VirtFS server where it can efficiently interact with the guest providing one of the efficient paravirtual network file system interfaces. VirtFS server is facilitate in QEMU by defining two types of devices.

One is virtio-9p-pci device, and this will be used to transport protocol messages and data between the host and the guest. Second one is fsdev device, this is used to define the export file system characteristics like file system type, security model (section 4.2)

A typical usage of QEMU to export a file system is:

-fsdev local,id=exp1,path=/tmp/,security_model=mapped -device virtio-9p-pci,fsdev=exp1,mount_tag=v_tmp

On the client it can be mounted with:

$ mount -t 9p -o trans=virtio v_tmp /mnt

4.2 Security Model

Security is one of the most important aspects of the file system design and it needs to be handled with care to cater specific needs of the exploiters. There are two major use cases that can make use of this new technology and their security requirements are quite different. One demands a complete isolation of guest user domain from that of the hosts hence practically eliminating any security issues related to setuid/setgid and root. This is a very practical use case for certain classes of cloud workloads where multi-tenancy is a requirement. A complete isolation of user domain can practically make two competing customers share the same file system.

The other use case is playing along with the traditional network file-serving methodologies like NFS and CIFS by sharing user domains of the host and guest. This method edges out by offering flexibility to export the same file system through other network file systems along with VirtFS.

Linux being POSIX complaint, offers a simple yet powerful file system permission model: Every file system object is associated with three sets of permissions that define access for the owner, the owning group, and for others.

Each set may contain Read (r), Write (w), and Execute (x) permissions. This scheme is implemented using only nine bits for each object. In addition to these nine bits, the Set User Id, Set Group Id, and Sticky bits are used for number of special cases.

Although this traditional model is sufficient for most of the use cases, it falls short of satisfying the needs of the modern world. The need for more granular permissions eventually resulted in a number of Access Control List (ACL) implementations on UNIX like Posix ACLs, Rich ACLs, NFSv4 ACLs etc. These ACL models were developed for specific needs and has very limited degree of commonality. This poses a major challenge for network file systems as it may have to support wide variety of file systems with different ACL models [5].

VirtFS, being one of the first file systems in the genre of paravirtual file systems need to consider all options and use cases. This type of file systems need to play a dual role, where it should be a viable alternative to net work file systems like NFS and CIFS, and also it needs to fill-in the new space where it should provide special optimizations and considerations for the needs of guest operating systems. To address these special needs and the use cases explained above, we came up with two types of security models for VirtFS: the mapped security model and the passthrough security model. QEMU administrator picks a model at the start-up and is expected to stick with that.

4.2.1 Security model: mapped

In this security model, VirtFS server intercepts and maps the file object create and get/set attribute requests. Files on the fileserver will be created with VirtFS server's (QEMU) user credentials and the client-user's credentials are stored in extended attributes. On the request to get attributes, server extracts the client-user's credentials from extended attributes and sends them to the client. Since the files are created on the fileserver with QEMU credentials, this model keeps the guests user domain completely isolated from the host's user domain. Host view shows QEMU as the owner of all files created by any user(including root) on the guest hence provides complete isolation and security.

The access permissions for user attributes are defined by the file permission bits. The file permission bits of regular files and directories are interpreted differently from the file permission bits of special files and symbolic links. For regular files and directories the file permission bits define access to the file's contents, while for device special files they define access to the device described by the special file. The file permissions of symbolic links are not used in access checks. These differences would allow users to consume file system resources in a way not controllable by disk quotas for group or world writable special files and directories. For this reason, extended user attributes are only allowed for regular files and directories only.

Given that the user space extended attributes are available to regular files only, special files are created as regular files on the fileserver and appropriate mode bits are added to the extended attributes. This method presents all special files and symlinks as regular files on the fileserver while they are represented as special files on the guest mount.

```
On Host:
# ls -l
drwx------. 2 virfsuid virtfsgid 4096 2010-05-11 09:19 adir
-rw-------. 1 virfsuid virtfsgid    0 2010-05-11 09:36 afifo
-rw-------. 2 virfsuid virtfsgid    0 2010-05-11 09:19 afile
-rw-------. 2 virfsuid virtfsgid    0 2010-05-11 09:19 alink
-rw-------. 1 virfsuid virtfsgid    0 2010-05-11 09:57 asocket1
-rw-------. 1 virfsuid virtfsgid    0 2010-05-11 09:32 blkdev
-rw-------. 1 virfsuid virtfsgid    0 2010-05-11 09:33 chardev

On Guest:
# ls -l
drwxr-xr-x 2 guestuser guestuser  4096 2010-05-11 12:19 adir
prw-r--r-- 1 guestuser guestuser     0 2010-05-11 12:36 afifo
-rw-r--r-- 2 guestuser guestuser     0 2010-05-11 12:19 afile
-rw-r--r-- 2 guestuser guestuser     0 2010-05-11 12:19 alink
srwxr-xr-x 1 guestuser guestuser     0 2010-05-11 12:57 asocket1
brw-r--r-- 1 guestuser guestuser  0, 0 2010-05-11 12:32 blkdev
crw-r--r-- 1 guestuser guestuser  4, 5 2010-05-11 12:33 chardev
```

Most of the file systems offer only one block for extended attributes. This limitation curbs the use of extended attributes to stored the target link location. Under this model, target link location is store as file data using write() and readlink reads it back through read()

```
On Guest:
# ls -l asymlink
lrwxrwxrwx 1 root root   6 2010-05-11 12:20 asymlink -> afile

On Host:
# ls -l asymlink
-rw-------. 1 root root    6 2010-05-11 09:20 asymlink
# cat asymlink
afile
#
```

Just like any security model, this has its own advantages and limitations. One of the main strength and weakness of this model is, the host file system will be VirtFSized. While the guest doesn't see any difference, host users and tools need to understand the security model to use the file system credentials on the host. This is a strength because it completely isolates the guest users address space, it allows the server to run as a non-privileged user, hence it involves no issues of root-squashing or setuid issues. Hence this security model makes it perfect for the guest to run in its own security island.

4.2.2 Security model: Passthrough.

In this security model, VirtFS server passes down all requests to the underlying file system. File system objects on the fileserver will be created with client-user's credentials. This can be done by setting setuid()/setgid() during creation or chmod/chown immediately after creation. At the end of create protocol request, files on the fileserver will be owned by client-user's credentials.

```
On Host:

# grep 611 /etc/passwd
hostuser:x:611:611::/home/hostuser:/bin/bash

# ls -l
-rwxrwxrwx. 2 hostuser hostuser 0 2010-05-12 18:14 file1
-rwxrwxrwx. 2 hostuser hostuser 0 2010-05-12 18:14 link1
srwxrwxr-x. 1 hostuser hostuser 0 2010-05-12 18:27 mysock
lrwxrwxrwx. 1 hostuser hostuser
5 2010-05-12 18:25 symlink1 -> file1

On Guest:
$ grep 611 /etc/passwd
guestuser:x:611:611::/home/guestuser:/bin/bash

$ ls -l
-rwxrwxrwx 2 guestuser guestuser 0 2010-05-12 21:14 file1
-rwxrwxrwx 2 guestuser guestuser 0 2010-05-12 21:14 link1
srwxrwxr-x 1 guestuser guestuser 0 2010-05-12 21:27 mysock
lrwxrwxrwx 1 guestuser guestuser 5 2010-05-12 21:25 symlink1 -> file1
```

This model lets the host tools understand the filessytem, and statistics collection and quotas enforcement will be easier. But this needs the server to run as a privileged user (root) and also exposes root/setuid security issues just like NFS

4.2.3 ACL Implemenation

Access Control Lists (ACLs) steps in where the traditional mode bits security model is not sufficient. ACLs allow fine grained control by the assignment of permissions to individual users and groups even if these do not correspond to the owner or the owning group. Access Control Lists are a feature of the Linux kernel and are currently supported by many common file systems and its support is crucial with the wide spread usage of file sharing among heterogeneous systems like Linux/Unix, and Windows. While ACLs are an essential part of comprehensive security scheme, the lack of universal standards often make the design complex and complicated and VirtFS is not an exception.

At the time of writing this paper, we are still at the design stage of implementing the following aspects of ACL.

A newly created file system object inherits ACLs from its parent directory. The common practice is a non-directory object inherits the parent default ACLs as its access ACLs and directory object inherits parent default ACLs as its default ACLs. To make things little more complicated, there are no standards on the inheritance algorithm and they differ for each ACL model.

In adition to the gid/uid/mode-bits, ACLs of the file system object will be checked before granting the requesting access to a file system object. This checking can be done either on the client or on the server. If the enforcement is on the server, it need to have the context of the client-user's credentials, which makes the protocol very bulky. For this reason it becomes a natural choice to have permission checks at the client.

We are leaning towards supporting at least NFSv4 level ACLs and employing client to do the ACL enforcement while the ACL inheritance is servers job. The server can choose to delegate it to the fileserver.

4.3 Current state of the project

At the time of writing this paper, the project is actively being worked on with a team of seven IBM engineers and the community is just starting to get excited. Several patches has been posted to the community mailing lists. Client side patches are being posted to the v9fs-developer@lists.sourceforge.net list and server side patches are being posted to the qemu-devel@nongnu.org list.

QEMU community blessed the project by accepting the VirtFS server feature patch set into the mainline. This is a significant milestone for the project. A patch set introducing the security model as explained above is also on the mailing list awaiting acceptance. We are also working towards allowing a more asynchronous model for the QEMU server which should boost performance significantly.

On the client side, the team has contributed dozens of patches to the 9P client of the Linux kernel. We are actively working on fixing pre-exisitng bugs and defining the 9P2000.L protocol extension. As mentioned above, the main intent and focus of this new protocol extension is to define an efficient Linux friendly protocol. In this process, we are contributing to the improvement and stabalization of the 9P client as a whole.

5 Performance

As mentioned earlier, VirtFS is intended to be the network file system specialist in the virtualization world. By the virtue of its design VirtFS is expected to yield better performance compared to its alternatives like NFS/CIFS. Though we are just getting started, lot of focus is given to the performance aspect of the implementation and the protocol design. This section covers the initial performance evaluation and comparisons with its counterparts.

5.1 Test Environment

The following test environment is used for the performance evaluation.

Host: 8 CPU 2.5GHz Intel Xeon server, 8 GB of memory, Qlogic 8Gb fiber channel controller, 24 disks JBOD. 2.6.34-rc3 kernel, mainline qemu.

Guest: 2 vCPU, 2GB memory running 2.6.34-rc3 kernel.

The following configuration is used for gathering performance numbers:

Configuration-1: Sequential read and write performance of VirtFS in comparison to NFS and CIFS. In this configuration, file system is mounted on the host and guest accesses the file system through VirtFS, NFS or CIFS.

Configuration-2: Sequential read and write performance of VirtFS in comparison to block-device performance. In this configuration, guest directly accesses the file system on the block device.

Setup

- For comparisons with blockdev, each block device is exported to guest as a block device (cache=writethrough). Filesystem is not mounted in the host and its mounted only on the guest for this testing.
- Each filesystem is stripped across 8 disks to eliminate single disk bottlenecks. 3 such filesystems are used for the performance analysis.
- Each filesystem is mounted in the host and exported (using defaults) to the guest over virtio-net.
- Filesystems are un-mounted, remounted and re exported before each read test to eliminate host level pagecache.

Commands

Used simple "dd" tests to simulate sequential read and sequential write patterns.

Write:
dd if=/dev/zero of=/mnt/fileX bs=<blocksize> count=<count>

Read:
dd if=/mnt/fileX of=/dev/null bs=<blocksize> count=<count>

blocksize - 8k, 64k, 2M.
count = number of blocks to do 8GB worth of IO.
All the tests are conducted in the guest with various IO block sizes.

5.2 VirtFS, NFS and CFS comparison

Figure-2 compares sequential read performance of VirtFS at various block sizes against NFS and CIFS. VirtFS clearly outperforms NFS and CIFS at all block sizes

Figure-3 compares sequential write performance of VirtFS at various block sizes against NFS and CIFS. Again, as expected VirtFS outperforms NFS and CIFS at all block sizes.

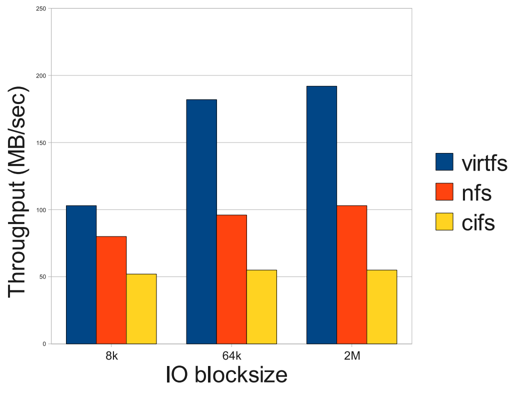


Figure 2: Comparing Sequential Read among VirtFs/NFS/CIFS

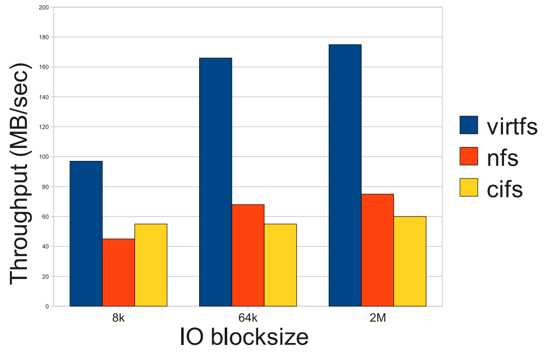


Figure 3: Comparing Sequential Write among VirtFs/NFS/CIFS

5.3 VirtFS, block device comparison

Figure-4 compares sequential read performance of VirtFS against local file system access by block device. At the time of writing this paper, VirtFS doesn't seem to scale well with number of file systems. This is due to single threaded implementation of VirtFS server in QEMU. Currently efforts are underway to convert it to multi threaded implementation.

Figure-5 compares sequential write performance of VirtFS against the block device access. As these are not synchronous writes, VirtFS is able to scale very well by taking advantage of the host and guest page caches.

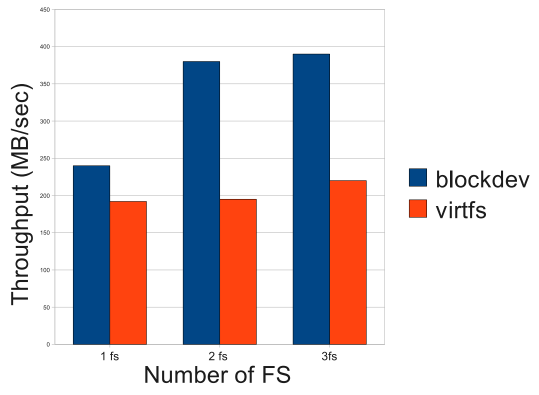


Figure 4: Comparing Sequential Read between VirtFs and Block device

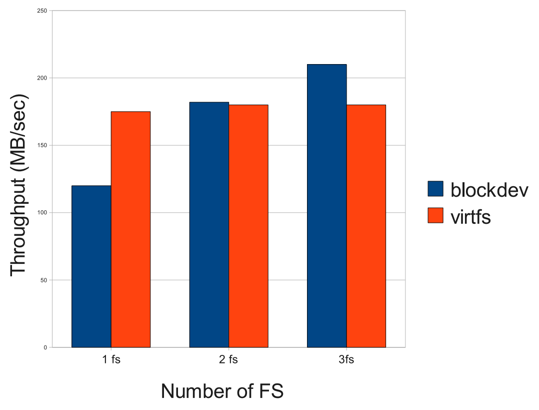


Figure 5: Comparing Sequential Write between VirtFs and Block device

6 Next Steps

6.1 Complete 9P2000.L

Defining an optimal and efficient protocol that is well suited for Linux's needs is our current focus. Implementing the defined 9P2000.L protocol messages both on the client and server and getting them into mainline will be our immediate priority. We are aiming to make the prototype available for early adopters and exploiters as quickly as possible and also plan for next releases with more advanced features.

6.2 Security

ACL implementation for network file systems is always a challenge as the server need to support different file systems with different ACL models. It becomes more complicated for VirtFS as we are supporting two different security models as mentioned above. The Linux kernel supports POSIX ACLs only but we are planning on supporting NFSv4 level ACLs. This may throw more challenges on the way.

6.3 Page Cache Sharing

Reading a file on the network file system mount on the guest makes the host to read the page onto its cache, send the data over the protocol to guests page cache before it is consumed by the user. In the case KVM, both host and guest are running on the same hardware and are using same physical resources. That means the page-cache block is being duplicated in different regions of the memory. One could extrapolate this problem to the worst case scenario where almost half of the system's page cache is wasted in duplication. We need to come up with a method where the host and guest share the same physical page. This eliminates the need for data copy between guest and host and provide better data integrity and protection to the application yielding extreme performance benefits. Sharing pages between two operating systems is challenging as we can run into various locking and distributed caching issues.

6.4 Interfacing with File system APIs

A user space server has an unique opportunity where it can interact directly with the file system API instead of going through the system call/VFS interface. VirtFS server, being a user space server can be modeled to plug directly into the fileservers API if one is available. This opens up speciality features offered by the fileserver to the guest through VirtFS and hence gives an opportunity to give a true pass-through representation of the fileserver. Simply put, this effort should provide a layer of indirection between the third party/specialized file systems on the host and virtual machines, enabling any application dependent on the special features of these file systems to run on the guests VirtFS mount.

7 Conclusions

In this paper we have motivated and described the development of a paravirtual system service, namely VirtFS. We have described its integration into KVM/QEMU and the Linux kernel, and discussed both its underlying protocol and the changes we are making to that protocol to improve its support for Linux VFS features. We have described few different use cases and included a discussion of different security models for deploying this paravirtual file systems in cloud environments. Finally, we have shown that our initial work has superior performance to the use of conventional distributed file systems and even reasonable performance when compared to the use of a paravirtualized disk. As described in the next steps (section 6) we plan on continuing the development to improve performance, stability, security, and features. It is our belief that as virtualization continues to becomes more pervasive, paravirtual system services will play a larger role – perhaps completely overtaking paravirtual devices and device emulation.

8 Acknowledgements

We would like to thank Sripathi Kodi for his valuable review comments.

We would like to thank Jim Garlick for his contributions to 9P2000.L. We'd also like to thank Ron Minnich and Latchesar Ionkov who have provided valuable feedback as well as being th eco-maintainers of the 9P subsystem in Linux. We'd also like to thank all those members of the Linux kernel development community who have contributed patches to the 9p base infrastructure we are leveraging.

References

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. pages 93–112, 1986.

[2] Glenn Ammons, Jonathan Appavoo, Maria Butrico, Dilma Da Silva, David Grove, Kiyokuni Kawachiya, Orran Krieger, Bryan Rosenburg, Eric Van Hensbergen, and Robert W. Wisniewski. Libra: a library operating system for a jvm in a virtualized execution environment. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 44–54, New York, NY, USA, 2007. ACM.

[3] Bell-Labs. Introduction to the 9p protocol. *Plan 9 Programmers Manual*, 3, 2000.

[4] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 251–266, New York, NY, USA, 1995. ACM.

[5] Andreas Grunbacher. Posix access control lists on linux. *USENIX paper - Freenix track*, 2003.

[6] Eric Van Hensbergen and Ron Minnich. Grave robbers from outer space using 9p2000 under linux. In *In Freenix Annual Conference*, pages 83–94, 2005.

[7] M. Tim Jones. Discover the linux kernel virtual machine - learn the kvm architecture and advantages.

[8] Anthony Liguori and Eric Van Hensbergen. Experiences with content addressable storage and virtual disks. In *In Proceedings of the Workshop on I/O Virtualization (WIOV)*, 2008.

[9] Edmund B. Nightingale, Chris Hawblitzel, Orion Hodson, Galen Hunt, and Ross Mcilroy. Helios: Heterogeneous multiprocessing with satellite kernels. In *In Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[10] Fabio Oliveira, Gorka Guardiola, Jay A. Patel, and Eric Van Hensbergen. Blutopia: Stackable storage for cluster management. In *CLUSTER*, pages 293–302, 2007.

[11] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[12] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The use of name spaces in plan 9. *SIGOPS Oper. Syst. Rev.*, 27(2):72–76, 1993.

[13] Rusty Russel. virtio: towardsa a de-facto standard for virtual I/O devices. In *Operating Systems Review*, 2008.

[14] Adrian Schupbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs. Embracing diversity in the barrelfish manycore operating system. In *In Proceedings of the Workshop on Managed Many-Core Systems*, 2008.

[15] Eric Van Hensbergen. P.R.O.S.E.: partitioned reliable operating system environment. *SIGOPS Operating Systems Review*, 40(2):12–15, 2006.

Taking Linux Filesystems to the Space Age: Space Maps in Ext4

Saurabh Kadekodi
Spring Computing Pvt. Ltd.
saurabhkadekodi@gmail.com

Shweta Jain
Clogeny Technologies Pvt. Ltd.
atewhs.jain@gmail.com

Abstract

With the ever increasing filesystem sizes, there is a constant need for faster filesystem access. A vital requirement to achieve this is efficient filesystem metadata management.

The bitmap technique currently used to manage free space in Ext4 is faced by scalability challenges owing to this exponential increase. This has led us to re-examine the available choices and explore a radically different design of managing free space called Space Maps.

This paper describes the design and implementation of space maps in Ext4. The paper also highlights the limitations of bitmaps and does a comparative study of how space maps fare against them. In space maps, free space is represented by extent based red-black trees and logs. The design of space maps makes the free space information of the filesystem extremely compact allowing it to be stored in main memory at all times. This significantly reduces the long, random seeks on the disk that were required for updating the metadata. Likewise, analogous on-disk structures and their interaction with the in-memory space maps ensure that filesystem integrity is maintained. Since seeks are the bottleneck as far as filesystem performance is concerned, their extensive reduction leads to faster filesystem operations. Apart from the allocation/deallocation improvements, the log based design of Space Maps helps reduce fragmentation at the filesystem level itself. Space Maps uplift the performance of the filesystem and keep the metadata management in tune with the highly scalable Ext4.

1 Introduction

Since linux kernel 2.6.28, Ext4 has been included in the mainstream and has become the default filesystem with most distributions. In a very short span of time, it has grown in popularity as well as stability. Over its predecessor Ext3, Ext4 brings many new features like scalability, delayed allocation, multiple-block allocation, improved timestamps[1] among others. One of its most important features is its use of extents.

1.1 Impact of extents

An extent is a combination of two integers, the first is the start block number and the second is the number of contiguous physical blocks ahead of the start block.

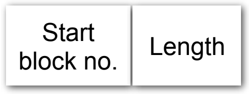


Figure 1: Extent

Inodes in Ext4 no longer use the indirect block mapping scheme. Instead they have extents which are used to denote range of contiguous physical blocks owned by a file. Huge files are split into several extents. Four extents can be stored in the Ext4 inode structure directly and if more extents are required (eg. in case of very large, highly fragmented or sparse files) they are stored on disk in the form of an Htree. Extents offer multiple advantages. With extents, the amount of metadata to be written to describe contiguous blocks is much lesser than that required by the double/triple indirect mapping scheme. This results in improved performance for sequential read/writes. They also greatly reduce the time to truncate a file as also the CPU usage[2]. Moreover, extents encourage continuous layouts on the disk, resulting in lesser fragmentation[4]. Extents have been shown to bring about a 25% throughput gain in large sequential I/O workloads when compared with Ext3. Tests conducted using the Postmark benchmark, which simulates a mail server, with creation and deletion of a large number of small to medium files also showed similar performance gain[2][10].

The crux of features like delayed allocation and persistent preallocation is the extent. Preallocation deals with pre-allocating/reserving contiguous disk blocks for a file without actually writing to them immediately. Due to this, the file remains in a more contiguous state, enhancing the read performance of the filesystem. Moreover, it also helps in reducing fragmentation to a great extent. With preallocation, the applications are assured that they will always get the space they need, avoiding situations where the filesystem gets full in the middle of an important operation[4].

With delayed allocation, block allocations are postponed to page flush time, rather than writing them to disk immediately. As a result there are no block allocations for short lived files, and several individual block allocation requests can be clubbed into one request[2]. Since the exact number of blocks required are known at flush time, the allocator tends to assign a contiguous chunk rather than satisfy short term needs, which probably would have split the file.

1.2 Extents in free space management

Ext4 also introduced extents in its free space management technique. To avoid changing the on-disk layout, extents were maintained only in-memory, eventually relying on the on-disk bitmap blocks. One of the problems faced by the Ext3 block allocator, which allocated one block at a time, was that it did not perform well for high speed sequential writes[5]. Alex Tomas addressed this problem and implemented *mballoc* - the multiple block allocator. Mballoc uses the classic *buddy* data structure.

Whenever a process requests for blocks, the allocator refers the bitmap to find if the goal block is available or not. After this point is where the traditional balloc and mballoc differ. Balloc would have returned the status of just one block and this would have continued for every block that the process requires. Mballoc, on the other hand constructs a buddy data structure as soon as it fetches the bitmap in memory. Buddy is simply an array of metadata, where each entry describes the status of a cluster of nth power of 2 blocks, classified as free or in use[5]. After the allocator confirms the availability of the goal block from the bitmap, it refers the buddy to find the free extent length starting from the goal block, and if found, returns the extent to the requesting process. In case the extent is not of appropriate length, the allocator continues to search for the best suitable extent. If after searching for a stipulated time, a larger extent is not found, then mballoc returns the largest extent found in that search.

Both, the bitmap and the buddy are maintained in the page cache of an in-core inode[3]. Before flushing the bitmap to disk, information from the buddy is reflected in the bitmap and the buddy is discarded.

The bitmap and buddy combination enabled mballoc to speed up the allocation process. The combination of delayed allocation and multiple block allocation have been shown to significantly reduce CPU usage and improve throughput on large I/O. Performance testing shows a 30% throughput gain for large sequential writes and 7% to 10% improvement on small files (e.g., those seen in mail-server-type workloads)[2]. The credit for this goes to mballoc's ability to report free space in the form of extents. However, this mechanism still raises certain issues:

1. Even though the buddy scheme of Ext4 is more efficient at finding contiguous free space than the bitmap-scanning scheme of Ext3, the overhead of fetching and flushing bitmaps is still involved. Updating the bitmaps in-memory is fast, seeking and fetching them is the bottleneck.

2. Initializing the buddy bitmaps entails some cost[3]. Every time a bitmap is fetched into memory, there is an extra overhead of constructing the buddy.

3. Usually, only one structure is used to define the free space status of the filesystem. However in case of mballoc, both the buddy and the bitmap are used. Both these structures have to be updated on every allocation/deallocation. This introduces redundancy.

4. The buddy technique consumes twice the amount of memory as compared to only bitmaps. Thus, lesser number of bitmaps can reside in memory, resulting in more seeks.

5. Whenever preallocation has to be discarded, there is a comparison done between the buddy and the on-disk bitmaps. The on-disk bitmaps need to be referred to find out the exact chunk of space utilized. This leads to even more seeks.

6. Finally, when a filesystem has significantly aged, the buddy structure will be of little use as the available disk space may hardly be available contiguously. In such cases we currently have to rely on the primitive bitmap technique which is inefficient and slow.

The above points clearly indicate that optimizations are possible in the Ext4 block allocator. Something like an in-memory list for more optimal free extent searching, would further assist mballoc[3].

The underlying phenomenon of the success of extents, is that the filesystem usually deals with blocks in chunks, unlike inodes which are dealt with individually. Creation/deletion of files/directories mostly involves dealing with chunk of blocks. It thus seems natural to represent their free space status, also in the form of chunks. We take this idea further and explore a technique that is based entirely on extents. This technique is called **Space Maps**.

2 Design Details

In implementing this technique, there is a change to Ext4's on-disk layout. In Ext4 without space maps, for 4K block size, each 128MB chunk is called a blockgroup and each blockgroup has a bitmap. For space maps, we have combined a number of such block groups, amounting to 8GB space, calling it a metablockgroup, and each metablockgroup has a space map. The 8GB size is the default size of a metablockgroup for a 4K block size filesystem. The metablockgroup size is tunable at mkfs time.

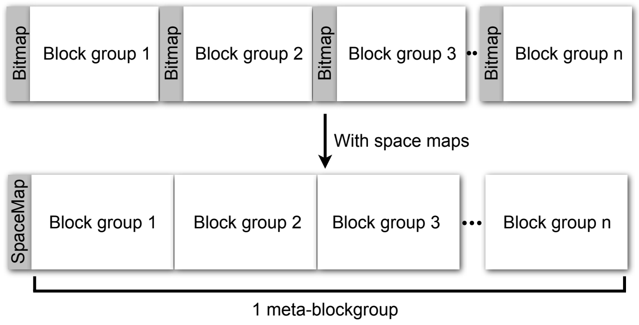


Figure 2: Metablockgroup

A space map consists of a red-black tree and a log. The nodes of the tree are extents of free space, sorted by offset and the log maintains records of recent allocations and frees. The tree and the log together provide the free space information.

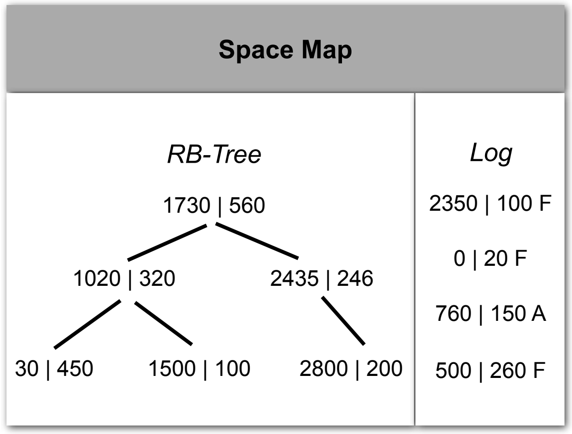


Figure 3: Structure of space maps

Bitmaps have a linear relationship with the size of the filesystem. This is not true with extent based techniques. Their size changes dynamically depending on the state of the filesystem. The tree has as many nodes as there are number of free space fragments (holes) in the metablockgroup. An experiment was conducted to get an idea of the space required by the space maps. *E2freefrag*[9] (a free space extent measuring utility) was executed on a 12GB Ext3 partition which was more than 90% full. Totalling the extents of various sizes, there were in all 1175 extents. Ext3 is good at avoiding fragmentation, but Ext4 is even better. Even then, as a safe cut-off mark, let us assume 2000 fragments in one metablockgroup i.e. 8GB, and calculate the space required for space maps. Here, one space map would require 2000 entries * 20 bytes per tree node entry = approximately only 40KB for the tree and let us keep aside 8KB for the log. Hence, space maps consume 48KB for 1 metablockgroup. This means that for an 8TB filesystem, where bitmaps would have required 256MB, space maps require merely 48MB i.e. only around 1/5th of the space required for bitmaps. This significant reduction in size enables space maps to reside entirely in memory at all times. To find free space, the allocator has to refer only the in memory structures and update them. This eliminates the huge I/O traffic from reading and writing bitmaps, which resulted from the fact that only a limited number could reside in memory at any given time.

The space maps are initialized at mkfs time. When the filesystem is mounted, each space map is read into memory. They persist in memory for the duration that the filesystem remains mounted. During this period they keep their interaction with on-disk structures to a minimum, such that filesystem integrity is maintained. On unmounting, space maps are flushed back to disk.

The detailed description of the tree and the log along with a reasoning of why they are chosen is given below.

2.1 In-memory structures

2.1.1 Red black tree

The red black tree[7] of the space map, as described above, consists of nodes which are extents of free space, sorted by offset. The red black property of the tree helps the tree adjust itself if it is skewing to any one side. This limits the height of the tree making searches efficient. The tree is the primary structure denoting the free spaces in a particular metablockgroup, while log is a secondary structure, temporarily noting recent operations.

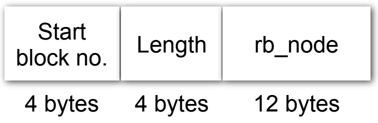


Figure 4: In-memory tree node

The tree node is 20 bytes in size. The start block number is relative to the first block of the metablockgroup to which the space map belongs. Hence, just 4 bytes suffice for the start block number and length fields.

2.1.2 Log

The log being an append-only structure, insertions to the log are very fast. Thus all operations are initially noted in the log, and then depending on their nature, they are either reflected in the tree instantly or in a delayed manner. The log assists the RB tree in maintaining a consistent state of the filesystem. Another reason for choosing the log is its ability to retain frees. In bitmaps, once the requested deallocation was performed, the freed space was forgotten. Due to this, an allocation following the free would be searched completely independent of the recently done free. This involved the tedious task of searching the bitmaps again, possibly from a completely different part of the platter. Moreover this also increased the chances of holes in the filesystem. In such a situation, the log acts as a scratchpad noting the recently done frees which can directly be used to satisfy the upcoming allocation requests, if any. Additionally, as allocations following frees fill up the recent frees from the log itself they help reduce holes in the filesystem. The working section along with an example will explain the behaviour of the log in much more detail.

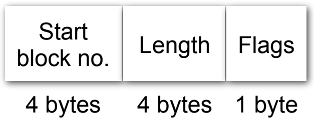


Figure 5: In-memory log entry

As each space map has a log, here too the start block number is relative to the start of the metablockgroup. The flags field is one byte with one of its bits denoting the type of the operation viz. allocation or free. Thus, a log entry is totally 9 bytes.

2.2 On-disk structures

2.2.1 Tree

For persistence across boots, the in-memory tree is stored on disk as an array of extents. At mount time, the extents from the on-disk tree are read to form the in-memory tree. The on-disk tree is updated only under two circumstances; when the filesystem is unmounting or when the on-disk log gets full.

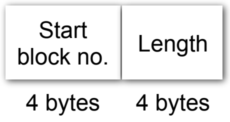


Figure 6: On-disk tree node

As shown in the preceeding figure, one on-disk tree entry consists of just one extent. Its size is 8 bytes.

2.2.2 Log

To avoid inconsistency in case of a crash, the operations noted in the in-memory log are also noted in the on-disk log in a transactional manner. As the log is an append-only structure only the last block of the on-disk log is required in memory. The exact operation is discussed in detail in the working of the technique.

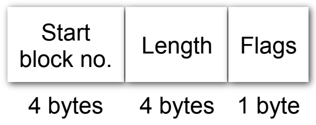


Figure 8: On-disk log entry

The on-disk log structure is same as that of the in-memory log.

3 Working

3.1 Allocation

The first flowchart outlines the allocation procedure.

3.2 Deallocation

The second flowchart explains the process of freeing space.

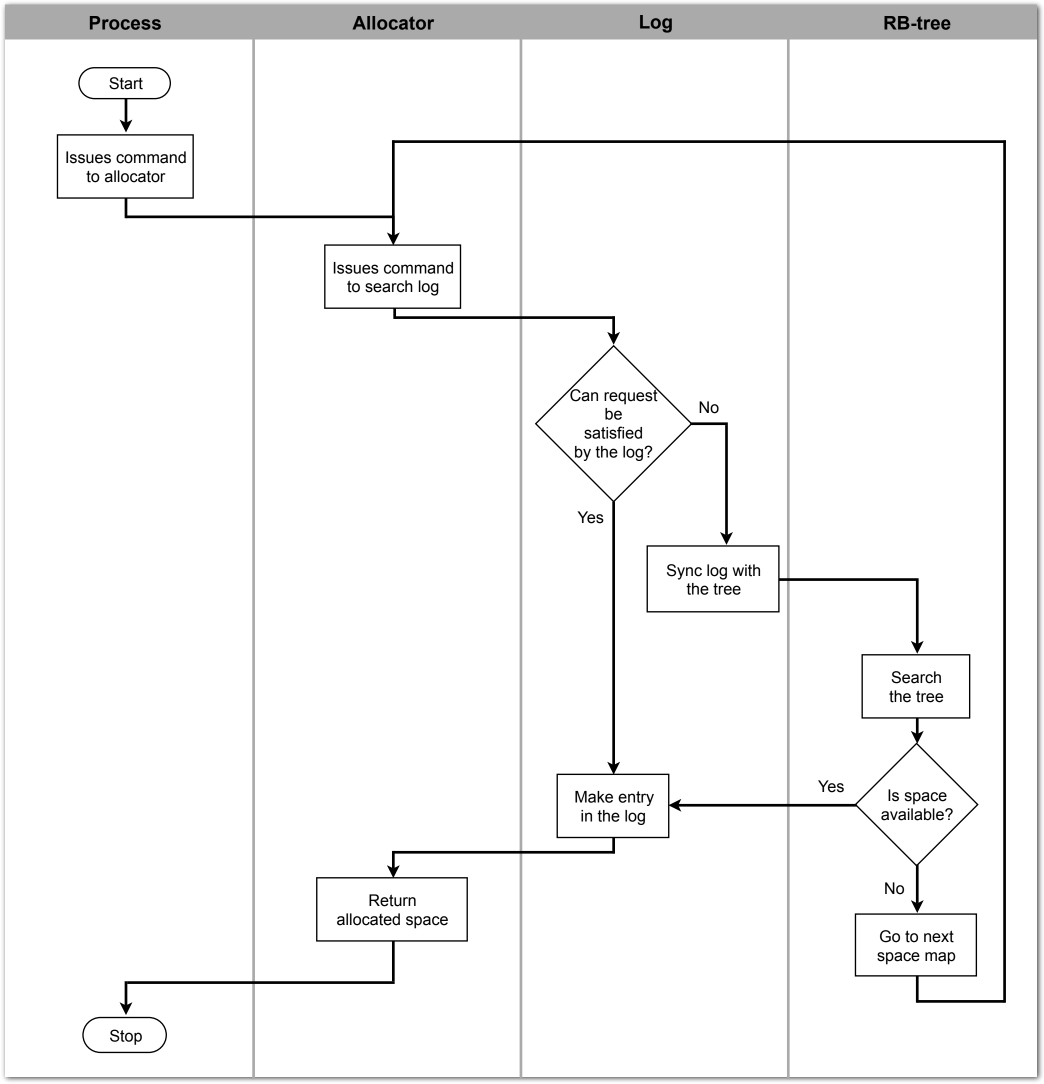


Figure 7: Allocation flowchart

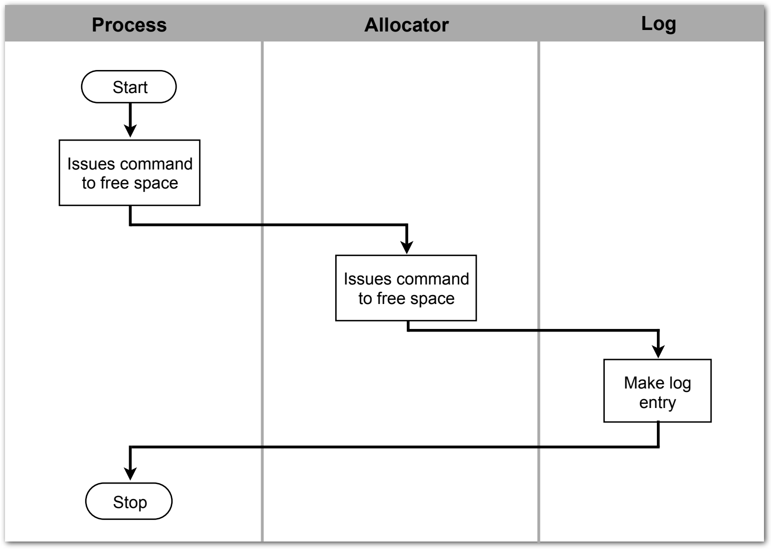


Figure 9: Free flowchart

The example below better illustrates the working of the system:

- Consider a newly made filesystem. As explained earlier, the log (left figure) is empty and the tree (right figure) consists of a single node. For the sake of this example, let us assume that a metablock-group consists of 5000 blocks. The single node of the tree indicates that the entire metablockgroup is free.

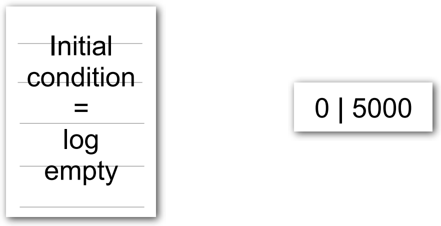


Figure 10: First scenario

- Suppose a process requests for 100 blocks. In this case, the allocator first searches the in-memory log of the goal metablockgroup for any recently done free operations that can satisfy the request. Since the log is empty, the tree is searched. As it is able to find a suitable extent, the process is allocated 100 blocks starting from, say, block number 0. Corresponding entries are made in the in-memory and on-disk logs in a single transaction. Note that neither logs are synced with the trees immediately.

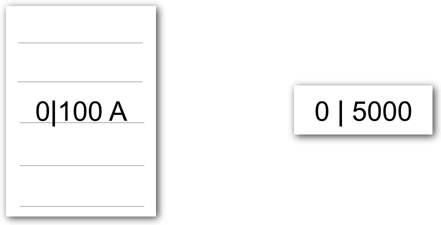


Figure 11: Second scenario

- Now, suppose there is a request for allocation of 150 blocks. As there is no entry in the log that can satisfy the request, the tree has to be searched. But since the tree does not reflect the updated state of the filesystem, we first need to sync the in-memory log with the in-memory tree. The in-memory log is then nullified. Here, the on-disk log is not synchronized with the on-disk tree as it would result in rewriting the entire on-disk tree. Writing the entire tree to disk every time any operation takes place would result in a lot of unnecessary writes to the filesystem. The beauty of this design is that as the on-disk structures are meant only to maintain space maps across boots, and are not referred for any allocation/deallocation, it suffices to just make a note of the operations somewhere on disk. The log serves this purpose. Hence, only the last block of the append-only on-disk log needs to be in memory at all times to which we append entries of operations. Assume that the allocator assigns 150 blocks starting from block number 450. Entries in the logs are made accordingly. Even if the in-memory structures and on-disk structures are different, the in-memory log + in-memory tree = on-disk log + on-disk tree; thus maintaining consistency. If there is a crash at this point, replaying the on-disk log onto the on-disk tree will give us the exact state of the filesystem as it was before the crash.

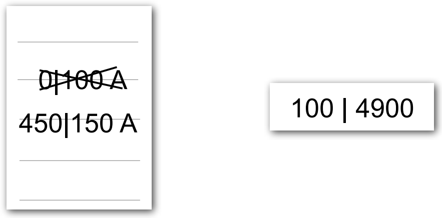


Figure 12: Third scenario

- Another allocation request of 200 blocks is tackled in a similar fashion.

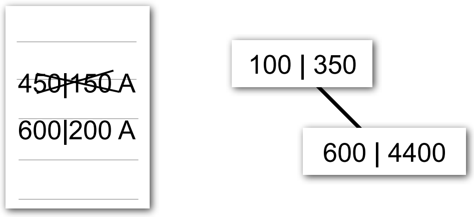


Figure 13: Fourth scenario

- Suppose now there is a request to free 150 blocks starting from block number 650. For a free request, ideally, there should be no need to search anything at all. All that is required is that the free space manager is informed about the blocks that are freed. Here, the bitmap technique faces a problem. In bitmaps, the specific bits of the particular bitmap block (in whose block group the file resided) are to be nullified. This causes a lot of seeks. This is where the log plays its most important role. The fastest way of informing space maps about a free is simply appending an entry to the logs. That is exactly what happens in space maps. So here, only the logs are updated with the entry suggesting that 150 blocks from block number 650 are freed. This maintains perfect locality of appends and results in very fast, virtually seekless operations. Furthermore, the deletion of a large, sparse or fragmented file requires many bitmaps to reside in memory. As against this, in space maps only the last block of the on-disk log (to which the appends are to be made) needs to be in memory. Hence, for an 8GB metablockgroup, where in the worst case (i.e. if the file/directory being deleted had occupied blocks in all 64 block groups comprising that metablockgroup) the bitmap technique would have required fetching all 64 bitmap blocks in memory, space maps require only 1 block (viz. the last block of the on-disk log) in memory. This not only reduces the memory consumption but also speeds up deallocation process.

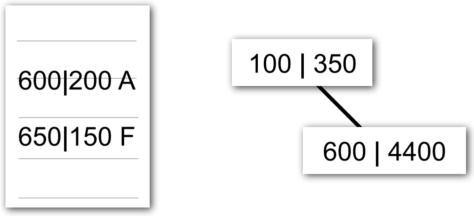


Figure 14: Fifth scenario

- Suppose there is another request for 150 blocks. In this case, the in-memory log does have a recent free which can satisfy the request. The 150 blocks are allocated from the log itself. This not only prevents another sync with the tree and a scan of the whole tree for an appropriate chunk, but also fills up a hole, thereby reducing potential free space fragmentation. The two entries for allocation and free are purged in the log itself.

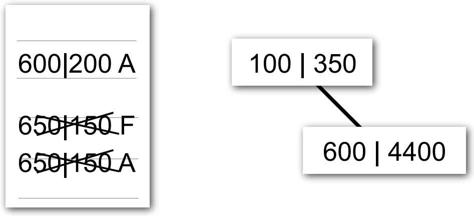


Figure 15: Sixth scenario

All further allocations and deallocations continue to happen in the above-mentioned way.

Since logs are append-only, they can keep on filling indefinitely. The log sizes cannot be kept infinite and thus the design has to incorporate the handling of logs getting filled up completely. Consider the following two scenarios:

- *In-memory log gets full.* In this case, before the next entry to the log is made, the log is synchronized with the in-memory tree and nullified. Thus, the filesystem is still consistent.

- *On-disk log gets full.* In this case, the in-memory log is first synchronized with the in-memory tree. The in-memory tree showing the then most up-to-date condition of the filesystem overwrites the on-disk tree. After this, both the logs are nullified.

4 Evaluation

Space maps were put to test using some standard benchmarks. In the tests below, Ext4 (as of kernel 2.6.33.2) is compared with Ext4 (of the same kernel version) with space maps implemented. To really stress the allocator, the tests were conducted on a 50GB partition with memory size as 384MB. Also the filesystem was made with 1K block size to increase the number of bitmaps to simulate the behaviour of large filesystems.

4.1 Small file handling using postmark

Postmark[10] is a tool that simulates the behaviour of a web server, typically a mail server. Thus its tests involves creation and deletion of a large number of small files. In the test below, postmark was run 5 times. Each time 100000 files were added to the previous test, starting with 100000 files.

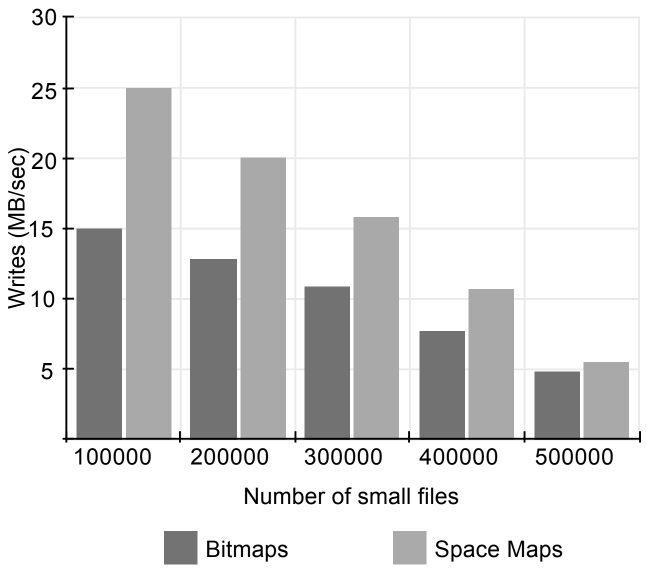


Figure 16: Graph of postmark

As predicted, better extents result in faster file writes. There is a stark difference in the speed of allocation initially between space maps and bitmaps, but the difference gradually reduces as the number of files goes up. Even then, at all times space maps allocation speed is higher than that of bitmaps.

4.2 Simultaneous small file and large file creation

Mballoc has the ability to treat large and small files differently. In order to stress mballoc, a test was conducted in which 5 linux kernels were untarred in different directories and 5 movie files (typically in the range of 700MB) were copied simultaneously. Operations like these jumble the allocator with large file and small file requests at the same time. In such scenarios, mballoc tends to make a lot of seeks. This is evident from the results below. The following statistics were taken when performing the tests on a newly made filesystem. The tool used below to measure seek count is called *seekwatcher*[8] by Chris Mason. It uses the output of the *blktrace*[11] utility and constructs a graph with time on the x-axis. It uses matplotlib to build the graphs.

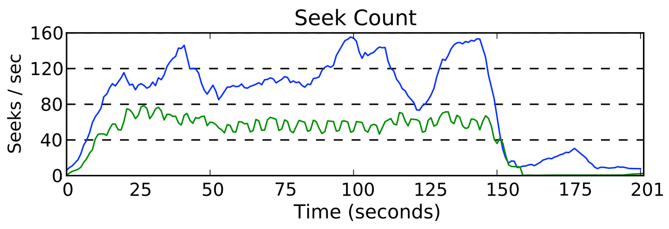


Figure 17: Stress test seek comparison run 1

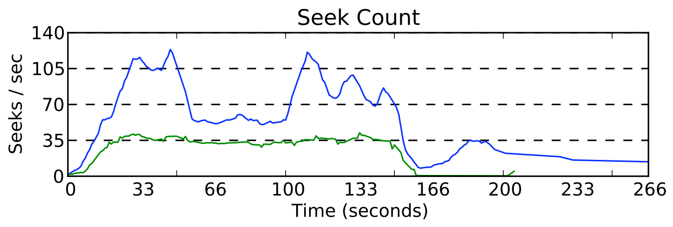


Figure 18: Stress test seek comparison run 2

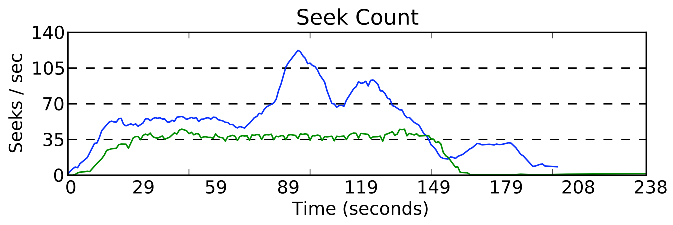


Figure 19: Stress test seek comparison run 3

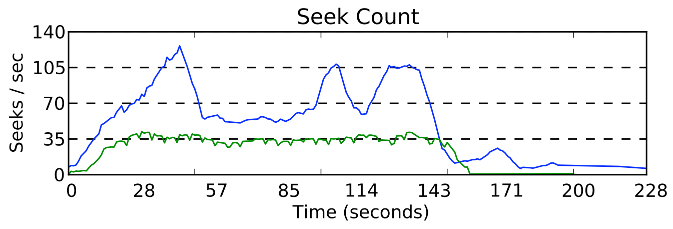


Figure 20: Stress test seek comparison run 4

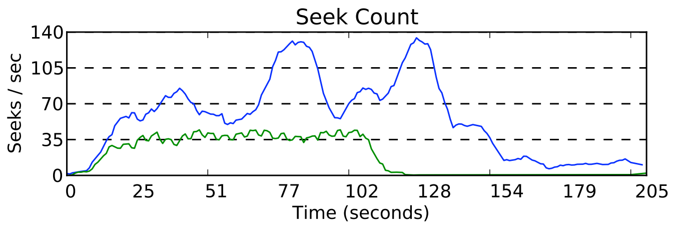


Figure 21: Stress test seek comparison run 5

The above test was conducted 5 times consequtively. As clearly visible in all the operations the seek count when allocation was done using space maps is less than half of the seeks required by the Ext4 that used bitmaps.

4.3 Free space fragmentation using e2freefrag

The following test measures the number and size of the fragments of free space in the filesystem. The test is just an extension to the previously performed simultaneous large and small file creation executed a total of 7 times. Fragmentation was measured at the end of each iteration. In Ext4 with bitmaps, we measure this attribute with *e2freefrag*[9], whereas in Ext4 with space maps, extents were nothing but the nodes of the trees. As clear below, the number of free space fragments of the filesystem go on reducing as the filesystem fills up. This is because the nodes of the tree get filled very efficiently resulting in lesser nodes, and thus lesser fragments.

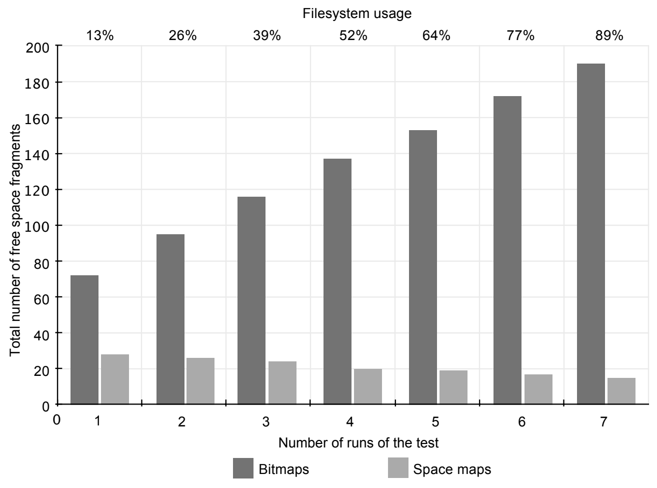


Figure 22: Graph of number of free space fragments

After having seen the number of fragments, let us have a look at the nature of the fragments in terms of their size. In general, the larger the extents of free space, the more chances of a future file residing contiguously on disk. The results show that even when the filesystem is 89% full, the extents of free space greater than 1GB are around 74%, whereas in bitmaps it falls down drastically to 36%. This confirms that the more extent oriented information is available, the more efficiently allocations can be carried out with minimum free space fragmentation.

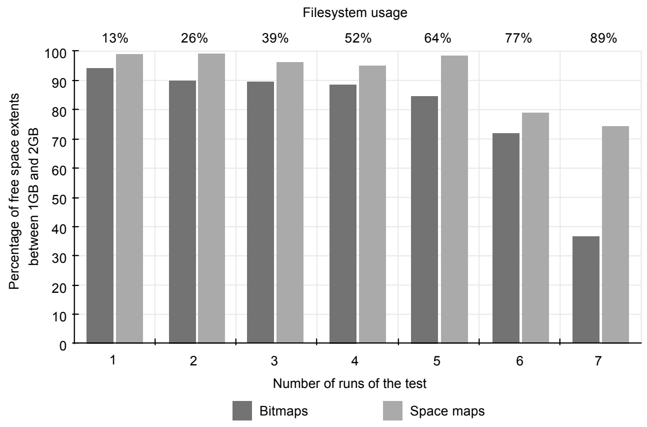


Figure 23: Graph of e2freefrag

4.4 File fragmentation using filefrag[12]

As stated earlier, the allocator tends to give better and more contiguous space to files if it recieves better extents. The graph below completely supports the claim. To measure the effects of file fragmentation, a test was conducted which involved copying of large 1GB files to a 10GB partition. The partition was made using the default flexible block group parameter of 16 block groups. Even then, the average number of fragments shown by files allocated using space maps are 1/5th of those allocated using bitmaps.

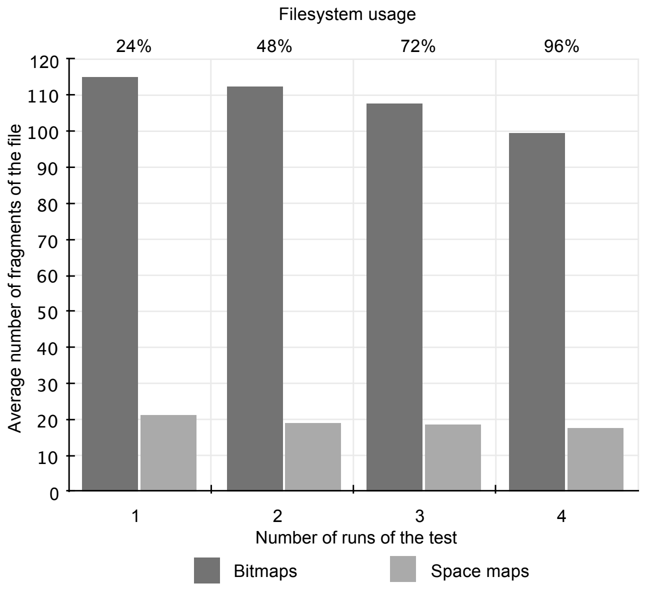


Figure 24: Graph of filefrag

5 Other benefits

- During mkfs, until the new uninitialized block groups feature was incorporated, all the bitmaps had to be invalidated to indicate that the entire filesystem was free. With uninitialized block groups you can now just set a field in the group descriptors indicating that the bitmap for this block group is invalid[2]. The actual bitmap block is nullified only when it is about to be used. This has significantly sped up the process of formatting Ext4. With space maps, the mkfs process involves just the entry of 1 node per metablockgroup indicating that the entire metablockgroup is free. This is as fast as the uninitilized block groups feature if not faster, as there are lesser number of metablockgroups than the group descriptors in a filesystem. Along with that, this completely takes care of marking the entire metablockgroup free as against just the invalidation in the group descriptor. So there is no extra operation required before using the particular metablockgroup for the first time and it is completely ready for usage.

- If the filesystem is made with 1K block size, then there are 16 times more bitmaps than the same filesystem made with the default 4K block size. Even in this case, as the space maps parameter for size of metablockgroup is tunable, the number and size of space maps can remain the same.

- Ext4 has done remarkably well to avoid fragmentation mainly due to the use of persistent preallocation. Even though this is the case, when the inode preallocation runs out of preallocated space, it may have to place a part of the file at a different offset. While doing this, fast access to flexible extents (extents not limited by block group size and/or not just as powers of 2) of free space eases the job of the allocator and results in lesser file fragments.

- Intelligent allocator heuristics will result in the reduction of the size of the tree as the filesystem goes on filling up. This will in-turn increase the allocation speed as tree lookups will be faster.

6 Limitations

- Every time the filesystem is mounted, the on-disk trees are read and the RB tree is constructed. This is time consuming as compared with the current bitmap technique as nothing has to be initialized with respect to bitmaps. Also while umounting, the trees have to again be traversed and stored onto disk in the form of a list of extents. That too is more time consuming as compared with the current scenario.

- In cases of extreme fragmentation (say every alternate disk block is empty) the memory consumption due to space maps will be higher than that of bitmaps.

7 Future enhancements

- One of the further optimizations could be the intelligent separation of the space maps based on file sizes. If we have separate space maps for large and small files, then the log entries in those space maps (for frees) will be of similar nature. Thus more allocation requests can be satified by the log itself without having to rely on the tree. This will result in maximum utilization of the log design.

- Another enhancement can be in designing a more efficient log. Currently, the log is simply an array of extents. Advanced data structures can enable much faster lookups of the log, resulting in even faster allocations/deallocations.

8 Related work

8.1 Space maps in ZFS

The concept of Space Maps is not new. ZFS, a solaris filesystem, also has the idea of space maps but the mechanism of implementation varies. Each virtual device on ZFS is divided into metaslabs, each having its own space map. Metaslab of ZFS is analogous to the metablock-group of Ext4. However, in case of ZFS, space map is simply a log of allocations and frees as they happen. Due to the use of the log, ZFS also benefits from the perfect locality of appends. Appends are made for allocations as well as frees.

Whenever the allocator wants to search for free space in a particular metaslab, it reads its space map and replays the allocations and frees into an in-memory AVL tree of free space, sorted by offset. At this time, it also purges any allocation-free pairs that cancel out. The on-disk space map is then overwritten with this smaller, in-memory version[6].

8.2 Comparison with space maps in Ext4

- In ZFS, space map is nothing but an on-disk log of allocations and frees. Also, an AVL tree is used which is the only in-memory structure. As against this, the Ext4 implementation has an RB tree along with an in-memory log helping reduce fragmentation and speed-up deallocations.

- Another difference is that the ZFS allocator has to update the tree for each and every request whether it is an allocation or a free. There can be cases when the entire tree needs to be reshuffled often. In a case where there are allocations following several frees and if the allocations can be satisfied by the recent frees, the Ext4 space maps can answer the request from the in-memory log itself instead of having to sync the tree time and again.

9 Conclusion

Space Maps demonstrate all the qualities essential for supporting fast free space allocation and deallocation in large filesystems common today. Finding free space can now be done entirely in memory and requires very little involvement of the disk. Space maps provide great scalability and are proved to maintain filesystem consistency. We believe that improvements in space maps will further bring in optimizations and lift the current performance of this technique even higher.

10 Acknowledgements

We wish to thank the following people for their contributions; Ms. Pavneet Kaur, who shared her thoughts during the development of space maps. Mr. Anuj Kolekar who assisted during the testing phase. Mr. Kalpak Shah, our mentor whose ideas and timely criticism helped us bring this to fruition. Mr. Jeff Bonwick, who laid the seed of the idea. Last, but not the least, we wish to thank the anonymous reviewers, who patiently read our paper and gave us valuable inputs which helped make significant improvements in this paper.

References

[1] Mathur A., Cao M., Bhattacharya S., Dilger A., Tomas A and Viver L., *The New ext4 filesystem: current status and future plans.* (2007) [Online] Available: `http://ols.108.redhat.com/2007/Reprints/mathur-Reprint.pdf`

[2] Avantika Mathur, Mingming Cao and Andreas Dilger, *ext4: the next generation of the ext3 file system* (2007) [Online] Available: `http://www.usenix.org/publications/login/2007-06/openpdfs/mathur.pdf`

[3] Aneesh Kumar K.V, Mingming Cao, Jose R Santos and Andreas Dilger, *Ext4 block and inode allocator improvements* (2008) [Online] Available: `http://www.linuxsymposium.org/archives/OLS/Reprints-2008/kumar-reprint.pdf`

[4] *Ext4* (2010) [Online] Available: `http://kernelnewbies.org/Ext4`

[5] Mingming Cao, et.al. *State of the Art: Where we are with the Ext3 filesystem* (2005) [Online] Available: `http://www.linuxsymposium.org/2005/linuxsymposium_procv1.pdf`

[6] Jeff Bonwick, *Space Maps* (2007) [Online] Available: `http://blogs.sun.com/bonwick/entry/space_maps`

[7] Rob Landley, *Red-black tree* Available: Linux kernel documentation

[8] Chris Mason, *Seekwatcher* [Online] Available: `http://oss.oracle.com/~mason/seekwatcher`

[9] Rupesh Thakare, Andreas Dilger, Kalpak Shah, *e2freefrag* [Online] Available: `http://manpages.ubuntu.com/manpages/lucid/en/man8/e2freefrag.8.html`

[10] Jeffrey Katcher, *PostMark: A New File System Benchmark* [Online] Available: `http://communities.netapp.com/servlet/JiveServlet/download/2609-1551/Katcher97-postmark-netapp-tr3022.pdf`

[11] Jens Axboe, Alan D. Brunelle and Nathan Scott, *blktrace User Guide* [Online] Available: `http://pdfedit.petricek.net/bt/file_download.php?file_id=17&type=bug`

[12] Theodore Tso, *filefrag* Available: Linux kernel documentation

Mobile Simplified Security Framework

Dmitry Kasatkin
Nokia Corporation
`dmitry.kasatkin@nokia.com`

Abstract

Linux kernel has already several security frameworks such SELinux, AppArmor, Tomoyo and Smack. After some studies we found out that they are not very suitable for mobile consumer devices such as mobile phones. They either require too complicated administration or do not really provide any security API, which can be used by applications providing services to verify credentials of their clients, and then decide if a particular client can access the provided service or not.

In this paper we present a new platform security framework developed by the Maemo security team specifically for mobile devices. The key subsystem of the Mobile Simplified Security Framework is the Access Control framework, which is used to bind privileges (resource tokens) to the application when the application is starting. Using a special API, different entities are able to verify possession of those resource tokens and allow/disallow access to protected resources. If any of the applications require an access to protected resources, a Manifest file with the credential request should be included in the package providing the application. The Manifest file is also used to declare new credentials, which are provided by an application coming from the package.

1 Introduction

The scope of this paper is to describe Mobile Simplified Security Framework (MSSF). This security framework has been developed specifically to be suitable for mobile consumer electronics devices such as smartphones.

Smartphones usually have limited resources such as memory, CPU power, and power supply, but at the same time have a network connectivity and allows the user to download and install applications. Malicious applications can pose a threat to the security of the system.

Platform security mechanisms must provide following:

- Protect the user.
  It must not be possible for malicious application to corrupt or steal the device owner's personal data. Also malicious application must not be able to misuse the device and incur costs by sending sms to pay numbers. If device is stolen, it should not be possible to access the user's private data.

- Protect the device.
  Device functionality and reliability must satisfy specification requirements. It must not be possible to change critical device parameters. Changing RF, WiFi values can cause device malfunctioning and violate regulatory requirements.

- Protect the business.
  Phones are sold via different channels. Operators often subsidize devices. Breaking a SIM/Subsidy lock immediately mean lose of business. Operators want product customization - certain applications and service should only be available on its devices, and possibly limit what can be installed on the device.

- Enable new services.
  Providing services such as Music Store or Application Store requires device to support copy-protection. Services like mobile payments and billing requires secure handling of customer data.

Comparing to personal computers where users have more control over the device and often prepared to perform administrative tasks, mobile device users do not expect that they need to make complicated configuration of the device. That requires security framework to provide protection without additional maintenance effort and to allow applications coming from different sources to get access to protected resource in controlled manner.

2 Existing Linux Security Frameworks

Linux kernel has already several security frameworks such as SELinux, Smack, Tomoyo. While providing Mandatory Access Control (MAC) implementation, they do not provide end-to-end solution for security, taking into account also software distribution and developers ecosystem. Here we provide some reasons why we decided to develop our own security framework.

2.1 Traditional Unix DAC

Unix DAC is a classical access control model which is based on restricting access to objects based on identity of the subjects and groups to which they belong. The main difficulty to use Unix DAC is a lack of process based access control.

Mobile device is normally a single user device where all processes are either running under root account or the same user account. Processes with the same user ID have unlimited access to the resources of each other. This pose a threat that malicious application can corrupt or steal the data of other applications.

The way to work around this issue is to run processes under different users accounts and groups. But this approach is not generic and requires administrative work to maintain needed information.

2.2 SELinux

SELinux is LSM-based MAC implementation [1]. SELinux is very powerful, but requires very complex and centralized policy administration. That is not problem for the servers which are usually centrally administered by professional people. Required administration effort makes it very complicated to use in smartphone platform.

2.3 Smack

Simplified Mandatory Access Control Kernel (SMACK) is LSM-based, relatively simple MAC implementation as alternative to SELinux [2]. It’s operational logic is simple: labels are attached to system components and access rules between the labels are defined by the system administrator. SMACK provides primitive security API, but doesn’t provide the application enough granularity to provide detailed access control.

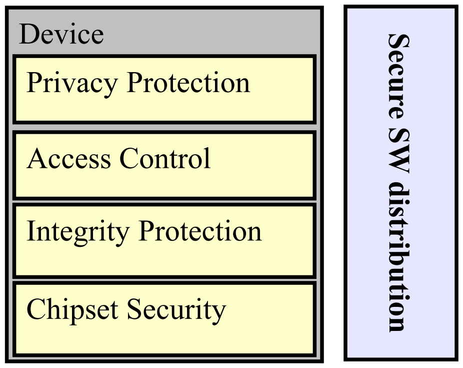


Figure 1: Security Frameworks Layers

2.4 Tomoyo

Tomoyo is a lightweight MAC implementation [3] . It performs pathname-based access control. TOMOYO utilizes "process invocation history" and requires administrative actions on the target system.

3 Mobile Simplified Security Framework

Mobile simplified security framework (MSSF) is a set of mechanisms to protect entire platform consisting of following layers (Figure 1):

- Chipset security.
  Provides secure cryptographic services for OS level security.
- Integrity protection.
  Ensure protection of TCB, applications and data. Provides protection against offline attacks.
- Access Control.
  Limits application access to critical resources. Provides protection against runtime attacks.
- Application privacy protection. Provides integrity and confidentiality protection for applications and services. Provides protection against offline attacks.

Security Framework relies on the secure software distribution model. The goal of Secure SW distribution is to ensure the authentication of a SW’s source.

4 Chipset Security

4.1 Features

Chipset security is the key subsystem on which whole security framework relies on. It provides tamper-resistant securure services and serves the same purpose as Trusted Platform Module (TPM) [4] or Mobile Trusted Module (MTM) [5]. It includes:

- Root symmetric devices specific key.
  The root symmetric device specific key is unique one-time programmable (OTP) key, which is used to derive keys used for local cryptography operations. It is also used to derive a unique public identifier of the device.
- Root public key.
  The root public key is OTP key and is used to verify that software components are coming from trusted source.
- Provides trusted boot (chain of trust).
  Root public key is used to verify integrity of the bootloader and SW image.
- Secure services.
  Secure key management and cryptography services.
- Provides Secure Execution Environment (SEE).
  SEE consists of secure ROM and RAM which is isolated from reset of the system. It allows execution of integrity protected applications, which can utilize secret device keys and provide specific secure services for the OS. Protected applications are needed by Protected Storage and DRM framework.

4.2 Operation modes

Nokia MeeGo 1.0 N device will have two operation modes: normal and open mode.

Devices shipped by Nokia come with original Nokia SW having device configured for normal mode Security functionality such as access control and integrity protection are enabled and enforced. Applications and services are able to use device keys and cryptographic services. In normal mode authorized applications are given access to copy protected content. Unauthorized modification of the security policy is impossible.

Developers who want to have unrestricted access to the platform resource, might turn the device into open mode by flashing an ”open mode” image.

Open mode is mostly needed only for low-level development and deep device customization. Ordinary application developers test their applications with normal mode as well. Open mode provides the same functionality as in normal mode except that there is no access to copy protected content. In open mode image security is enforced but allows developers to modify the policy and to allow their applications to access more device resources without the need of application certification process. In open mode it is possible to use own kernel. However in open mode, chipset security generates different keys which are incompatible with normal mode keys. This makes it impossible to get an access to copy protected content.

4.3 Boot process

Boot process is shown on figure 2.

Bootloader image is verified with Root Public key.

If bootloader verification fails, device is automatically resetted.

In the case when kernel verification fails, device either restarted or booted to open mode. If device is SIM locked, it is not allowed to boot into open mode unless explicitly allowed by the device customization.

5 Integrity Protection

MSSF has an integrity protection subsystem called Validator, which protects the integrity of kernel modules, executables and libraries and data files. The primary goal is to protect integrity of SW components which belongs to Trusted Computing Base (TCB). Trusted Computing Base includes all hardware, firmware and software components that are critical to the security of the entire platform.

The integrity subsystem is shown on Figure 3.

Validator is implemented as LSM kernel module and is based on the DigSig project [7]. The difference is that

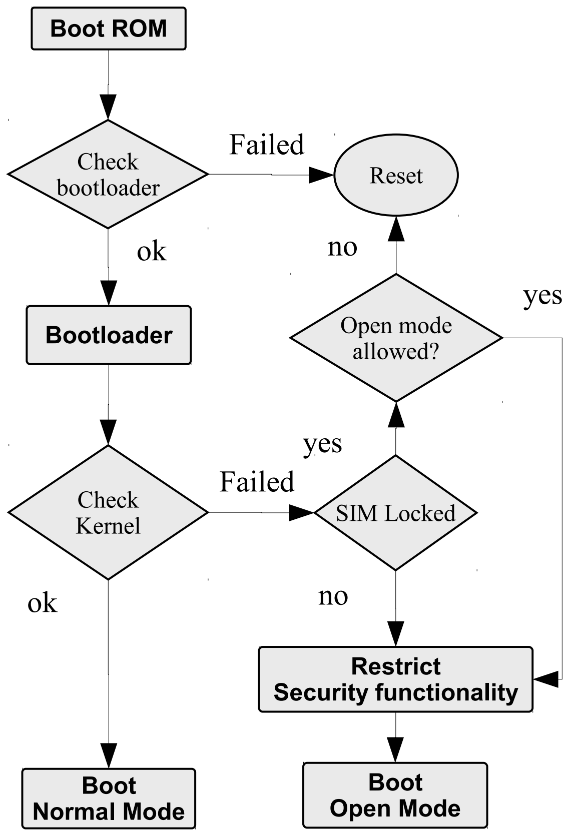


Figure 2: Boot process

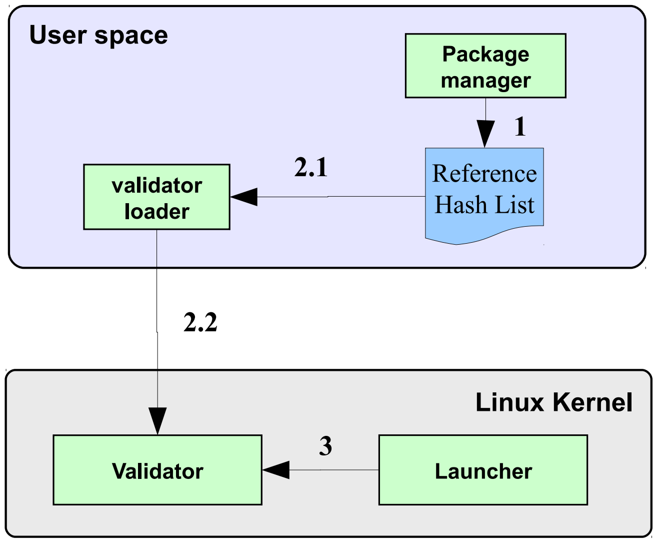


Figure 3: Integrity Protection Subsystem

instead of using ELF header, device maintains a reference hash list (`/var/lib/mssf/refhashlist`) of all protected binaries. Integrity of the reference hash list is also protected by the device signature.

Reference hash list includes SHA1 hash of the file, file attributes, and AC related data.

Debian package contains sha1 hashes of all executables and important data files. Package manager updates reference hash list upon package installation, removal or upgrade.

Use of single reference hash list instead of ELF signed binaries and EA has certain advantages. It allows to have protection for scripts and data files, which do not have ELF header. It does not require to verify integrity of EA itself for every file using digital signatures. Integrity verification of reference hash list is more mobile device friendly. Also EA is a subject to offline removal attack, which cannot be detected.

Installation includes following steps:

1. Package Manager installs new binaries and updates reference hash list.
2. Validator loader loads new or updated hash list into the kernel.
3. Validator calculates and compares hash and file attributes upon execve() call. It also verifies hashes of shared libraries upon mmap() call.

Integrity protection policy defines action when integrity verification fails. Currently it only blocks the execution.

Validator also has a support for integrity protection of non-modifiable data files. That is used for protection of critical configuration files.

Source code of this module is located in:
linux/security/mssf/validator

6 Access Control

6.1 Introduction

Access Control framework provides runtime protection.

We had following design goals for access control:

- Process-based access control to protected resources.
- Minimal changes to the default Linux model.
- No need for centralized security policy administration.

Access control framework includes following components:

- Manifest file.
  Manifest file is included to the package and contains a list of executables and its credentials.
- Device Security Policy.
  Located on the device and defines repository trust level and credentials, which can be granted to packages coming from that repository.
- Credentials Policy.
  It is a file which contains mapping of credentials to executables. Package Manager updates this file when packages are installed, upgraded or removed.
- Package Manager.
  In addition to installing the application, Package Manager updates Credentials Policy database.
- Credentials Policy loader.
  It is called during boot to read and import credentials policy into the kernel.
- Credentials Manager.
  Provides credentials management and assignment to the process. It is implemented as a set of kernel modules (see Implementation Details).

6.2 Credentials

Access control in Linux is based on credentials. Conventional credentials in Linux are UIDs, GIDs and POSIX capabilities.

MSSF access control framework uses term protected resource to denote any virtual object which represents some functionality or data, such as tasks, files, sockets, devices.

MSSF access control framework extends credential set with **resource tokens** and **application identifier**.

6.2.1 Resource Tokens

Each protected resource is assigned a resource token, which is just a string representation of the resource, for example Cellular, UserData. Other security frameworks often use the term ”label”.

Applications or services need to declare requested or provided resources. For that purpose, package, which hosts those executables, must include the Manifest file.

Resource tokens can either be global or package specific. Global tokens comes from special package. Package specific tokens are declared as pkgname::token.

Access Control model does not define subject labels, object labels and access control rules, such as ’subjectlabel objectlabel access’, but resource tokens play the role of both subject and object labels. Resource tokens are assigned to the process subjective context upon startup. Enforcement mechanism just needs check if a process possess a token.

Rules are not enforced by MSSF access control model automatically, but processes need to perform enforcement manually when task is being accessed by retrieving client tokens. Currently supported access method is via Unix domain sockets.

API is provided to get credentials of the process.

6.2.2 Application identifier

Application identifier is used to derive application specific keys (see Privacy Protection). Application identifier is defined as:

AppID =
{SourceID, Package Name, Application Name}

for example {ovi.com, CoolTools, AddressBookPlugIn}.

Application name is defined in manifest file.

Application identifier has following properties:

- Unforgeable & Trustworthy.
  SourceID is defined in Device Security Policy and protected by repository keys.

- Unique.
  System can only have one package with the same name.
- Persistant.
  It remains the same between reboots, application updates, and for different instances of the same application.

6.3 Device Security Policy

Software packages are distributed via Debian-like repositories. Repository contains a package list, which is signed with repository private key and verified by the package manager using repository public key. Package based signing using PGP and X.509 is also supported.

The purpose of the Device Security Policy is to define repository trust level and credentials, which can be granted to packages coming from that repository.

Device Security policy contains entries which have following format:
{SourceID : Trust Level : Public Key : Allowed credentials}, where:

SourceID is a meaningful name for the origins of the repository, for example in a form of domain name.

Trust Level is an ordinal number and defines repository ranking. During update, package can only be updated from repository which has the same or higher trust level. It will prevent possibility for some 3rd party repositories by mistake or on purpose to replace trusted package with untrusted one.

Allowed credentials is a list of credentials, which can be granted by this repository.

Public Key is a repository public key which is used to verify repository package list.

Example of policy entry can look like:
{nokia.com : 1 : ABCDEF : UserData, Cellular}.

Package Manager uses device security policy when packages are installed or upgraded. Granted credentials, which is added to the Credentials Policy, are the result of 'intersection' operation over credentials set from Manifest file and security policy. Only allowed credentials are added to the Credentials Policy.

Figure 4 shows the concept of distribution model.

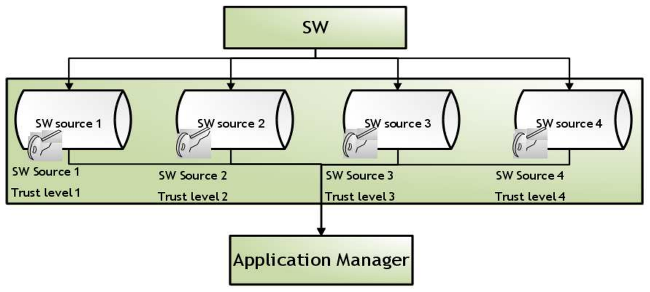


Figure 4: SW Distribution Model

6.4 Manifest File

If an application requests or provides some credentials, the package is expected to ship with the Manifest file `<package>.mssf` with description of credentials.

Package manager updates Credentials Policy based on the manifest file and constraints from the device security policy as described in the previous section.

Manifest file is written in XML and defines following tags:

- **<request>**
  requested credentials
- **<provide>**
  provided credentials
- **<credential name="credential name">**
  credential name
- **<for path="path">**
  absolute path to the program executable
- **<dbus name="dbus service name">**
  D-bus service name
- **<bus="bus type">**
  D-bus type (system or session)
- **<own="credential name">**
  Credential to bind to a specific d-bus service name
- **<interface name="interface name">**
  D-Bus interface name

6.4.1 Manifest file for client-server example

In the example bellow, server defines resource token **UserData**, which is needed by the client to access the server.

```
<mssf>
  <provide>
    <credential name="UserData"/>
  </provide>
</mssf>
```

In the example bellow client declares that it requires a token **UserData** and **Cellular**

```
<mssf>
  <request>
    <credential name="UserData"/>
    <credential name="Cellular"/>
    <for path="/usr/bin/userdatamanager">
    <for path="/usr/bin/userdataclient">
  </request>
</mssf>
```

In this example both applications **userdatamanager** and **userdataclient** will get the same credentials.

6.4.2 Manifest file for traditional credentials

Manifest file can be used also to assign conventional credentials such as UID, GID and POSIX capabilities.

```
<mssf>
  <request>
    <credential name="UID::email"/>
    <credential name="GID::email"/>
    <credential name="CAP::cap_sys_rawio"/>
    <for path="/usr/bin/mssf-dbus-server"/>
  </request>
</mssf>
```

6.4.3 Manifest file for policy update

Manifest file is also used to update device security policy. Policy update is done via the special authorized package.

```
<mssf>
  <domain name="MyDomain" rank="30">
    <allow>
      <credential match="*"/>
      <deny>
          <credential name="drm"/>
      </deny>
    </allow>
    <origin>
      <keyinfo>
        mQGiBE...O6XB
     </keyinfo>
    </origin>
  </domain>
</mssf>
```

6.4.4 Manifest file for DBUS

We implemented a DBUS extension which uses credentials API to verify client credentials. Manifest file may have dbus specific tags, which are used by the Package manager to generate DBUS policy.

Manifest file for DBUS-server is shown on Figure 5.

Generated DBUS policy file is shown on Figure 6.

DBUS client uses the same manifest file as with peer-to-peer access control (Figure 7).

6.5 Package Installation

Installation of new applications and services is done via packages (Figure 8).

Package installation includes following steps:

1. Package arrives to the Package Manager together with Manifest file.
2. Package Manager checks the Device Security policy for the information.
3. Package Manager updates the Credentials Policy according to the ”Intersection rule”.
4. Package Manager possibly updates D-Bus policy.
5. Package Manager updates runtime credentials policy in the kernel.

6.6 Startup

Startup process is shown on Figure 9.

1. At a boot, Credentials Policy loader reads Credentials Policy and loads it into the kernel.

```
<mssf>
<provide>
     <credential name="access"/>
     <dbus name="com.meego.mssf.example" own="mssf-dbus-server" bus="session">
         <node name="/">
             <interface name="mssf.Example">
                  <annotation name="com.meego.secure.Access" value="access"/>
             </interface>
       </node>
     </dbus>
</provide>
<request>
     <for path="/usr/bin/mssf-dbus-server"/>
</request>
</mssf>
```

Figure 5: DBUS server manifest

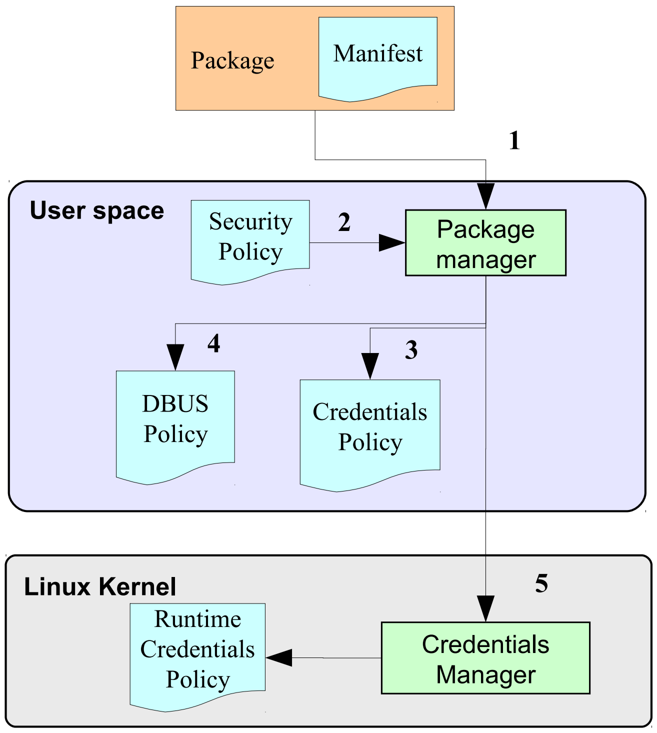


Figure 8: Package instllation

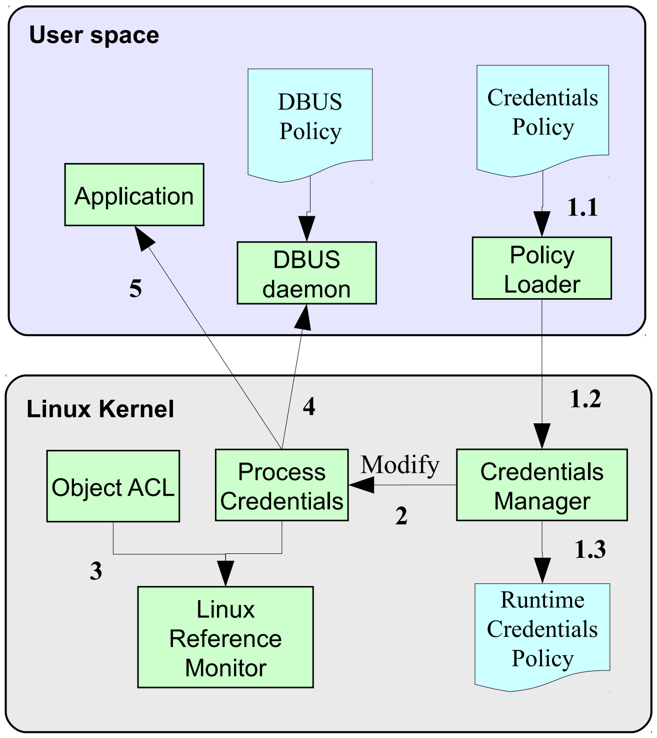


Figure 9: Startup

```
<busconfig>
  <policy context="default">
    <deny own="com.meego.mssf.example"/>
  </policy>
  <policy creds="mssf-dbus-server::mssf-dbus-server">
    <allow own="com.meego.mssf.example"/>
  </policy>
  <policy context="default">
    <deny send_destination="com.meego.mssf.example" send_interface="mssf.Example"/>
    <deny receive_sender="com.meego.mssf.example" receive_interface="mssf.Example"/>
  </policy>
  <policy creds="mssf-dbus-server::access">
    <allow send_destination="com.meego.mssf.example" send_interface="mssf.Example"/>
    <allow receive_sender="com.meego.mssf.example" receive_interface="mssf.Example"/>
  </policy>
</busconfig>
```

Figure 6: DBUS policy

```
<mssf>
  <request>
     <credential name="mssf-dbus-server::access"/>
     <for path="/usr/bin/mssf-dbus-client"/>
   </request>
</mssf>
```

Figure 7: DBUS client manifest

2. Upon application startup, Policy Manager modifies process' credentials according to the received credentials.

3. File AC.
   Validator checks process credentials using kernel API.

4. D-Bus.
   D-Bus daemon checks client credentials using libcreds (see DBUS Integration).

5. Client-server.
   Application checks client credentials using libcreds.

6.7 Credentials APIs

When a client issues a request to a server, the server may wish to check whether the client is authorized for the requested operation. It is done using the **libcreds** library, which gives the server a way to read the credentials of the client process and to permform the desired credential checks.

Kernel credentials API is also available.

Example of using API is shown on Figure 10.

creds_str2creds() converts token string to internal format.

creds_getpeer() retrieves credentials of the client process.

creds_have_p() checks if the client process has required credential.

6.8 DBUS support

6.9 File System Access Control

Debian packages often contain installation scripts which runs under the root. It allows them to modify any files

```
creds_value_t value;
creds_type_t type;
require_type = creds_str2creds("UserData", &require_value);
fd = accept(sockfd, &cli_addr, &clilen);
ccreds = creds_getpeer(fd);
allow = creds_have_p(ccreds, require_type, require_value);
if (allow)
   write(fd, MESSAGE("GRANTED\n"));
else
   write(fd, MESSAGE("DENIED\n"));
```

Figure 10: Code example

on the system which can make device unusable. In order to prevent that is necessary to protect access to certain files and folders.

Validator reference hash list also contains list of tokens, required to access files and folders. Validator uses resource tokens kernel API to verify process's permission to access the file.

6.10 Kernel implementation details

Credentials Manager is implemented as set of kernel modules: **restok**, **credp**, and **creds**.

6.10.1 restok

restok module provides a persistent mapping of strings to unique dynamically assigned identifier numbers.

The generated identifiers are used as supplementary group numbers in the task structure and provide additional, dynamically configured credentials for processes. An access to service is protected by requiring a presence of specific credential in the task context (supplementary groups).

Although these numbers are used as supplementary groups, they are not persistent and cannot be used as file system groups in permanent storage.

Once the string has been assigned an identifier, this assignment cannot be changed while restok module is loaded. If the module is compiled into the kernel, the assignments are permanent until the next boot.

To provide different name spaces, the strings form a forest of trees. The string corresponding the identifier, is the path from the ground up to the node that defines the identifier. Within the path "::" is used as a separator.

The module creates a special default tree with an empty string as a name of the root. A string without any "::" is assumed to be a direct child of this default root. For any other identifiers, the string must be a full path from one of the roots to the defining node.

The above rule would make it impossible to address any other root nodes. Thus, the module implements a special case, where a string containing "name::name" collapses into "name". Some examples:

foo -> "::foo" (symbol under default root)
foo::foo -> "foo" (root level symbol, different from previous)
foo::foo::foo -> "foo" (a repeated name is reduced to single instance)
::foo -> "::foo"
:: -> "" (= default root)

The purpose of the "default root" is to provide applications a place to define simple symbols, which do not conflict with the root names, which are used for identifying different name spaces.

The string used in resolving an identifier (function 'restok_locate') is always a full path or a string under the default root.

Strings are defined one level at a time (function 'restok_define'). The identifier of the parent must be supplied. A zero as parent creates a new root.

A malicious application could create a huge number of mapped strings. This is the only reason for limiting the capability of creating new mappings.

For debugging purposes, restok can be compiled as a module, but real usage requires that it is built in.

Source code of this module is located in:
linux/security/mssf/restok

6.10.2 credp

credp module provides credentials management and assignment to the process.

This module maintains a runtime credentials policy, which is a mapping of credentials to an executable or identifier. The module provides a user space API via securityfs entry **/sys/kernel/security/credp/policy**, which is used by tools to add and remove rules from the runtime policy. When adding a new rule, kernel performs translation of resource token strings to identifiers using kernel API provided by the **restok** module.

Credentials Policy database is located in the **/var/lib/mssf/restok/restok.conf** file. During boot, the policy loader **mssf-loader** reads rules from the policy database and imports them into the kernel. Upon installing a new package, package manager, in addition to updating the policy database, imports new rules into the kernel.

To perform credentials assignment, the credp module registers a hook that is called when new executable is about to be started. To achieve that, we implemented a small patch for **security/commoncap.c:cap_bprm_set_creds()**, which allows modules to register credentials assigner operations.

Operations has **apply()** function, which is called from cap_bprm_set_creds() upon executables startup via **execve**.

Source code of this module is located in:
linux/security/mssf/credp

6.10.3 creds

creds module provides an API for user space access control in client/server architecture. The module provides a user space API via securityfs entry **/sys/kernel/security/creds/read**, which is used by **libcreds** library.

This module gives the server a way to read the credentials of the client process and to perform the desired credential checks. Because this is targeted for access control, the returned credentials are the *effective* credentials.

Without this service, getting information about the credentials of another process, is only possible by parsing the "/proc/<pid>/status" content, which is fragile to format changes and it only provides maximum of 32 supplementary groups.

In addition to credentials retrieval, this also provides translations between string and numeric values of credentials. Currently only capabilities names need to be provided and handled by the kernel.

If a companion module 'restok' is compiled, this provides a gateway for translations of symbols defined there. The restok defined symbols are currently mapped into credentials via use of supplementary groups. Other mappings, like defining a totally new credential type for those, are possible in future.

Source code of this module is located in:
linux/security/mssf/creds

7 Privacy Protection

7.1 Protected Storage

Protected Storage provides protection against offline attacks.

Mobile device can be lost or stolen. For that reason it may be a good idea to store sensitive data such as contacts in encrypted form.

Also some security related configuration data such as security policies, credentials policy, reference hash list, certificates, and other configuration data requiring protection against unauthorized modifications.

For that purpose MSSF provides Protected Storage service. Protected storage can be global (G), private (P) or shared between applications (S). It can be used for integrity protection (s) or also for confidentiality (e) protection.

Protected storage implementation uses chipset cryptographic services, and is based on application id and resource tokens.

	Global	Private	Shared
Signed	Gs	Ps	Ss
Encrypted	Ge	Pe	Se

Table 1: Protected Storage types

Private storage uses an application specific key, which is derived from an application id: **K(device key, AppID)**. Global and Shared storages use a shared key, which is derived from a resource token: **K(device key, Resource Token)**. Keys are device specific and ensure copy protection.

If the protected storage is based on a resource token, only those applications that have the resource token can manipulate the store. If the protected storage is based on an application id, only those binaries that share the same application id can manipulate the store.

Applications need to use special API in order to use protected storage.

7.2 Security FS

In order to provide easy-to-use protected storage for such applications, where it doesn't make sense to use proprietary API, MSSF provides a FUSE-based user space file system for similar functionality through the normal POSIX file handling API.

This means in practice that the encryption is transparent for each application, in a similar way with a normal block-device (disk partition) encryption. But unlike with partition-wide encryption, applications cannot see and/or decrypt each other's files unless they have proper credentials, regardless of whether they are running in the same or different user-id.

Manifest file is extended to describe mount points and type of the storage.

Security FS is under heavy development now.

8 Performance

MSSF has slight effect on system performance. Boot time, application startup, runtime performance are affected.

8.1 Integrity protection

Integrity protection (Validator) has most significant implication to system performance. Binary startup time increases, because Validator needs to calculate the SHA1 hash. Verification is done only when binary is loaded for the first time.

Performance of Validator is heavily depends on use and performance of SHA1 HW accelerator. Nokia MeeGo 1.0 N device has SHA1 HW accelerator which is, according to our measurements, quite CPU and power efficient. In our case, application startup time increases by 5 to 10%, and total boot time by 2 to 3%.

8.2 Access Control

Performance penalties given by Access Control framework is insignificant.

Credentials Policy is loaded during boot and requires time insignificant to the total boot time.

Access Control affects application startup time, because Credentials Manager needs to find a policy, and to assign credentials if one exists. It increases startup time by 2.5% (or 6ms in our case).

9 Conclusions and Future work

Mobile Simplified Security Framework is a comprehensive, light-weight alternative to heavy security frameworks for mobile devices. Secure SW distribution model is a important part of MSSF end-to-end security model.

Latest Linux kernel provides integrity subsystem called IMA [6], but verification module (EVM) has not been integrated yet. We will consider possibility to use it when all components are available in the kernel.

MSSF Access Control has similarities to SMACK and we currently investigating possibility for co-operation.

MSSF project can be found on [8]. There libraries, tools and patches for the Linux kernel can be found.

10 Acknowledgements

I would like to thank my Nokia colleagues Ilhan Gurel, Vesa Jaaskelainen, Janne Karhunen, Markku Kylanpaa, Juhani Makela, Janne Mantyla, Elena Reshetova, and Markku Savela for some material, suggestions and review of the article.

References

[1] SELinux - Security Enhanced Linux, `http://www.nsa.gov/research/selinux/index.shtml`

[2] SMACK - Symplified Mandatory Access Control Kernel for Linux, `http://schaufler-ca.com/`

[3] TOMOTYO Linux - MAC implementation for Linux, `http://tomoyo.sourceforge.net/`

[4] Trusted Computing Group, TPM main specification, `https://www.trustedcomputinggroup.org/specs/TPM/`

[5] TGG Mobile Trusted Module specification, 2006, `https://www.trustedcomputinggroup.org/specs/mobilephone/`

[6] Linux kernel integrity subsystem, `http://linux-ima.sourceforge.net/`

[7] DigSig project, `http://disec.sourceforge.net/`

[8] Mobile Simplified Security Framework Project, `http://meego.gitorious.org/meego-platform-security`

Automating Virtual Machine Network Profiles

Vivek Kashyap
IBM
kashyapv@us.ibm.com

Arnd Bergman
IBM
arndb@de.ibm.com

Stefan Berger
IBM
stefanb@us.ibm.com

Gerhard Stenzel
IBM
Gerhard.Stenzel@de.ibm.com

Jens Osterkamp
IBM
Jens.Osterkamp@de.ibm.com

Abstract

With the explosion of use of virtual machines in the data-center/cloud environments there is correspondingly a requirement for automating the associated network management and administration. The virtual machines share the limited number of network adapters on the system among them but may run workloads with contending network requirements. Furthermore, these workloads may be run on behalf of customers desiring complete isolation of their network traffic. The enforcement of network traffic isolation through access controls (filters) and VLANs on the host adds additional run-time and administrative overhead. This is further exacerbated when Virtual Machines are migrated to another physical system as the corresponding network profiles must be re-enforced on the target system. The physical switches must also be reprogrammed.

This paper describes the Linux enhancements in kernel, in libvirt and layer-2 networking, enabling the offloading of the switching function to the external physical switches while retaining the control in Linux host. The layer 2 network filters and QoS profiles are automatically migrated with the virtual machine on to the target system and imposed on the physical switch port without administrative intervention.

We discuss the proposed IEEE standard (802.1Qbg) and its implementation on Linux for automated migration of port profiles when a VM is migrated from one system to another.

1 Introduction

A network adapter is shared across multiple virtual machines(VM) on a Linux host. To accommodate VM-VM and VM to external network communication a virtual bridge is included in the Linux host. The Linux virtual bridge relays unicast traffic between the virtual and physical interfaces attached to it. It further replicates and forwards broadcast and multicast transmission received on its port(s).

For network isolation and security, the iptables/ebtables rules might be enforced on the system. The more common rules are to prevent ARP or IP spoofing, block sending a link level broadcast, or to allow only specific set of protocol traffic.

Different workloads running in the VMs might also be specifically restricted within certain limits based on the workload requirements, priorities and possibly based on the bandwidth purchased by the user.

For the purposes of this paper the layer-2 forwarding, multiplexing and filtering(ebtables) function is collectively considered part of the virtual-bridge.

The per-packet processing for bridging function - packet relaying, replicating, evaluation against filter rules, or bandwidth control - imposes a heavy burden that takes away CPU resources that could be utilized more gainfully. This problem gets more and more exacerbated as the number of virtual machines supported on a single host increases in the multi-core, and large memory systems being used in the data-center/cloud environments.

Another problem faced in large deployments is the need for maintaining consistent view of the port profiles. The switch fabric and the embedded switches in the hypervisor(Linux KVM host) may have different capabilities and also be under separate administrative control.

This is further exacerbated as the VMs (running important workloads) migrate from one system to another.

The filter rules and QoS enforcement must follow the VM and be quickly in place. Not all policies may be deployable in the hypervisor and the switch must be informed of the migration.

As the MAC address associated with the virtual interfaces migrates as well, the physical switch at the target does not know if the MAC, now visible at another port, is a migrated VM or a different VM using the same MAC. Thereby, it needs to be informed of the port-profile policy to deploy.

The proposal 'Edge Virtual Bridging' in IEEE [802.1Qbg] addresses these issues through offloading of the switching function to the adjacent bridge i.e. the physical switch in the network.

1.1 Edge Virtual Bridging: IEEE 802.1Qbg

The Edge Virtual Bridging(EVB) proposal defines the protocols, configuration and control required across the physical end station, such as the Linux host, and the adjacent switch.

This section provides an introduction to the IEEE 802.1Qbg proposal and the subsequent sections describe our implementation of the same on Linux to configure KVM guests in 'VEPA' mode.

Reflective Relay

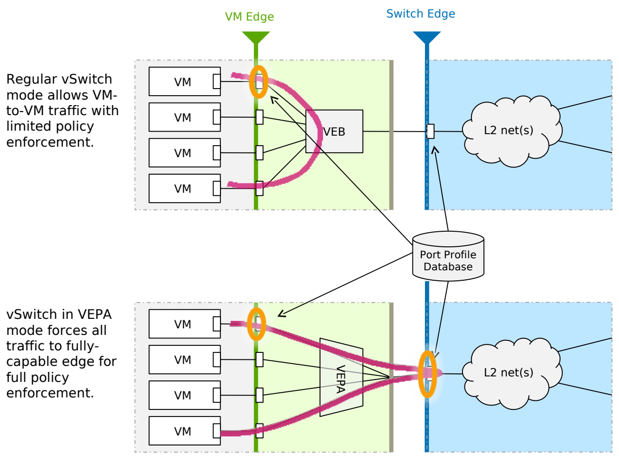
The proposed standard extends the adjacent bridge capabilities to the virtual machines by ensuring that all the packets sent by the VM's are first sent to the switch port. The switch then consults its tables, and if the packet is destined to another VM on the same host, sends the packet back to the host. The packet is then forwarded to the destination VM.

This packet flow enables the switch to enforce fabric wide packet filtering and control policies across all network traffic. Otherwise the policy and rules might need to be co-ordinated and enforced across the switch and the hypervisor leading to administrative overhead and inefficiencies outlined earlier.

Existent switches do not reflect the packet back on the port on which it was received. Therefore, a new mode, referred to as the 'reflective relay' mode is introduced in the Edge Virtual Bridging(EVB) proposal.

Virtual Ethernet Port Aggregator

The component in the physical end-station that works together with the adjacent switch to support EVB is called "Virtual Ethernet Port Aggregator" or VEPA.

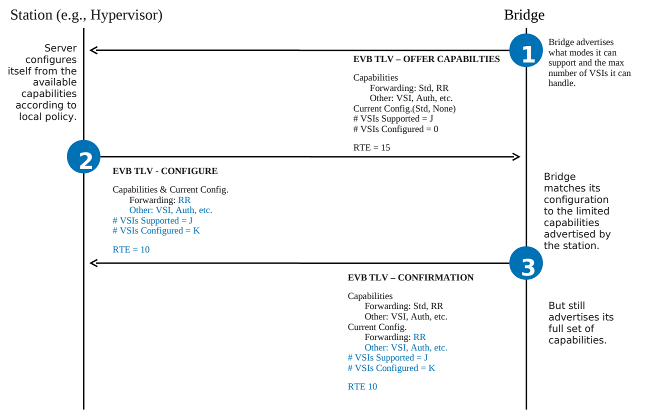

VM Edge
Switch Edge
Regular vSwitch mode allows VM-to-VM traffic with limited policy enforcement.
VM
VM
VM
VM
VEB
L2 net(s)
Port Profile Database
vSwitch in VEPA mode forces all traffic to fully-capable edge for full policy enforcement.
VM
VM
VM
VM
VEPA
L2 net(s)

A VEPA is very simplified version of an Ethernet bridge that allows multiple downlink ports to communicate with a single uplink port but not with each other. Ethernet frames from one of the downlink ports get sent directly to the uplink, and Ethernet frames arriving at the uplink port get forwarded to just the destination with the matching MAC address, or flooded to all downlink ports in case of broadcast. A VEPA does not support unknown unicast frames, which get silently dropped.

The VEPA component therefore, is able to provide VM to VM communication in conjunction with an uplink port which is configured in 'reflective relay' mode.

Dynamic discovery and configuration for VEPA mode

The Link-level Discovery Protocol(LLDP)[LLDP] is used for layer-2 discovery operations. The EVB proposal extends the TLVs (configuration messages) supported to include advertisement of 'reflective relay' capabilities and setting of the adjacent bridge's port in reflective relay mode. The TLV also includes additional parameters such as the the number of virtual interfaces (VSI or Virtual station interfaces in EVB parlance) that a station or bridge can support. See [802.1Qbg] for description of all capabilities and parameters exchanged.

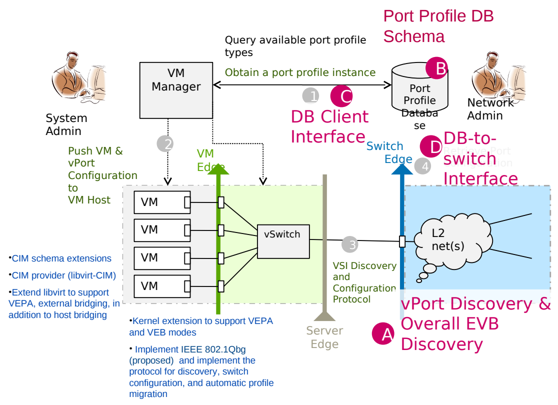

Station (e.g., Hypervisor)
Bridge
Server configures itself from the available capabilities according to local policy.
1
EVB TLV – OFFER CAPABILTIES
Capabilities
Forwarding: Std, RR
Other: VSI, Auth, etc.
Current Config.(Std, None)
# VSIs Supported = J
# VSIs Configured = 0
RTE = 15
Bridge advertises what modes it can support and the max number of VSIs it can handle.
2
EVB TLV - CONFIGURE
Capabilities & Current Config.
Forwarding: RR
Other: VSI, Auth, etc.
# VSIs Supported = J
# VSIs Configured = K
RTE = 10
Bridge matches its configuration to the limited capabilities advertised by the station.
3
EVB TLV – CONFIRMATION
Capabilities
Forwarding: Std, RR
Other: VSI, Auth, etc.
Current Config.
Forwarding: RR
Other: VSI, Auth, etc.
# VSIs Supported = J
# VSIs Configured = K
RTE 10
But still advertises its full set of capabilities.

The bridge periodically advertises its capabilities. The Station receives the TLV, and based on its configuration, may respond to configure the port in 'reflective relay' (RR) mode.

VSI Discovery and Configuration

The switching function is offloaded from the end-station with VEPA and dynamic configuration of the switch port to RR mode. This however does not fully address the problems outlined above. A mechanism is required to associate the virtual interface (VSI) to specific profile for filtering, VLAN and bandwidth control.

This is achieved by informing the switch of the MAC address and VLAN ID pair and the profile that must be used.


Port Profile DB Schema
Query available port profile types
Obtain a port profile instance
VM Manager
System Admin
Network Admin
Port Profile Database
B
1
C
DB Client Interface
2
Push VM & vPort Configuration to VM Host
VM Edge
Switch Edge
D
DB-to-switch Interface
4
VM
VM
VM
VM
vSwitch
3
L2 net(s)
VSI Discovery and Configuration Protocol
•CIM schema extensions
•CIM provider (libvirt-CIM)
•Extend libvirt to support VEPA, external bridging, in addition to host bridging
•Kernel extension to support VEPA and VEB modes
• Implement IEEE 802.1Qbg (proposed) and implement the protocol for discovery, switch configuration, and automatic profile migration
Server Edge
A
vPort Discovery & Overall EVB Discovery

The bridge on receiving the association request gets the profile details from a database and configures its port accordingly. This mechanism also addresses the issue of 'reincarnated' or reused MAC address and a MAC address that has appeared on the port after a VM migration since the switch is informed of the exact profile to impose.

The VSI Discovery protocol (VDP) defined by EVB, therefore defines a set of states and commands exchanged between the bridge and the station. These are:

- *Pre-associate:* Inform the switch of the VSI and port-profile and intention to associate
- *Pre-associate with Resource Reservation:* Same as Pre-associate except also reserve resources at the switch for a future association.
- *Associate:* Associate the VSI. This causes all resources to be allocated at the switch and the imposition of the VSI port profile.
- *DeAssociate:* De-associate the VSI from the port and profile

Edge Control Protocol

The discovery and exchange of bridge capabilities is performed over LLDP. LLDP is an unacknowledged protocol and has limitations on frequency and number of transmissions.

For VDP, the EVB has proposed a new link-level transport called the "Edge Control Protocol"(ECP) which acknowledges the messages exchanged. This enables the End Station to transmit discovery operations more frequently. The VDP protocol messages will be carried in TLVs over ECP. The TLVs carry the VDP state requests and responses.

2 Design and Implementation

The mapping of IEEE protocol to the Linux implementation can be broken into a few distinct interlocked components. These are:

- Extend Linux kernel to support 'VEPA' mode
- Extend libvirt interface xml to define VEPA mode and VSI state
- Implement user-space daemon to support EVB link level protocols

2.1 MACVTAP: Supporting a ’VEPA’ interface

When applied to Linux, a VEPA lets us share a single Ethernet NIC between multiple KVM guests that all get access to the Ethernet segment, but without the need to set up an actual bridge that has both administrative and performance overhead associated with features like MAC address learning, spanning tree protocol or filtering.

For our needs, we developed our solution over the preexisting ’macvlan’ driver supported in Linux. Macvlan provides virtual Ethernet interfaces that can be created using the ”ip link” command and that can be used by applications, virtual machines or containers just like any other Ethernet interface.

One significant drawback for our needs in macvlan implementation was that it cannot easily be connected to Qemu/KVM, which expects a tun/tap device instead of a network interface. In addition, the VEPA implementation has to ensure that broadcast and multicast frames never get delivered to the source port but do get forwarded to all other ports that want them.

In order to connect macvlan devices to a kvm virtual machine, a ”macvtap” device driver was implemented. Macvtap plugs into the macvlan device driver, and lets each of its downstream ports show up in the system as a character device rather than a network interface. This character device implements a subset of the API of the tun/tap driver that is typically used to connect a KVM guest to the host network.

In the implementation of macvtap, a few shortcuts could be taken compared to the combination of tun/tap with a bridge connected to an external NIC. Most importantly, frames sent from the guest to the tap are not injected into the receive path of the host but directly into the transmit queue of the outbound interfaces. Similarly, inbound frames do not need to get received by the host and then sent out to a tun/tap device but simply get put into the guests receive path when they get intercepted by the receive function of the uplink interface.

2.2 libvirt:VEPA interface for the KVM guest

With the necessary infrastructure for VEPA in place with the macvtap/macvlan implementation we needed to integrate the capability into the libvirt domain xml.

Specifying the VEPA interface

Since the macvlan/macvtap are tied specifically to an interface that provides the uplink port for the ’VEPA’ function we decided to provide a strong linkage between the macvtap interfaces and the backing up device.

This is therefore specified in the guest’s domain definition as described below. The example assumes the use of ’eth0’ as the source device.

```
<interface type='direct'>
  <source dev='static' mode='vepa'/>
  <model type='virtio'/>
</interface>
```

Libvirt creates a macvtap interface when a virtual machine with a direct network interface type is started or such an interface type attached to a running virtual machine.

Libvirt opens the tap device to get a filedescriptor for Qemu to write packets to and receive packets from. A macvtap device works similar to a tap device in that an interface is created on the host and a filedescriptor is subsequently passed to Qemu. However, a macvtap device requires more active management by libvirt during device teardown. Whereas for a tap device it is sufficient that Qemu closes the tap filedescriptor for the tap device’s network interface in the host to disappear, libvirt must actively tear down the macvtap device after Qemu was detected to have terminated. Creation and teardown of a macvtap device is done using netlink messages that libvirt sends to the kernel device driver. The command line parameters for Qemu are the same as for a tap device and pass the filedescriptor.

```
[...]  -net nic,
macaddr=52:54:00:0c:dd:47,
vlan=1,model=virtio,
name=net1 -net tap,fd=15,
vlan=0,name=hostnet1
[...]
```

Specifying the VSI state information

The VSI state comprising of the profile to be imposed at the switch port is specified in the interface description as well. The VSI state along with the MAC address, and the VLAN id is therefore forwarded to the user-space daemon implementing the VDP/ECP protocols.

As of writing of this paper we are working to extend the LLDPAD[e1000] daemon to support both the EVB TLVs for RR mode, the ECP and VDP protocols.

The libvirt daemon is extended to send netlink messages with the relevant details to the protocol daemon, LLDPAD. The daemon will then initiate the setup protocol with the switch to enable the packet flow. For this LLDPAD will register the MAC/VLAN pair with the switch along with the VSI data. The success (or failure) will be reported back to libvirt and the guest will be brought online (or failed). In case of failure, libvirt will not start the VM or declare the hot-plugging of an interface to have failed.

The VSI state is specified as in the following example:

```
<interface type='direct'>
  <source dev='static' mode='vepa'/>
  <model type='virtio'/>
  <vsi managerid='12'
   typeid='0x123456'
   typeidversion='1'
   instanceid='insert-uuid-here' />
</interface>
```

VSI states

The ongoing libvirt extensions will send the 'Associate' command when the guest is being brought online and will send a 'DeAssociate' command when the guest is shutdown or suspended or has been migrated.

3 Emulating VEPA bridge

As of this writing there are no switches in the market that support the 802.1Qbg VEPA or reflective relay mode. Therefore, for testing purposes, we utilized the support for 'hairpin mode' or reflective-relay mode already included in the Linux kernel. This enables us to test the raw packet flow that will occur after the switch setup protocol has been successfully completed. Testing of the switch setup protocol will need to be done later.

```
brctl addif <bridge> <interface>
cd /sys/class/net/<interface>/brport
echo 1 > hairpin_mode
```

The VEPA enabled system was then be tested against the Linux host configured as above.

4 Future Work

The libvirt and the LLDPAD work described above are still in progress. The implementation will cover the EVB TLVs, ECP and VDP protocols. The 802.1Qbg proposal also describes 'multi-channel' support and the corresponding 'Channel Discovery and Configuration Protocol' (CDCP). We will extend our implementation to support CDCP in the future.

A large set of management applications use the DMTF's CIM protocol to discover, create and manage virtual machines. There is work ongoing to implement libvirt-CIM providers such that network profile automation can be managed by CIM clients.

Migration of VMs

With the current implementation, at the time of the migration the target system must be put into 'Associate' state with the switch when the migration commences since there is no hook or mechanism to insert additional function in the migration process as implemented by qemu/kvm. It would be advantageous to be able to insert a 'PreAssociate' state on the target and then move to 'Associate' state only when the source is suspended. This will avoid overlapping associates from two systems at the same time since that can cause unpredictable results in the layer-2 fabric if the source and target are connected to the same switch. This issue is mitigated since migrating VMs are only active in one place, either on the source or the target host, but never on both hosts at the same time.

5 Acknowledgements

We would like to thank the EVB working group, IEEE authors group and the Linux community members with whom we have worked closely to define and implement the 802.1Qbg standard. We would specifically like to thank Renato Recio, Mike Krause, Rakesh Sharma Jeff Lynch, Daniel Berrange, Daniel Veillard, and Chris Wright for various discussions on protocol development, implementation and code reviews. Thanks to Anna Fischer for enabling the "hairpin mode" in the Linux bridge.

6 References

[802.1Qbg] http://www.ieee802.org/1/pages/802.1bg.html

[LLDP] http://standards.ieee.org/getieee802/download/802.1AB-2005.pdf

[e1000] http://e1000.sf.net

Coverage and Profiling for Real-time tiny Kernels

Sital Prasad Kedia
Your affiliation
`your-address@example.com`

Abstract

The advantages of a Kernel Sub-Maintainer

Jeff Kirsher
LAN Access Division, Intel® Corporation
`jeffrey.t.kirsher@intel.com`

Abstract

Last year, approximately 9,500 patches were submitted to the Linux kernel networking sub-system. Of these 9.500 patches, roughly 8% of those patch submissions were against the in-kernel Intel® wired LAN drivers. In addition, over the last 2 years, the number of in-kernel Intel® wired LAN drivers went from 3 drivers (e100, e1000 & ixgb) to 8 drivers (e100, e1000, e1000e, igb, igbvf, ixgb, ixgbe & ixgbevf). With the increase in Intel® wired LAN kernel drivers and the large number of kernel patches, support and maintaining of the in-kernel drivers faced several challenges.

To address the issues in maintaining and supporting the Intel® wired LAN in-kernel drivers, we needed a sub-maintainer to deal with all of these challenges. I will go on to explain the obstacles we overcame and the advantages we found by having a sub-maintainer and the processes we use to assist us in our daily routine.

1 Introduction

When submitting patches toward the kernel for our Intel® wired LAN drivers, ran into issues that caused a number of problems. Either the patches did not apply cleanly, because there were changes made to the in-kernel driver that developers were not aware of, or patches were not submitted at the proper time to make a particular kernel version.

Having over 8 developers, who are all trying to keep up-to-date with what the latest networking kernel tree to use and/or the proper order and format the patches needed to be submitted in was a nightmare. Individual developers had to keep up with what was going on in the community while still maintaining their internal work load, and for some that was too much.

From the community standpoint, seeing patches and responses from several different developers at Intel® caused confusion as to who they needed to contact or deal with when problems arose with the Intel® wired LAN kernel drivers. With these problems, we needed to find a resolution which could both service the needs of the community and our internal needs. That is where I come in as the kernel sub-maintainer for Intel® wired LAN drivers...

2 Reasoning

Increased Drivers

Over the last 3 years, Intel® has gone from supporting 3 drivers (e100, e1000 and ixgb) to supporting 8 drivers (e100, e1000, e1000e, igb, igbvf, ixgb, ixgbe, and ixgbevf) with a 40 GbE driver on the way as well. The increasing number of drivers to support means an increase in the amount of work to be done as well as community support.

Increased patches

With the additional in-kernel drivers, comes additional internal and external patches submitted against our drivers. The Linux kernel networking maintainer (David Miller) already has a large number of patches to review and test personally, so any offload of the work would greatly assist Mr. Miller.

3 Advantages

3.1 Consistency

In the past, multiple developers were submitting patches using different techniques and/or tools. The kernel maintainers felt like they had to inform developers of their preferences and tendencies every time a patch was submitted. In addition, developers would submit patches once or twice a year which made it difficult to remember the preferred methods and procedures for submitting patches. By having a single, local sub-maintainer, the patch submittal process followed a more regulated schedule and the patches were consistent in their format to ensure that they followed the kernel coding style. I am also able to keep in constant communication with David Miller to make sure that we are aware of Mr. Miller's current preferences and tendencies.

One of the by products of having consistently formatted patches is that the kernel maintainers and community can focus on reviewing the changes being made rather than focusing on patch formatting. An additional advantage is that patches tend to be smaller and easier to review because patches are being sent out at a consistent and regular pace, rather than "bulk" patches sent every three months.

3.2 Single Point of Contact

With having a dedicated maintainer for the Intel® wired LAN drivers, both the community and kernel maintainers have a single point of contact for questions and/or support for any of our drivers. This reduces the amount of confusion as to who to contact when questions or problems arise and I am able to ensure that the appropriate developers are made aware of any issues.

3.3 Reduced Patch Problems

Patches submitted by the community can often times conflict with patches submitted by Intel® which would either cause the driver to break or have issues passing traffic. In having a single sub-maintainer, David Miller can have confidence that patches I send to him have been tested and will apply cleanly to his networking git trees.

3.4 Increased Patch Testing

Every patch submitted against our drivers, either from the community or internally, goes through our BAT (Basic Acceptance Testing) to ensure that the patch does not cause any issues. The testing includes compile testing, module load/unload, and passing of traffic. In addition to these basic tests, additional targeted testing is done to ensure that the patch does what it is intended to do.

4 Processes (Internal/External)

4.1 Internal Mailing List & Netdev

Community patches come through the kernel networking (Netdev) mailing list and sometimes through the kernel mailing list (LKML). I import these patches and mail them out through our internal kernel patch mailing list. This ensures that Intel® developers and testers who have not seen the patch on the community mailing lists, have a chance to review the community patches.

4.2 Patchworks

In addition to David Miller's Netdev patchwork project, we have an internal patchworks server to keep the status of the patches (internal and community) that have been submitted against our drivers. We have modified the patch status field to include a more granular status of the patch. Here are the additional states that a patch could be in:

Status: New

This is the same status used in the community, new patches which have been submitted that need to be assigned a tester and placed under review for others to see and comment.

Status: Under LAD Review

Patches are under review internally (driver owners specifically) and have been assigned a tester and placed under review for others to see and comment.

Status: Under LAD Review - Critical

This status is reserved for internal and community patches which need immediate attention for review and testing.

Status: Under LAD Testing

The validation team is currently working on validating the patch. Validation of a patch usually takes 24 hours to complete the Basic Acceptance Testing (BAT).

Status: Passed LAD Testing

The testing has passed the Basic Acceptance Testing (BAT) and can be sent to Netdev for inclusion.

Status: Failed LAD Testing

Testing has failed the testing and changes are required. The patch is assigned back to me, along with feedback on why the patch has failed.

Status: Mailed to Kernel/Netdev ML

The patch has passed testing and has been submitted to the appropriate kernel maintainer and kernel mailing list. In some cases, this is a re-submittal (for community patches) to the kernel maintainer and kernel mailing list.

5 Problems

Public Git Tree

Currently we do not have a public git tree for the community to pull from. This can be an issue when I import patches submitted from the community into our internal git tree because there may be several internal patches currently applied to the tree which are under testing. I have been able to minimize this by maintaining several branches in my git trees which keeps the patch count in each branch lower.

Patch Status

Community patch submitters do not have a way to view the status of their patch while the patch goes through our internal patch process. David Miller sets the status of the patch, originally submitted to the community, to "Awaiting Upstream" in the Netdev project of patchworks. This is not a direct reflection of the current status of the patch in our internal git tree.

6 Conclusion

The addition of a sub-maintainer for Intel® wired LAN drivers has helped David Miller in the processing and review of patches. Since the adding of a sub-maintainer, the patch acceptance rate went from 45%, over three years ago, to 97% acceptance rate in 2009. This paper shows some of the benefits of having a sub-maintainer and some of the processes that can be used.

Linux-CR: Transparent Application Checkpoint-Restart in Linux

Oren Laadan
Columbia University
orenl@cs.columbia.edu

Serge E. Hallyn
IBM
serue@us.ibm.com

Abstract

Application checkpoint-restart is the ability to save the state of a running application so that it can later resume its execution from the time of the checkpoint. Application checkpoint-restart provides many useful benefits including fault recovery, advanced resources sharing, dynamic load balancing and improved service availability. For several years the Linux kernel has been gaining the necessary groundwork for such functionality, and now support for kernel based transparent checkpoint-restart is also maturing. In this paper we present the implementation of Linux checkpoint-restart, which aims for inclusion in Linux mainline. We explain the usage model and describe the user interfaces and some key kernel interfaces. Finally, we present preliminary performance results of the implementation.

1 Introduction

Application checkpoint-restart can provide many benefits, including fault recovery by rolling back applications to a previous checkpoint, better response time by restarting applications from checkpoints instead of from scratch, and better system utilization by suspending jobs on demand. Application migration is useful for dynamic load balancing by moving applications to less loaded hosts, fault resilience by migrating applications off of faulty hosts, and improved availability by evacuating applications before host maintenance so that they continue to run with minimal downtime.

While application checkpoint-restart can be performed at different levels, choosing the correct level to transparently support unmodified applications is crucial in practice to enable deployment and widespread use. The two main approaches for providing application checkpoint-restart are in userspace or in the operating system kernel. Userspace approaches are simple to implement and use, but lack transparency and severely limit the types of applications that can be supported. In contrast, kernel approaches utilize operating system support to provide full application transparency, without requiring any changes to applications.

Given the benefits of the kernel level approach, it has been the focus of several projects that aim to provide application checkpoint-restart for Linux [5, 9, 12, 14, 16, 20]. However, none of them is integrated into the mainstream Linux—they are implemented as kernel modules or sets of kernel patches instead. As such, they incur a burden both on users because they are cumbersome to install, and on developers because maintaining them on top of quickly changing upstream kernels is a sisyphean task and they quickly fall behind.

We present Linux-CR, an implementation of transparent application checkpoint-restart in Linux, which aims for inclusion in the Linux mainline kernel. Linux-CR builds on the experience garnered through the previous out-of-mainstream projects. It benefits from many of the supporting features needed for checkpoint and restart that are available upstream today, including the ability to isolate applications inside containers using namespaces, and to selectively freeze applications using the freezer control group. Linux-CR's checkpoint-restart is transparent, secure, reliable, and efficient, and does not adversely impact the performance or code quality of the rest of the Linux kernel.

The remainder of the paper is organized as follows. Section 2 describes Linux checkpoint-restart usage model from a user's point of view. Section 3 presents in detail the checkpoint-restart architecture. Section 4 provides an overview of the in-kernel API for checkpoint-restart. Section 5 presents experimental results. Section 6 discusses related work. Finally, we present some concluding remarks.

2 Usage

The granularity of checkpoint-restart is a process hierarchy. A checkpoint begins at a task which is the root of the hierarchy, and proceeds recursively to include all the descendant processes. It generates a checkpoint image which represents the state of the process hierarchy. A restart takes a checkpoint image as input to create an equivalent process hierarchy and restore the state of the processes accordingly.

Before a checkpoint begins, and for the duration of the entire checkpoint, all processes in the hierarchy must be frozen. This is necessary to prevent them from modifying system state while a checkpoint is underway, and thus avoid inconsistencies from occurring in the checkpoint image. Freezing the processes also puts them in a known state–just before returning to userspace–which is useful because it is a state with only a trivial kernel stack to save and restore.

Linux-CR supports two main forms of checkpoint: *container checkpoint* and *subtree checkpoint*. The distinction between them depends on whether the checkpointed hierarchy is "self contained" and "isolated". The term "self contained" refers to a hierarchy that includes all the processes that are referenced in it. In particular, it must include all parent, child, and sibling processes, and also orphan processes that were re-parented. The term "isolated" refers to a hierarchy whose resources are only referenced by processes that belong to the hierarchy. For example, open file handles held by processes in the hierarchy may not be shared by processes not in the hierarchy. A key property of hierarchies that satisfy both conditions is that their checkpoints are consistent and reliable, and therefore guarantee a successful restart.

Container checkpoint operates on process hierarchies that are both isolated and self contained. In Linux, isolated and self-contained hierarchies are created using *namespaces* [4], which facilitate the provision of private sets of resources for groups of processes. In particular, the PID-namespace can be used to generate a sub-hierarchy that is self contained. A useful management tool for this is Linux Containers [13], which leverages namespaces to encapsulate applications inside virtual execution environments to give them the illusion of running privately.

Checkpointing an entire container ensures that processes inside the container do not depend on processes from the outside. To ensure also that the checkpoint is consistent, the state of shared resources in use by processes in the container must remain unmodified for the duration the checkpoint. Because the processes in the container are frozen they may not alter the state. Thus it suffices to require that, at checkpoint time, none of the resources are referenced by processes from the outside. Combined, these properties guarantee that a future restart from a container checkpoint will always succeed. Note that to leverage container checkpoint, users must launch those application that they wish to checkpoint inside containers.

Subtree checkpoint operates on arbitrary hierarchies. Subtree checkpoints are especially handy since they do not require users to launch their applications in a specific way. For example, casual users that execute long-running computations can simply checkpoint their jobs periodically. However, subtree checkpoints cannot provide the same guarantees as container checkpoints. For instance, because checkpoint iterates the hierarchy from the top down, it will not reach orphan processes unless it begins with the `init(1)` process, so orphan processes will not be recreated and restored at restart. Instead, it is the user's responsibility to ensure that dependencies on processes outside the hierarchy and resource sharing from outside the hierarchy either do not exist or can be safely ignored. For example, a parent process may remain outside the hierarchy if we know that the child process will never attempt to access it. In addition, if outside processes share state with the container, they must not modify that state while checkpoint takes place.

We support a third form of checkpoint: *self-checkpoint*. Self-checkpoint allows a running process to checkpoint itself and save its own state, so that it can restart from that state at a later time. It does not capture the relationships of the process with other processes, or any sharing of resources. It is most useful for standalone processes wishing to be able to save and restore their state. Self-checkpoint occurs by an explicit system call and always takes place in the context of the calling process. The process need not be frozen for the duration of the checkpoint. This form of checkpoint is analogous to the `fork` system call in that the system call may return in two different contexts: one when the checkpoint completes and another following a successful restart. The process can use the return value from the system call to distinguish between the two cases: a successful checkpoint operation will return a "checkpoint ID" which is a non-zero positive integer, while a successful restart operation always returns zero.

2.1 Userspace Tools

The userspace tools consist of three programs: *checkpoint* to take a checkpoint of a process hierarchy, *restart* to restart a process hierarchy from a checkpoint image, and *ckptinfo* to provide information on the contents of a checkpoint image. A fourth utility, *nsexec*, allows users to execute applications inside a container by creating an isolated namespace environment for them. These tools provide the most basic userspace layer on top of the bare kernel functionality. Higher level abstractions for container management and related functionality are provided by packages like lxc [3] (of Linux Containers) and libvirt [2].

Figure 1 provides a practical example that illustrates the steps involved to launch an application inside a container, then checkpoint it, and finally restart it in another container. In the example, the user launches a session with an `sshd` daemon and a `screen` server inside a new container.

Lines 1–3 create a freezer control group to encapsulate the process hierarchy to be checkpointed. Lines 5–13 create a script to start the processes, and the script is executed in line 15. In line 7 the script joins the freezer control group. Descendant processes will also belong there by inheritance, so that later it will be possible to freeze the entire process hierarchy. Lines 8–10 close the standard input, output and error to decouple the new container from the current environment and ensure that it does not depend on `tty` devices.

In line 15, *nsexec* launches the script inside a new private set of namespaces. Lines 18–21 show how the application is checkpointed. First, in line 18 we freeze the processes in the container using the freezer control group. We use the *checkpoint* utility in line 19 to checkpoint the container, using the script's `PID` to indicate the root of the target process hierarchy. In the example, we kill the application once the checkpoint is completed, and thaw the (now empty) control group. We restart the application in line 23. Finally, in line 25 we show how to examine the contents of the checkpoint image using the *ckptinfo* utility.

```
 1   $ mkdir -p /cgroup
     $ mount -t cgroup -o freezer cgroup /cgroup
     $ mkdir /cgroup/1

 5   $ cat > myscript.sh << EOF
     #!/bin/sh
     echo $$ > /cgroup/1/tasks
     exec 0>&-
     exec 1>&-
10   exec 2>&-
     /usr/sbin/sshd -p 999
     screen -A -d -m -S mysession somejob.sh
     EOF

15   $ nohup nsexec -tgcmpiUP pid myscript.sh &

     $ PID=`cat pid`
     $ echo FROZEN > /cgroup/1/freezer.state
     $ checkpoint $PID -l clog -o image.out
20   $ kill -9 $PID
     $ echo THAWED > /cgroup/1/freezer.state

     $ restart -l rlog -i image.out

25   $ ckptinfo -ve < image.out
```

Figure 1: A simple checkpoint-restart example

The log files provided to both the *checkpoint* and *restart* commands are used for status and error reports. Should a failure occur during either operation, the log file will contain error messages from the kernel that carry more detailed information about the nature of the error. Further debugging information can be gained from the checkpoint image itself with the *ckptinfo* command, and from the restart program by using the "-vd" switches.

2.2 System Calls

The userspace API consists of two new system calls for checkpoint and restart, as follows:

- **`long checkpoint(pid, fd, flags, logfd)`**

This system call serves to request a checkpoint of a process hierarchy whose root task is identified by `@pid`, and pass the output to the open file indicated by the file descriptor `@fd`. If `@logfd` is not `-1`, it indicates an open file to which error and debug messages are written. Finally, `@flags` determines how the checkpoint is taken, and may hold one or more of the values listed in Table 1.

On success the system call returns a positive checkpoint identifier. In the future, a checkpoint image may optionally be briefly preserved in kernel memory. In such cases, the identifier would serve to reference that image. On failure the return value is `-1`, and `errno` indicates a suitable error value.

In self-checkpoint, where a process checkpoints itself, it is necessary to distinguish between the first return from

Operation	Flags	Flag description
Checkpoint	`CHECKPOINT_SUBTREE`	perform a subtree checkpoint
	`CHECKPOINT_NETNS`	include network namespace state
Restart	`RESTART_TASKSELF`	perform a self-restart
	`RESTART_FROZEN`	freeze all the restored tasks after restart
	`RESTART_GHOST`	indicate a process that is a place-holder
	`RESTART_KEEP_LSM`	restore checkpointed MAC labels (if permitted)
	`RESTART_CONN_RESET`	force open sockets to a closed state

Table 1: System call flags (checkpoint and restart)

a successful checkpoint, and a subsequent return from the same system call but following a successful restart. This distinction is achieved using the return value, similarly to `sys_fork`. Specifically, the system call returns plain `0` if it came from a successful restart. In either case, when the operation fails the return value is `-1`, and `errno` indicates a suitable error value.

• **`long sys_restart(pid, fd, flags, logfd)`**

This system call serves to restore a process hierarchy from a checkpoint image stored in the open file indicated by the file descriptor `@fd`. It is intended to be called by all the restarting tasks in the hierarchy, and by a special process that coordinates the restart operation. When called by the coordinator, `@pid` indicates the root task of the hierarchy (as seen in the coordinator's PID-namespace). The root task must be a child of the coordinator. When not called by the coordinator, `@pid` must remain `0`. If `@logfd` is not `-1`, it indicates an open file to which error and debug messages are written. Finally, `@flags` determines how the checkpoint is taken, and may hold one or more of the values listed in Table 1.

On success the system call returns in the context of the process as it was saved at the time of the checkpoint. The exact behavior depends on how and when the checkpoint was taken. If the process was executing in userspace prior to the checkpoint, then the restart will arrange for it to resume execution in userspace exactly where it was interrupted. If the process was executing a system call, then the return value will be set to the return value of that system call whether it completed or was interrupted. For a self-checkpoint, the restart will arrange for the process to resume execution at the first instruction after the original call to `sys_checkpoint`, with the system call's return value set to `0` to indicate that this is the result of a restart. On failure the return value is `-1` and `errno` indicates a suitable error value.

3 Architecture

The crux of checkpoint-restart is a mechanism to serialize the execution state of a process hierarchy, and to restore the process hierarchy and its state from the saved state. For checkpoint-restart of multi-process applications, not only must the state associated with each process be saved and restored, but the state saved and restored must be globally consistent and preserve process dependencies. Furthermore, for checkpoint-restart to be useful in practice, it is crucial that it transparently support existing applications.

To guarantee that the state saved is globally consistent among all processes in a hierarchy, we must satisfy two requirements. First, the processes must be frozen for the duration of the checkpoint. Second, the resources that they use must not be modified by processes not in the hierarchy. These requirements ensure that the state will not be modified by processes in or outside the hierarchy while the execution state is being saved.

To guarantee that the operation is transparent to applications, we must satisfy two more requirements. First, the state must include all the resources in use by processes. Second, resource identifiers in use at the time of the checkpoint must be available when the state is restored. These requirements ensure not only that all the necessary state exists when restart completes, but also that it is visible to the application as it was before the checkpoint, so that the application remains unaware.

Linux-CR builds on the *freezer* subsystem to achieve quiescence of processes. This subsystem was created to allow the kernel to freeze all userspace processes in preparation for a full system suspend to disk. To accommodate checkpoint-restart, the *freezer* subsystem was recently re-purposed to enable freezing of groups of processes, with the introduction of the *freezer* control group.

Linux-CR leverages *namespaces* [4] to encapsulate processes in a self-contained unit that isolates them from other processes in the system and decouples them from the underlying host. Namespaces provide virtual private resource names: resource identifiers within a namespace are localized to the namespace. Not only are they invisible to processes outside that namespace, but they do not collide with resource identifiers in other namespaces. Thus, resource identifiers, such as `PID`s, can remain constant throughout the life of a process even if the process is checkpointed and later restarted, possibly on a different machine. Without it, identifiers may in fact be in use by other processes in the system. To accommodate checkpoint-restart, namespaces were extended to allow restarting processes to select predetermined identifiers upon the allocation of their resources, so that those processes can reclaim the same set of identifiers they had used prior to the checkpoint.

For simplicity, we describe the checkpoint-restart mechanism assuming *container checkpoint*. We also assume a shared storage (across participating machines), and that the filesystem remains unmodified between checkpoint and restart. In this case, the filesystem state is not generally saved and restored as part of the checkpoint image, to reduce checkpoint image size. Available filesystem snapshot functionality [6, 10, 15] can be used to also provide a checkpointed filesystem image. We focus only on checkpointing process state; details on how to checkpoint filesystem, network, and device state are beyond the scope of this paper.

3.1 Kernel vs. Userspace

Previous approaches to checkpoint-restart have run the gamut from fully in-kernel to hybrid to fully userspace implementations. While many properties of processes and resources can be recorded and restored in userspace, some state exists that cannot be recorded or restarted from userspace. To provide application transparency and allow applications to use the full range of operating system services, we chose to implement checkpoint-restart in the kernel. In addition, an in-kernel implementation is not limited to user visible APIs such as system calls–it can use the full range of kernel APIs. This not only simplifies the implementation, but also allows use of native locking mechanisms to ensure atomicity at the desired granularity.

Checkpoint is performed entirely in the kernel. Restart is also done in the kernel, however, for simplicity and flexibility, the creation of the process hierarchy is done in userspace. Moving some portion of the restart to userspace is an exception, which is permitted under two conditions: first, it must be straightforward and leverage existing userspace APIs (i.e. not introduce specialized APIs). Second, doing so in userspace should bring significant added value, such as improved flexibility. Also, all userspace work must occur before entering the kernel, to avoid transitions in and out of the kernel.

For instance, the incentive to do process creation in userspace is because it is simple to use the `clone` system call to do so, and because it allows for great flexibility for restarting processes to do useful work after the process hierarchy is created and before the rest of the restart takes place. Indeed, the entire hierarchy is created before in-kernel restart is performed. Likewise, it is desirable to restore network namespaces in userspace[1]. Doing so will allow reuse of existing userspace network setup tools that are well understood instead of replicating their high-level functionality inside the kernel. Moreover, it will allow users to easily adjust network settings at restart time, e.g. change the network device or its setup, or add a firewall to the configuration.

3.2 Checkpoint

A checkpoint is performed in the following steps (steps 2–4 are done by the `checkpoint` system call):

1. Freeze the process hierarchy to ensure that the checkpoint is globally consistent.
2. Record global data, including configuration and state that are global to the container.
3. Record the process hierarchy as a list of all checkpointed processes, their `PID`s, and relationships.
4. Record the state of individual processes, including credentials, blocked and pending signals, CPU registers, open files, virtual memory, etc.
5. Thaw the processes to allow them to continue executing, or terminate the processes in case of migration. (If a filesystem snapshot is desired, it is taken prior to this step.)

Checkpoint is done by an auxiliary process, and does not require the collaboration of processes being checkpointed. This is important since processes that are not runnable, e.g. stopped or traced, would not be able to

[1]However, as of the writing of this paper, this is yet undecided.

perform their own checkpoint. Moreover, this can be extended in the future to multiple auxiliary processes for faster checkpoint times of large process hierarchies.

Much effort was put to make checkpoint robust in the sense that if a checkpoint succeeds then, given a suitable environment, restart will succeed too. The implementation goes to great extents to be able to detect whether checkpointed processes are "non-restartable". This can happen, for example, when a process uses a resource that is unsupported for checkpoint-restart, therefore it will not be saved at all. Even if a resource is supported, it may be temporarily in an unsupported state. For example, a socket that is in the process of establishing a connection is currently unsupported.

3.3 Restart

A restart is performed in the following steps (step 3 is done by the `restart` system call):

1. Create a new container for the process hierarchy, and restore its configuration and state.
2. Create the process hierarchy as prescribed in the checkpoint image.
3. Restore the state of individual processes in the same order as they were checkpoint.
4. Allow the processes to continue execution[2].

Restart is managed by a special *coordinator* process, which supervises the operation but is not a part of the restarted process hierarchy. The coordinator process creates and configures a new container, and then generates the new process hierarchy in it. Once the hierarchy is ready, all the processes execute a system call to complete the restart of each process in-kernel.

To produce the process hierarchy, it is necessary to preserve process dependencies, such as parent-child relationships, threads, process groups, and sessions. The restored hierarchy must satisfy the same constraints imposed by process dependencies at checkpoint. Because the process hierarchy is constructed in userspace, these dependencies must be established at process creation time (to leverage the existing system calls semantics). The order in which processes are created is important, because some dependencies are not reflected directly from the hierarchical structure. For instance, an orphan process must be recreated by a process that belongs to the correct session group to correctly inherit that group.

[2]It is also possible to freeze the restarted processes, which is useful for, e.g., debugging.

Linux-CR builds on two algorithms introduced by Zap [12] to reconstruct the process hierarchy. The algorithms are designed to create a hierarchy that is equivalent to the original one at checkpoint. The first algorithm, *DumpForest*, analyzes the state of a process hierarchy. It runs in linear time with the number of `PID`s in use in the hierarchy. The output is a table with an entry for each `PID`; The table encodes the order and the manner in which processes should be restarted. The second algorithm, *MakeForest*, reconstructs the hierarchy. It works in a recursive manner by following the instructions set forth by the table. It begins with a single process that will be used as the root of the hierarchy to fork its children, then each process creates its own children, and so on. For a detailed discussion of these algorithms refer to [12].

In rebuilding the process hierarchy, there are two special cases of `PID`s referring to terminated processes that require additional attention. One case is when a `PID` of a dead process is used as a `PGID` of another process. In this case, the restart algorithm creates a "ghost" process–a placeholder that lives long enough so that its `PID` can be used as the `PGID` of another process, but terminates once the restart completes (and before the hierarchy may resume its execution, to avoid races). Another case is when a `PID` represents a *zombie* process that has exited but whose state has not been cleaned up yet. In this case, the restart algorithm creates a process, restores only minimal state such as its exit code, and finally the process exits to become a zombie.

Because the process hierarchy is created in userspace, the restarting processes have the flexibility to do useful work before eventually proceeding with in-kernel restart. For instance, they might wish to create a new custom networking route or filtering rule, create a virtual device which existed at the host at the time of checkpoint, or massage the mounts tree to mask changes since checkpoint.

Once the process hierarchy is created, all the processes invoke the `restart` system call and the remainder of the restart takes place in the kernel. Restart is done in the same order that processes were checkpointed. The restarting processes now wait for their turn to restore their own state, while the *coordinator* orchestrates the restart.

Restart is done within the context of the process that is restarted. Doing so allows us to leverage the available kernel functionality that can only be invoked from within that context. Unlike checkpoint, which requires observing process state, restart is more complicated as it must create the necessary resources and reinstate their desired state. Being able to run in process context and leverage available kernel functionality to perform these operations during restart significantly simplifies the restart mechanism.

In the kernel, the `restart` system call depends on the caller. The *coordinator* first creates a common restart context data structure to share with all the restarting process, and waits for them to become properly initialized. It then notifies the first process to start the restart, and waits for all the restarting tasks to finish. Finally, the *coordinator* notifies the restarting tasks to resume normal execution, and then returns from the system call.

Correspondingly, restarting processes first wait for a notification from the *coordinator* that indicates that the restart context is ready, and then initialize their state. Then, each process waits for its turn to run, restores the state from the checkpoint image, notifies the next restarting process to run, and waits for another signal from the *coordinator* indicating that it may resume normal execution. Thus, processes may only resume execution after all the processes have successfully restored their state (or fail if an error has occurred), to prevent processes from returning to userspace prematurely before the entire restart completes.

3.4 The Checkpoint Image

The checkpoint image is an opaque *blob* of data, which is generated by the `checkpoint` system call and consumed by the `restart` system call. The blob contains data that describes the state of select portions of kernel structures, as well as process execution state such as CPU registers and memory contents. The image format is expected to evolve over time as more features are supported, or as existing features change in the kernel and require to adjust their representation. Any changes in the blob's format between kernel revisions will be addressed by userspace conversion tools, rather than attempting to maintain backward compatibility inside the `restart` system call.

Internally, the blob consists of a sequence of records that correspond to relevant kernel data structures and represent their state. For example, there are records for process data, memory layout, open files, pending signals, to name a few. Each record in the image consists of a header that describes the type and the length of the record, followed by a payload that depends on the record type. This format allow userspace tools to easily parse and skim through the image without requiring intimate knowledge of the data. Keeping the data in self contained records will also be suitable for parallel checkpointing in the future, where multiple threads may interleave data from multiple processes into a single stream.

Records do not simply duplicate the native format of the respective kernel data structures. Instead, they provide a representation of the state by copying those individual elements that are important. One justification is that during restart, one already needs to inspect, validate and restore individual input elements before copying them into kernel data structures. However, the approach offers three additional benefits. First, it improves image format compatibility across kernel revisions, being agnostic to data structure changes such as reordering of elements, addition or deletion of elements that are unimportant for checkpoint-restart, or even moving elements to other data structures. Second, it reduces the total amount of state saved since many elements may be safely ignored. Per process variables that keep scheduler state are one such example. Third, it allows a unified format for architectures that support both 32-bit and 64-bit execution, which simplifies process migration between them.

The checkpoint image is organized in five sections: a header, followed by global data, process hierarchy, the state of individual processes, and a finally a trailer. The header includes a magic number (to identify the blob as a checkpoint image), an architecture identifier in little-endian format, a version number, and some information about the kernel configuration. It also saves the time of the checkpoint and the flags given to the system call. It is followed by an architecture dependent header that describes hardware specific capabilities and configuration.

The global data section describes configuration and state that are global to the container being checkpointed. Examples include Linux Security Modules (LSM) and network devices and filters. In the future container-wide mounts may also go here. The process hierarchy section that follows provides the list of all checkpointed processes, their `PIDs` and their relationships, e.g., parent-child, siblings, threads, and zombies. These two section are strategically placed early in the image for two reasons: first, it allows restart to create a suitable environment for the rest of the restart early on, and second, it allows to do so in userspace.

The remainder of the checkpoint image contains the state of all of the tasks and the shared resources, in the order that they were reached by the process hierarchy traversal. For each task, this includes state like the task structure, namespaces, open files, memory layout, memory contents, CPU state, signals and signal handlers, etc. Finally, the trailer that concludes the entire image serves as a sanity check.

The checkpoint-restart logic is designed for streaming to support operation using a sequential access device. Process state is saved during checkpoint in the order in which it needs to be used during restart. An important benefit of this design is that the checkpoint image can be directly streamed from one machine to another across the network and then restarted, to accomplish process migration. Using a streaming model provides the ability to pass checkpoint data through filters, resulting in a flexible and extensible architecture. Example filters include encryption, signature/validation, compression, and conversion between formats of different kernel versions.

3.5 Shared Resources

Shared resources may be referenced multiple times, e.g. by multiple processes or even by other resources. Examples of resources that may be shared include files descriptors, memory address spaces, signal handlers and namespaces. During checkpoint, shared resources will be considered several times as the process hierarchy is traversed, but their state need only be saved once.

To ensure that shared resources are saved exactly once, we need to be able to uniquely identify each resource, and keep track of resources that have been saved already. More specifically, when a resource is first discovered, it is assigned a unique identifier (*tag*) and registered in a hash-table using its kernel address (at checkpoint) or its tag (at restart) as a key. The hash-table is consulted to decide whether a given resource is a new instance or merely a reference to one already registered. Note that the hash-table itself is not saved as part of the checkpoint image; instead, it is rebuilt dynamically during both checkpoint and restart, and discarded when they complete.

During checkpoint, shared resources are examined by looking up their kernel addresses in the hash-table. If an entry is not found, then it is a new resource–we assign a new tag and add it to the hash-table, and then record its state. Otherwise, the resource has been saved before, so it suffices to save only the tag for later reference.

During restart the state is restored in the same order as has been saved originally, ensuring that the first appearance of each resource is accompanied with its actual recorded state. As with checkpoint, saved resource tags are examined by looking them up in the hash-table, and if not found, we create a new instance of the required resource, restore its state from the checkpoint image, and add it to the hash-table. If an entry is found, it points to the corresponding (already restored) resource instance, which is reused instead of creating a new one.

3.6 Leak Detection

In order to guarantee that a *container checkpoint* is consistent and reliable, we must ensure that the container is isolated and that its resources are not referenced by other processes from outside the container. When container shared resources are referenced from outside, we say that they *leak*; the ability to detect leaks is a prerequisite for container checkpoint to succeed.

Because the shared objects hash-table already tracks shared resources, it also plays a crucial role in detecting resource leaks that may obstruct a future restart. The key idea behind leak detection is to explicitly count the total number of references to each shared object inside the container, and compare them to the global reference counts maintained by the kernel. If for a certain resource the two counts differ, it must be because of an external (outside the container) reference.

Leak detection begins in a pre-pass that takes place prior to the actual checkpoint, to ensure that there are no external references to container shared objects. In this pass we traverse the process hierarchy like in the actual checkpoint but do not save the state. Instead, we collect the shared resources into the hash-table and maintain their reference counts. When this phase ends, the reference count of each object in the hash-table reflects the number of in-container references to it. We now iterate through all the objects and compare that count to the one maintained by the kernel. The two counts match[3] if and only if there are no external references.

[3]Actually, the hash-table count should be one less, because it does not count the reference that the hash-table itself takes.

This procedure is non-atomic in the sense that the reference counts from the hash-table and the kernel are compared only after all the resources have been collected. This is racy because processes outside the container may modify the state of shared resources, create new ones or destroy existing ones before the procedure concludes. For instance, consider two processes, one inside a container and the other not, that share a single file descriptor table. Suppose that after the pre-pass collects the file table and the files in it, the outside process opens a new file, closes another (existing) file, and then terminates. At this point, the new file is left out of the hash-table, while the other (closed) file remains there unnecessarily. Moreover, when the pre-pass concludes it will not detect a file table leak because the outside process exited, and the file table is only referenced inside the container. It will not detect a file leak even though the new file may be referenced outside, because the new file had not even been tracked.

To address these races we employ additional logic for leak detection during the actual checkpoint. This logic can detect in-container resources that are not tracked by the hash-table, or that are tracked but are no longer referenced in the container. Untracked resources are easy to detect, because their lookup in the hash-table will fail. To detect deleted resources, we mark every resource in the hash-table that we save during the checkpoint, and then at the end of the checkpoint, we verify that all the tracked resources are marked. A tracked but unmarked resource must have been added to the hash-table and then deleted before being reached by the actual checkpoint. In either case, the checkpoint is aborted.

3.7 Error Handling

Both checkpoint and restart operations may fail due to a variety of reasons. When a failure does occur, they must provide proper cleanup. For checkpoint this is simple, because the checkpoint operation is non-intrusive: the process hierarchy whose state is saved remains unaffected. For restart, cleanup is performed by the coordinator, which already keeps track of all processes in the restored hierarchy. More specifically, in case of failure the coordinator will send a fatal signal to terminate all the processes before the system call returns. Because the cleanup is part of the return path from the `restart` system call exit path, there is no risk that cleanup be skipped should the coordinator itself crash.

When a checkpoint or restart fails, it is desirable to communicate enough details about the failure details to the caller to determine the root cause. Using a simple, single return value from the system call is insufficient to report the reason of a failure. For instance, a process that is not frozen, a process that is traced, an outstanding asynchronous IO transfer, and leakage of shared resource leakage, are just a few failure modes that result all in the error `-EBUSY`.

To address the need to report detailed information about a failure, both `checkpoint` and `restart` system calls accept an additional argument: a file descriptor to which the kernel writes diagnostic and debugging information. Both the checkpoint and restart userspace utilities have options to specify a filename to store this log.

In addition, checkpoint stores in the checkpoint image informative status report upon failure in the form of (one or more) error objects. An error object consists of a mandatory pre-header followed by a null character (`'\0'`), and then a string that describes the error. By default, if an error occurs, this will be the last object written to the checkpoint image. When a failure occurs, the caller can examine the image and extract the detailed error message. The leading `'\0'` is useful if one wants to seek back from the end of the checkpoint image, instead of parsing the entire image separately.

3.8 Security Considerations

The security implications of in-kernel checkpoint-restart require careful attention. A key concern is whether the system calls should require privileged or unprivileged operation. Originally our implementation required tasks to have the `CAP_SYS_ADMIN` capability, while we optimistically asserted our intent to eventually remove the need for privilege and allow all users to safely use checkpoint and restart. However, it was pointed out that letting unprivileged users use these system calls is not only beneficial to users, but also has the useful side effect of forcing the checkpoint-restart developers to be more careful with respect to security throughout the design and development process. In fact, we believe this approach has succeeded in keeping us more on our toes and catching ways that users otherwise would have been able to escalate privileges through carefully manipulated checkpoint images, for instance bypassing `CAP_KILL` requirements by specifying arbitrary `userid` and signals for file owners.

At checkpoint, the main security concern is whether the process that takes a checkpoint of other processes in some hierarchy has sufficient privileges to access that state. We address this by drawing an analogy between checkpointing and debugging processes: in both it is necessary for an auxiliary process to gain access to internal state of some target process(es). Therefore, for checkpoint we require that the caller of the system call will be privileged enough to trace and debug (using `ptrace`) all of the processes in the hierarchy.

For restart, the main concern is that we may allow an unprivileged user to feed the kernel with random data. To this end, the restart works in a way that does not skip the usual security checks. Process credentials, i.e. `UID`, `EUID`, and the security context[4] currently come from the caller, not the checkpoint image. To restore credentials to values indicated in the checkpoint image, restarting processes use the standard kernel interface. Thus, the ability to modify one's credentials is limited to one's privilege level when beginning the restart.

Keeping the restart procedure to operate within the limits of the caller's credentials means that scenarios consisting of privileged application that reduce their privilege level cannot be supported. For instance, a "setuid" program that opened a protected log file and then dropped privileges will fail the restart, because the user will not have enough credentials to reopen the file. The only way to securely allow unprivileged users to restart such applications is to make the checkpoint image tamper-proof.

There are a few ways to ensure the a checkpoint image is authentic. One method is to make the userspace utilities privileged using "setuid" and use cryptographic signatures to validate checkpoint images. In particular, checkpoint will sign the image and restart will first verify the signature before restoring from it. For instance, TPM [11] can be used to sign the checkpoint image and produce a keyed hash using a sealed private key, and to refuse restart in the absence of the correct hash. Another method is to create an assured pipeline for the checkpoint image, from the invocation of the checkpoint and restart system calls. Assured pipelines are precisely a target feature of SELinux, and could be implemented by using specialized domains for checkpoint and restart. Note, however, that even with a tamper-proof checkpoint image, a concern remains that the checkpoint image amounts to a persistent privileged token, which a clever user could find ways to exploit in new and interesting ways.

[4]Security contexts are part of Linux Security Modules (LSM).

4 Kernel Internal API

The kernel API consists of a set of functions for use in kernel subsystems and modules to provide support for checkpoint and restart of the state that they manage. All the kernel API calls accept a pointer to a checkpoint context (`ctx`), that identifies the operation in progress.

The kernel API can be divided by purpose into several groups: functions to handle data records; functions to read and write checkpoint images; functions to output debugging or error information; and functions to handle shared kernel objects and the hash-table. Table 2 lists the API groups and their naming conventions. Additional API exists to abstract away the details about checkpointing and restoring instances of some objects types, including memory objects, open files, and LSM (security) annotations.

The `ckpt_hdr_...` group provides convenient helper functions to allocate and deallocate buffers used as intermediate store for the state data. During checkpoint they store the saved state before it is written out. During restart they store data read from the image before it is consumed to restore the corresponding kernel object.

The `ckpt_write_...` and `ckpt_read_...` groups provide helper functions to write data to and read data from the checkpoint image, respectively. These are wrappers that simplify the handling, for example by adding and removing record headers, and by providing shortcuts to handle common data such as strings and buffers.

The `ckpt_msg` function writes an error message to the log file (if provided by the user), and, when debugging is enabled, also to the system log. The `ckpt_err` function is used when checkpoint or restart cannot succeed. It accepts the error code to be returned to the user, and a formatted error message which is written to the user-

Group	**Description**
`ckpt_hdr_...`	record handling (alloc/dealloc)
`ckpt_write_...`	write data/objects to image
`ckpt_read_...`	read data/objects from image
`ckpt_msg`	output to the log file
`ckpt_err`	report an error condition
`ckpt_obj_...`	manage objects and hash-table

Table 2: Kernel API by groups

provided log and the system log. If multiple errors occur, e.g. during restart, only the first error value will be reported, but all messages will be printed.

The `ckpt_obj_...` group includes helper functions to handle shared kernel objects. They simplify the hash-table management by hiding details such as locking and memory management. They include functions to add objects to the hash-table, to find objects by their kernel address at checkpoint or by their tag at restart, and to mark objects that are saved (for leak detection).

Dealing with shared kernel objects aims to abstract the details of how to checkpoint and restart different object types, and push the code to do so near the native code for those objects. For example, code to checkpoint and restart open files and memory layouts appears in the `file/` and `mm/` subdirectories, respectively. The motivation for this is twofold. First, placing the checkpoint-restart code there improves maintainability because it make the code more visible to maintainers. Higher maintainers' awareness increases the chances that they will adjust the checkpoint-restart code when they introduce changes to the other native code. Second, it is more friendly to kernel objects that are implemented in kernel modules, because it means that the code to checkpoint-restart such objects is also part of the module.

To abstract the handling of shared kernel objects we associate a set of operations with each object type (similar to operations for files, sockets, etc). These include methods to checkpoint and restore the state of an object, methods to take or drop a reference to the objects so that it can be referenced when in the hash-table, and a method to read the reference count (in the hash-table) of an object. The function `register_checkpoint_obj` is used to register an operations set for an object type. It is typically called from kernel initialization code for the corresponding object, or from module initialization code as part of loading a new module. Each of the `ckpt_obj_...` takes the object type as one of its argument, which indicates the object-specific set of operation to use.

During checkpoint, for each shared kernel object the function `checkpoint_obj` is called. It first looks up the object in the hash-table, and, if not found, invokes the `->checkpoint` method from the corresponding operations set to create a record for the object in the image, and adds the object to the hash-table. During restart, records of shared kernel objects in the input stream are passed to the function `restore_obj`, which invokes the `->restore` method from the corresponding operations set to create an instance of the object according to the saved state, and adds the newly created object to the hash-table.

Several objects in the Linux kernel are already abstracted using operations set, which contain methods describing how to handle different versions of objects. For instance, open files have `file_operations`, and seeking in *ext3* filesystem is performed using a different method than in *nfs* filesystem. Likewise, virtual memory areas have `vm_operations_struct`, and the method to handle page faults is different in an area that corresponds to private anonymous memory than one that corresponds to shared mapped memory.

For such objects, we extend the operations to also provide methods for checkpoint, restore, and object collection (for leak detection), as follows:

To checkpoint a virtual memory area in a task's memory map, the `struct vm_operations_struct` needs to provide the method for the `->checkpoint` operation:

```
  int checkpoint(ctx, vma)
```

and at restart, a matching callback to restore the state of a new virtual memory area object::

```
  int restore(ctx, mm, vma_hdr)
```

Note that the function to restore cannot be part of the operations set, because it needs to be known before the object instance even exists.

To checkpoint an open file, the `struct file_operations` needs to provide the methods for the `->checkpoint` and `->collect` operations:

```
  int checkpoint(ctx, file)
  int collect(ctx, file)
```

and at restart, a matching callback to restore the state of an opened file:

```
  int restore(ctx, file, file_hdr)
```

Here, too, the restore function cannot be part of the operations set. For most filesystems, generic functions are sufficient: `generic_file_checkpoint` and `generic_file_restore`.

To checkpoint a socket, the `struct proto_ops` needs to provide the methods for the `->checkpoint`, `->collect` and `->restore` operations:

```
  int checkpoint(ctx, sock);
  int collect(ctx, sock);
  int restore(ctx, sock, sock_hdr)
```

Number of processes	Image size	Checkpoint time (to file)	Checkpoint time (no I/O)	Restart time
10	0.87 MB	8 ms	3 ms	10 ms
100	8.0 MB	72 ms	24 ms	72 ms
1000	79.6 MB	834 ms	237 ms	793 ms

Table 3: Checkpoint-restart performance

5 Experimental Results

We evaluated the current version of Linux Checkpoint-Restart to answer three questions: (a) how long checkpoint and restart take; (b) how much storage they require; and (c) how they scale with the number of processes and their memory size.

The measurements were conducted on a Fedora 10 system with two dual-core 2 GHz AMD Opteron CPUs with 1 GB L2 cache, 2 GB RAM, and a 73.4 GB 10025 RPM local disk. We used Linux 2.6.34 with the Linux-CR v21 patchset, with most debugging disabled and SELinux disabled. All optional system daemons on the test system were turned off.

For the measurements we used the *makeprocs* test from the checkpoint-restart test-suite [1]. This test program allows a caller to specify the number of child processes, a memory size for each task to map, or a memory size for each task to map and then dirty. We repeated the tests thirty times, and report average values.

Table 3 presents the results in terms of checkpoint image size, checkpoint time and restart time, for measurements with 10, 100 and 1000 processes. Checkpoint times were measured once with the output saved to a local file, and once with the output discarded (technically, redirected to `/dev/null`). Restart times were measured from when the coordinator begins and until it returns from the kernel after a successful operation. Restart times were measured reading the checkpoint images from a warm-cache.

The results show that the checkpoint image size and the time for checkpoint and restart scale linearly with the number of processes in the process hierarchy. The average total checkpoint time is about 0.8 ms per process when writing the data to the filesystem, and drops to well under 0.3 ms per process when filesystem access is skipped. Writing the data to a file triples the checkpoint time. This suggests that buffering the checkpoint output until the end of a checkpoint is a good candidate for a future optimization to reduce application downtime at checkpoint. Average total restart time from warm cache is about 0.9 ms per process. The total amount of state that is saved per process is also modest, though it highly depends on the applications being checkpointed.

To better understand how much of the checkpoint and restart times is spent for different resources, we instrumented the respective system calls to measure a breakdown of the total time. For both checkpoint and restart, saving and restoring the memory contents of processes amounted to over 80% of the total time. The total memory in use within a process hierarchy is also the most prominent component of the checkpoint image size.

To look more closely at the impact of the process size, we measured the checkpoint time for ten processes each with memory sizes increasing from 1 MB to 1 GB that the process allocated using the `mmap` system call. We repeated the test twice. In one instance the processes only allocate memory but do not touch it. In the second instance the processes also dirty all the allocated memory. In both cases, the output was redirected to avoid expensive I/O. The results are given in Table 4. The results show strong correlation between the memory footprint of processes and checkpoint times. Even when memory is untouched, the cost associated with scanning the process's page tables is significant. Checkpoint times increase substantially with dirty memory as it requires the contents to actually be stored in the checkpoint image.

Task size	Checkpoint time (clean)	Checkpoint time (dirty)
1 MB	0.5 ms	1.3 ms
10 MB	0.7 ms	6.4 ms
100 MB	1.7 ms	58 ms
1 GB	10.6 ms	337 ms

Table 4: Checkpoint times and memory sizes

6 Related Work

Checkpoint-restart has been the subject of extensive research [17, 18, 19], spanning all four approaches: application level, library mechanisms, operating system mechanisms, and hardware virtualization; See [12] for a detailed discussion on these approaches.

Many application checkpoint-restart mechanisms have been implemented in Linux, some in userspace [7, 8] and others in the kernel [5, 9, 14, 12, 20]. Ckpt [7] modifies tasks to let them checkpoint themselves. The checkpoint images are in the form of executable files which, when executed, restart the original process. CryoPID [8] also uses an executable file for the checkpoint image, but relies on `/proc` information to checkpoint a task. Userspace approaches do not capture or are unable to restore some parts of a process's system state, and are limited in which applications they support. In contrast, Linux-CR is an operating system mechanism that can save and restore all relevant state, and can do so completely transparently to the application.

EPCKPT [9] and CRAK [20] provide partial support for checkpoint-restart for Linux 2.4 series, as a kernel patch and kernel module respectively. BLCR [5] is aimed primarily at HPC users. It consists of a library and a kernel module. Applications must be checkpoint-aware so as to discard unsupported resources. None of these provide virtualization. Zap [12, 16] and OpenVZ [14] implement both containers and checkpoint-restart. OpenVZ consists of an invasive out-of-tree kernel patch, and Zap is a kernel module.

The Linux-CR project emerged as a unifying framework to provide checkpoint-restart in the Linux kernel. It builds on Zap, but it is implemented in the kernel. Linux-CR improves on earlier work in that the design and implementation are done in a clean way and geared for inclusion in the mainstream Linux kernel, so that it will be maintained as part of the kernel.

Linux-CR can detect when resources leak outside of containers. This leak detection logic was inspired by OpenVZ, which was the first to propose this mechanism. However, leak detection in OpenVZ is incomplete because it does not handle race condition when scanning for resources, allowing untracked and deleted resources to remain undetected. Linux-CR addresses this by extending the mechanism to explicitly discover and handle resources that escape the initial scan.

7 Conclusions

Previous work on application checkpoint and restart for Linux does not address the crucial issue of integration with the mainstream Linux kernel. We present in detail Linux-CR, an implementation of checkpoint-restart, which aims for inclusion in the mainline Linux kernel. Linux-CR provides transparent, reliable, flexible, and efficient application checkpoint-restart. We discuss the usage model and describe the user interfaces and some key kernel interfaces of Linux-CR. We present preliminary performance results of the implementation.

A key ingredient to a successful upstream implementation is the understanding by the kernel community of the usefulness of checkpoint-restart. Without this we would fail to receive from the community the invaluable review and advice upon which we have relied. That we have in fact received much such help shows that the usefulness of checkpoint-restart is recognized by the community. We therefore have high hopes that, with the community's help, the project will succeed in providing checkpoint-restart functionality in the upstream kernel.

Legal Statement

Acknowledgments

The Linux-CR project is the result of the hard work of several developers. Major contributors include Matt Helsley, Dave Hansen, Sukadev Bhattiprolu, Dan Smith, and Nathan Lynch. Additional guidance has come from Arnd Bergmann, Ingo Molnar, Louis Rilling, Alexey Dobriyan and Andrey Mirkin. Various reviewers gave countless valuable comments for the patchset. Jason Nieh provided helpful comments on earlier drafts of this paper. This work was supported in part by the DARPA under its Agreement No. HR0011-07-9-0002, by NSF grants CNS-0914845 and CNS-0905246, and AFOSR MURI grant FA9550-07-1-0527.

References

[1] Checkpoint/restart testsuite. `http://www.linux-cr.org/git/?p=tests-cr.git;a=summary`.

[2] Libvirt LXC container driver. `http://www.libvir.org/drvlxc.html`.

[3] Linux Containers. `http://lxc.sf.net`.

[4] S. Bhattiprolu, E. W. Biederman, S. E. Hallyn, and D. Lezcano. Virtual Servers and Checkpoint/Restart in Mainstream Linux. *SIGOPS Operating Systems Review*, 42(5), 2008.

[5] Berkeley Linux Checkpoint/Restart User's Guide. `http://mantis.lbl.gov/blcr/doc/html/BLCR_Users_Guide.html`.

[6] BTRFS. `http://oss.oracle.com/projects/btrfs`.

[7] ckpt. `http://pages.cs.wisc.edu/~zandy/ckpt/`.

[8] CryoPID - A Process Freezer for Linux. `http://cryopid.berlios.de`.

[9] EPCKPT. `http://www.research.rutgers.edu/~edpin/epckpt/`.

[10] EXT4. `ext4.wiki.kernel.org`.

[11] T. Group. Tcg tpm specification version 1.2 - part 1 design principles, 2005.

[12] O. Laadan and J. Nieh. Transparent Checkpoint-Restart of Multiple Processes on Commodity Operating Systems. In *Proceedings of the 2007 USENIX Annual Technical Conference*, Santa Clara, CA, June 2007.

[13] LXC: Linux container tools. `http://www.ibm.com/developerworks/linux/library/l-lxc-containers`.

[14] A. Mirkin, A. Kuznetsov, and K. Kolyshkin. Containers checkpointing and live migration. In *Proceedings of the 2008 Ottawa Linux Symposium*, July 2008.

[15] Network Appliance, Inc. `http://www.netapp.com`.

[16] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, Dec. 2002.

[17] J. S. Plank. An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance. Technical Report UT-CS-97-372, Dept. of Computer Science, University of Tennessee, July 1997.

[18] E. Roman. A Survey of Checkpoint/Restart Implementations. Technical Report LBNL-54942, Lawrence Berkeley National Laboratory, July 2002.

[19] J. C. Sancho, F. Petrini, K. Davis, R. Gioiosa, and S. Jiang. Current Practice and a Direction Forward in Checkpoint/Restart Implementations for Fault Tolerance. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 18*, Washington, DC, Apr. 2005.

[20] H. Zhong and J. Nieh. CRAK: Linux Checkpoint/Restart As a Kernel Module. Technical Report CUCS-014-01, Columbia University , 2001.

Optimizing processes on multicore for speed and latency

Christoph H. Lameter
Graphe Inc.
cl@gentwo.org

Abstract

High Performance Computing is coming to every file server and every desktop machine these days. The processing power of the average server will grow significantly with the introduction of Westwere (24 threads dual socket), Nehalem EX for quad socket (64 threads) and 8 socket machines (128 threads) which will make entirely new applications possible. In the financial market it is then possible to run a complete trading system setup on a single machine as demonstrated by running a NYSE simulation at the IDF conference by Intel. However, the same issues already show up with a smaller effect even on the run of the mill dual quad core systems.

Intel Nehalem processors support NUMA – a technology so far only known from large supercomputers. The NUMA effects are small on todays dual quad core file servers but as the number of processors rises the distances of processors to memory will also increase and put more demands on the operating system and application software to obtain memory that a processor can reach in an efficient manner. Temporal locality issues dominate even within a core because the most effective storage is in the L1 cache that is local to one execution context but unreachable from another. It is vital that techniques originally developed for HPC are used to exploit the full potential that todays hardware provides.

We will discuss Westmere, Nehalem EX, temporal and spatial locality management techniques, managing cpu caches, hyperthreading, latency and performance in Linux.

Open Source issues? Avoiding the ipo(a)ds ahead..

Christoph H. Lameter
Graphe Inc.
`cl@gentwo.org`

Abstract

Have you ever wondered why Linux does not make progress in certain areas? Why have we not conquered the desktop yet? Why are many "commercial" applications not for Linux? Why is Apple dancing circles around us with Iphones, Ipods and Ipads?

All these things require funding, a certain frame of mind that focuses oni the end user and an autonomy on the part of the developer of new applications. Open source developers are frequently caught in an evil web of pressure by employers to work on proprietary ideas which sucks off the majority of time that is available for productive work, the necessity to maintain relationships with (like-minded) open source developers working on the same projects which results in a closed mind to end users and the inability to start something new and creative with somewhat controllable effort and benefit monetarily from it so that further efforts can be made.

It seems that the ipod world has found a solution to these issues and enabled developers to create useful apps in a minimal time frame, benefit from it and grow the usefulness of their software. Sadly this world is controlled by a commercial entity, source code is not available.

The impression to the general public is that open source contribution is something like a heroic effort. A renunciation of the riches that could be had and a taking of a vow of poverty. The motives of commercial entities can be in collaboration but what we have seen the contributions are mainly driven by commercial benefits derived from open source contribution. The motivation is not to provide a well designed easy to use application to the end user.

The author thinks that we need to change the development process so that the design and creation of easily usable end user applications is rewarded and that at the same time it must be possible for others to use and modify the code created.

Deploying Preemptible Linux in the Latest Camcorder

Geunsik Lim
Samsung Electronics
geunsik.lim@samsung.com

Jupyung Lee
Samsung Electronics
jupyung.lee@samsung.com

Sangbum Suh
Samsung Electronics
sbuk.suh@samsung.com

Abstract

Currently, the Linux kernel is well equipped to compete with the soft realtime operating system. Linux has been the choices of the operating system. We adjusted optimized Linux kernel to the camcorder's system architecture which is equipped with ARM cortex-A8 and implemented open-source based tool-chain, audio zoom calculation, and realtime HDMI I2C communication and userspace realtime thread program. Samsung has introduced Consumer Electronics Show(CES) this year with its new S-Series of full HD digital camcorders. These product is the world's first commercially available camcorder which includes built-in Wi-Fi and DLNA connectivity.

This paper describes our trouble shooting, cross-compiler issues, technical experiences and best practice in reducing latency in Linux and applications for developing an embedded product like camcorder. This discussion focuses on how commercial platform can optimize the realtime extensions available in Linux kernel, but it is also relevant to any software developer who may be concerned with finding a suitable tradeoff between throughput and responsiveness for embedded systems. Furthermore, many methods which implemented to further improve the system performance will be presented as well.

1 Introduction

Since 2000, commercial RTOSs such as pSOS, LynxOS, QNX, Nucleus, Vxworks that have been used in embedded devices have been replaced by Linux kernel. For home appliances such as Digital TV, camcorder, digital camera, printer, etc., electronic companies have improved their differentiation and competitiveness by modifying open-source software for each product use since they adopted Linux which has the advantage not to require paying royalty. For mobile platform such as mobile phone, variable commercialization projects based on Linux such as Maemo, Moblin, Android, and Palm-Pre are in progress.

Particularly, when starting with Linux version 2.6, realtime functionalities[19] such as O(1) Scheduler, Threaded Interrupt Handling, Preemptible Kernel[8], Priority Inheritance, High Resolution Timer[5], Tickless Timer and Userspace Realtime Mutex[2] have been added so that they are being used for embedded devices which require responsiveness as important factor.

For example, Commercial RHEL-RT[18] that its realtime property is improved by Ingo Molnar[9] who is one of the realtime system developers, currently working for Red Hat has been applied for USS Zumwalt (DDG-1000, U.S. Navy Next-Generation Destroyer) to ensure that computers response correctly without halt & stop of computers or failure of synchronization with other events.

However, when originally development of the Linux kernel began, it was not designed for the purpose of realtime property. That is why Linux may be suitable for the system that required the soft realtime property, not the hard realtime property.

Although Linux is continuously being developed to be closer to the hard realtime property by many open-source developers[1, 23] non-preemptible critical sections and interrupt off sections still exist in the RT patched Linux kernel.

In the case of full HD camcorder, commercial camcorders have been developed by adopting RTOSs such as Vxworks, TronOS instead of Linux for the OS in order to meet realtime property. However, commercialization was attempted for the full HD camcorders from the Samsung camcorder S-Series based on the Linux 2.6.29 kernel considering the advantages that the payment for royalty is not required and the source can be freely modified.

The Samsung camcorder S-Series offer both built-in Wi-Fi and DLNA connectivity and feature Samsung AllShare that allows users to easily access, manage, and share content including their full-HD videos across other DLNA certified devices. The Samsung full HD camcorder is commercially available in the worldwide market.

2 Responsiveness VS. Throughput

The goal of Real Time Operating System(RTOS) is to provide a predictable and deterministic environment. The primary purpose is not to increase the speed of the system[6], or lower the latency between an action and response, although both of these increase the quality of a RTOS[14].

Many companies have basically adopted RT patched Linux kernel for their embedded products recently in order to meet realtime property. When running software in such a system, tradeoff between responsiveness and throughput is often found. Otherwise, responsiveness and throughput are inversely related.

The overhead for realtime preemption is applicable for these cases [16].

- Mutex operations more complex than spinlock operations
- Priority inheritance on mutex increases task switching
- Priority inheritance increases Worst-Case Execution Time(WCET)

And, flexible design allows much better worst case scenarios in embedded products.

- Realtime tasks are designed to use kernel resources in managed ways then delays can be eliminated or reduced

For example, For recording mass files, if we try to unmap memory that we allocated with 100MB for recording in camcorder, we have to wait for more than 3seconds to change mode from play mode to recording mode. This results from the unit of memory unmapped size when we are recording mass files like camcorder particularly.

When performing memory unmap with system call to virtual memory area used by user, it improves realtime property by reducing the size of non-preemptible sections associated with the unmap operation through the minimization of "ZAP_BLOCK_SIZE". However, the memory unmap operation results in delay due to the minimized "ZAP_BLOCK_SIZE". In this case, we recommend that default memory range to unmap allocated memory from 8 pages to 1,024 pages. Figure 1 shows operation comparison between the memory unmap ranges. We can not find side-effect during the sotware quality assurance test after adjusting this approach .

The unmap operations were repeated 3,328 times with 8 pages units in order to reset the 100MB of memory, and check out whether any task with higher realtime priority is waiting every time unmapping is completed in each 8 pages units for realtime property. If any task with higher realtime priority exists, the task preempts the current task and the memory unmap operation will be performed again after finishing the task. At the moment, It takes more than 3 seconds to unmap 100MB of memory with 8 pages units in our test, and the average share of CPU during the time accounted for 92% - 98%.

For a camcorder in this paper, as mass files have to be recorded, memory has to be allocated with 100MB unit and reset. When changing the boundary between responsiveness and throughput to 1,024 pages from 8 pages through repeated tests, it showed the most realistic performance. After changing the policy of memory unmap which changes minimum unit for memory unmapping to 1,024 pages, the switching time from play mode to recording mode in mass production has been improved from 3 seconds and more to less than 1 second. In our experiment, We decided that the best page size of unmap operation was 1,024 pages without the more page size like 2,048 pages and 4,096 pages to consider both throughput and responsiveness.

When unmapping 100MB that was allocated for recording video files through the Software Quality Assurance(SQA) test process before and after changing the unit in camcorders, it showed that the performance of unmap_page_range() with 1,024 page unit is effective to solve the memory unmap operation issue is delayed due to the minimized "ZAP_BLOCK_SIZE".

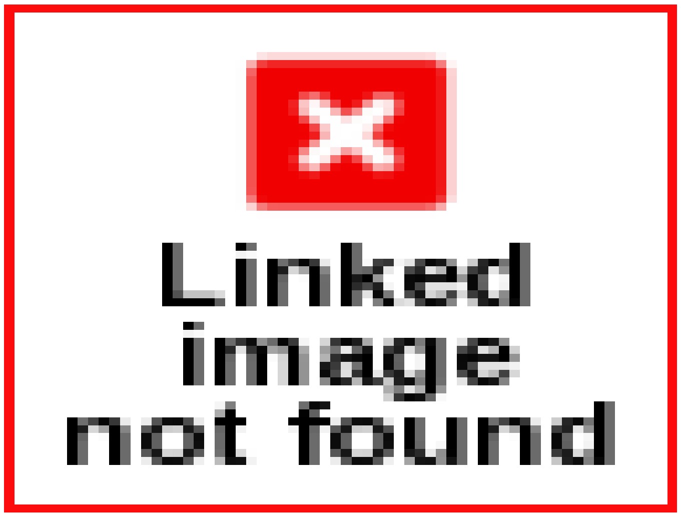


Figure 1: Operation comparison between the memory unmap ranges

3 Cross Compiler to solve low clock speed

While we prepared for the mass production of the Samsung camcorder, we tried that the clock speed for the product should be changed from 800Mhz to 600 Mhz to reduce overall power consumption of camcorder. It means that the lower clock speed is suitable for commercial product if we can. After the change of the clock speed is adjusted, Video recording test is failed on SQA test before production release because of low clock speed.

3.1 S5PC110 hardware specification

Before explaining the problem, We first presents hardware characteristics of Samsung camcorder. The S5PC110 is targeted for small form-factor connected devices such as multimedia intensive smart phones in Figure 2, while the S5PV210 is aimed at portable computing devices such as netbooks that demand high performance and design flexibility.[21]

High speed 3D graphics rendering and high resolution video support are two key differentiating features for advanced mobile devices. Both the S5PC110 and S5PV210 are equipped with a powerful built-in POWERVR SGX 3D graphics engine, licensed from Imagination Technologies, to support sophisticated 3D UI and high-caliber games. In addition, the two processors integrate a1080p full HD codec engine that supports 30fps full HD video playback and recording. A built-in HDMI1.3 interface allows output of captured or downloaded mobile multimedia contents to an external high definition digital display.

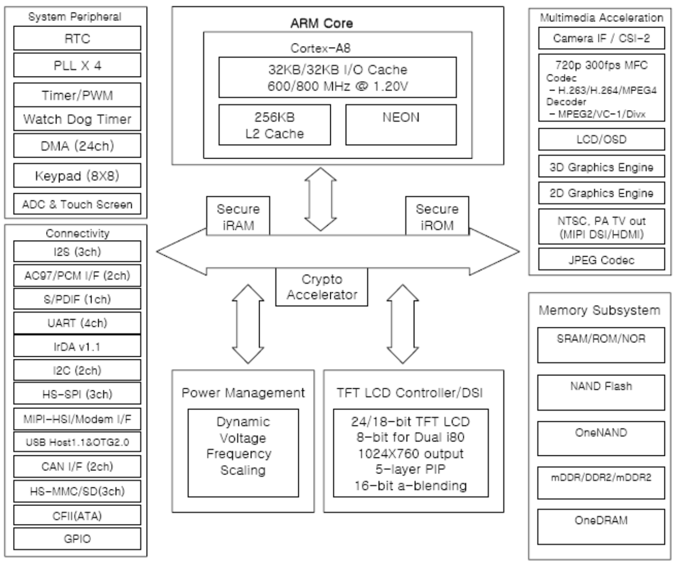


Figure 2: S5PC110(Cortex-A8) Architecture map

3.2 How to overcome low clock speed

First of all, We considered software technologies to minimize code size and optimized throughput according to the compiler option tuning, As a result, we passed 68% with GCC 4.4.3 based tuned toolchain and passed others with updated AudioZoom engine library. Fortunately, we overcomed the problem of low clock speed with software approaches to observe the development schedule that we had to work successfully for commericial product.

3.3 GCC optimization for performance & code size

This optimization can provide significant increases in performance and equally significant reductions in code size for camcorder[4].

The default inline options for camcorder are following.

```
-finline-functions
-fno-inline-functions-called-once
```

Below are options that we found via mobile software platform study like Android[22], Moblin, Maemo. Especially, Parameters setting is a key driver in performance and size optimization. We searched for a configuration that reduces size the most using compiler option search approach.

```
-finline
-fno-inline-functions
-finline-functions-called-once
--param max-inline-insns-auto=50
--param inline-unit-growth=0
--param large-unit-insns=0
--param inline-call-cost=4
```

- -fno-inline: Don't pay attention to the inline keyword. Normally this option is used to keep the compiler from expanding any functions inline. Note that if you are not optimizing, no functions can be expanded inline.

- -finline-functions: Integrate all simple functions into their callers. The compiler heuristically decides which functions are simple enough to be worth integrating in this way. Enabled at level '-O3'.

- -finline-functions-called-once: Consider all static functions called once for inlining into their caller even if they are not marked inline. Enabled at levels '-O1', '-O2', '-O3' and '-Os'.

- max-inline-insns-auto: When you use '-finline-functions' (included in '-O3'), a lot of functions that would otherwise not be considered for inlining by the compiler will be investigated. To those functions, a different limit compared to functions declared inline can be applied. The default value is 50.

- inline-unit-growth: Specifies maximal overall growth of the compilation unit caused by inlining. The default value is 30 which limits unit growth to 1.3 times the original size.

- large-unit-insns: The limit specifying large translation unit. Growth caused by inlining of units larger than this limit is limited by '–param inline-unit-growth'.

- inline-call-cost: Specify cost of call instruction relative to simple arithmetics operations(having cost of 1). Increasing this cost disqualifies inlining of non-leaf functions and at the same time increases size of leaf function that is believed to reduce function size by being inlined. The default value is 12.

3.4 Code size comparison among the compilers

Figure 3 is the comparison as to how the size of userspace library of camcorder changes by each GCC version. When compiling the source of system library by using tuned GCC 4.4.3 version, 6.60% of size reduction was obtained compared to existing GCC 4.2.0 version in Figure 3.

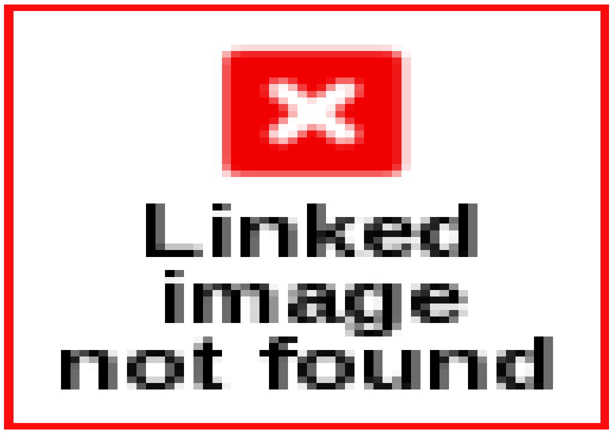


Figure 3: Code size comparison among the cross compilers

3.5 Arithmetic speed for recording functions

Below is evaluation result among the cross compilers to solve the low clock speed problem when running recording function. From the test result, the arithmetic speed of AudioZoom library engine for recoding functions by using tuned GCC 4.4.3 version was increased by 10% compared to existing performance speed in Table 1.

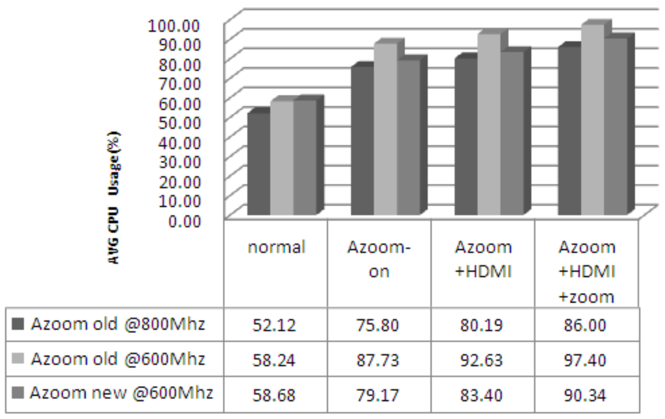
- Test environment: Preemptible Linux + Lightweight root file system for embedded device
- GCC 4.2.X option: -march=armv7 -O3 -ffast-math -fomit-frame-pointer -funroll-all-loops -pipe
- GCC 4.4.3 option: -finline -fno-inline-functions -finline-functions-called-once –param max-inline-insns-auto=50 –param inline-unit-growth=0 –param large-unit-insns=0 –param inline-call-cost=4

3.6 CPU usage comparison

AudioZoom library performs arithmetic functions and the largest computing tasks when recording video on the camcorder. Figure 4 shows CPU usage monitoring result after moving latest GCC version and optimization.

GCC Version	*CodeSourcery(Lite) 4.2.1*	*CrossTool-0.43 4.2.0*	*Montavista 5.0 4.2.0*	*Opensource(tuned) 4.4.3*
Binary Size	175,034	175,034	170,377	156,019
1st Test	12,970,182	13,138,915	13,258,427	12,060,383
2nd Test	13,030,439	13,163,295	13,212,784	12,157,852
3rd Test	13,345,300	13,038,169	13,075,400	12,279,066
4th Test	12,889,146	13,353,949	13,171,671	12,277,243
5th Test	13,000,646	13,147,129	13,158,650	12,192,311
Average	13,047,143	13,168,291	13,175,386	12,193,371
Time(normalization)	1	1.009285	1.001402	0.916656

Table 1: Performance result of AudioZoom engine among the cross compilers



	normal	Azoom-on	Azoom +HDMI	Azoom +HDMI +zoom
■ Azoom old @800Mhz	52.12	75.80	80.19	86.00
■ Azoom old @600Mhz	58.24	87.73	92.63	97.40
■ Azoom new @600Mhz	58.68	79.17	83.40	90.34

Figure 4: CPU usage comparison after improvement

3.7 SQA Test after changing cross compiler

After changing GCC version to 4.4.3, optimizing the inline function, and improving "Azoom(AudioZoom)" library engine for arithmetic process in recording, video recording for 16 SD cards from different manufacturers was normally performed in Software Quality Assurance(SQA) test though the clock speed of CPU was lowered from @800MHz to @600MHz like Table 2.

3.8 GPL license issues

As we all know, Glibc open-source community selected to encourage open-source based commercial products. Our GCC 4.4.3 based tuned toolchain consists of Glibc 2.11. Recently, Glibc 2.6.1+ is "LGPLv2+ and LGPLv2+ with exceptions and GPLv2+". Many companies want to close sources of userspace application of camcorder for competitiveness thereby selecting LGPL based Glibc open-source software. We can release our software with "GCC 4.4.3 + Glibc 2.11" because some sources of glibc are that defined with GPLV2+ are as Executable Linking File(ELF) binary for debugging and configuration setting only without shared object('x.so').

When compiling the Glibc source files that GPLv2 is applied in Glibc 2.11 source, 'x.so' files for library use in rootFS(root file system) don't exist. However, as they are ELF Binary files for debugging and test, license issue for application codes open doesn't come out when using glibc 2.6.1 version or higher to embedded product. The source that requires GPLV2 license applied for shared library('x.so') files has not been used so far, as seen in Table 3. If GPLV2 is adopted to some shared library('x.so') files of the specified version that was updated for new feature and bugfix in the future, another C library like uClibc, eGlibc, BSD C library should be considered in order to protect the application codes developed among the companies for commercial products from being disclosed.

4 Needs to move the latest Linux version

We use the term backporting to describe the situation case where we take a fix for a security flaw out of the most recent version of an upstream software package, and apply that fix to an older version of the package we release for server solutions and embedded devices like RHEL(Redhat Enterprise Linux), Montavista [26], Windlinux.

Backporting is the action of taking a certain software modification (patch) and applying it to an older version of the software than it was initially created for. It is part of the maintenance step in a software development

Maker	*Type*	*Cap*	*Write(Kb/s)*	*800Mhz*	*600Mhz*	*600Mhz*	*600Mhz*
				AudioZoom Ver 1.18	AudioZoom Ver 1.18	AudioZoom Ver 1.18	AudioZoom Ver 1.22
				GCC 4.2.0 (Montavista)	GCC 4.2.0 (Montavista)	GCC 4.4.3 (tuned)	GCC 4.4.3 (tuned)
A-DATA	SDHC	8GB	10,556	ok	fail	ok	ok
AnyFlash	SDHC	8GB	8,687	ok	fail	ok	ok
imation	SDHC	8GB	9,945	ok	fail	ok	ok
imation	microSD	8GB	5,981	ok	fail	ok	ok
inx	SDHC	8GB	5,876	ok	fail	ok	ok
Jaydisk	SDHC	8GB	8,802	ok	fail	ok	ok
Memorette	SDHC	8GB	10,343	ok	fail	fail	ok
Memorette	SDHC	8GB	7,062	ok	fail	ok	ok
Memory4U	SDHC	8GB	8,260	ok	fail	fail	ok
Samsung	SDHC	8GB	9,690	ok	fail	ok	ok
Sandisk	SDHC	8GB	8,148	ok	fail	fail	ok
Sandisk	SDHC	8GB	11,725	ok	fail	ok	ok
Sandisk	SDHC	8GB	8,148	ok	fail	fail	ok
Timu	SDHC	8GB	10,172	ok	fail	ok	ok
Toshiba	SDHC	8GB	5,873	ok	fail	fail	ok
Toshiba	SDHC	8GB	10,072	ok	fail	ok	ok

Table 2: Recording test among the 16 SD cards

process. It's important that we examine how the Linux kernel source changed recently.[27]

When considering the cost for labor and total costs for the period of development in regard to R&D for commercial embedded products, change of existing released kernel version that has been commercialized and well working into the latest version can be a risk to companies in terms of stability. For this reason, many embedded product manufacturers are hesitating to change the existing Linux kernel version to the latest version.

However, if an existing Linux kernel version is applied for new products, it has a drawback that debugged functions and new kernel features can't be used. If the fact that new Linux kernel version is actually released every couple of months is taken into account, backporting can be a practical choice as a method for considering both product stability and new kernel features.

4.1 Case study: BUG: swapper: xxxxxx Error message when booting

- Problem: "BUG:swapper: 1 task might have lost a preemption check!" message.
- Reason: When we enabled 'CONFIG_DEBUG_PREEMPT' feature, Kernel display the status of preemption mode every times When booting.
- Solution: Backporting from Linux 2.6.30-RT

4.2 Case study: Allocation error with GCC 4.4.X

- Problem: Relocation error when we compile linux-2.6.29 Sources with GCC 4.4.3 toolchain (Undefined relocation type in Linux kernel module *module.c*)

  Relocation section '.rel.text' at offset 0x1adae0 contains 1960 entries:

```
Offset          Type              Sym. Name
00000088   R_ARM_MOVW_ABS_NC   jiffies
00000090   R_ARM_MOVT_ABS      jiffies
000001f4   R_ARM_MOVW_ABS_NC   .LANCHOR
000000200  R_ARM_MOVT_ABS      .LANCHOR
000000264  R_ARM_ABS32         .text
```

- Reason: Linux 2.6.29 don't define relocation type in module.c for GCC 4.4.X
- Solution: Backporting from Linux 2.6.30

Source files under GPLV2+	*Binary file name*	*Description*
elf/ldconfig.c, elf/readlib.c	ldconfig	ELF file
elf/cache.c, elf/chroot_canon.c	ldconfig	ELF file
nscd/*.c (22files)	nscd	A Name Service Caching daemon
catgets/gencat.c	getcat	ELF file to create formatted msg catalog
locale/programs/*.c (38 files)	locale, localedef libBrokenLocale.so	iconv config file
posix/getconf.c	getconf	Binary to get glibc setting information
iconv/dummy-repertoire.c	iconv_prog, iconvconfig	Iconv ELF file, Iconv config file
iconv/iconv_charmap.c	iconv_prog, iconvconfig	Iconv ELF file, Iconv config file
iconv/iconvconfig.c	iconv_prog, iconvconfig	Iconv ELF file, Iconv config file
iconv/iconv_prog.c	iconv_prog, iconvconfig	Iconv ELF file, Iconv config file
malloc/memusagestat.c	memusagestat	ELF file
sysdeps/.../linux/nscd_setup_thread.c	nscd	To create Thread on nscd Daemon

Table 3: Glibc 2.11 source files under GPLV2+

4.3 Case study: Undefined reference to '_gnu_mcount_nc'

- Problem: We often enable 'ftrace(function tracer)' feature for tracing internal behavior of Linux. But We can't compile Linux 2.6.29 with GCC 4.4.X

```
...undefined ref to `__gnu_mcount_nc'
...undefined ref to `__gnu_mcount_nc'
.../mach-c110/built-in.o: In function
...undefined ref to `__gnu_mcount_nc'
...undefined ref to `__gnu_mcount_nc'
...more undefined ref to
`__gnu_mcount_nc' follow
make: *** [.tmp_vmlinux1] Error 1
```

- Reason: Since GCC 4.4's the name and calling convention is changed for function profiling like ftrace on ARM.

- Solution: Backporting from `http://marc.info/?l=linux-arm-kernel&m=124946219616111&w=2` to support new EABI compatible profiler *__gnu_mcount_nc*. With this patch both types are supported. Since the first submission We folded in the EXPORT_SYMBOL changes by Anand and removed the question mark after "gcc 4.4". (Using __gnu_mcount_nc was introduced in r140974 to gcc- Fix ARM EABI profiling.)

4.4 Case study: Compiler error of Linux-2.6.29-RT with SLUB features

- Problem: We can't boot Linux kernel that patched Ingo' RT-Patch with SLUB(Unqueued Allocator) features by default.

- Reason: This reason is summarized by Jonathan Corbet on Oct-03-2007. Quite a few other changes can be found in this tree. The SLUB allocator is not an option for Realtime kernels. Instead, this tree uses a modified version of the slab allocator which replaces interrupt-based single-CPU locking with a set of specific per-CPU locks.

  The global files_lock has been removed in favor of tightly-locked per-CPU lists. To help with the creation of such lists, a new locked-list type has been added. The tasklet code has been reworked for better threading, but the tasklet elimination patch is not present. There's also quite a few architecture-specific patches adding various features (such as clock_events) needed by the Realtime tree and fixing problems.

- Solution: Understanding the Realtime tree about SLUB/SLAB[11]

5 IRQ Handler

Under Linux, hardware interrupts are called IRQ's (Interrupt Requests).[17] There are two types of IRQ's, short and long. A short IRQ is one which is expected to take a very short period of time, during which the rest of the machine will be blocked and no other interrupts will be handled. A long IRQ is one which can take longer, and during which other interrupts may occur but not interrupts from the same device. If at all possible, it's better to declare an interrupt handler to be long.

When the CPU receives an interrupt, it stops whatever it's doing, saves certain parameters on the stack and calls the interrupt handler.

This means that certain things are not allowed in the interrupt handler itself, because the system is in an unknown state. The solution to this problem is for the interrupt handler to do what needs to be done immediately, usually read something from the hardware or send something to the hardware, and then schedule the handling of the new information at a later time (this is called the "bottom half") and return. The kernel is then guaranteed to call the bottom half as soon as possible – and when it does, everything allowed in kernel modules will be allowed.

How can we run IRQ Processing with preemptible linux on camcorder? The realtime for Linux patchset does not guarantee adequate realtime behavior for all target platforms. When using realtime Linux on a new platform you should expect to have to tune the kernel and drivers to provide performance that matches your specific requirements.[3]

In our case, Camcorders have to finish hard IRQ processing most rapidly after HARD IRQ need Realtime characteristics obtain CPU resources above all. And, an architect designed to work lightweight tasks as hardware about video decoding.

If we consider realtime characteristics about entire system in general, threaded hard interrupt is a good choice. But, we have to proceed Hard Interrupt case most rapidly in our Camcorder. Otherwise, our camcorder has to support realtime responsiveness assurance that don't must delay Hardware Interrupt in Figure 5.

For example, when we connect interface between Digital TV and HDMI(High-Definition Multimedia Interface), communication start between DDC(Display Data Channel) and I2C. If the delay happen when IRQ happen, abnormal data transfer often happen by timeout on Samsung Digital TV.

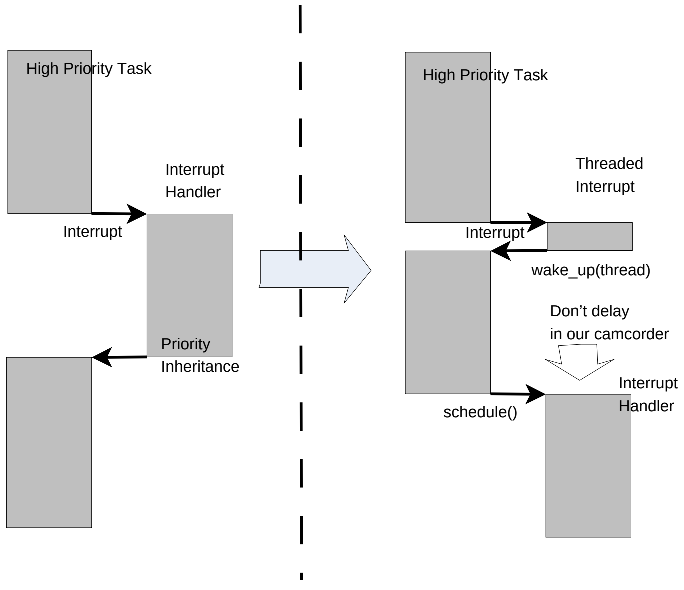


Figure 5: Area for preventing Hard IRQ Delay in Samsung camcorder

"Thread Hardirqs" option reduces the latency of the kernel by "Thread Hardirqs". This means that all hardirqs will run in their own kernel thread context. While this helps latency, this feature can also reduce performance. We don't enable "Thread Hardirqs" in our Camcorder for lightweight Hard IRQ processing at "make menuconfig" menu.[13]

We measured IRQ execution time to decide finally before we disable "Hard thread IRQ" feature in Table 4. For test, we save the execution time with do_gettimeofday( ) in asm_do_IRQ, run "`target> cat /proc/interrupts`" after modifying show_interrupt( ). At the Table 4, 'Deadline(RT-irq)' field means RT-irqs that don't have to be delayed by the higher priority task during the IRQ processing.

6 Further Enhancements

"Complete Preemption Mode" does not mix with 3rd Party's binary kernel modules in our development for commercial full HD camcorder.

We selected "Preemptible Mode" because of 3rd Party's preemptible device driver verification issues. In order to further reduce the impact of human error on driver reliability, we will consider "device drivers verification process" of third party for improved realtime characteristics as further enhancements on next products.

According to the NICTA research, Figure 6 shows a summary of their study of 500 bugs found in Linux device drivers.[12]

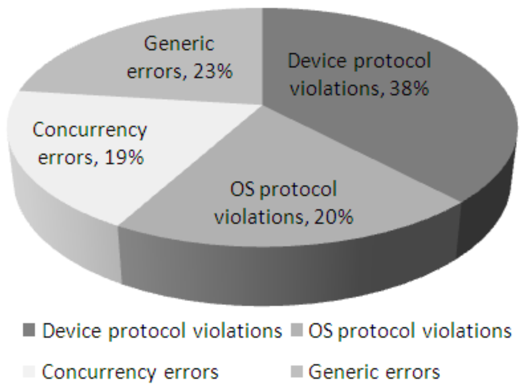


Figure 6: Summary of bugs found in Linux device drivers

We focus on three types of bugs. device protocol violations, i.e when the driver behaves in a way that violates the required hardware protocol, and concurrency bugs, i.e. race conditions and deadlocks resulting from incorrect synchronization of OS threads inside the driver[10], and OS protocol violations, i.e. situations when the driver fails to behave the way the OS expects it to. The common property of these types of bugs is that both of them are related to how the driver interacts with the OS.[20]

Where do bugs come from?

- > 70% of the kernel code is in drivers.
- Drivers contain 3x - 7x bugs per loc compared to the rest of the kernel.

Since device specifications are independent of any operating system, drivers for different systems can be synthesised from a single device specification. As a result, the likelihood of errors due to incorrect specifications will be reduced because these specifications are shared by many drivers[15].

7 Conclusions

In contrast to the origianl Linux for non-realtime environment, embedded developers often need a high level concurrency in their thread application based on realtime priority, with predictable application response to system events rapidly.

Camcorder system requires realtime operating system for deterministic control and rich operating system for running camcorder applications (e.g: wireless, tcp/ip protocol) and exploiting legacy libraries.

Linux kernel subsystem for Camcorder's recording is implemented using memory unmapping approach for realtime control of recording mode and play mode. And, We figiured out needs to move the lates Linux and GCC version according to the experience of backporting from recent version for the enhancement and bug fixes we need is always in the next revision.

Camcorder specific realtime application logic is implemented with many threads in NPTL(Native Posix Thread Library[25]) model based userspace with priority queuing, robust FUTEX[24], priority inheritance[7].

For commercial Camcorder, Linux with proper tuning showed satisfactory performance for a preemptible linux based Camcorder system including RT-patch of Ingo Molnar and others. Application of low latency and preemptible kernel patches make it possible to record video files normally.

References

[1] Real-Time Linux Wiki. Project site: `http://rt.wiki.kernel.org`.

[2] RT-mutex subsystem with PI support. Linux kernel documentation: `kernel/Documentation/rt-mutex.txt`.

[3] Sony Frank Rowand. Adventures in real-time performance tuning. In *Embedded Linux Conference*, 2008.

[4] FSF & GNU. *GCC 4.4.4 online documentation*, April 2010.

[5] Thomas Gleixner and Douglas Niehaus. Hrtimers and beyond: Transforming the linux time subsystems. In *Ottawa Linux Symposium*, 2006.

[6] Darren V. Hart. realtime linux latency comparisons. In *Ottawa Linux Symposium*, 2007.

[7] Ingo Molnar. PI-futex. people.redhat.com: `http://lwn.net/Articles/102216/`.

[8] Ingo Molnar. remove the Big Kernel Lock, this time for real. Linux kernel mailing list: `http://lwn.net/Articles/102216/`.

[9] Ingo Molnar. RT-patch. Index of /mingo/realtime-preempt:`http://www.kernel.org/pub/linux/kernel/projects%/rt/`.

[10] Jonathan Corbet. Driver porting: completion events. Linux Weekly News: `http://lwn.net/Articles/102216/`.

[11] Jonathan Corbet. What's in the Realtime tree. Linux Weekly News: `http://lwn.net/Articles/102216/`.

[12] Ihor Kuz Leonid Ryzhyk, Peter Chubb and Gernot Heiser. Dingo: Taming device drivers. In *Proceedings of the 4th EuroSys Conference, Nuremberg, Germany*, 2009.

[13] Linus Tovalds. The Linux Kernel Archives. kernel.org: `http://www.kernel.org`.

[14] Paul E McKenny. 'real time' vs 'real fast': How to choose? In *Ottawa Linux Symposium*, 2008.

[15] Microsoft. *Architecture of the kernel-mode driver framework(KMDF),2006/2007*, 2007.

[16] Montavista. *Real-time Application Programmer's Guide*, 2009.

[17] Peter Jay Salzman. The Linux kernel Module Programming Guide. TLDP: `http://tldp.org/LDP/lkmpg/2.4/html/`.

[18] Redhat. *Red Hat Enterprise MRG 1.2 Realtime Tuning Guide*, 2009.

[19] Steven Rostedt. Internals of the rt patch. In *Ottawa Linux Symposium*, 2007.

[20] Leonid Ryzhyk. On the construction of reliable device drivers. In *PhD Thesis, School of Computer Science and Engineering, University of NSW*, 2010.

[21] SAMSUNG. SAMSUNG Opens the Door to PC-Level Performance on Mobile Devices with 1Ghz Low-power Application Processors. Samsung News: `http://www.samsung.com/global/business/semiconductor/newsView.do?news_i%d=1043`.

[22] Shih-wei Liao. Smaller and Faster Android. COSCUP(Conference for Open Source Coders, Users and Promotoers): `http://coscup.org/2009/zh-tw/program/`.

[23] Thomas Gleixsner. Cyclictest. `http://rt.wiki.kernel.org/index.php/Cyclictest`.

[24] Ulrich Drepper. Futexes Are Tricky. people.redhat.com: `http://people.redhat.com/drepper/futex.pdf`.

[25] Ulrich Drepper. The native POSIX Thread Library for Linux. people.redhat.com: `http://people.redhat.com/drepper/nptl-design.pdf`.

[26] Klaas van Gend. Using realtime linux common pitfalls, tips & tricks. In *Embedded Linux Conference*, 2008.

[27] Wikipedia Contributors. Backporting terminology. Wikipedia, the free encyclopedia: `http://en.wikipedia.org/wiki/Backporting`.

IRQ No.	*Deadline(RT-irq)*	*Chip Name*	*Action Name*	*Description*
16	-	s3c-uart	s5pc100-uart	UART ch0 RX
18	-	s3c-uart	s5pc100-uart	UART ch0 TX
26	-	s3c-uart	NULL	UART ch2 TX
45	500(Passed)	s3c_vic_eint	hpd	HDMI Hot Plug Detect
50	-	sVIC	PDMA0	Physical DMA Ch-0
51	900(Passed)	VIC	PDMA1	Physical DMA Ch-1
52	-	VIC	samspi-dma	SPI DMA
53	-	s3c-timer	pwm-tick	pwm timer0 for log key
54	-	s3c-timer	pwm-tick	pwm timer1 for time-print
55	-	s3c-timer	pwm_timer2	pwm timer2 for hrt clock_source
57	-	s3c-timer	pwm_timer4	pwm timer4 for hrt tick-timer
73	-	VIC	Pata	ATA
74	-	VIC	3D	3D
76	-	VIC	HDMI	HDMI
79	-	VIC	sam-spi	spi ch 0
80	-	VIC	sam-spi	spi ch 1
81	-	VIC	sam-spi	spi ch 2
83	-	VIC	s3c2410-i2c.2	i2c ch2 for audio-DAc
88	-	VIC	s3c-udc	usb OTG
93	-	VIC	s3c-csis	MIPI
96	-	VIC	s3cfb	LCD[0]
97	500(Passed)	VIC	s3cfb	LCD[1]
101	100(Passed)	VIC	s3c-fimc0	fimc0
102	100(Passed)	VIC	s3c-fimc1	fimc1
103	100(Passed)	VIC	s3c-fimc2	fimc2
104	-	VIC	s3c-jpg	jpeg
106	-	VIC	PowerVR	ad converting
107	100(Passed)	VIC	s5p-tvout	tv mixer
109	100(Passed)	VIC	s5p-ddc	i2c ch1 for HDMI ddc
110	-	VIC	s3c-mfc	mfc
112	-	VIC	s3c-i2s-v50	i2s v50 for audio ch0
113	-	VIC	s3c-i2s-1	i2s for audio ch1
118	-	VIC	s5pc1xx_spdif	spdif
130	500(Passed)	VIC	MMC0	mmc0 for wireless lan
131	-	VIC	s5p-cec	HDMI CEC

Table 4: Deadline measurement result of IRQs that need realtime characteristic after disabling "Hard thread IRQ"
.

User Space Storage System Stack Modules with File Level Control

Sumit Narayan
Dept. of E.C.E.
Univ. of Connecticut, USA
sumitn@engr.uconn.edu

Rohit K. Mehta
E.C.S., School of Engineering
Univ. of Connecticut, USA
rohitm@engr.uconn.edu

John A. Chandy
Dept. of E.C.E.
Univ. of Connecticut, USA
chandy@engr.uconn.edu

Abstract

Filesystem in Userspace (FUSE) is a typical solution to simplifying writing a new file system. It exports all file system calls to the user-space, giving programmer the ability to implement actual file system code in the user-space but with a small overhead due to context switching and memory copies between the kernel and the user-space. FUSE, however, only allows writing non-stackable file systems. The other alternative to simplify writing file system code is to use File System Translator (FiST), a tool that can be used to develop stackable file systems using template code. FiST is limited to the kernel space and requires learning a slightly simplified file system language that describes the operation of the stackable file system. In this work, we combine FUSE with FiST and present a stackable FUSE module which will allow users to write stackable file systems in the user-space. To limit the overhead of context switching operations, we provide this module in combination with our previously developed ATTEST framework that provides ways to filter files so that only those with specific extended attributes are exported to the user-space daemon. Further, these attributes can also be exported to user-space where multiple functions can behave as stackable modules with dynamic ordering. Another advantage of such a design is that it allows non-admin users to have stackable file system implemented and mounted, for example, on their respective home directories. In our experiments, we observe that having stackable modules in user-space has an overhead of around 26% for writes and around 39% for reads when compared to the standard stackable file systems.

1 Introduction

The file system is often seen as one of the most critical part of an operating system. It handles the task of storing and organizing user files and their data on the underlying storage devices. It is comprised of very complex C kernel code which takes several months to develop and stabilize and is usually written for a particular operating system platform. The file system code must interact with the operating system's virtual file system manager to receive system calls from the user-space, with virtual memory manager for page allocation and memory management within the kernel and with the virtual device layer to communicate with the storage devices and store data. This makes the file system code remarkably complex to understand and very hard to develop. An average modern file system is comprised of around 50,000–60,000 lines of code and supports a variety of features, such as B-tree based search, flexible data extents, access control lists, extended attributes, etc. [28]. This low-level kernel code is very difficult to program and is often the origin of bugs in a storage system [9, 17]. To add any new feature in a file system, a programmer needs to have a thorough understanding and working of the file system. Apart from programming, providing support and maintenance for such large and complex file systems with several features and diverse mount options is also very hard. Thus, file system development and maintenance is always considered to be the work of the select few who have a very deep knowledge of the file system and also the operating system.

Several techniques have been suggested to simplify the process of file system development. To address the need to quickly develop and incorporate new features in an existing file system, the Linux kernel has provisions for implementing stackable file systems. Stackable file systems [21, 10] give developers a quicker way to add new features to a file system through an extensible file system interface. It reduces the complexity of developing a newer file system, in that it allows features to be added incrementally in steps instead of creating a new file system from scratch, or modifying an existing one. However, to obtain the best performance, these file systems are tightly integrated into the Linux kernel or are designed and developed to run as a kernel module, thus requiring the uphill task of understanding the kernel before starting to develop a file system. File System Translator (FiST) [27] is a file system generation tool that simplifies the task of creating stackable file systems by generating most of the code from a standard file system template. The programmer is required to provide code only for the main functionality of the file system, which is then called from another code written in FiST language and fed to the FiST file system generation tool. The resulting code can then be inserted into a live system as a loadable kernel module. However, the simplified coding now requires learning a new file system template language.

Kernel-space file systems are not always the best way to develop a file system and suffer from several drawbacks. They cannot be ported across different platforms and they also do not provide any options for non-privileged users to mount a file system. File system in User Space (FUSE) is another solution to simplify writing a file system and can be ported across different operating system platforms. It has been integrated into the Linux kernel tree and has ports available for other major operating systems. FUSE exports all file system calls within the kernel to the user-space through a simple application programming interface (API) by connecting to a daemon that is running in the user-space. FUSE provides a good way to write virtual file systems, in that the file systems do not store any data themselves. Writing a file system in user-space is several folds easier when compared to writing a kernel-space file system. FUSE also has provisions to permit non-privileged users to mount FUSE-based file systems. These user-space file systems however come with a small overhead due to context switches and memory copies made during the data transfer operations [25, 26].

As discussed above, FiST and FUSE are two very common solutions to simplify the process of writing file systems. But, as observed, performance, portability and availability to non-privileged users, all cannot be achieved together. In this work we propose a stackable FUSE architecture that will allow developing stackable file systems in user-space. And to limit the overhead due to context switching between the kernel space and user-space, we propose to combine this stackable FUSE design with ATTEST, an attribute-based storage framework that allows defining policies to filter files and thus

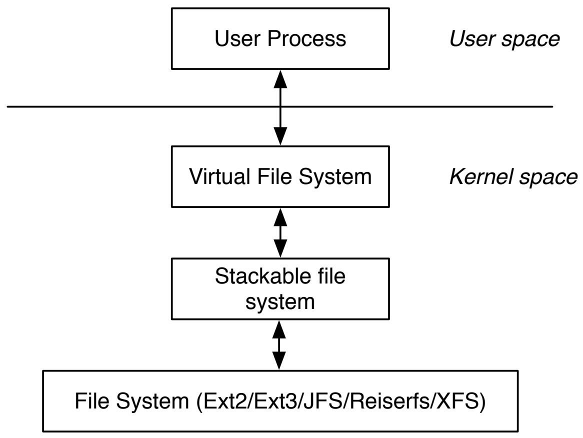


Figure 1: Stackable file systems.

file system operations on a per-file or per-directory basis [15].

The remainder of this paper is organized as follows. Section 2 discusses the stackable file system model, while Section 3 describes the FUSE architecture. Section 4 describes the previously developed ATTEST framework for an extendable storage system. In Section 5, we provide details of our design for stackable FUSE module, Section 6 gives a brief description of our implementation approach and Section 7 shows the performance results. Section 8 is an overview of the related work. We end this paper with a section on our conclusions and proposed future work.

2 Stackable File Systems

The idea of stackable or layered file systems was adapted from the *vnode* interface first implemented on SunOS in 1984 [12]. Stackable file systems [20, 21, 24, 22] are stand-alone file systems that can be mounted on top of an existing file system mount point. Figure 1 shows the typical arrangement of a stackable file system present between the Virtual File System (VFS) and a lower-level file system, which may or may not be a device-based file system. The advantage of developing a stackable file systems is that they can be used to extend the functionality of an existing file system without changing the code of the original file system. A stackable file system creates a *vnode* with its own operations that is inserted on top of the *vnode* belonging to the underlying file system. This allows a stackable file system to perform operations in between the VFS and the lower file system calls. For example, an encryption process can take place before the data is written on the lower file system, or, a decryption function can be run after the data is read from the lower file system. Stackable file systems can be used to add many functionalities such as compression, encryption, caching, etc. to an existing file system. Other examples of stackable file systems include WrapFS [26], UnionFS [18], RAIF [11], AVFS [14], etc.

fistgen is a set of File System Translator (FiST) language and tool that allows a developer to create a stackable file system by only describing the core functionalities of the file system [27]. The file system generator tool, generates the code for a file system that can be directly loaded as a kernel module into a live Linux system. To add some of the functionalities in FiST, however, requires learning a new language. Since loading a kernel module in a system is restricted to privileged users only, file systems generated using FiST can only be used if inserted into the system previously by an administrator or a privileged user.

3 File system in User space (FUSE)

Filesystem in Userspace (FUSE) is a combination of a user-space library and a kernel module for Unix-like operating systems that allows non-privileged users to create their own file systems without editing the kernel code [2]. This is achieved by running the file system code in user-space, while the FUSE module only provides a bridge to the actual kernel interfaces through a set of APIs. FUSE's kernel module simply redirects the Virtual File System (VFS) calls to the user-space daemon. Figure 2 shows the internal architecture of FUSE. Several FUSE-based file systems are already in common use. FUSE can be particularly useful in providing a POSIX interface for files which are accessible over the network through different network protocols. Some of the file systems based on such a design are *sshfs* [6], *httpfs* [3], *CurlFtpFs* [1], etc. FUSE file systems are easier to maintain since they run in user-space. They are also easier to code and debug compared to the kernel file systems. Running file system in user-space also implies access to more libraries. Thus, FUSE file systems can be written in any language that has a binding to the FUSE libraries, including Ruby and Python.

However, file systems created using FUSE are always the lowest file system in the storage stack. This means that all FUSE requests must return from the FUSE

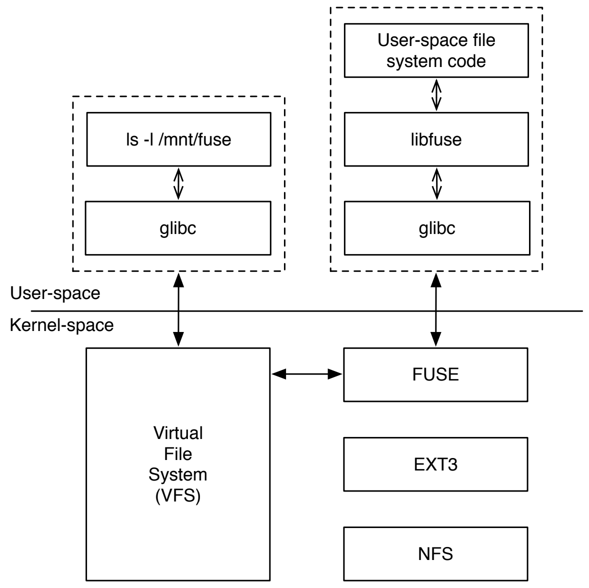


Figure 2: FUSE architecture showing kernel module and user-space library.

layer to the user-space applications without going to the lower-level file system within the kernel, similar to a stackable file system. Thus, FUSE, in its current form is not a solution for developing a stackable user-space file system.

4 ATTribute-based Extendable STorage (ATTEST)

ATTEST is an attribute-based extendable storage framework that allows policy decisions to be made at file-level granularity and at all levels of the storage stack through the use of file's, or directory's extended attributes [15]. These attributes can be used to enable or disable stackable file systems thus behaving like plugins and also allow the user to define rules to identify redundancy or throughput requirements on per-file basis to select the device for storing data. ATTEST allows user to set define file-based rules or directory-based policies that will selectively enable or disable options at each layer within the storage system stack. The rules set on a file are stored in the file's extended attributes and move with the file while directory-based policies are set on all files created under it. By allowing per-file attribute-based policies, it becomes possible to implement storage policies on a much smaller granularity. The ATTEST framework also pushes these attributes to the operating systems' storage device layer called the *logical volume manager*

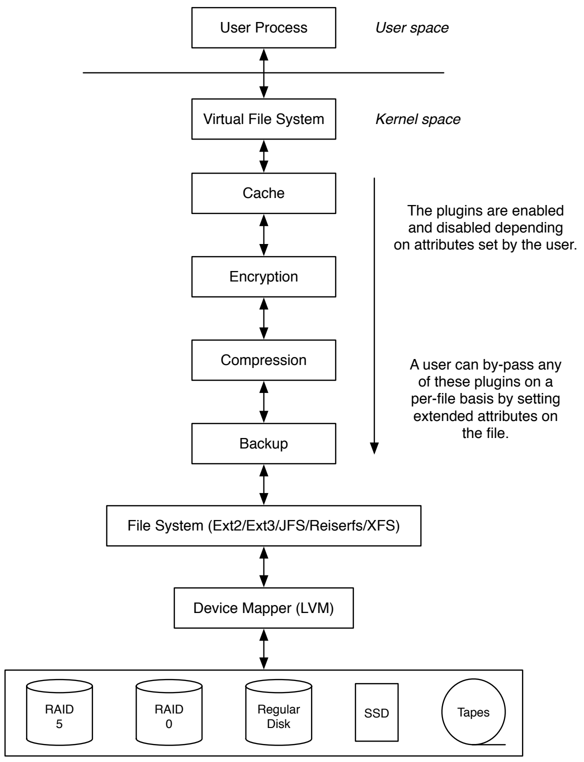

User Process
User space
Virtual File System
Kernel space
Cache
Encryption
Compression
Backup
The plugins are enabled and disabled depending on attributes set by the user.
A user can by-pass any of these plugins on a per-file basis by setting extended attributes on the file.
File System (Ext2/Ext3/JFS/Reiserfs/XFS)
Device Mapper (LVM)
RAID 5
RAID 0
Regular Disk
SSD
Tapes

Figure 3: ATTEST architecture with cache, encryption, compression, backup plugins and RAID-5, RAID-0, regular disk, SSD and tapes.

(LVM). This allows the user to also pass a file's properties and its data redundancy or throughput requirements. ATTEST framework will then select the device based on those attributes to store data blocks belonging to that file.

The ATTEST framework was designed with the objective to include computationally expensive, but necessary functionalities at the file system layer, or in the device manager layer under a single file system mount-point but with user-controlled rules or policies to determine which files or directories will really be applying them. Such a scheme would allow including functionalities in file systems which are otherwise typically ignored simply to avoid distributing their computation overhead across all files present on the file system. Typical examples of such functionalities include encryption, compression, redundancy, etc.

5 Stackable FUSE

As already mentioned, FiST and FUSE are techniques to design new file systems with lower learning curve as

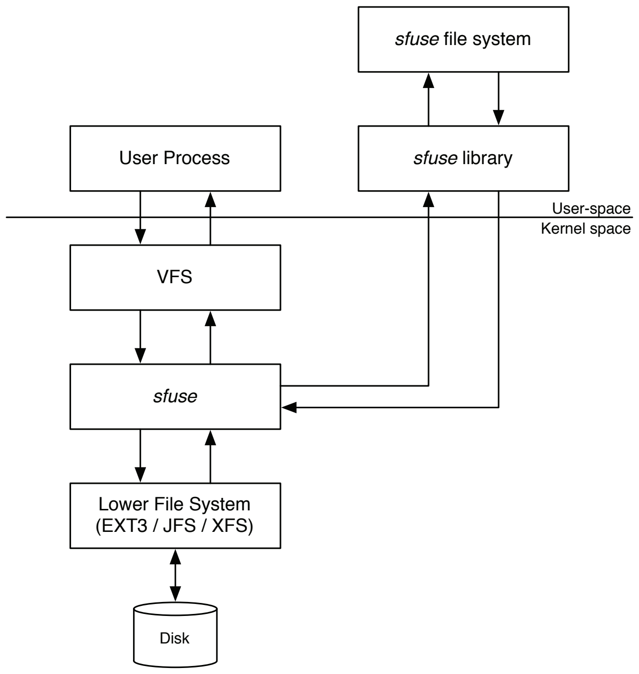


Figure 4: Operation flow in *sfuse* with a user-space *sfuse* file system.

compared to writing a standard kernel-level file system from scratch. FiST helps by extending functionalities of an existing file system while FUSE allows easy programming and maintenance of file system in the user-space. FUSE also provides the ability for non-privileged users to mount file systems and use it.

In this work, we propose a new stackable file system module called *sfuse* that will provide users with a FUSE-like interface in user-space to write their own file system in the user-space. The added advantage of *sfuse* and the difference compared to FUSE is that it will provide stackability similar to that available using FiST. Thus, data in all I/O operations will be sent to the user-space, copied or modified, and returned to kernel-space to be pushed to the lower-level file system. Since a FUSE-based file system have a cost due to context switching and memory copies, we plan to extend the idea of ATTEST and limit the overhead only to files that require the user-space functionality implemented in the file system.

There are several advantages of porting file system stackability to the user-space. One of the main advantages is that such a scheme would allow any user on the machine to mount a stackable file system without the need of administrator privileges. Along with stackability, per-file control on the files will allow the user more control on how the files are treated. *sfuse* also avoids the need for the user to understand the FiST language in developing a stackable file system. Figure 4 shows the operation flow with a *sfuse*-based file system present in the user-space. In case of READ, the data would flow away from the lower file system, while during WRITE operation, the data would flow towards the lower-level file system. Section 6 provides more detail on the internals of the *sfuse* file system module.

6 Implementation

We implemented our stackable FUSE-like file system *sfuse* on Linux kernel version 2.6.24. We started by first creating a stackable base file system *basefs* using FiST. We used a patched source of FiST version 0.2.1 to create the stackable file systems which was compatible with the Linux kernel version present on our machine. *sfuse* is designed to export all file system operations to the user-space daemon, similar to the default FUSE module. To export I/O functions to the user-space daemon, we modified the user-space FUSE library to receive requests from the kernel even without any previous file OPEN operation. All I/O requests are forwarded to the user-space file system, irrespective of whether an OPEN operation was performed on that file. This is in contrast to FUSE, where an I/O operation can be performed only after an OPEN call is made. This step is required in FUSE to open the actual file on the ported file system and obtain a file handle in the user-space. The file handle information is later used in identifying the file on which I/O needs to be performed. In *sfuse*, the file is actually opened within the *sfuse*'s Linux kernel module and multiple operations on the same file are handled within the kernel module.

A user may however also opt to also use the exported OPEN function in the user space, depending on the requirement of the stacked file system. As an example, for a simple encryption file system, encode and decode functions can be run without having any knowledge of the file handle. In another case, if the stackable file system is designed to count number of times a file is opened, OPEN functions will have to be implemented in the user-space.

To implement stackability in FUSE, we also require the user-space FUSE library to return the request data buffer back to the kernel after performing the stack function. This is done in the same way as any write operation would be performed in FUSE module. File systems developed based on *sfuse* are mounted in the same way as a FUSE-based file system. The mount binary file requires two parameters – the mount point directory and the directory which needs to have the stacked functionality on top of it. In the kernel, data structures for storing the file system's private information must include information regarding the lower-level file system along with the connection pointers of the user-space FUSE daemon.

The user can control the files which must be exported to the user-space by using ATTEST. The user can lay rules or set policies for each file by directly setting the file's extended attributes, or by including the rules in the ATTEST config file. More details on how to set the rules and policy in an ATTEST framework is explained in [15]. Stackable FUSE also allows attributes set by the ATTEST framework to be passed from the kernel-space to the user-space as tags along with any I/O request. This will allow the user to perform dynamic ordering of multiple stacked functions in the user-space without going back into the kernel-space.

Our current implementation only supports synchronous operations. This means that all operations can return to the kernel space only after the user-space functions have returned. As part of our future work, we plan to support asynchronous operations in *sfuse* library, which will allow the requests to be appended to a queue in the user-space file system. This queue will be cleaned by a thread running continuously on the system. One place where such a mechanism can be very useful is in performing lazy data backup and deduplication on a per-file or a per-directory basis. With the assistance of ATTEST rules, the user can also define policies, such as, if the files need to backed up after compression, or encryption, or neither.

7 Results

We evaluated the performance of *sfuse* by running IOZone [4], a popular benchmarking tool that performs synthetic read/write tests to determine the throughput of the system over a variety of file system configurations. We conducted our experiments on a 1.8 GHz dual-core dual processor AMD Opteron machine with 2 GB RAM and two 40 GB hard disk drives running Linux kernel 2.6.24. The experiment was run for a file size of 2 GB with record size set to 128 KB. Figure 5 shows the results with overhead of using *sfuse* compared to other file system configurations.

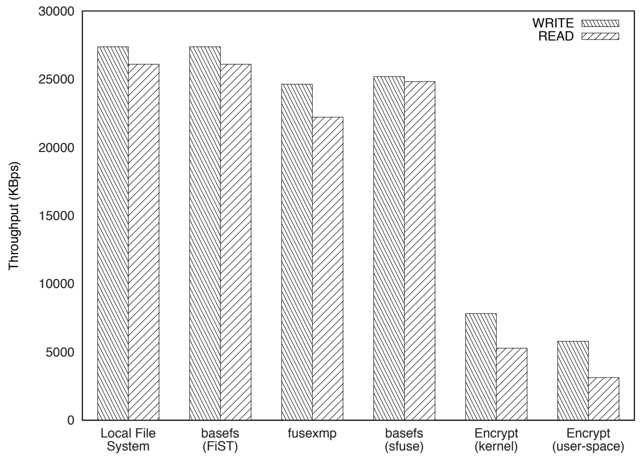


Figure 5: Throughput comparison for different setup of file system using IOZone benchmarking tool on a 2 GB file and record size of 128 KB.

We first ran IOZone on a default EXT3 formatted non-root partition to obtain the base performance of the system without any stacks. Next, we ran IOZone on *basefs*, a stackable file system generated by FiST. *basefs* is an empty file system in that it simply forwards all calls to the lower-level file system. By running IOZone on *basefs*, we evaluate the overhead of including a stackable file system between VFS and disk-based file system such as EXT3. *basefs* is available along with the source code of *fistgen*. From Figure 5, we can see that including an empty stackable file system has a very negligible overhead.

Our third experiment was run on a *fusexmp* file system mount. *fusexmp* is a FUSE-based file system that simply mirrors the root directory of the system on the mounted directory. It is available freely along with the FUSE source code [2]. Our experiments confirm the overhead that is expected to be present on any FUSE-based file system due to memory copies between user-space and the kernel-space. We observed almost 10% overhead for writes and slightly less than 15% for reads in this test. Our fourth setup was an empty file system similar to *basefs*, but set up in *sfuse*. This file system simply returned the request back to the kernel space without doing anything to the transferred data. We conducted this experiment to observe the real overhead of context switching and memory copy operations. In our experiments, we observed an overhead of around 8% for writes and around 5% for reads when compared to the local file system.

Our fifth experiment was done on a slightly modified version of *cryptfs* stackable file system available with the *fistgen* source code. We disabled the encryption of file names in *cryptfs* and only allowed data block encryption. We implemented the same encryption algorithm in user-space and implemented a *sfuse* file system for it. We observed almost 26% overhead for writes and around 39% overhead for reads by porting the code to the user-space. The overhead in this case is primarily because of context switching between the user-space and kernel space. The data buffer memory is copied two times in each I/O operation for each direction of the data flow, i.e., once from the kernel to the user-space, and then, from the user-space back to the kernel space. This overhead also comes from several other aspects within the operating system like processor registers that need to be saved and restored, cache entries that need to be evicted and reloaded for the incoming processes, etc. [13, 19].

While the overhead in our experiments are certainly non-negligible and casts doubt over the need to port file systems into the user-space, we remind the reader the benefits such as the ability for non-administrator accounts to control their data, simpler programming and debugging in user-space with FUSE bindings available in many programming languages other than C and per-file granularity control sufficient to make this a useful solution. Further, our *sfuse* code has not been highly optimized and could be improved significantly to lower this overhead.

8 Related Work

Significant effort has been put into providing users with control of their data in terms of where their data is placed in the storage system. Redundant Array of Independent Filesystems (RAIF) [11], for example, is a stackable file system that allows user to define rules to determine data placement policy. RAIF allows users to distribute data across different file systems and define redundancy across it. UmbrellaFS [8] is another solution that allows the users to define distribution policy, but across different devices with each device having their own redundancy and throughput limitation. AT-TEST differs from these existing solutions, in that AT-TEST allows rules that enable or disable stackable plugins mounted in the system by the administrator and also allows the user to define rules on data placement by selecting underlying storage devices, each with their own redundancy or throughput characteristics.

Apart from FUSE, there also exist other solutions to writing file systems in user-space. UserFS [7] was an idea proposed in 1993 which exported file system requests to the user-space through a file descriptor. *puffs* is an export of FUSE-like library on the NetBSD operating system [5].

However, all these solutions limit themselves by either allowing the user only to decide where the data is stored or not providing stackability in the file system. To the best of our knowledge, we know of no solution that provides the users a mount point interface that will allow the user to modify or copy the data in a stackable fashion without the need for having any root-privileges.

9 Conclusions and Future Work

In this paper, we have presented a stackable user-based file system model which can be controlled based on user defined rules. Most of the existing file systems make compromise on adding costly functionalities because there is no way to make policy decisions at a finer granularity. In this work, we reinforce our commitment to providing users more control over policy decisions on the files by using ATTEST. Stackable file systems in user-space opens up a variety of opportunities to design file systems. However, the stackability of these file systems are expensive due to context switching. By using the ATTEST framework, we can select files and enable stackability and absorb the overheads only for files that require the stackable functions enabled.

In the future, we plan to implement a stackable module in the LVM layer that will send data blocks present at the disk level to the user-space daemon in a similar fashion. This can be used to write user-spaced disk block manipulation functions such as data deduplication or snapshotting. Such a design will also allow users to write their own disk layout algorithms [23, 16] from the user space.

References

[1] CurlFtpFS - a FTP filesystem based on cURL and FUSE. `http://curlftpfs.sourceforge.net/`.

[2] FUSE: Filesystem in user space. `http://fuse.sourceforge.net`.

[3] HTTP filesystem. `http://httpfs.sourceforge.net/`.

[4] IOZone. `http://www.iozone.org`.

[5] puffs - pass-to-userspace framework file system. `http://www.netbsd.org/docs/puffs/`.

[6] SSH filesystem. `http://fuse.sourceforge.net/sshfs.html`.

[7] Jeremy Fitzhardinge. Userfs. `http://www.goop.org/~jeremy/userfs/`.

[8] John A. Garrison and A. L. Narasimha Reddy. Umbrella file system: Storage management across heterogeneous devices. In *ACM Transactions on Storage*, volume 5, New York, USA, March 2009.

[9] Jim Gray. A census of tandem system availability between 1985 and 1990. *IEEE Transactions on Reliability*, 39(4):409–418, October 1990.

[10] John S. Heidemann and Gerald J. Popek. File system development with stackable layers. In *ACM Transactions on Computer Systems*, volume 12, pages 58–89, February 1994.

[11] Nikolai Joukov, Arun M. Krishnakumar, Chaitanya Patti, Abhishek Rai, Sunil Satnur, Avishay Traeger, and Erez Zadok. RAIF: Redundant array of independent filesystems. In *IEEE Conference on Mass Storage Systems and Technologies*, pages 199–214, San Diego, CA, September 2007.

[12] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proceedings of the Summer USENIX Technical Conference*, pages 238–247, Atlanta, GA, 1986.

[13] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *Workshop on Experimental Computer Science*, San Diego, CA, 2007.

[14] Yevgeniy Miretskiy, Abhijith Das, Charles P. Wright, and Erez Zadok. Avfs: An on-access anti-virus file system. In *13th USENIX Security Symposium*, San Diego, CA, August 2004.

[15] Sumit Narayan and John A. Chandy. ATTEST: ATTributes-based Extendable STorage. *Journal of Systems and Software*, 83(4):548–556, April 2010.

[16] James A. Nugent, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Controlling your PLACE in the file system with gray-box techniques. In *Proceedings of the Annual USENIX Technical Conference*, pages 311–323, San Antonio, TX, June 2003.

[17] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON file systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 206–220, Brighton, UK, October 2005.

[18] David Quigley, Josef Sipek, Charles P. Wright, and Erez Zadok. UnionFS: User- and community-oriented development of a unification file system. In *Proceedings of 2006 Ottawa Linux Symposium*, pages 349–362, Ottawa, Canada, June 2006.

[19] Aditya Rajgarhia and Ashish Gehani. Performance and extension of user space file systems. In *Proceedings of the 25th ACM Symposium on Applied Computing*, Sierre, Switzerland, March 2010.

[20] David S. H. Rosenthal. Evolving the vnode interface. In *Proceedings of the USENIX Technical Conference*, pages 107–118, Anaheim, CA, Summer 1990.

[21] David S. H. Rosenthal. Requirements for a "stacking" vnode/VFS interface. Technical Report SD-01-02-N014, UNIX International, 1992.

[22] Josef Sipek, Yiannis Pericleous, and Erez Zadok. Kernel support for stackable file systems. In *Proceedings of 2007 Ottawa Linux Symposium*, Ottawa, Canada, June 2007.

[23] Jun Wang and Yiming Hu. PROFS: Performance-oriented data reorganization for log-structured file system on multi-zone disks. In *Proceedings of IEEE/ACM 9th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 285–293, Cincinnati, OH, August 2001.

[24] Erez Zadok and Ion Badulescu. A stackable file system interface for Linux. In *LinuxExpo 99*, pages 141–151, May 1999.

[25] Erez Zadok, Ion Badulescu, and Alex Shender. Cryptfs: A stackable vnode level encryption file system. Technical Report CUCS-021-98, Computer Science Department, Columbia University, June 1998.

[26] Erez Zadok, Ion Badulescu, and Alex Shender. Extending file systems using stackable templates. In *Proceedings of the Annual USENIX Technical Conference*, pages 57–70, Monterey, CA, June 1999.

[27] Erez Zadok and Jason Nieh. FiST: A language for stackable file systems. In *Proceedings of 2000 USENIX Annual Technical Conference*, pages 55–70, San Diego, CA, June 2000.

[28] Erez Zadok, Vasily Tarasov, and Priya Sehgal. The case for specialized file systems, or, fighting file system obesity. *;login: The USENIX Magazine*, 35(1):38–40, February 2010.

expect-lite

Automation for the rest of us

Craig Miller
Ciena
cvmiller@gmail.com

Abstract

Developers can always use a tool that will save money and keep the boss happy. Automation boosts efficiency while skipping drudgery. But how to implement automation for the rest of us without slowing down the real work of developing software? Introducing expect-lite. Written in expect, it is designed to directly map an interactive terminal session into an automation script. As easy as cutting and pasting text from a terminal window into a script, and adding '>' and '<' characters to the beginning of each line with advanced features to take you further. No knowledge of expect is required!

In this paper, you'll get an introduction to expect-lite, including applications where complex testing environments can be solved with just a few lines of expect-lite code. Although expect-lite is targeted at the software verification testing environment, its use is not limited to this environment, and it has been used world-wide for several years in router configuration, Macintosh application development, and FPGA development. expect-lite can be found at: `http://expect-lite.sf.net/`

1 Introduction

expect-lite was born out of the need to boost efficiency and skip drudge work. While written in Expect, no knowledge of expect is required. In fact, expect-lite was created to avoid scripting in Expect.

Because Expect is an interpretive language, quite often there are lines which are not executed, except when something goes wrong. There is a chance that a typo, or some other error may cause the Expect script *crash*, printing a call trace. If that script is part of a larger framework, it is possible the overnight regression, for example, will come to a screeching halt.

expect-lite avoids this problem in its simplicity. By using simple characters such as > and <, it is near impossible to have a typo, or syntax error.

2 Problem

The problem is that developers need to test software, without having to think about an entire automation framework and yet another specialized language. expect-lite was developed out of a need to automate testing, and make it as easy as possible.

2.1 History

expect-lite is based on the industry standard Expect language, a TCL extension written by Don Libes of NIST (National Institute of Standards and Technology in the U.S.). It was used and improved internally from 2005 to 2007 when Freescale Semiconductor graciously allowed it to become an open source project. In 2008, over 24,000 lines of expect-lite code were running daily as part of a nightly regression.

2.2 Licensing

Because of the open nature of the foundation software TCL (BSD-Style License) and Expect (Public Domain), this open source project is licensed under the BSD-Style License. Any modifications need not be re-integrated into the open source project. Even commercial products can be created using this software, as long as the copyright header remains intact.

This seemed like the right thing to do, since the underlying software is licensed under less restrictive terms.

3 Features

expect-lite has been expanded over its five years of use, starting with the simple **send** and **expect** constructs ('>' and '<' respectively), and adding additional functionality, such as if statements, looping, multi-session support, and instant debugging.

The following subsections will high-light some of the more important features which give expect-lite the power to solve complex automation problems with minimal lines of script.

3.1 Remote Login

expect-lite was born into a multi-host environment, which created an early requirement to remote login to a host (usually a Linux machine). The remote host was allocated by a sharing facility such as the commercial package, LSF. Additionally, the remote host must be specified on the command line where the script will wake up and begin executing. Three methods of remote login are supported:

1. telnet
2. ssh with password
3. ssh with keys (no password)

ssh with keys is the preferred method, since no password is required, however this requires some environment setup before hand.

Even when no remote host is required, it is best to log into the localhost (shown in yellow in Figure 1) since the underlying Expect has problems with synchronization (between send and expect strings) when a remote host is not specified. Therefore it is possible to setup the localhost with ssh keys using the provided shell script *setup_local_ssh.sh*.

3.2 Special Character Sequences

expect-lite interprets special characters at the beginning of each line in the script as shown in Table 1.

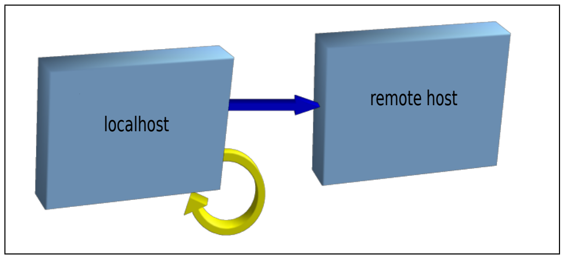


Figure 1: Connection to remote host (blue) and localhost (yellow) .

Table 1: Special Characters

Char	Action
>	send string to the remote host
>>	send string to remote host, without waiting for prompt
<	string/regex MUST be received from the remote host in the alloted timeout or the script will FAIL!
<<	literal string MUST be received (similar to 'lessthan' without regex evaluation)
-<	if string/regex IS received from the remote host the script will FAIL!
#	used to indicate comment lines, and have no effect
;	are also used to indicate comment lines, but are printed to stdout (for logging)
;;	similar to above, but no extra newlines are printed (useful for printing script help)
@num	changes the expect timeout to number of seconds
$var=	static variable assignment at script invocation
+$var=	dynamic variable assignment
+$var	increment value of $var by 1 decimal
-$var	decrement value of $var by 1 decimal
=$var	math functions, perform bitwise and arithmetic operations: << >> & | ^ * / % + -
!	indicates an embedded expect line
?	c-style if/then/else in the format ?cond?action::else_action
%	label - used for jumping to labels
~filename	includes a expect-lite automation file, useful for creation of common variable files, or 'subprograms/subroutines'

3.3 Regular Expression (RegEx) Evaluation

The full power of TCL/Expect RegEx is available to expect-lite. Thus permitting complex expect statements, with the following limitations:

1. Support of RegEx in standard expect (e.g. including anchors, char-classes and repeats)
2. Support of RegEx meta characters (e.g. \t \n are supported, \d is not)
3. expect-lite only evaluates lines using RegEx which begin with '<' '-<' and '+' (dynamic variables)

3.4 Static and Dynamic Variables

Variables are always prefaced with the $. Variables bound at script invocation are considered static, and those which are bound during execution of the script are dynamic. Static variables are assigned with the $ as the first character of a line, such as:

```
$five=5
```

Dynamic variables are assigned using RegEx capture and begin with +$. For example, text output from a command may captured into a variable, such as:

```
>hostname
+$host=\n([a-z]+)
```

If, however, the dynamic var is not successfully captured, expect-lite will print:

```
Assigned Var: \
   somevar=__NO_STRING_CAPTURED__
```

3.5 Constants

Constants are a very powerful feature of expect-lite. Constants permit writing a generalized script and overriding internal variables in the script with command line specified constants. They are are immutable, and cannot be changed. For example the constant local_eth can be defined on the command line as:

```
expect-lite remote_host=remote-host-018 \
   cmd_file=basic_ping_test.elt \
   local_eth=eth2
```

All references to $local_eth in the script will be set to eth2. This allows one to change the behaviour of the script without requiring a script change.

3.6 Plays well with Bash

Although not limited to working with bash, bash is invoked upon logging into the remote host, and therefore will be discussed more here.

Since the bash shell is well documented, and supported, it can be leveraged to assist in expect-lite's limitations such as looping and branching. A simple bash loop inside expect-lite can be created for example:

```
$set=1 2 3 4 5
>for i in $set #expanded by expect-lite
>{
>echo $i #expanded by bash
>}
```

In the above example, $set is an expect-lite variable, not a bash variable. An expect-lite variable is always declared before its use (e.g. $set = 1 2 3 4 5). If a variable cannot be dereference by expect-lite it is passed to the shell. The loop will execute after the final "}" line is sent to the remote-host.

Using Bash to create executable expect-lite scripts, it is possible to make expect-lite scripts executable, with the help of a small embedded bash helper script. Insert the following at the top of the expect-lite script:

```
#!/usr/bin/env bash
# make this auto-runnable!
# Little bootstrap bash script to run \
#  kick start expect-lite script
# Convert --parms to expect-lite \
#  param=value format
PARAMS=`echo $* | /bin/sed -r \
's;--([a-z]+) ([0-9a-zA-Z]+);\1=\2;g'`

echo $PARAMS
expect-lite r=none c=$0 $PARAMS
exit $?
\
```

The bash scriptlet does the following:

- Converts command line arguments from –arg value to arg=value format, making the script more linux-like

- Call the expect-lite script with default arguments (in this example that is r=none) and converted arguments
- Exit with the same exit code of the expect-lite script (0 success, 1 failure)

expect-lite ignores the bash scriptlet when invoked, because none of the lines begin with expect-lite special characters.

3.7 Include Files

Include files are a quick way to develop script snippets which can be included into larger scripts or to include a common variable file. When an include file is executed, it is as if the file were just pasted into the script file, and therefore has access to the variable space of the main script, and can modify that variable space as well. In this example, a common variable file is "sourced":

```
# Source common variable file
~asic_vars.inc
```

Common functions, such as telnet'ing to the DUT, are a good use of include files:

```
; === Connect to DUT
~dut_connect.inc
```

Include filenames can also be assigned in a variable, such that the file names can be declared at the top of the script but used later within the script. For example:

```
# Source Var file to be used
$asic_include=asic_vars.inc
...
~$asic_include
```

3.8 Not Expect

A test can also fail should text appear that is unexpected (such as an error). This does not clear the expect input buffer, and should be used before a valid expect. How long should the script wait for the un-expected? In order to reduce delays in script running time, the Not Expect feature only waits for 100ms. This is usually enough time to detect quick error responses such as "file not found".

For example: Fail device doesn't exist

```
>ls -l /dev/linux
-<No such file
```

3.9 If Statement

Conditional (if/then/else) statements are natively supported in expect-lite. The conditional uses a c-style syntax with a question mark (?) at the beginning of the line, and double colon to indicate the else statement, using the format ?cond?action::else_action

Although spaces are not required around the conditional characters (? and ::), it is recommended for ease of reading. The comparison operators are: '==', '!=' ,'>=', '<=', '>' and '<'. If the compared values can be evaluated as a numbers, then larger and less than will yield expected results. A simple conditional example:

```
$age=56
?if  $age >= 55 ? \
  >echo "freedom at 55!" :: \
  > echo "keep working!"
```

In the above example if $age is larger than or equal to 55 then the action 'echo "freedom at 55!". If $age is less than 55 then the action 'echo "keep working!"' will be sent.

3.10 Looping with Conditionals & Labels

Conditionals are limited to a single line. Sometimes this is too limiting, as it would be nice to have several commands be executed based on the success of a conditional. To support this, the concept of labels has been introduced. A label is defined as having the first character a '%'. Although the label line itself does nothing, it provides a location to the conditional to Jump To Label.

Simple looping is now supported by allowing jump to label backwards. The Repeat Loop is the easiest loop to implement:

```
; ======== Incrementing Loop ========
$max=5
$count=3
%REPEAT_INC_LOOP
```

```
    >echo $count
    # increment variable
    +$count
?if $count <= $max ?%REPEAT_INC_LOOP
```

Because of jump to label can jump backwards, it is important to assign unique looping labels, such as %REPEAT_INC_LOOP. Unexpected results will occur if non-unique loop label names are used. Non-looping labels, as illustrated in the previous section, are not required to be unique.

Also included in the above example is incrementing an expect-lite variable: +$count This will add 1 to the value of $count. If $count is not an integer, the value of $count will remain unchanged (can't add 1 to a string).

As part of the looping enhancement, there is infinite loop protection. The maximum amount of looping is defined in expect-lite itself with the variable _el_infinite_loop. This value is decremented with each iteration of all loops for the entire script. Typically this would be in the range of 100 to 1000 to be safe. For example, if a complex expect-lite script had 4 loops each with 100 iterations, the _el_infinite_loop should be set larger than 400.

3.11 User Defined Prompt */prompt/

With each '>' command an implied "wait for prompt" occurs. The predefined prompts are based on standard shell prompts (>%$#). However, it is quite possible that expect-lite will not be interacting with a shell, but another application (such as gdb) or device which does not issue a shell-like prompt.

As of version 3.1.5, the user may define a custom prompt using the following command:

```
*/my_prompt /
```

The succeeding '>' lines will interpret a prompt as the standard shell prompts and 'my_prompt '. The user defined prompt definition between the slashes is interpreted as regex, and therefore regex rules such as escaping applies. For example, to set a user defined prompt for gdb the following would be used:

```
*/\(gdb\) /
```

Only one (1) user defined prompt can be active at a time, however multiple prompts can be represented using the regex OR '|', for example, creating a gdb prompt and my_prompt:

```
*/\(gdb\) |my_prompt /
```

Clearing a user defined prompt. There may be times where a previously user defined prompt is causing problems by output which falsely triggers the user defined prompt. It is possible to clear the user defined prompt by:

```
*//
```

The scope of the user defined prompt is global, that is, it extends to the main script as well as all include files referenced.

3.12 Multiple Sessions *FORK

Until version release 3.5.0, expect-lite has been intentionally limited to a single session keeping it simple. However, there are certain environments where it might be advantageous for a single script to control/monitor both a client (e.g wget) and a server (httpd log). Without multiple session support this type of environment would be difficult to automate with expect-lite.

What is a new session? A new session starts with a new shell on the remote host. All commands '>' and received text '<' are constrained to that session. It is possible to use multiple sessions on the localhost using r=none, however r=none support is limted due to timing issues. If multiple sessions on the localhost (the same host where expect-lite is running), it is better to ssh to the localhost by specifying r=localhost. This loop back method ensures that the timing between commands and received text stay in sync.

With the version of 3.5.0, expect-lite supports nearly limitless multiple sessions. However, it is a good guideline to use as few sessions as needed. For backward compatibility, the first session is the "default" session. Additional sessions are assigned a name at invocation with the *FORK <session_name> directive:

```
*FORK Server
INFO: FORK session is: Server
```

expect-lite will print an "INFO" line stating the current session name Session names may not contain spaces, and must be unique for each session. To switch back to a previously started session, re-use the session name. The session name "default" (without quotes) is reserved for the first session.

```
*FORK default
INFO: FORK session is: default
```

Print the current session name by using the *FORK with no session name:

```
*FORK
INFO: FORK session is: Server
```

*FORK and variables. It is possible to assign a session name to a variable, and then create a new session with that name. For example:

```
$my_session=Client
*FORK $my_session
INFO: FORK session is: Client
```

All sessions will be automatically closed when the script terminates.

4 Debugging your scripts

Although expect-lite is designed to implement automation as easily and quickly as possible, it is possible that by using one of the more complex features, some script debugging may be required.

4.1 Interact

Interact may be the quickest, easiest, and overall best debugging aid. Interact is a mode which turns control of the keyboard over to the user, so that one may type directly to the process on the remote host With version 3.5.0 there are two methods to invoke Interact: programmatic, and instant-interact.

Programmatic Interact is called in the script with the following command:

```
*INTERACT
```

expect-lite will pause at this point in the script, and connect the keyboard to the remote session (which may be at a prompt). Any command may be entered and responses observed. Typing '+++' (3 pluses) will return control to the script, and it will continue. This is very helpful for automation assist, allowing the script to perform complicated setup commands, before turning control over to the user for an interactive session.

The other method is instant-interact. This feature requires a tcl package TclX to be installed, and will automatically be enabled when the package is present. With the feature enabled, the user can press '^\' (control+backslash) at anytime and enter Interact. This is the easiest and fastest way to debug a script.

4.2 Debugging with the el_shell

To make debugging of scripts even easier, both methods of interact support a limited expect-lite "shell". In this "el_shell", expect-lite script commands can be typed, it is possible to even assign variables inside the paused script, for example:

```
$MYVAR=today

*SHOW VARS
Var:0     Value:0
Var:MYVAR Value:today
Var:TEST  Value:/proc/cpuinfo

>>pwd
pwd
/home/user
```

In the previous example, $MYVAR is assigned, all variables their respective values are displayed, and the 'pwd' command is sent.

5 General Tips for writing scripts

Although expect-lite is designed to be simple, there are a few things to watch out for when writing a script. Here are some simple tips which will make script writing go more smoothly:

1. Use reasonable timeouts, if 30 seconds is needed to get a response, set the timeout at 45 or 60 seconds, not 600.

   - There is no cost to changing the timeout, timeout values can also be variables

2. Beware of expect-lite using regex, when creating lines such as: `<0.005 secs (5 micro secs)`

   - The parentheses is used by the regex engine, instead escape these characters: `<0.005 secs \(5 micro secs \)`
   - or use '<<' which does not use regex, and does not require escaping: `<<0.005 secs (5 micro secs)`

3. Use the expect character '<' or '<<' often. Check for valid results when possible. A script which expects nothing will never fail!

4. Use printable comments ';' often. Think of it as writing a note to oneself, it will make reading log files much easier.

6 Example Script

Problem: you need to write an automated script which monitors an ethernet port for incoming packets (using tcpdump) requiring sudo privilages, while generating network traffic (a web request).

Solution: generate a request with wget and use tcpdump to verify the return packet

In this script, two sessions are created with the `*FORK` command, one for the packet capture, and another for the wget request. The tcpdump output is filtered with egrep to remove packets of non-interest, and limit the hex packet output to 8 lines. Using two sessions permits the monitoring of network traffic while generating the wget request.

After setting up the monitor, and generating the wget request, the script switches back to the `tcpdump` session to verify the client and server id strings. If the strings are correct, then the script prints texttt0verall Result: PASS.

```
#
#       OLS 2010 Example Script
# Script uses sudo, and tcp dump to
# monitor network interface while
# initiating web traffic with wget
#
# set timeout to 10 seconds
@10
$SUDO_PASS=mysudopassword
$WEB_SERVICE=http://www.w3.org/

# start packet capture
*FORK tcpdump
>echo $SUDO_PASS | sudo -S tcpdump \
 -i eth0 -s 256  -e -X portrange 80 \
  | egrep -A 8 'www'
# force expect-lite to wait
# until tcpdump is ready
<\nlistening on
<link-type

# initiate web request with wget
*FORK wget
; === get page without saving a copy, \
    show HTTP headers
>wget -S --spider $WEB_SERVICE
# verify server in wget output
<<Server: Apache
>

# check packets captured
*FORK tcpdump
# verify client
<User-Agent
<Wget
# verify server
<Server
<Apache/2
>>^C

#*INTERACT
>>
>
```

Note near the `*INTERACT` command near the end of the script. It was used to initially pause the script inside the `tcpdump` session during development and is now commented out.

The output of the script appears in Figure 2.

7 Summary

Creating automation scripts is as easy as cutting and pasting text from a terminal window into a script, and adding '>' and '<' characters to the beginning of each line. Advanced features take you even further. No knowledge of expect is required!

Although expect-lite is targeted at the software verification testing environment, it is not limited to this environment, and has been used world-wide for several years in router configuration, Macintosh application development, and FPGA testing.

expect-lite really is *Automation for the rest of us*.

Figure 2: Script output

```
cvmiller@sawtoothy:~/Expect-lite$ ./expect-lite r=halaconia c=test_sudo.elt

INFO: Instant-Interact '^\' feature is enabled

spawn ssh halaconia

Setting Expect Timeout to: 10

INFO: FORK session is: tcpdump, Active sessions are: default tcpdump

cvmiller@halaconia:~$
cvmiller@halaconia:~$
cvmiller@halaconia:~$ export PS1='$ '
$ set prompt = "$ "
$ echo mysudopassword | sudo -S tcpdump -i eth0 -s 256 -e -X portrange 80 | egrep -A 8 'www'
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 256 bytes


INFO: FORK session is: wget,    Active sessions are: default tcpdump wget

 === get page without saving a copy, show HTTP headers

cvmiller@halaconia:~$
cvmiller@halaconia:~$ export PS1='$ '
$ set prompt = "$ "
$ wget -S --spider http://www.w3.org/
--11:02:01--  http://www.w3.org/
           => `index.html'
Resolving www.w3.org... 128.30.52.51, 128.30.52.53, 128.30.52.54, ...
Connecting to www.w3.org|128.30.52.51|:80... connected.
HTTP request sent, awaiting response...
  HTTP/1.1 200 OK
  Date: Mon, 03 May 2010 15:02:01 GMT
  Server: Apache/2
  Content-Location: Home.html
  Vary: negotiate,accept
  TCN: choice
  Last-Modified: Mon, 03 May 2010 14:51:30 GMT
  ETag: "6dfe-485b1ba692080;89-3f26bd17a2f00"
  Accept-Ranges: bytes
  Content-Length: 28158
  Cache-Control: max-age=600
  Expires: Mon, 03 May 2010 15:12:01 GMT
  P3P: policyref="http://www.w3.org/2001/05/P3P/p3p.xml"
  Connection: close
  Content-Type: text/html; charset=utf-8
Length: 28,158 (27K) [text/html]
200 OK

$
```

```
INFO: FORK session is: tcpdump, Active sessions are: default tcpdump wget

11:02:01.093847 00:11:24:e1:db:c8 (oui Unknown) > 00:60:08:12:8f:95 (oui Unknown), ethertype
 IPv4 (0x0800), length 74: halaconia.hoomaha.net.57016 > stu.w3.org.www: S
  1982196051:1982196051(0) win 5840 <mss 1460,sackOK,timestamp 22265167 0,nop,wscale 6>
        0x0000:  4500 003c 541c 4000 4006 2740 0a01 010e  E..<T.@.@.'@....
        0x0010:  801e 3433 deb8 0050 7625 e953 0000 0000  ..43...Pv%.S....
        0x0020:  a002 16d0 74ac 0000 0204 05b4 0402 080a  ....t...........
        0x0030:  0153 bd4f 0000 0000 0103 0306            .S.O........
11:02:01.130855 00:60:08:12:8f:95 (oui Unknown) > 00:11:24:e1:db:c8 (oui Unknown), ethertype
IPv4 (0x0800), length 74: stu.w3.org.www > halaconia.hoomaha.net.57016: S
3973601193:3973601193(0) ack 1982196052 win 5792 <mss 1460,nop,nop,timestamp 3243392802 22265167>
        0x0000:  4500 0038 0000 4000 3206 8960 801e 3433  E..8..@.2..`..43
        0x0010:  0a01 010e 0050 deb8 ecd8 57a9 7625 e954  .....P....W.v%.T
        0x0020:  9012 16a0 46e2 0000 0204 05b4 0101 080a  ....F...........
        0x0030:  c152 3f22 0153 bd4f de71 7b8c            .R?".S.O.q{.
11:02:01.130944 00:11:24:e1:db:c8 (oui Unknown) > 00:60:08:12:8f:95 (oui Unknown), ethertype
IPv4 (0x0800), length 66: halaconia.hoomaha.net.57016 > stu.w3.org.www: . ack 1 win 5840
<nop,nop,timestamp 22265176 3243392802>
        0x0000:  4500 0034 541d 4000 4006 2747 0a01 010e  E..4T.@.@.'G....
        0x0010:  801e 3433 deb8 0050 7625 e954 ecd8 57aa  ..43...Pv%.T..W.
        0x0020:  8010 16d0 5e66 0000 0101 080a 0153 bd58  ....^f.......S.X
        0x0030:  c152 3f22                                .R?"
11:02:01.131988 00:11:24:e1:db:c8 (oui Unknown) > 00:60:08:12:8f:95 (oui Unknown), ethertype
IPv4 (0x0800), length 165: halaconia.hoomaha.net.57016 > stu.w3.org.www: P 1:100(99) ack 1 win
5840 <nop,nop,timestamp 22265176 3243392802>
        0x0000:  4500 0097 541e 4000 4006 26e3 0a01 010e  E...T.@.@.&.....
        0x0010:  801e 3433 deb8 0050 7625 e954 ecd8 57aa  ..43...Pv%.T..W.
        0x0020:  8018 16d0 bfe9 0000 0101 080a 0153 bd58  .............S.X
        0x0030:  c152 3f22 4845 4144 202f 2048 5454 502f  .R?"HEAD./.HTTP/
        0x0040:  312e 300d 0a55 7365 722d 4167 656e 743a  1.0..User-Agent:
        0x0050:  2057 6765 742f 312e 3130 2e32 0d0a 4163  .Wget/1.10.2..Ac
        0x0060:  6365 7074 3a20 2a2f 2a0d 0a48 6f73 743a  cept:.*/*..Host:
        0x0070:  2077 7777 2e77 332e 6f72 670d 0a43 6f6e  .www.w3.org..Con
--
11:02:01.171913 00:60:08:12:8f:95 (oui Unknown) > 00:11:24:e1:db:c8 (oui Unknown), ethertype
IPv4 (0x0800), length 70: stu.w3.org.www > halaconia.hoomaha.net.57016: . ack 100 win 5792
<nop,nop,timestamp 3243392813 22265176>
        0x0000:  4500 0034 6887 4000 3206 20dd 801e 3433  E..4h.@.2.....43
        0x0010:  0a01 010e 0050 deb8 ecd8 57aa 7625 e9b7  .....P....W.v%..
        0x0020:  8010 16a0 5e28 0000 0101 080a c152 3f2d  ....^(.......R?-
        0x0030:  0153 bd58 8acc f50f                      .S.X....
11:02:01.172825 00:60:08:12:8f:95 (oui Unknown) > 00:11:24:e1:db:c8 (oui Unknown), ethertype
IPv4 (0x0800), length 529: stu.w3.org.www > halaconia.hoomaha.net.57016: P 1:460(459) ack 100
win 5792 <nop,nop,timestamp 3243392813 22265176>
        0x0000:  4500 01ff 6888 4000 3206 1f11 801e 3433  E...h.@.2.....43
        0x0010:  0a01 010e 0050 deb8 ecd8 57aa 7625 e9b7  .....P....W.v%..
        0x0020:  8018 16a0 b148 0000 0101 080a c152 3f2d  .....H.......R?-
        0x0030:  0153 bd58 4854 5450 2f31 2e31 2032 3030  .S.XHTTP/1.1.200
        0x0040:  204f 4b0d 0a44 6174 653a 204d 6f6e 2c20  .OK..Date:.Mon,.
        0x0050:  3033 204d 6179 2032 3031 3020 3135 3a30  03.May.2010.15:0
        0x0060:  323a 3031 2047 4d54 0d0a 5365 7276 6572  2:01.GMT..Server
        0x0070:  3a20 4170 6163 6865 2f32 0d0a 436f 6e74  :.Apache/2..Cont
--
11:02:01.172857 00:11:24:e1:db:c8 (oui Unknown) > 00:60:08:12:8f:95 (oui Unknown), ethertype
IPv4 (0x0800), length 66: halaconia.hoomaha.net.57016 > stu.w3.org.www: . ack 460 win 6432
<nop,nop,timestamp 22265186 3243392813>
        0x0000:  4500 0034 541f 4000 4006 2745 0a01 010e  E..4T.@.@.'E....
        0x0010:  801e 3433 deb8 0050 7625 e9b7 ecd8 5975  ..43...Pv%....Yu
        0x0020:  8010 1920 59d3 0000 0101 080a 0153 bd62  ....Y........S.b
        0x0030:  c152 3f2d                                .R?-
sending ^C

9 packets captured
9 packets received by filter
0 packets dropped by kernel
$
$
##Overall Result: PASS
```

Impediments to institutional adoption of Free/Open Source Software

Peter St. Onge
Information + Technology Services, University of Toronto
pete.stonge@utoronto.ca

Abstract

Free and Open Source Software (FOSS) have a number of characteristics that make it highly desirable in institutional settings: prevention of lock-in, cross-platform availability and version consistency across platforms, internationalization support, breadth and depth of choices, availability of updates and cost. Despite these manifold advantages, institutional uptake of FOSS has been limited. This paper discusses some of the key factors limiting adoption, and presents some suggestions on how these barriers can be overcome.

1 Introduction

This paper examines some of the strengths and weaknesses of FOSS as these relate to their use in institutional settings, environments where a large number of computers are in use – Typically this would involve anywhere from several tens of systems, upwards to several thousand systems in active central management.

Given that organizations are driven to save money and resources by realizing economies of scale wherever possible, the most important consideration in such a setting is the ability to manage three major components – users, systems, and services – as efficiently as possible. The user base is generally highly variable in terms of technical aptitudes, ability, and acceptance of any performance issues or anything impacting their ability to work, perceived or otherwise. The practical upshot of this approach is that any element requiring manual configuration or other "tweaking" is not suitable for use in an institutional context.

There are excellent examples of where FOSS-based systems are used as the foundation for institutional computing. This would include complex, multi-site file and print services for Windows client systems [30], institutional non-web single sign on [11], and corporate directories and authentication [3].

While extensive mention is made of Debian and derivative distributions in this paper, including the packaging mechanism, it is understood that similar facilities exist in RedHat/Fedora and other distributions of Linux and that the same points apply to these distributions as well.

2 Packaging and Distribution

Similar to commercial software development, the practices and processes underlying the development, packaging, and delivery of FOSS applications and systems can be highly variable even where best practices (eg. revision control, source code conventions, documentation conventions, etc) are relatively well known.

2.1 Distribution

While the distribution of software from both commercial and FOSS efforts can be done by the developers themselves, in the form of source tarballs or pre-compiled binaries, greater mindshare is typically achieved by making the application available via existing channel-based distribution systems, such as those used by Debian, RedHat, and derivatives. Beyond their philosophical differences underlying how FOSS-based systems should work (eg. Debian vs Ubuntu or RedHat or Fedora), all of the distributions of Linux serve the particularly crucial role of being the primary distribution channel for software in convenient form – packages. The importance of the distribution channel – the different Linux distributions – manifests in the fact that upstream developers often integrate distribution-specific tools to facilitate packaging the application (eg. in the Makefile); in this way, the amount of work required to effectively package an application for different distributions is minimized.

The distribution systems (eg. package repositories) as well as the packages themselves are difficult to subvert, thanks to the good use of md5 hashes for repository content lists themselves, for each of the many packages in that repository, and for files inside packages via manifest as required by the various distributions' policies[24]. It is thus possible to audit the integrity of system and application binaries on these FOSS-based systems, something is in theory possible but not mandatory when creating an MSI or EXE-based installation package. Newer, smaller organizations often do not have the expertise or resources to test their packages via build/install farm typical of larger organizations or projects[8].

The different "distros" each also provide quality assurance for those packages they make available. Distribution-specific policies set high standards for both source (eg. the dreaded FTBFS[13]) and binary packages[12]; packages are not permitted to enter into the distribution system until the policy requirements are met. Broadly-distributed tools such as linitian[25] and the like, in conjunction with the distributions' automatic build testing, help to ensure the ability to consistently install and uninstall individual packages cleanly and effectively.

In addition, these distribution mechanisms also allow for the provisioning of a package's dependencies (eg. shared libraries) in a separate package; in non-FOSS systems, these shared components were often shipped as part of a dependant package, often leading to conflicts in different versions of the same libraries installed on the system known as 'DLL Hell'[1], although this has become less of a problem in recent years[4].

Ultimately, these channels also provide a mechanism to provide patches back to the upstream source maintainers to ensure that underlying bugs can be resolved in a coordinated manner.

It is interesting to note that in the year 2010, the paradigm for distribution of software on the Windows operating system platforms is still primarily either the retail channel, or via downloads from the developers' sites, and that no central distribution channel exists. Moreover, no consist ant update mechanism has achieved any real traction: Microsoft's update mechanisms ("Windows Update" and "Microsoft Update") have focused primarily on their software with what appears to be limited participation by hardware vendors for driver updates (NVidia, ATI, Dell, Intel have occasional but infrequent updates), and other approaches like InstallShield's distribution service appear to have lost momentum after not gathering any real buy-in from software manufacturers.

Unfortunately, FOSS offerings for non-FOSS systems are also distributed in this same *ad hoc* manner for the most part, which acts to limit the knowledge of and access to the many multi-platform applications that would otherwise help build FOSS' mindshare outside of the existing areas.

This vacuum presents a tremendous opportunity for advocates of "World Domination" (as quoted in [7]) to put forward a packaging / distribution / update channel effort that would function effectively on non-FOSS platforms, typical of the home or unmanaged user. Being able to conveniently and easily install a FOSS package on a home user's non-FOSS system, in the way that **synaptic** would work, would be highly attractive to technical and non-technical users alike.

Not only would this approach help serve to bring FOSS to a much wider audience, it would also solve the problem of ensuring that end users can keep their applications up to date with minimal effort (eg. similar to how **update-notifier** already works).

Previous efforts have attempted to bridge this gap, strictly providing a distribution mechanism for packages: WPM[27] appears to have stalled during design phase, Win-Get[23] had more success as an active distribution channel, but the apps in channel appear to be quite dated (2.0.0.x for Firefox, Thunderbird). A third attempt, Appupdater[19] combines a distribution mechanism with a user application that tracks available and installed packages, and facilitates updates. It appears to be the most successful approach thus far, at least for unmanaged users in Windows, and continues to have a modest selection of updated software packages – 88 packages, some of which are FOSS (eg. Thunderbird, Firefox), some not (eg. Adobe PDF reader). This is rather a small number of applications, however, given that there are a rather large number of FOSS applications currently available for Windows.

2.2 Packaging

Even with a delivery mechanism to provide FOSS software to non-FOSS operating systems, however, there are other problems that remain particularly challenging for institutional adoption of FOSS.

From the perspective of the distribution, the main goal of a package is not just to facilitate the installation and removal of that package's application; it should also be to facilitate the future replacement of that package by another more updated version as it becomes available. Although it is possible to build software for distribution to non-FOSS platforms, it is generally not possible to build the distribution package for non-FOSS platforms (exe, msi) in the same way that is done for FOSS platforms (rpm, deb, etc).

Firefox, in particular, is a good example as a web browser popular with non-technical and technical people for the relative speed, flexibility, and general resistance to hostile sites[15]. Although there is a Windows binary installer package available, it cannot be used in managed Windows environments without first installing it to a test system, creating an MSI file based on the pre-install and post-install snapshots, then tested in a number of environments prior to being deployed via Group Policy Object (GPO)[33]. As the work required to package the application and test it properly is considerable, and given the frequency with which updates become available, it is no surprise that few administrators have opted to repackage applications themselves. This does create opportunities for enterprising individuals to provide such a service[17], and this has helped penetration of Firefox into some managed environments (eg. ours). Unless these are done as part of normal FOSS project activities, however, the project has no control over or voice in how the application is packaged and provided.

At present, there are multiple approaches used to package FOSS applications for non-FOSS operating systems. NSIS[6] is an open-source suite used to build executable (EXE) based installation packages; many commercial offerings exist as well (eg. InstallShield). These have all had a great deal of success in packaging FOSS applications for end users of Windows systems. From the perspective of managed systems, however, the standard packaging format for applications on Windows systems is the MSI[33]. Although other mechanisms for software installation over large numbers of centrally managed non-FOSS systems may exist, the best known and most used software delivery mechanism in managed environments typical of institutions is the MSI installer package installed automatically via GPO.

Ultimately, the ability to build MSI packages for FOSS applications for distribution on non-FOSS platforms, with the same care and attention as other packaging formats (deb, rpm, etc) to allow for package auditing and assurance, as well as the development of a distribution system similar to how FOSS software distribution already works, would remove a substantial barrier to awareness of the broad availability of FOSS on non-FOSS platforms in general, and to institutional adoption of FOSS on centrally-managed non-FOSS systems in particular.

3 Software

The advantages of FOSS in general have been treated extensively and exhaustively elsewhere[26, 34]. From the institutional perspective, there can be substantial wins in adopting FOSS: the cross-platform availability of given applications, support for internationalization, the tendency towards frequent and non-disruptive updates, and the ability to access support resources, both internal and external.

There are, however, substantial downsides that must be considered.

3.1 Cross-platform availability

Like numerous commercial applications, many established FOSS applications like OpenOffice or Firefox are available for multiple operating systems. Unlike many commercial applications that produce different versions of an application for different operating systems, however, FOSS applications tend to have consistent versions and interfaces across different operating systems.

The ability to have the same user experience from an application across multiple operating systems has important ramifications institutionally.

In terms of support, it becomes considerably simpler to diagnose and rectify user issues with the application, produce useful internal documentation to support the use of that application, and build institutional knowledge around the use of that applications.

From the user perspective, the application becomes the important element rather than the operating system. This can ease some pain points in heterogeneous computing environments. In particular, it allows users to be more "mobile" in terms of what operating system they use: A consistent interface across platforms minimizes user disorientation with the application, particularly when they already know that application well, even when the underlying operating system user interface differs from their "usual" platform.

This mobility offers an organization a great deal of flexibility in developing their IT strategy, as they can choose from different platforms based on their requirements while minimizing disruption from transitions.

3.2 Internationalization

In addition to the ability to have consistent versions of an application over multiple platforms, a further advantage of using established FOSS applications is well-known and well-supported ability for a single binary application to present its user interface elements in any number of languages[16].

Lacking such a well-documented practice, I have noticed that many commercial applications have separate releases to support individual languages. In many cases, this is effectively a different version release of a given application; the overhead of maintaining multiple concurrent versions of a commercial application makes integrating bug and feature fixes more complicated. The end result we witnessed was an incompatibility between an operating system of one language, and an application of another, despite the fact that both were from the same company. This has improved in Windows relatively recently[20] but still appears considerably more complex than how distros manage locales for i18n, and existing Mac OS X support.

In an organization spanning different areas of languages, dealing with language-specific software issues – particularly if a common vocabulary does not exist – adds considerable complexity to support and administration of these environments.

Conversely, a single binary application that has support for relevant languages is far more easily deployed and supported broadly, allowing for economies of scale and leveraging of administrative efforts required in an institutional context.

3.3 Frequent, small changes

The development cycle and distribution process of packaged binaries of FOSS applications provide a fairly painless means to integrate the relatively frequent and incremental updates over time. In the established or "stable" dists, security and bug fixes are unremarkable from the user standpoint as these minor internal issues and not major changes to user interface and other elements.

Even in cases where one decides to use the more dynamic "testing" or "unstable" dists in Debian, the experience of significant breakage is quite small in my own experience. This does not, of course, obviate the need to test packages prior to wide deployment.

This capacity for accommodating frequent minor changes provides the institutional administrator the means to take company-specific or resource-specific packages and maintain these in a state where minor changes be readily propagated in response to corporate, managerial or new policy requirements.

3.4 Support resources

In a large corporate organization, it is not unusual to have local staff whose primary role is to coordinate with a vendor for a particular product (or set of products), to manage pending bug or function issues, feature requests, and such. Although large software vendors often have well-developed self-help and similar documentation resources, dealing directly with support resources often requires the dedicated staff to handle contract and entitlement issues to ensure prompt access to updates and support resources.

After considerable work ensuring that entitlements and contracts are maintained, it was our experience as a relatively small organization (a large University) that days would usually pass between a support request and an initial response, we felt generally poorly served by such support arrangements. Especially in light of the cost of the support contracts for expensive software: Typically this was on the order of 20-25% of the original purchase price, with the percentage increasing on an annual basis.

As a result, serious consideration of alternative open source alternatives for some of these applications is being made at present[10]. Although in this case, justification to technical and managerial oversight was simple due to the organization's receptiveness to the use of FOSS, other organizations may be more hesitant for various reasons; perhaps the most important is the lack of formal "support" mechanisms.

As the number of organizations with similar problems opt for the FOSS solution, the number of active developers and participants involved with projects increases – each scratching their employers' itches as well as their own – these projects grow in capability and dependability. Having individuals in a position to support critical applications employed on-site, the responsiveness of support of FOSS in the form of immediate and direct access to bug fixes is a valuable asset. Ultimately, if critical vendor applications would require staff dedicated to support local customizations as well as coordinate with remote vendors, it is difficult to see how dedicating local staff to work on a FOSS project would imply any greater staffing costs. Furthermore, local staff involved in the development and support of internally developed applications can also be the source of considerable innovation for those organizations.

Greater visibility of corporate or institutional involvement in FOSS projects outside of the kernel, where this involvement is well known, would provide additional comfort to organizations considering adopting FOSS applications in important and visible areas.

3.5 Limitations

The breadth and depth of mature FOSS applications is a testament to the effectiveness of FOSS development methods. These applications, however, have a number of weaknesses that would have to be address to make them suitable for institutional use.

As one example, it is currently not possible for most of these very useful applications find all or part of their configuration elsewhere. User configuration (user name, email address, organization, incoming & outgoing mail server configurations) for a mail user agent (eg. Thunderbird), for instance, is still a usually a task for the user, or for a tech support representative. Web browser proxy configuration is a similar issue, particularly in corporate environments. Given the highly variable nature of user technical aptitudes in organizations, expecting users to enter in such information – simple as it may seem to many readers – can easily lead to unexpected overburdening of support resources and the loss of user acceptance of these applications.

In an environment where the mail service is managed centrally, and all users are known *a priori*, the inability to provision user data to the application on behalf of the user reflects poorly on the technology and those who administer it.

The ultimate goal of such efforts is to limit the need for manual configuration of production user applications – be they mail clients, web browsers, database clients, databases, ODBC connections, etc. This not only allows for a more productive user experience (minimizing time lost due to configuration issues), it also provides a means to minimize impacts of future service changes (different server software, different IP address used, etc).

4 Operating Systems

FOSS-based operating systems have had good support via PAM for external identity management systems for some time now, Kerberos and LDAP have been two of the most commonly used of FOSS-based systems[28, 11] for that purpose.

The primary goal of external authentication frameworks like these is twofold. First, to shrink user credential space by reducing the number of credentials that users in an organization have to use to prove their identity to non-critical services on a daily basis. Making it easier for the user to remember their user name and one good password (which can be enforced via password policy, of course) rather than requiring many user names and corresponding (and usually bad) passwords on different systems – usually on systems where password policies cannot be implemented – can drastically reduce the number of password reset requests and thus reduce demands on IT help desks. Moreover, the common user name across multiple systems allows for more effective auditing of user activities across these systems, simplifying the detection of anomalous behaviour (eg. access to systems from "local" IP addresses concurrently with access via VPN from "foreign" IP addresses).

The second, and perhaps more important goal, is to provide means for identity to be vouched for by a separate trusted system (eg. Kerberos) once the user successfully authenticates themselves; this process is often referred to as Single Sign On[32], or SSO. From the user's perspective, SSO reduces issues and inconveniences around identification and authentication by having these handled transparently after the initial authentication. Log in once, and never again for that day.

From the institutional perspective, however, SSO provides another important value. SSO reduces password exposure by drastically reducing the number of information system that have to handle passwords and hence reducing the avenues for user credential compromise. In other words, by relying on the Kerberos (or similar) service for authentication, servers/services do not ever handle the user's password, and their compromise is less likely to facilitate exploiting other systems via captured user credentials.

Moreover, SSO simplifies services to some extent by obviating the need to design and implement user credential (user/pass) facilities, allowing developers to re-use code dealing with the underlying SSO service for those purposes.

Single sign on does not obviate the need for additional levels of protection and authentication around critical institutional resources; services such as payroll, finance, etc. require greater protection than the individual user's machine, for instance, and the use of two-factor (or more) authentication is called for in these circumstances.

Kerberos is perhaps the most mature SSO systems, and remains important, particularly because it remains the only SSO solution that bridges both system-level authentication and web single sign on through web browsers[21].

Given the competing paradigms for identity management in play (Active Directory[5], LDAP[2], Kerberos[11], PubCookie[22], Shibboleth[14], and others), it would appear that institutional systems will need to accommodate more than one of these approaches in heterogeneous institutional environments.

5 Practice

Although the technological capacity for FOSS-based systems to support institutional is already well-established through the use of directory services[31], the understanding of what directory services can provide is still very limited by most administrators of FOSS systems.

In discussions with colleagues both near and far involved in managing large numbers of machines in different environments, and generally the adoption of directory services to support administration is minimal. As a result, the community of practice around the use of directories to support FOSS-based or more heterogeneous environments remains relatively underdeveloped.

5.1 Directory services

The Lightweight Directory Access Protocol[35, 29] (LDAP), is a directory service allowing for user, group, machine, authorization, and service information to be maintained centrally and in a secure manner. OpenLDAP provides a standards-based implementation of LDAP[3], and Active Directory extends LDAP to support Microsoft Windows-based systems and services[5].

Central control of any resource usually invokes political and other concerns. One of the advantages of directory services is the ability to delegate control over parts of the directory tree to particular individuals and groups, allowing local control over access and facilitating managerial tasks through a common GUI or web-based interface.

The adoption of LDAP and AD by FOSS applications manifests in many different areas. What follows is not exhaustive by any means, but does serve to show possibilities.

Mail user agents (Thunderbird and Evolution are only two examples among others), for instance, can access corporate directory information from an LDAP directory. The PostgreSQL database can have its service list in stored in LDAP, which facilitates connecting to remote institutional database servers.

Many office appliances (scanners, copiers, faxes) can make use of user information from an LDAP server.

The ISC DHCP and BIND servers can both use LDAP to contain their respective data; not only can this allow for the management of data across multiple DNS / DHCP servers across the institutional network, but they also provide a means to ensure service redundancy inside the institution.

Samba can use directories to manage user, group, and machine elements in large organizations through the use of LDAP to contain user account information[30].

PAM can also make use of user and group info in LDAP, and with the proper components, new user home directories can be created at the initial login. Further, the automounter can use information in LDAP to mount the user's remote home directory via NFS3 or NFS4.

The Puppet datacenter automation tool can make use of LDAP to store configuration info for the machines that it controls[18].

5.2 What's missing...

Even with delegation and the breadth of applications able to make use of directory services, there are a number of application shortcomings that remain.

As mentioned previously, the ability to tell user applications where to look for their configurations (eg. having the user's email application find the user's info in a particular directory location) as a means to auto-provision applications is an obvious institutional example.

For applications or services serving multiple users or systems, however, similar benefits could accrue. Storing application configurations in a directory, where appropriate, would allow for the ability to check and modify the application's configuration remotely and non-intrusively.

The benefits of directories are not always well-known, particularly since these usually only become used in larger environments not commonly experienced by a sizable majority of FOSS users and developers.

The perception that the cost of one-off changes to systems to allow them to keep working or participate in the local environment is far smaller than the benefit that a well-designed directory service can provide; in aggregate, however, the reverse is true. Not only would directory services provide am effective way to manage users and equipment, but as mentioned above, would allow user data to be provisioned appropriately across platforms.

Another barrier in the FOSS world, the directory server tree has no default population at install time. The nature of most system administrators is to take the precautionary approach in that they would want to understand how a system works prior to configuring it. Combined with the highly flexible nature of directories, in my experience the ethereal nature of a directory makes it harder for most to grasp. In addition, configuring individual machines or services to use the directory is a similarly involved task, at least initially.

Unlike FOSS directory servers like OpenLDAP, Active Directory-based directory servers come pre-populated to accommodate the majority of common tasks (eg. management of users, groups, and machines), primarily through information gathered when the server is initially installed. The process of enrolling Windows machines is also relatively straightforward at install time.

In order to realize the manifold benefits of directory services supporting the use of FOSS in organizations, a greater community of practice around the use of these techniques is needed. As this becomes more established, other shortcomings at the level of software are more likely to be addressed[9]. As in other communities around technologies (eg. Samba), an intrinsic activity this community should undertake is to document examples where directory services are used to support FOSS systems, the problems encountered and lessons learned (good and bad) in the process of set up and maintenance of the directory, as well as best practices to ensure continuity.

6 Conclusions

Presently, the underlying technology required to productively support the use of FOSS-based systems at the institutional level exists at both the software application and operating system levels. The majority of the technical limitations remaining involve the ability to for applications to obtain user-specific information through directory services.

It is expected that as FOSS applications are increasingly adopted by institutional users that support for packaging FOSS applications for non-FOSS platforms will become more commonplace, akin to how FOSS applications are packaged for FOSS-based platforms, although this would likely require a FOSS means to create MSI files.

Finally, the most important limit to the potential of directory services is the general lack of knowledge of how directory services work and how to properly design directories for particular situations.

7 Acknowledgements

This paper benefited immensely from discussions with a number of people, and I would like to thank them: Ian

Thomas, Martin Loeffler, Mike Wiseman, David Auclair, Peter Eden, John DiMarco, Ted Sikorski, David Sutherland, Roger Dingledine, and Richard Sanford.

Any inaccuracies or errors are but my own.

References

[1] Rick Anderson. The End of DLL Hell. `http://msdn.microsoft.com/en-us/library/ms811694.aspx`.

[2] Brian Arkills. *LDAP Directories Explained: An Introduction and Analysis*. Addison-Wesley Professional, 2003.

[3] Gerald Carter. *LDAP System Administration*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2003.

[4] Raymond Chen. Windows Confidential: Getting Out of DLL Hell. `http://technet.microsoft.com/en-ca/magazine/2007.01.windowsconfidential%.aspx`.

[5] Brian Desmond, Joe Richards, Robbie Allen, and Alistair Lowe-Norris. *Active Directory: Designing, Deploying, and Running Active Directory*. O'Reilly Media, Inc., 2008.

[6] The NSIS developers. The Nullsoft Scriptable Install System (NSIS). `http://nsis.sourceforge.net/Main_Page`.

[7] Chris DiBona, Sam Ockham, and Mark Stone. *Open Sources: Voices from the Open Source Revolution*. O'Reilly Media, 1999.

[8] Andrew Dunstan. PostgreSQL BuildFarm. `http://buildfarm.postgresql.org/`.

[9] The Apache Software Foundation. Apache Directory Project. `http://directory.apache.org/`.

[10] The Kuali Foundation. About the Kuali Community. `http://www.kuali.org/about`.

[11] Jason Garman. *Kerberos: The Definitive Guide*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2003.

[12] The Debian QA Group. Debian quality assurance. `http://qa.debian.org/`.

[13] The Debian QA Group. Debian wiki: FTBFS. `http://wiki.debian.org/qa.debian.org/FTBFS`.

[14] Internet2. Shibboleth. `http://shibboleth.internet2.edu/`.

[15] Brian Krebs. A Peek Inside the 'Eleonore' Browser Exploit kit. `http://krebsonsecurity.com/2010/01/a-peek-inside-the-eleonore-browser-e%xploit-kit/`.

[16] Tomohiro Kubota. Introduction to i18n. `http://www.debian.org/doc/manuals/intro-i18n/`.

[17] Ing-Long Eric Kuo. Firefox MSI. `http://www.frontmotion.com/Firefox/`.

[18] Puppet Labs. Storing Node Information in LDAP. `http://projects.puppetlabs.com/projects/puppet/wiki/Ldap_Nodes`.

[19] Neil McNab. Appupdater. `http://www.nabber.org/projects/appupdater/`.

[20] Microsoft. Guide to Windows Vista Multilingual User Interface. `http://technet.microsoft.com/en-us/library/cc721887(WS.10).aspx`.

[21] University of Maryland Office of Information Technology. Configuring Web Browsers for Kerberos Authentication. `http://www.helpdesk.umd.edu/topics/applications/kerberos/4782/`.

[22] University of Washington Techology Services. Pubcookie: open-source software for intra-institutional web authentication. `http://www.pubcookie.org/`.

[23] Ryan Proctor. Win-get. `http://windows-get.sourceforge.net/`.

[24] The Debian Project. Debian Policy Manual. `http://www.debian.org/doc/debian-policy/`.

[25] The Debian Project. Lintian. `http://lintian.debian.org/`.

[26] Eric S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001. Foreword By-Young, Bob.

[27] Edward Ropple. WPM - A Windows Package Manager. `http://blacken.superbusnet.com/oss/wpm/`.

[28] Andrew Ryan. HOWTO-pam. `https://mon.wiki.kernel.org/index.php/HOWTO-pam`.

[29] J. Sermersheim. RFC4511: Lightweight Directory Access Protocol (LDAP): The Protocol. `http://tools.ietf.org/html/rfc4511`.

[30] John H. Terpstra. *Samba-3 by Example: Practical Exercises to Successful Deployment (Bruce Perens Open Source)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[31] Wikipedia. Directory Sservices. `http://en.wikipedia.org/wiki/Directory_service`.

[32] Wikipedia. Single sign-on. `http://en.wikipedia.org/wiki/Single_sign-on`.

[33] Wikipedia. Windows Installer. `http://en.wikipedia.org/wiki/Windows_Installer`.

[34] Sam Williams. *Free as in Freedom: Richard Stallman's Crusade for Free Software*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[35] K. Zeilenga. RFC4510 Lightweight Directory Access Protocol (LDAP): Technical Specification Road Map. `http://tools.ietf.org/html/rfc4510`.

Linux kernel support to exploit phase change memory

Youngwoo Park, Sung Kyu Park and Kyu Ho Park
Korea Advanced Institute of Science and Technology (KAIST)
`(ywpark,skpark)@core.kaist.ac.kr and kpark@ee.kaist.ac.kr`

Abstract

Recently, phase change memory (PRAM) has been developed as a next generation memory technology. Because PRAM can be accessed as word-level using memory interface of DRAM and offer more density compared to DRAM, PRAM is expected as an alternative main memory device. Moreover, it can be used as additional storage of system because of its non-volatility. However, PRAM has several problems. First, the access latency of PRAM is still not comparable to DRAM. It is several times slower than that of DRAM. Second, PRAM can endure hundreds of millions of writes per cell. Therefore, if PRAM does not be managed properly, it has negative impact on the system performance and consistency. In order to solve these problems, we consider the Linux kernel level support to exploit PRAM in memory and storage system. We use PRAM with a small size DRAM and both PRAM and DRAM are mapped into single physical memory address space in Linux. Then, the physical memory pages, which are used by process, are selectively allocated based on the access characteristics. Frequently updated hot segment pages are stored in DRAM. PRAM is used for read only and infrequently updated pages. Consequently, we minimize the performance degradation caused by PRAM while reducing 50% energy consumption of main memory. In addition, the non-volatile characteristic of PRAM is used to support file system. We propose the virtual storage that is a block device interface to share the non-volatile memory pages of PRAM as a storage alternative. By using 256MB PRAM for virtual storage, we can decrease more than 40% of access time of disk.

1 Introduction

For several decades, DRAM has been the main memory of computer systems. Since the memory requirement is growing to support the increasing number of cores and concurrent applications, DRAM based main memory significantly increases the power and cost budget of a computer system. Recent studies [8, 7] have shown that 30-40% of modern server system energy is consumed by the DRAM memory. Moreover, it is expected that DRAM scaling will be clamped by the limitation in cell-bitline capacitance ratio [4, 10]. Therefore, new memory technologies such as Phase-change RAM (PRAM), Ferroelectric RAM (FRAM), and Magnetic RAM (MRAM) have been proposed to overcome the limitation of DRAM.

Among these memories, PRAM is the most promising technology for future memory. Figure 1 shows the basic structure of PRAM cell. PRAM uses phase change material (GST: Ge2Sb2Te5). It has two phases;an amorphous or a crystalline phase. Since the amorphous and the crystalline phase have a large variance on their resistance, the data is read by measuring the current of PRAM. The phase of GST can be changed by heating the material. The moderate and long current pulse crystallizes GST. On the other hand, short current pulse melts and quenches GST quickly and makes it amorphous.

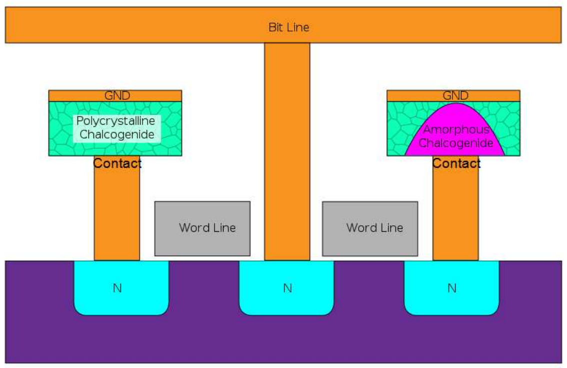


Figure 1: PRAM cell structure [2]

Basically, PRAM is byte-addressable like DRAM. The great advantages of PRAM are the scalability and low energy consumption. PRAM does not require implementing capacitor for memory cell. PRAM provides superior density relative to DRAM. Because the phase of PRAM is maintained persistently, PRAM is non-volatile memory and has negligible leakage energy. Therefore, PRAM can be used to provide the memory that has much higher capacity and lower power consumption than DRAM.

However, the long current pulse for crystallizing increases the latency of PRAM writes. Although PRAM access latency is tens of nanoseconds, it is still not comparable to DRAM access latency. The frequent access of PRAM can impact on the overall system performance. Also, the PRAM write energy consumption and endurance are limitations of PRAM. The high pulse for phase change increases the dynamic energy consumption. PRAM writing makes the thermal expansion and contraction of material. It degrades the electrode-storage contact and reduces the reliability of programming current. This degrades the write endurance of PRAM cells. PRAM can sustains 10^8 rewrite per cell [11].

In this paper, we consider the Linux level support to exploit PRAM in current computer system. First of all, we decide to use PRAM with a small size DRAM to overcome the limitation of PRAM. PRAM and DRAM are mapped into single physical memory address space. Then, the physical main memory of Linux consists of one small fast region (DRAM) and one large slow region (PRAM). Therefore, the PRAM is used to increase the size of main memory and eliminate a lot of page faults. At the same time, we can use DRAM to reduce the overall main memory access latency. Based on this main memory architecture, we propose a new Linux physical page management mechanism. The physical memory pages, which are used by process, are selectively allocated based on the segment type. Consequently, we minimize the performance degradation and endurance problems caused by PRAM while achieving a large scale and low power main memory.

In addition, we also discuss the block device interface to share the non-volatile main memory pages of PRAM as a storage alternative. It gives a lot of advantage for file system to access metadata and small size file because it can read or write the data as single word level and avoid unnecessary seek latency of disk.

2 Hybrid main memory architecture

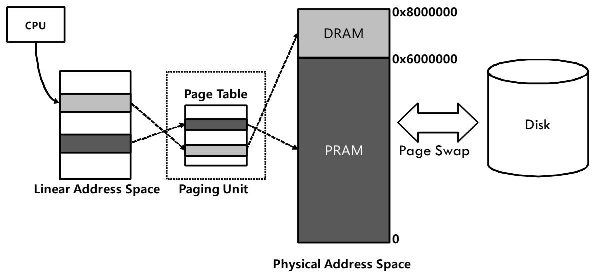


Figure 2: Hybrid main memory architecture

Figure 2 shows the proposed hybrid main memory architecture. Both PRAM and DRAM are used as a main memory. This architecture reduces the cost and power budget because large portion of main memory is replaced by PRAM. Using small size of DRAM, it minimizes the performance degradation. Similar to the traditional main memory architecture, there is memory page swap between hybrid main memory and the second level storage. However, the number of page swapping can be reduced because the main memory capacity is increased by PRAM.

In hybrid main memory architecture, PRAM and DRAM are assigned to single physical address space. All memory pages of PRAM and DRAM can be directly managed by the Linux kernel. However, it is necessary for kernel to distinct the PRAM and DRAM region. We assume that DRAM always has lower physical address than PRAM and the size of each memory is provided by the kernel option. The, Linux kernel has information of the exact physical address range of PRAM and DRAM as shown in Figure 2. Physical pages of PRAM and DRAM can be distinguished by the physical address.

3 Hybrid main memory management

3.1 Free page management

For the hybrid main memory architecture, Linux kernel needs to manage DRAM and PRAM region separately. In current Linux kernel, the physical memory of the system can be partitioned and managed in several nodes. Also, physical memory of each node is divided into several zones. If we can map DRAM and PRAM physical pages into separate memory nodes, memory management facilities can be used in proposed hybrid main memory architecture. However, Linux node is only proposed for specific NUMA hardware and Linux zone is proposed to cope with the hardware constraints. Instead of making new node or zone for DRAM and PRAM, we use additional *free area* for each zone as shown in Figure 3.

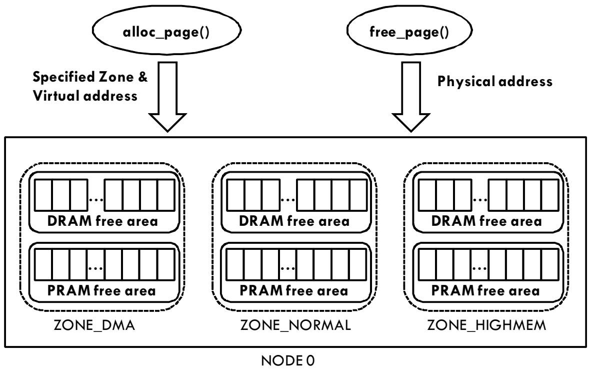


Figure 3: Free page management for hybrid main memory list

Proposed Linux kernel has two *DRAM free area* and *PRAM free area* which contain only the free pages of DRAM and PRAM, respectively. Although both *free areas* are used by the same *buddy system* page allocator, the contiguous DRAM and PRAM block is independently handled. In addition, When the kernel invokes a memory allocation function, the page frames are selectively allocated to one of the *free area* of specified zone. The *free area* selection considers the characteristics of allocated data. We will describe this allocation policy in the section 3.2

After we divide the *free areas*, we also need to consider the page reclamation method. Basically, Linux reclaims free pages when there is not enough number of pages in zones. Although we separately manage the free page list of DRAM and PRAM, the page reclamation occurs when the total number of pages in both DRAM and PRAM *free areas* is lower than the threshold. Therefore, we can fully utilize the main memory region before swapping. Even though one of the memory devices is fully utilized, the free pages of another memory device are contributed as main memory. The large size PRAM main memory region reduces a lot of page swapping.

3.2 Selective allocation

In hybrid main memory architecture, there are two different memory devices. Particularly, PRAM is very different from the conventional DRAM. The physical memory management of Linux kernel, which is only developed for the uniform memory devices, should be changed. The first thing which needs to be addressed is the page allocation.

The goal of the page allocation of Linux is to serve the memory allocation request from the Linux kernel and the user processes. Conventional Linux kernel allocates physical memory pages when kernel functions like *alloc_page()* and *__get_free_page()* are called. These page allocation functions are successfully returned when the free pages of requested size are found and allocated. The conventional *buddy system* allocates groups of contiguous page frames to solve the external fragmentation.

Previously, it is not important where the memory pages are located in main memory because the characteristics of memory pages are always same in uniform memory device. However, in our hybrid main memory architecture, the location of pages can have significant effect on the performance and power consumption of main memory. As we mentioned in section 1, PRAM write operation is much slower than read and require high energy. If memory pages that are frequently updated are allocated in PRAM, it can increase the dynamic power consumption and decrease the overall access latency of main memory. Moreover, it reduces the lifetime of main memory because PRAM has limited write endurance. Therefore, the key of our page allocation is to assign frequently updated data into DRAM instead of PRAM.

The question is how to find the write intensive data before page allocation. Although it is very difficult to predict the future access pattern of each page, we can use the general characteristics of pages for page allocation. Traditionally, the process address space of Linux is partitioned in several linear address intervals called segments. It is well known that the access pattern of data in same segment is almost similar. For example, text segment includes the executable code which is read-only data. Stack segment contains the return address, parameters and local variables which are frequently read write. Table 1 summarizes the general access patterns of Linux segments.

Figure 4 shows the design of selective allocation. The memory segments are identified by the variables which are included in the *mm_struct* memory descriptor. For example, *start_code* and *end_code* store the initial and

Segment type	Access pattern
Text segment	Read only
Initialized data segment	Read centric
Uninitialized data segment	Infrequent read/write
Stack segment	Frequent read/write
Heap segment	Frequent read/write

Table 1: Access pattern of Linux segments

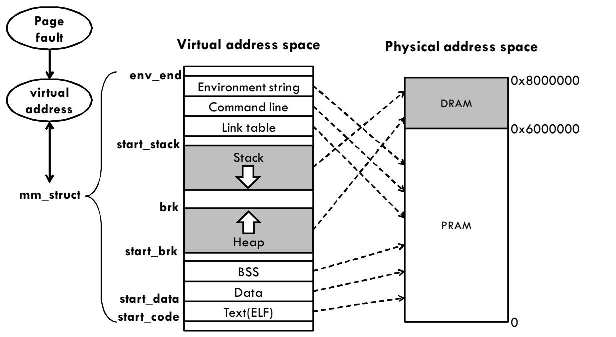


Figure 4: Design of selective allocation

final virtual address of the segments. *start_stack* store the start virtual address of stack segments. Also, we can get the virtual address, where physical page should be mapped. It is delivered by the page fault handler before page allocation. Thus, only if the variable of the *mm_struct* memory descriptor is compared to the virtual address that cause the page fault, we can decide the segment type of page before allocation. After finding the segment type, we selectively allocate a physical page between DRAM and PRAM.

In current design of selective allocation, the page allocation policy is fixed. The pages of heap and stack segments are allocated in DRAM. All other memory pages are allocated into PRAM as shown in Figure 4. However, if we implement new system call *salloc_policy()*, the selective allocation policy is changed by each application. For example, a user can allocate stack and heap pages in PRAM. Also, this system call can be used to decide the allocation policy of file cache, mmap and library pages.

4 Hybrid main memory for virtual storage

4.1 Virtual stroage

Many previous researches prove that non-volatile memory is very effective to reduce the overhead of disk based storage because it is free from seek latency and favorable for small size random accesses data [3, 6, 12]. In proposed hybrid main memory, a part of main memory (PRAM) can be used for the byte-addressable non-volatile storage. In order to exploit PRAM memory as storage, we propose the virtual storage which is a block level interface to use PRAM region of hybrid main memory for storage as shown in Figure 5.

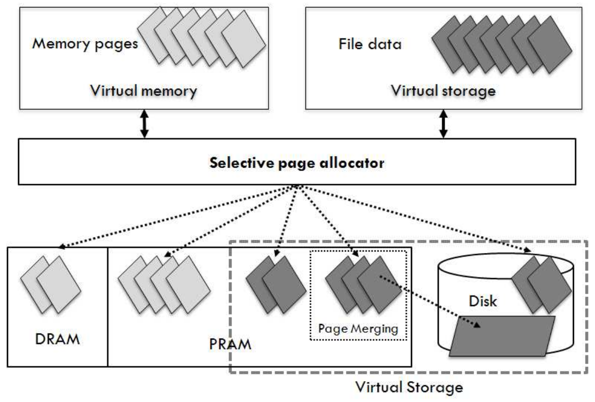


Figure 5: Virtual storage architecture

Similar to the virtual memory, the virtual storage is an abstraction of single and contiguous storage. Physically, it uses PRAM main memory region and disk as a storage device. If the file system writes data to the virtual storage interface, it selectively allocates data into PRAM memory page or disk. However, we do not reserve PRAM pages for storage use. All free pages in PRAM are dynamically allocated for memory page and file page. The mapping from virtual page to physical page in PRAM or disk is maintained by the storage page table which contains the mapping of allocated virtual pages as tree. In order to preserve the mapping information after power-off, it should be stored in PRAM and the root of table is managed in a fixed location of PRAM.

4.2 Selective allocation for virtual storage

In section 3.2, we describes that the memory pages are allocated based on the type of segments. On the other hand, the basic metric for file page allocation algorithm is data size. It is generally known that a small size file is frequently and randomly accessed. Because PRAM has fast access time and much smaller capacity than disk, it is better to keep only data of a small size file in PRAM. If the requested data size to virtual storage is over several tens of KB, contiguous physical pages in disk are selected to store the data.

Also, the virtual storage is designed to support write request merging. Although the data size of a request is small, if it is a sequential request that has a nearby virtual address of previous one, the selective allocator decides that those pages are in the same file and move them into disk later. It can allocate a large file, whose size is continuously increased, to disk. Consequently, the write request merging reduces the waste of PRAM space and number of disk access.

5 Evaluation

5.1 Hybrid main memory

Currently, PRAM is not available as a main memory. Instead of hardware, we evaluate proposed hybrid main memory and its management schemes using the M5 simulator [9]. In order to implement the hybrid main memory architecture, we use additional memory modules in a physical main memory space. Two memory modules are used to simulate DRAM and PRAM memory, respectively. The PRAM and DRAM memories are mapped into same physical address space.

In addition, we add the memory access monitoring module. It monitors all memory access of DRAM and PRAM. Because we separately use two memory modules, the memory monitoring module can monitors both DRAM and PRAM at the same time. Although we monitors the total read/write access counter and size of memory, the memory monitors can be extended to get an any information that is related with memory access. Finally, the monitored read/write access count and size is used to calculate the access latency and energy consumption of overall main memory.

Then, the selective allocation for main memory is implemented and evaluated on Linux 2.6.27 which is operated on the M5 simulator. We execute benchmarks on M5 using a simple execution ALPHA processor running at 1GHz. We assume that the processor does not have caches to focus on main memory evaluation. The total main memory size is 256MB. For hybrid main memory, 64MB DRAM and 198MB PRAM is used. The access latency and energy consumption of DRAM and PRAM is calculated by the parameters of Numonyx [11]. Table 2 summarizes the parameters used for DRAM and PRAM. For our workloads, we use MiBench benchmark suite.

Parameter	DRAM	PRAM
Read latency	50ns	50ns
Write latency	50ns	1us
Read Energy	0.1nJ/b	0.05nJ/b
Write Energy	0.1nJ/b	0.5nJ/b
Idle Energy	1W/GB	0.005W

Table 2: DRAM and PRAM characteristics [11]

First of all, we compare the energy consumption of proposed hybrid main memory with DRAM. For this evaluation, we should assume the idle time of main memory. It has been well-established that the average utilization of server is even below 30% [5]. Therefore, in all evaluations of energy consumption, we assumes that memory is only accessed 40% time.

Figure 6 and Figure 7 shows the reduction in memory energy consumption. The hybrid main memory achieves around 30% energy savings. Moreover, if we use the selective allocation, we can reduce the PRAM write operation which consumes more energy than DRAM read/write and PRAM read operation. The total energy saving ratio is increased by 50%. The selective allocation exploits the read-friendliness of PRAM as well as writes friendliness of DRAM, and hence achieves better overall energy efficiency.

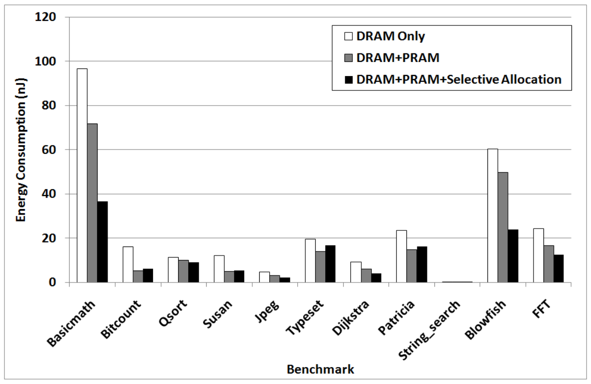


Figure 6: Total energy consumption

Although PRAM is good for scalable main memory, it can increase the overall latency of main memory. Figure 8 shows total access latency and Figure 9 shows the latency overhead against the DRAM main memory. In our experiments, the hybrid main memories that use typical Linux allocation algorithm have more than 100%

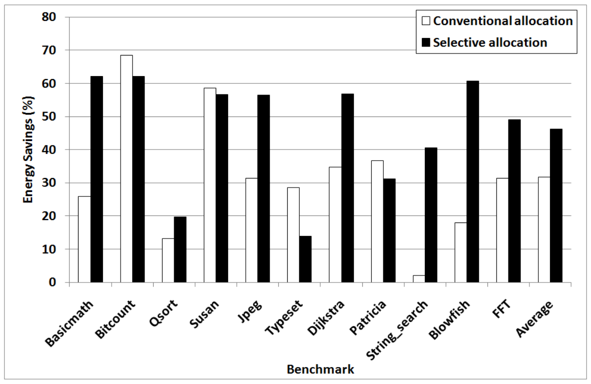


Figure 7: Energy savings against DRAM main memory

latency overhead. However, if we use the selective allocation, we can allocate the infrequently accessed page in PRAM and reduce the 50% latency overhead. Moreover, the latency of 6 applications is only increased under 20%, while reduce much more than 50% energy consumption.

However, some benchmarks use much global variables and update the variables frequently. It causes a lot of write in data segment. The latency of these benchmarks is much larger than DRAM main memory. For Typeset application, selective allocation rather increases the latency and reduces energy consumption.

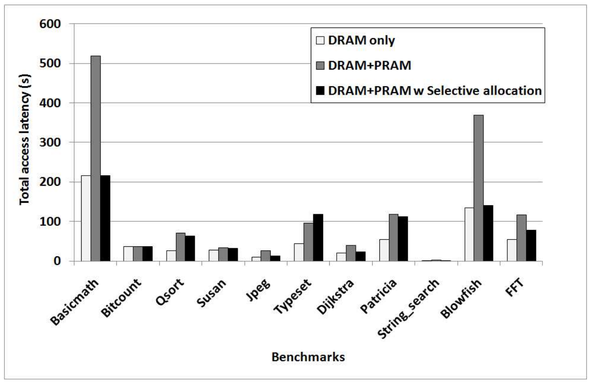


Figure 8: Total access latency

5.2 Virtual storage

In order to evaluate the performance of virtual storage, we estimate the total access time when executing OLTP trace [1]. In virtual storage, each OLTP request can be allocated to PRAM or disk. We use PRAM access latency of Table 2 and assume 5ms disk access latency for evaluation. Then, three allocation policies (random, selective, and selective merging) are compared in this

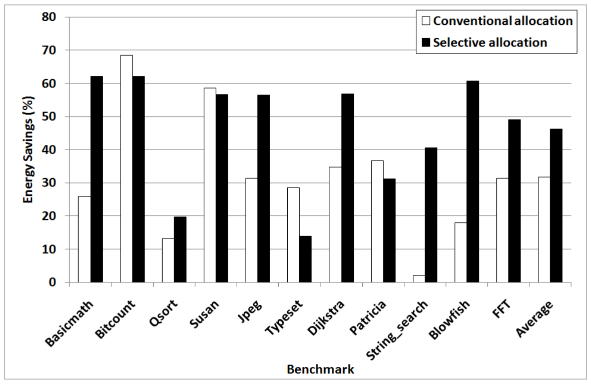


Figure 9: Latency overhead against DRAM main memory

evaluation. The random allocation randomly assigns a request to PRAM or disk. The selective allocation uses PRAM only for the request whose size is under 64KB. The selective merging uses both selective allocation and write request merging which is proposed in section 4.2. Figure 10 shows the evaluation result.

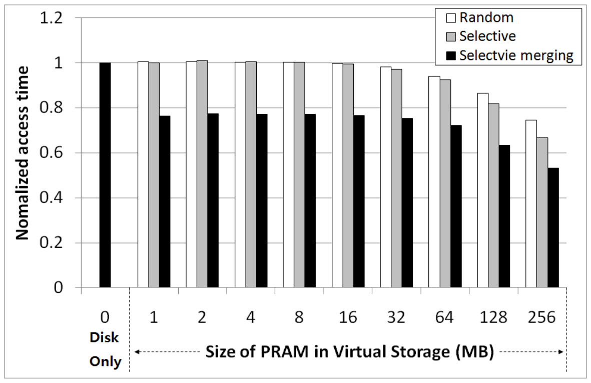


Figure 10: Total storage access time of virtual storage

Figure 10 presents that the use of PRAM decrease the total access time of disk. It is because PRAM is free from seek latency and favorable for small size random access. However, random allocation cannot fully use the PRAM because it dissipates PRAM for large size sequential data. Although the selective allocation reduces the access time of virtual storage, it does not effective when the size of PRAM is small. If the size of PRAM is small, PRAM space is filled with fragmented sequential requests soon. Many of small size random requests do not have the benefits of PRAM.

On the other hand, selective allocation with write request merging is very effective because request merging help to find more sequential data. The virtual storage can allocate the more random requests to PRAM. It reduces the number of disk accesses and more increase the performance of virtual storage. If 256MB PRAM is used as storage in virtual storage, we decrease more than 40% of access time of disk.

6 Further work

Although the selective allocation statically allocates memory pages by using the general characteristics of segments, it cannot fully reflect the dynamic access pattern of memory page. In order to minimize the access latency of hybrid main memory, it is necessary to dynamically balance and move pages between DRAM and PRAM. For example, if some memory pages are frequently updated in a moment, it is better to be migrated to DRAM at that time. If a page in DRAM is occasionally read, it needs to be migrated to PRAM. The page migration also increases the endurance of PRAM because the frequently updated page will be migrated to DRAM.

In order to implement the memory migration, we may use the LRU lists of OS kernel to manage. There are *active_list* and *inactive_list* of pages in Linux kernel. If we always select the page of *inactive_list* to migrate into PRAM, recently accessed page is stored in DRAM. However, the LRU list of OS kernel does not fully monitor the memory access because memory access is occurred without interruption of kernel. Therefore, we need to think about hardware and software design of kernel LRU list to reflect the memory access pattern.

Also, we described that PRAM is good for storage alternatives. However, all previous file systems statically assign non-volatile RAM only for storage although PRAM can be used both for main memory and storage system. In order to maximize the advantage of PRAM, we need to develop unified management of PRAM for memory and storage devices. All PRAM pages need to be freely allocated for main memory and storage. Linux kernel should be implemented to control the use of PRAM according to overall system status.

7 Conclusion

PRAM will be widely used in the future computing system as memory or storage alternatives. In this paper, we consider the Linux kernel level support to exploit PRAM. We use PRAM for hybrid main memory and propose new page allocation algorithm. Consequently, we minimize the performance degradation and endurance problems caused by PRAM while reducing 50% energy consumption of main memory. Also, we propose the virtual storage which is a new block level interface using PRAM for storage. It allocates small size random access data into PRAM and reduces the number of disk access. We can decrease more than 40% of access time of disk.

References

[1] OLTP Application I/O and Search Engine I/O. http://traces.cs.umass.edu/index.php/Storage/Storage.

[2] Phase-change memory. http://en.wikipedia.org/wiki/Phase-change_memory.

[3] An-I A. Wang, et al. Conquest: Better Performance through a Disk/Persistent-RAM Hybrid File System. In *Proceedings of 2002 USENIX Annual Technical Conference*, 2002.

[4] Benjamin C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting Phase Change Memory as a Scalable DRAM Alternative. In *Proceedings of the 36th annual International Symposium on Computer Architecture*, 2009.

[5] David Meisner, David Meisner, Thomas F. Wenisch. PowerNap: eliminating server idle power. In *roceeding of the 14th international conference on Architectural support for programming languages and operating systems*, 2009.

[6] Ethan L. Miller, Scott A. Brandt, and Darrell D. E. long. HeRMES: High-performance reliable MRAM-enabled storage. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, 2001.

[7] G. Dhiman, R. Ayoub, and T. Rosing. PDRAM:A Hybrid PRAM and DRAM Main Memory System. In *Proceedings of the 46th Annual Design Automation Conference*, 2009.

[8] Moinuddin K. Qureshi, V. Srinivassan, and Jude A. Rivers. Scalable High Performance Main Memory System Using Phase-Change Memory Technology. In *Proceedings of the 36th annual*

International Symposium on Computer Architecture, 2009.

[9] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52–60, 2006.

[10] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology. In *Proceedings of the 36th annual International Symposium on Computer Architecture*, 2009.

[11] S. Eilert, M. Leinwander, and G. Crisenza. Phase Change Memory: A new memory enables new memory usage models. *2009 IEEE International Memory Workshop*, pages 1–2, 2009.

[12] Y. Park, S. H. Lim, C. Lee, K. H. Park, et. al. PFFS:A Scalable Flash Memory File System for the Hybrid Architecture of Phase change RAM and NAND Flash. In *Proceedings of the 2008 ACM symposium on Applied computing*, 2008.

The Virtual Contiguous Memory Manager

Zach Pfeffer
Qualcomm Innovation Center (QuIC)
zpfeffer@quicinc.com

Abstract

An input/output memory management unit (IOMMU) maps device addresses to physical addresses. It also insulates the system from spurious or malicious device addresses and allows fine-grained mapping attribute control. The Linux kernel core does not contain a generic API to handle IOMMU mapped memory; device driver writers must implement device specific code to interoperate with the Linux kernel core. As the number of IOMMUs increases, coordinating the many address spaces mapped by all discrete IOMMUs becomes difficult without in-kernel support.

To address this complexity the Qualcomm Innovation Center (QuIC) created the Virtual Contiguous Memory Manager (VCMM) API. The VCMM API enables device independent IOMMU control, VMM interoperation and non-IOMMU enabled device interoperation by treating devices with or without IOMMUs and all CPUs with or without MMUs, their mapping contexts and their mappings using common abstractions. Physical hardware is given a generic device type and mapping contexts are abstracted into Virtual Contiguous Memory (VCM) regions. Users "reserve" memory from VCMs and "back" their reservations with physical memory. We have implemented the VCMM to manage the IOMMUs of an upcoming ARM based SoC. The implementation will be posted to the Code Aurora Foundation's site.

1 Motivation and Opportunities

Driver writers who control devices with IOMMUs must contend with device control and memory management. Driver writers have a large device driver API that they can leverage to control their devices, but they are lacking a unified API to help them program mappings into IOMMUs and share those mappings with other devices and CPUs in the system.

Sharing is complicated by Linux?s CPU centric VMM. The CPU centric model generally makes sense because average hardware only contains a MMU for the CPU and possibly a graphics MMU. If every device in the system has one or more MMUs, a CPU centric memory management (MM) programming model breaks down.

The VCMM was built to allow abstract device programming and mapping interoperation.

2 VCMM Abstractions

Abstracting IOMMU programming into a common API has already begun in the Linux kernel. It was built to abstract the difference between AMD's and Intel's IOMMUs to support x86 virtualization on both platforms. The interface is listed in kernel/include/linux/iommu.h. It contains interfaces for mapping and unmapping as well as 'domain management.' This interface has not gained widespread use outside the x86; PA-RISC, Alpha and SPARC architectures and ARM and PowerPC platforms all use their own mapping modules to control their IOMMUs. The VCMM contains an IOMMU programming layer, but since its abstraction supports map management independent of device control, the layer is not used directly. This higher-level view enables a new kernel service, not just an IOMMU interoperation layer.

Looking at mapping from a system-wide perspective reveals a general graph problem. The VCMM's API is built to manage the general mapping graph. Each node that talks to memory, either through an MMU or directly (physically mapped) can be thought of as the device end of a mapping edge. The other edge is the physical memory (or intermediate virtual space) that is mapped.

In the direct mapped case the device is assigned a 'one-to-one' MMU. This scheme allows direct mapped devices to participate in general graph management.

The CPU nodes can also be brought under the same mapping abstraction with the use of a light overlay on the existing VMM. This light overlay allows VMM managed mappings to interoperate with the common API. The light overlay enables this without substantial modifications to the existing VMM.

In addition to CPU nodes that are running Linux (and the VMM), remote CPU nodes that may be running other operating systems can be brought into the general abstraction. Routing all memory management requests from a remote node through the central memory management framework enables new features like system-wide memory migration. This feature may only be feasible for large buffers that are managed outside of the fast-path, but having remote allocation in a system enables features that are impossible to build without it.

The fundamental objects that support these abstractions are:

- Virtual Contiguous Memory Regions
- Reservations
- Associated Virtual Contiguous Memory Regions
- Memory Targets
- Physical Memory Allocations

In a nut-shell, users allocate Virtual Contiguous Memory Regions and associate those regions with one or more devices by creating an Associated Virtual Contiguous Memory Region. Users then create Reservations from the Virtual Contiguous Memory Region. At this point no physical memory has been committed to the reservation. To associate physical memory with a reservation a Physical Memory Allocation is created and the Reservation is backed with this allocation.

3 Virtual Contiguous Memory Regions

A Virtual Contiguous Memory Region (VCM) abstracts the memory space a device 'sees.' The addresses of the region are only used by the devices which are associated with the region. This address space would normally be implemented as a device page-table.

A VCM is created and destroyed with three functions:

```
vcm_id = vcm_create(start_addr, len);
```

```
vcm_id = vcm_create_from_prebuilt(ext_
vcm_id);
```

```
vcm_free(vcm_id);
```

start_addr is an offset into the address space where allocations will start from. len is the length from start_addr of the VCM. Both functions generate a vcm_id which is an opaque instance of a VCM.

ext_vcm_id is used to pass a request to the VMM to generate a vcm_id. In the current implementation the call simply makes a note that the vcm_id is a VMM vcm_id for other interfaces usage. This 'muxing' is seen throughout the implementation.

vcm_create() and vcm_create_from_prebuilt() produce vcm_ids for virtually mapped devices (IOMMUs and CPUs). To create a one-to-one mapped VCM users pass the start_addr and len of the physical region. The VCMM matches this and records that the vcm_id is a one-to-one VCM.

The newly created vcm_id can be passed to any function that needs to operate on or with a virtual contiguous memory region. Its main attributes are a start_addr and a len as well as an internal setting that allows the implementation to mux between true virtual spaces, one-to-one mapped spaces and VMM managed spaces.

The current implementation uses the genalloc library to manage the VCM for IOMMU devices.

4 Reservations

A Reservation is a contiguous region allocated from a VCM. There is no physical memory associated with it.

A Reservation is created and destroyed with:

```
res_id = vcm_reserve(vcm_id, len, attr);
```

```
vcm_unreserve(res_id);
```

A vcm_id is a VCM created above. len is the length of the request. It can be up-to the length of the VCM region the reservation is being created from. attr are mapping attributes: read, write, execute, user, supervisor, secure, not-cached, write-back/write-allocate, write-back/no write-allocate, write-through. These attrs can be changed to match to any architecture.

The implementation calls gen_pool_alloc() for IOMMU devices, alloc_vm_area() for VMM areas and is a pass through for one-to-one mapped areas.

5 Associated Virtual Contiguous Memory Regions and Activation

An Associated Virtual Contiguous Memory Region (AVCM) is a mapping of a VCM to a device. The mapping can be active or inactive.

An AVCM is managed with:

```
avcm_id = vcm_assoc(vcm_id, dev_
id, attr);
```

```
vcm_deassoc(avcm_id);
```

```
vcm_activate(avcm_id);
```

```
vcm_deactivate(avcm_id);
```

A vcm_id is a VCM created above. dev_id is an opaque device handle that's passed down to the device driver the VCMM muxes in to handle a request. attr are association attributes: split, use-high or use-low. split controls which address hit a 'high-address' page-table and which addresses hit a "low-address" page-table. For instance, all addresses whose most-significant-bit is one would use the "high-address" page-table, any other register would use the 'low address' page-table. One vcm_id can be associated with many devices and many vcm_ids can be associated with one device.

An AVCM is only a link. To program and deprogram a device with a VCM the user calls vcm_activate() and vcm_deactivate().For IOMMU devices, activating a mapping programs the base address of a page-table into an IOMMU. For VMM and one-to-one based devices, mappings are active immediately; the API does require an activation call for them for internal reference counting.

6 Memory Targets

A Memory Target is a platform independent way of specifying a physical pool; it abstracts a pool of physical memory. The physical memory pool may be physically discontinuous, need to be allocated from in a unique way or have other user-defined attributes.

7 Physical Memory Allocation and Reservation Backing

Physical memory is allocated as a separate step from reserving memory. This allows multiple reservations to back the same physical memory. A Physical Memory Allocation is managed using the following functions:

```
physmem_id = vcm_phys_
alloc(memtype, len, attr);
```

```
vcm_phys_free(physmem_id);
```

```
vcm_back(res_id, physmem_id);
```

```
vcm_unback(res_id);
```

attr can include an alignment request, a specification to map memory using various block sizes and/or to use physically contiguous memory. memtype is one of the memory types listed in Memory Targets.

The current implementation manages two pools of memory. One pool is a contiguous block of memory and the other is a set of contiguous block pools. In the current implementation the blocks pools contain 4K, 64K and 1M blocks. The physical allocator does not try to split blocks from the contiguous block pools to satisfy requests.

The use of 4K, 64K and 1M blocks solves a problem with some IOMMU hardware. IOMMUs are placed in front of multimedia engines to provide a contiguous address space to the device. Multimedia devices need large buffers and large buffers may map to a large number of physical blocks. IOMMUs tend to have small translation lookaside buffers (TLBs). The number of physical blocks that map a given range needs to be small or else the IOMMU will continually fetch new translations during a typical streamed multimedia flow since the TLB is small. By using a 1 MB mapping (or 64K mapping) instead of a 4K mapping the number of misses can be minimized, allowing the multimedia block to meet its performance goals.

8 Low Level Control

It is necessary to access attributes of the abstractions. The API contains many functions but the two that are typically used are:

```
devaddr = vcm_get_dev_addr(res_id);
```

```
vcm_hook(dev_id, user_
handler, void *data);
```

The first function, vcm_get_dev_addr() returns a device address given a reservation. This device address is a virtual IOMMU address for reservations on IOMMU VCMs, a virtual VMM address for reservations on VMM VCMs and a 'virtual' (physical) address for one-to-one devices.

The second function, vcm_hook allows a caller in the kernel to register a user_handler. The handler is passed the data during a fault. The user can return 1 to indicate that the underlying driver should handle the fault and retry the transaction or can return 0 to halt the transaction. If the user doesn't register a handler the low-level driver will print a warning and terminate the transaction.

9 A Detailed Walk Through

The following call sequence walks through a typical allocation sequence. In the first stage the memory for a device is reserved and backed. This occurs without mapping the memory into a VMM VCM region. The second stage maps the first VCM region into a VMM VCM region so the kernel can read or write it. The second stage is not necessary if the VMM does not need to read or modify the contents of the original mapping. Figure 1 shows the mappings schematically.

Stage 1: Map and Allocate Memory for a Device

The call sequence starts by creating a VCM region:

```
vcm_id = vcm_create(start_addr, len);
```

The next call associates a VCM region with a device:

```
avcm_id = vcm_assoc(vcm_id, dev_
id, attr);
```

To activate the association users call vcm_activate() on the avcm_id from the associate call. This programs the underlining device with the mappings.

```
vcm_activate(avcm_id);
```

Once a VCM region is created and associated it can be reserved from.

```
res_id = vcm_reserve(vcm_id, res_
len, res_attr);
```

A user allocates physical memory:

```
physmem_id = vcm_phys_
alloc(memtype, len, phys_attr);
```

To back the reservation with the physical memory allocation the user calls:

```
vcm_back(res_id, physmem_id);
```

Stage 2: Map the Device's Memory into the VMM's VCM region

If the VMM needs to read and/or write the region that was just created the following calls are made.

The first call creates a prebuilt VCM:

```
vcm_vmm_id = vcm_from_prebuit(ext_vcm_
id);
```

The prebuilt VCM is associated with the CPU device and activated:

```
avcm_vmm_id = vcm_assoc(vcm_vmm_id, dev_
cpu_id, attr);
```

```
vcm_activate(avcm_vmm_id);
```

A reservation is made on the VMM VCM:

```
res_vmm_id = vcm_reserve(vcm_vmm_
id, res_len, attr);
```

Once the topology has been set up a vcm_back() allows the VMM to read the memory using the physmem_id generated in stage 1:

```
vcm_back(res_vmm_id, physmem_id);
```

10 Mapping IOMMU, one-to-one and VMM Reservations

Figure 3 demonstrates mapping IOMMU, one-to-one and VMM reservations to the same physical memory. It shows the use of phys_addr and phys_size to create a contiguous VCM for one-to-one mapped devices. Figure 2 shows the mappings schematically.

11 Summary

The VCMM is an attempt to abstract attributes of three distinct classes of mappings into one API. The VCMM allows users to reason about mappings as first class objects. It also allows memory mappings to flow from the traditional 4K mappings prevalent on systems today to more efficient block sizes. Finally, it allows users to manage mapping interoperation without becoming VMM experts. These features will allow future systems with many MMU mapped devices to interoperate simply and therefore correctly.

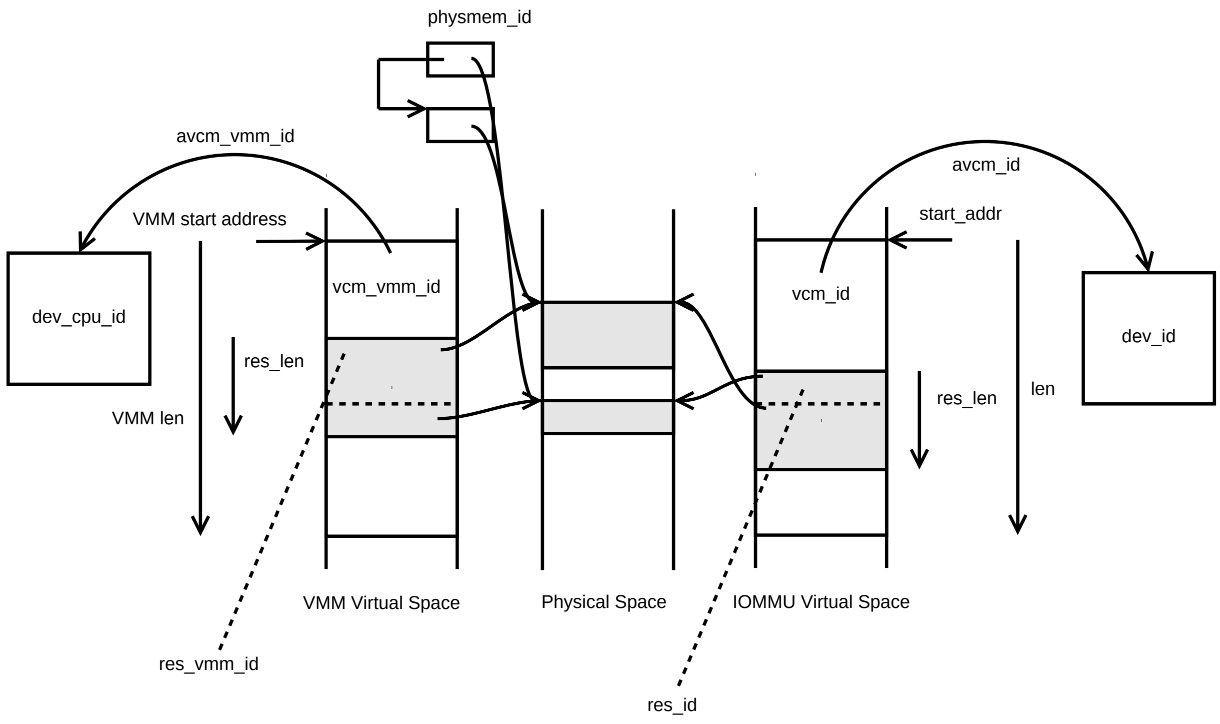


Figure 1: Walk Through

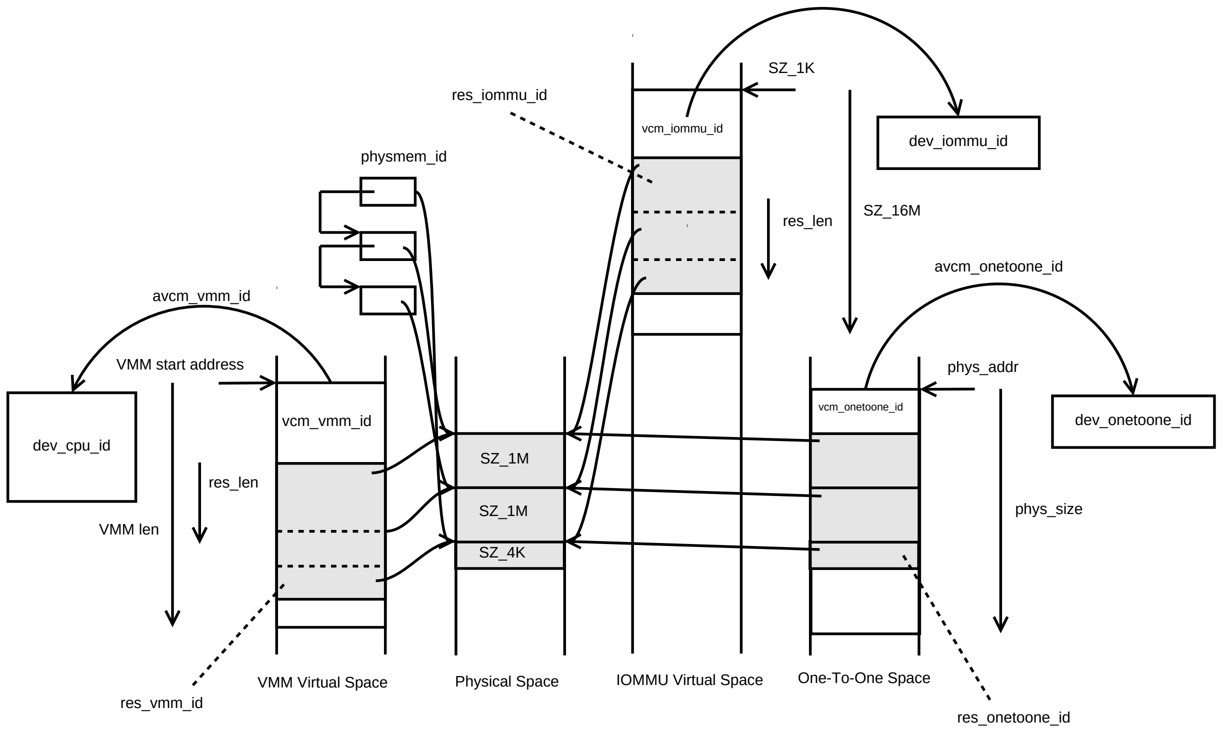


Figure 2: Mapping IOMMU, One-to-One and VMM Reservations

```
physmem_id = vcm_phys_alloc(memtype, SZ_2MB + SZ_4K, CONTIGUOUS);}
vcm_iommu_id = vcm_create(SZ_1K, SZ_16M);}
vcm_onetoone_id = vcm_create(phys_addr, phys_size);}
vcm_vmm_id = vcm_from_prebuit(ext_vcm_id);}

avcm_iommu_id = vcm_assoc(vcm_iommu_id, dev_iommu_id, attr0);}
avcm_onetoone_id = vcm_assoc(vcm_onetoone_id, dev_onetoone_id, attr1);}
avcm_vmm_id = vcm_assoc(vcm_vmm_id, dev_cpu_id, attr2);}

vcm_activate(avcm_iommu_id);}
vcm_activate(avcm_onetoone_id);}
vcm_activate(avcm_vmm_id);}

res_iommu_id = vcm_reserve(vcm_iommu_id, SZ_2MB + SZ_4K, attr);}
res_onetoone_id = vcm_reserve(vcm_onetoone_id, SZ_2MB + SZ_4K, attr);}
res_vmm_id = vcm_reserve(vcm_vmm_id, SZ_2MB + SZ_4K, attr);}

vcm_back(res_iommu_id, physmem_id);}
vcm_back(res_onetoone_id, physmem_id);}
vcm_back(res_vmm_id, physmem_id);}
```

Figure 3: Mapping IOMMU, One-to-One and VMM Reservations Example

Transactional system calls on Linux

Donald E. Porter
The University of Texas at Austin
porterde@cs.utexas.edu

Emmett Witchel
The University of Texas at Austin
witchel@cs.utexas.edu

Abstract

Have you ever had to manually back out an unsuccessful software install? Has a machine ever crashed on you while adding a user, leaving the group, password and shadow files inconsistent? Have you struggled to eliminated time-of-check-to-time-of-use (TOCTTOU) race conditions from an application? All of these problems have a single underlying cause: programmers cannot group multiple system calls into a single, consistent operation. If users (and kernel developers) had this power, there are a variety of innovative services they could build and problems they could eliminate. This paper describes system transactions and a variety of applications based on system transactions. We add system calls to begin, end, and abort a transaction. A system call that executes within a transaction is isolated from the rest of the system. The effects of a system transaction are undone if the transaction fails.

This paper describes a research project that developed transactional semantics for 152 Linux system calls and abstractions including signals, process creation, files, and pipes. The paper also describes the practical challenges and trade-offs in implementing transactions in Linux. The code changes needed to support transactions are substantial, but so are the benefits. With no modifications to dpkg itself, we were able to wrap an installation of OpenSSH in a system transaction. The operating system rolls back failed installations automatically, preventing applications from observing inconsistent files during the installation, and preserving unrelated, concurrent updates to the file system. Overheads for using transactions in an application like software installation range from 10-70%.

1 Introduction

A number of programming tasks are impossible to write robustly using the POSIX API. For example, backing out a failed software installation or upgrade is a major hassle for system administrators because the software spans multiple files with tightly coupled dependences. For instance, a new version of a binary may expect a new configuration file format or a new binary may not link with previous versions of the supporting libraries. If a software installation fails or the machine crashes during installation, these tight dependences are broken, often rendering the software unusable. If the failed upgrade is for a core system utility, such as the shell, the entire system may stop working. Software installation tools, such as `yum` and `apt`, have evolved to provide sophisticated support for tracking package dependences and automatically uninstalling libraries that are no longer needed, but even these systems fail—truly robust failure recovery remains elusive.

Even simple changes to system settings, such as adding user accounts, are prone to subtle errors or race conditions with other administrators. Local user accounts are stored across three files that need to be mutually consistent: `/etc/passwd`, `/etc/shadow`, and `/etc/group`. Utilities like `vipw` and `useradd` help ensure that these account files are formatted correctly and mutually consistent. These utilities create lock files to prevent concurrent modifications, but this cannot prevent a careless administrator from ignoring the lock file and editing the password files directly. Moreover, these utilities cannot ensure that updates to these files are mutually consistent if the system crashes during an operation. For instance, suppose the system crashes after `useradd` writes `/etc/passwd` but before it writes `/etc/shadow`. After rebooting the system, the new user will not be able to log on, yet `useradd` will fail because it thinks the user already exists, leaving the system administrator to manually repair the database files.

Race conditions for OS-managed resources, including the file system namespace, can cause security problems for programs that run as root. Despite their conceptual simplicity, time-of-check-to-time-of-use, or TOCT-

```
   Victim                    Attacker
if(access('foo')){
                          symlink('secret','foo');
  fd=open('foo');
  write(fd,...);
  ...
}
```

```
   Victim                    Attacker
                          symlink('secret','foo');
sys_xbegin();
if(access('foo')){
  fd=open('foo');
  write(fd,...);
  ...
}
sys_xend();
                          symlink('secret','foo');
```

Figure 1: An example of a TOCTTOU attack, followed by an example that eliminates the race using system transactions. The attacker's symlink is serialized (ordered) either before or after the transaction, and the attacker cannot see partial updates from the victim's transaction, such as changes to `atime`.

TOU, races [7] have created over 600 vulnerabilities in real, deployed applications [10]. A TOCTTOU race most commonly occurs as depicted in Figure 1, when a malicious program changes the file system namespace with a `symlink`, just between the check (`access`) and the use (`open`) in an application with root privilege. This is a common attack vector for privilege escalation, as these race conditions can trick a process with root-privilege to overwrite a sensitive file, such as the password database.

Each of these seemingly unrelated problems share an underlying cause: developers cannot group multiple system calls into a single, consistent operation. The ideal software installer would be able to atomically replace multiples files on the system at once; when the installation finished, either all of the updates take effect or they are all rolled back. Similarly, adding a user should atomically update each relevant configuration file, and prevent a concurrent user from interfering with the updates. Finally, an application with root privileges should be able to request that a permissions check and subsequent file open be executed in isolation from potentially interfering applications.

Some of these problems, such as TOCTTOU races, are being addressed in the kernel by adding more functionality to existing system calls. The `open` system call has acquired a number of flags that bundle in tasks like checking that the file doesn't exist and conditionally creating the file. The current `open` implementation is more complex than a simple `open`, `create`, and `stat` combined. Similarly, the `rename` system call has been heavily used by applications, such as editors, to atomically replace a single file. The `rename` implementation is so complex that Linux uses a single, file-system wide mutex to synchronize renames in all but the simplest cases, harming system scalability. In order to address TOCTTOU specifically, `openat` and over a dozen similar variants have been added to Linux. These calls essentially allow applications to reimplement their own private `dcache`, at a substantial performance and complexity cost to the application [16].

As an alternative, *system transactions* allow developers to compose a series of simple system calls into a more complex operation. The kernel guarantees that a system transaction appears to execute as one isolated, atomic operation. System transactions eliminate the need for complex work-arounds in applications, and even obviate the need for such semantically heavy system calls as `rename`. Windows Vista and later have already adopted a transactional file system and registry to address problems arising from crashes during software installation [14]. Rather than having to petition kernel developers for a point solution to the next race condition or crash-consistency issue, system transactions give developers the tools to solve their own problems.

This paper describes ongoing research at the University of Texas at Austin to develop transactional system calls on a variant of Linux, called TxOS. The work has appeared in previous research venues [11, 12]; this paper reviews the design of the system with a focus on the needed changes to the Linux source code and the rationale for the design decisions. Section 2 provides an overview of the TxOS design and Section 3 describes the implementation in more detail. Sections 4 and 5 measure the performance of system transactions. Section 6 describes why system transactions are a better solution than file locking or a transactional file system, Section 7 describes ongoing and future directions for the project, and Section 8 concludes.

2 TxOS Overview

System transactions provide ACID semantics for updates to OS resources, such as files, pipes, and signals.

Subsystem	Tot.	Part.	Examples
Credentials	34	1	getuid, getcpu, setrlimit (partial)
Processes	13	3	fork, vfork, clone, exit, exec (partial)
Communication	15	0	rt_sigaction, rt_sigprocmask, pipe
Filesystem	63	4	link, access, stat, chroot, dup, open, close, write, lseek
Other	13	6	time, nanosleep, ioctl (partial), mmap2 (partial)
Totals	138	14	Grand total: 152

Unsupported		
Processes	33	nice, uselib, iopl, sched_yield, capget
Memory	15	brk, mprotect, mremap, madvise
Filesystem	29	mount, sync, flock, setxattr, io_setup, inotify
File Descriptors	14	splice, tee, sendfile, select, poll
Communication	8	socket, ipc, mq_open, mq_unlink
Timers/Signals	12	alarm, sigaltstack, timer_create
Administration	22	swapon, reboot, init_module, settimeofday
Misc	18	ptrace, futex, times, vm86, newuname
Total	151	

Table 1: Summary of system calls that TxOS completely supports (Tot.) and partially supports (Part.) in transactions, followed by system calls with no transaction support. Partial support indicates that some (but not all) execution paths for the system call have full transactional semantics. Linux 2.6.22.6 on the i386 architecture has 303 total system calls.

In this programming model, both transactional and non-transactional system calls may access the same system state; the OS imposes a global order for all accesses and arbitrates contention fairly. The interface for system transactions is intuitive and simple, allowing a programmer to wrap a block of unmodified code in a transaction simply by adding `sys_xbegin()` and `sys_xend()`.

TxOS implements system transactions by isolating data read and written in a transaction (making it invisible to unrelated kernel threads) using existing kernel memory buffers and data structures. When an application writes data to a file system or device, the updates generally go into an OS memory buffer first, allowing the OS to batch updates to the underlying device. By making these buffers copy-on-write for transactions, TxOS isolates transactional data accesses until commit. In TxOS, transactions must fit into main memory, although this limit could be raised in future work by swapping uncommitted transaction state to disk.

TxOS isolates updates to kernel data structures using recent implementation techniques from object-based software transactional memory systems. These techniques are a departure from the logging and two-phase locking approaches of databases and historic transactional operating systems, such as QuickSilver [15] and Locus [17] (Section 2.3). TxOS's isolation mechanisms are optimistic, allowing concurrent transactions on the assumption that conflicts are rare.

Table 1 summarizes the system calls and resources for which TxOS supports transactional semantics, including the file system, process and credential management, signals, and pipes. A partially supported system call means that some processing paths are fully transactional, and some are not. For example, `ioctl` is essentially a large switch statement, and TxOS does not support transactional semantics for every case. When the user makes an unsupported system call or a partially supported call cannot support transactional semantics, the system logs a warning or aborts the transaction, depending on the flags passed to `sys_xbegin()`.

Ideal support for system transactions would include every reasonable system call. TxOS supports a subset of Linux system calls as shown in Table 1. The count of 152 supported system calls shows the relative maturity of the prototype, but also indicates that it is incomplete. The count of unsupported system calls does not proportionately represent the importance or challenge of the remaining work because many resources, such as network sockets, IPC, etc., primarily use the common file system interfaces. For instance, extending transactions to include networking (a real challenge) would increase the count of supported calls by 5, whereas transaction support for extended file attributes (a fairly straightforward extension) would add 12 system calls. The remaining count of system calls falls into three categories: substantial extensions (memory management, communication), straightforward, but perhaps less common or important (process management, timers, most remaining file interfaces), and operations that are highly unlikely

Function Name	Description
int sys_xbegin (int flags)	Begin a transaction. The flags specify transactional behavior, including automatically restarting the transaction after an abort, ensuring that committed results are on stable storage (durable), and aborting if an unsupported system call is issued. Returns status code.
int sys_xend()	End of transaction. Returns whether commit succeeded.
void sys_xabort (int no_restart)	Aborts a transaction. If the transaction was started with restart, setting no_restart overrides that flag and does not restart the transaction.

Table 2: TxOS API

to be useful inside a transaction (e.g., `reboot`, `mount`, `init_module`, etc.). TxOS supports transactional semantics for enough kernel subsystems to demonstrate the power and utility of system transactions.

2.1 System transactions for system state

Although system transactions provide ACID semantics for system state, they do not provide these semantics for application state. System state includes OS data structures and device state stored in the operating system's address space, whereas application state includes only the data structures stored in the application's address space. When a system transaction aborts, the OS restores the kernel state to its pre-transaction state, but it does not revert application state.

For most applications, we expect programmers will use a library or runtime system that transparently manages application state as well as system transactions. In simple cases, such as the TOCTTOU example, the developer could manage application state herself. TxOS provides single-threaded applications with an automatic checkpoint and restore mechanism for the application's address space that marks the pages copy-on-write (similar to Speculator [9]), which can be enabled with a flag to `sys_xbegin()` (Table 2). In a recent paper [11], we describe how system transactions integrate with hardware and software transactional memory, providing a complete transactional programming model for multi-threaded applications.

2.2 Communication model

Code that communicates outside of a transaction and requires a response cannot be encapsulated into a single transaction. Communication outside of a transaction violates isolation. For example, a transaction may send a message to a non-transactional thread over an IPC channel, which the system will buffer until commit. If the code waits for a reply to the buffered message, the application will deadlock. The programmer is responsible for avoiding this send/reply idiom within a transaction.

Communication among threads within the same transaction is unrestricted. This paper only considers system transactions on a single machine, but future work could allow system transactions to span multiple machines.

2.3 Managing transactional state

Databases and historical transactional operating systems typically update data in place and maintain an undo log. This approach is called **eager version management** [5]. These systems isolate transactions by locking data when it is accessed and holding the lock until commit. This technique is called two-phase locking, and it usually employs locks that distinguish read and write accesses. Because applications generally do not follow a globally consistent order for data accesses, these systems can deadlock. For example, one thread might read file A then write file B, while a different thread might read file B, then write file A.

The possibility of deadlock complicates the programming model of eager versioning transactional systems. Deadlock is commonly addressed by exposing a timeout parameter to users. Setting the timeout properly is a challenge. If it is too short, it can starve long-running transactions. If it is too long, it can destroy the performance of the system.

Eager version management degrades responsiveness in ways that are not acceptable for an OS kernel. If an interrupt handler, high priority thread, or real-time thread aborts a transaction, it must wait for the transaction to process its undo log (to restore the pre-transaction state) before it can safely proceed. This wait jeopardizes the system's ability to meet its timing requirements.

In contrast, transactions in TxOS operate on private copies of data structures, known as **lazy version management**. Transactions never hold kernel locks across system calls. Lazy versioning requires TxOS to hold locks only long enough to make a private copy of the relevant data structure. By enforcing a global ordering for kernel locks, TxOS avoids deadlock. TxOS can abort transactions instantly—the winner of a conflict does not wait for the aborted transaction to process its undo log.

The primary disadvantage of lazy versioning is the commit latency due to copying transactional updates from the speculative version to the stable version of the data structures. As we discuss in Section 3, TxOS minimizes this overhead by splitting objects, turning a `memcpy` of the entire object into a pointer copy.

2.4 Interoperability and fairness

TxOS allows flexible interaction between transactional and non-transaction kernel threads. TxOS efficiently orders transactions with non-transactional accesses inside the kernel by requiring all system calls follow the same locking discipline, and by requiring that transactions annotate accessed kernel objects. When a thread, transactional or non-transactional, accesses a kernel object for the first time, it must check for a conflicting annotation. The scheduler arbitrates conflicts when they are detected. In many cases, this check is performed at the same time as a thread acquires a lock for the object.

Interoperability is a weak spot for previous transactional systems. In most transactional systems, a conflict between a transaction and a non-transactional thread (called an **asymmetric conflict** [13]) must be resolved by aborting the transaction. This approach undermines fairness. In TxOS, because asymmetric conflicts are often detected before a non-transactional thread enters a critical region, the scheduler has the option of suspending the non-transactional thread, allowing for fairness between transactions and non-transactional threads.

3 Implementation

This section describes how system transactions are implemented in the TxOS kernel and the reasons why the TxOS implementation deviates from the Linux kernel. TxOS provides transactional semantics for 152 of 303 system calls in Linux, presented in Table 1. The supported system calls include process creation and termination, credential management operations, sending and receiving signals, and file system operations.

System transactions in TxOS add roughly 3,300 lines of code for transaction management, and 5,300 lines for object management. TxOS also requires about 14,000 lines of minor changes to convert kernel code to use the new object type system and to insert checks for asymmetric conflicts when executing non-transactionally. Compared to the overall size of the kernel, these changes are small; however, some changes are invasive at points and this section explains why the changes were necessary and potential alternatives.

3.1 Object versioning

TxOS maintains multiple versions of kernel data structures so that system transactions can isolate the effects of system calls until transactions commit (i.e., hide the effects from other kernel threads), and in order to undo the effects of transactions if they cannot complete. Data structures private to a process, such as the current user id or the file descriptor table, are versioned with a simple checkpoint and restore scheme. For shared kernel data structures, however, TxOS implements a versioning system that borrows techniques from software transactional memory systems [3] and recent concurrent programming systems [4].

When a transaction accesses a shared kernel object, such as an `inode`, it acquires a private copy of the object, called a **shadow** object. All system calls within the transaction use this shadow object in place of the **stable** object until the transaction commits or aborts. The use of shadow objects ensures that transactions always have a consistent view of the system state. When the transaction commits, the shadow objects replace their stable counterparts. If a transaction cannot complete, it simply discards its shadow objects.

Any given kernel object may be the target of pointers from several other objects, presenting a challenge to replacing a stable object with a newly-committed shadow object. A naïve approach might update the pointers to an object when that object is committed. This naïve approach is impractical, as some objects (e.g., inodes) are pointed to by a substantial number of other data structures which the object itself doesn't reference.

Splitting objects into header and data In order to allow efficient commit of lazy versioned data, TxOS decomposes objects into a stable **header** component and

```
struct inode_header {
  atomic_t      i_count; // Reference count
  spinlock_t    i_lock;
  inode_data    *data;   // Data object
  // Other objects
  address_space i_data;  // Cached pages
  tx_data xobj;   // for conflict detection
  list i_sb_list;  // kernel bookkeeping
};

struct inode_data {
  inode_header *header;
  // Common inode data fields
  unsigned long i_ino;
  loff_t        i_size; // etc.
};
```

Figure 2: A simplified inode structure, decomposed into header and data objects in TxOS. The header contains the reference count, locks, kernel bookkeeping data, and the objects that are managed transactionally. The inode_data object contains the fields commonly accessed by system calls, such as stat, and can be updated by a transaction by replacing the pointer in the header.

a volatile, transactional **data** component. Figure 2 provides an example of this decomposition for an inode. The object header contains a pointer to the object's data; transactions commit changes to an object by replacing this pointer in the header to a modified copy of the data object. The header itself is never replaced by a transaction, which eliminates the need to update pointers in other objects; pointers point to headers. The header can also contain data that is not accessed by transactions. For instance, the kernel garbage collection thread (kswapd) periodically scans the inode and dentry (directory entry) caches looking for cached file system data to reuse. By keeping the data for kernel bookkeeping, such as the reference count and the superblock list (i_sb_list in Figure 2), in the header, these scans never access the associated inode_data objects and avoid restarting active transactions.

Decomposing objects into headers and data also provides the advantage of the type system ensuring that transactional code always has a speculative object. For instance, in Linux, the virtual file system function vfs_link takes pointers to inodes and dentries, but in TxOS these pointers are converted to the shadow types inode_data and dentry_data. When modifying Linux, using the type system allows the compiler to find all of the code that needs to acquire a speculative object, ensuring completeness. The type system also allows the use of interfaces that minimize the time spent looking up shadow objects. For example, when the path name resolution code initially acquires shadow data objects, it then passes these shadow objects directly to helper functions such as vfs_link and vfs_unlink. The virtual file system code acquires shadow objects once on entry and passes them to lower layers, minimizing the need for filesystem-specific code to reacquire the shadow objects.

Multiple data objects TxOS decomposes an object into multiple data payloads when it houses data that can be accessed disjointly. For instance, the inode_header contains both file metadata (owner, permissions, etc.) and the mapping of file blocks to cached pages in memory (i_data). A process may often read or write a file without updating the metadata. TxOS versions these objects separately, allowing metadata operations and data operations on the same file to execute concurrently when it is safe.

Read-only objects Many kernel objects are only read in a transaction, such as the parent directories in a path lookup. To avoid the cost of making shadow copies, kernel code can specify read-only access to an object, which marks the object data as read-only for the length of the transaction. Each data object has a transactional reader reference count. If a writer wins a conflict for an object with a non-zero reader count, it must create a new copy of the object and install it as the new stable version. The OS garbage collects the old copy via read-copy update (RCU) [6] when all transactional readers release it and after all non-transactional tasks have been descheduled. This constraint ensures that all active references to the old, read-only version have been released before it is freed and all tasks see a consistent view of kernel data. The only caveat is that a non-transactional task that blocks must re-acquire any data objects it was using after waking, as they may have been replaced and freed by a transaction commit. Although it complicates the kernel programming model slightly, marking data objects as read-only in a transaction is a structured way to eliminate substantial overhead for memory allocation and copying. Special support for read-mostly transactions is a common optimization in transactional systems, and RCU is a technique to support efficient, concurrent access to read-mostly data.

```
static inline struct inode_data *
tx_get_inode(struct inode *inode,
       enum access_mode mode){
   if(!aborted_tx())
      return error;
   else if(!live_transaction()){
      return inode->inode_data;
   else {
      contend_for_object(inode, mode);
      return get_private_copy(inode);
   }
}
```

```
    struct inode *inode;
    // Replace idata = inode->inode_data with
    inode_data *idata = tx_get_inode(inode, RW);
```

Figure 3: Pseudo-code for the hook used to acquire an inode's data object, and an example of its use in code.

State	Description
`exclusive`	Any attempt to access the list is a conflict with the current owner
`write`	Any number of insertions and deletions are allowed, provided they do not access the same entries. Reads (iterations) are not allowed. Writers may be transactions or non-transactional tasks.
`read`	Any number of readers, transactional or non-transactional, are allowed, but insertions and deletions are conflicts.
`notx`	There are no active transactions, and a non-transactional thread may perform any operation. A transaction must first upgrade to `read` or `write` mode.

Table 3: The states for a transactional list in TxOS. Having multiple states allows TxOS lists to tolerate access patterns that would be conflicts in previous transactional systems.

3.2 Impact of data structure changes

The largest source of lines changed in TxOS comes from splitting objects such as inodes into multiple data structures. After a small amount of careful design work in the headers, most of the code changes needed to split objects was rather mechanical.

A good deal of design effort went into assessing which fields might be modified transactionally and must be placed in the data object, and which can remain in the header, including read-only data, kernel-private bookkeeping, or pointers to other data structures that are independently versioned. A second design challenge was assessing when a function should accept a header object as an argument and when it should accept a data object. The checks to acquire a data object are relatively expensive and would ideally occur only once per object per system call. Thus, once a system call path has acquired a data object, it would be best to pass the data object directly to all internal functions rather than reacquire it. This has to be balanced against forcing needless object acquisition in order to call a shared function that only uses the data object in the uncommon case.

Once the function signatures and data structure definitions are in place, the remaining work is largely mechanical. The primary change that must be propagated through the code is replacing certain pointer dereferences with hooks (Figure 3), so that TxOS can redirect requests for a data object to the transaction's private copy where appropriate. It is in this hook code where TxOS checks for conflicts between transactions. By encapsulating this work in a macro, we hide much of the complexity of managing private copies from the rest of the kernel code, reducing the chances for error.

A benefit of changing the object definitions is that it gives us confidence in the completeness of our hook placement. In order to dereference a field that can be modified in a transaction, the code must acquire a reference to a data object through the hook function. If the hook is not placed properly, the code will not compile. A question for future work is assessing to what degree these changes can be automatically applied during compilation using a tool like CIL [8]. This "header crawl" technique leads to more lines of code changed, but increases our confidence that the changes were made throughout the large codebase that is the Linux kernel.

3.3 Lists

Linked lists are a key data structure in the Linux kernel, and they present key implementation challenges. Simple read/write conflict semantics for lists throttle concurrent performance, especially when the lists contain directory entries. For instance, two transactions should both be allowed to add distinct directory entries to a single list, even though each addition is a list write. TxOS adopts techniques from previous transactional memory systems to avoid conflicts on list updates that do not semantically conflict [3]. TxOS isolates list updates with a lock and defines conflicts according to the states described in Table 3. For instance, a list in the `write` state allows concurrent transactional and non-transactional writers, so long as they do not access the same entry. Individual entries that are transactionally added or removed are annotated with a transaction pointer that is used to detect conflicts. If a writing transaction also attempts to read the list contents, it must upgrade the list to `exclusive` mode by aborting all other writers. The `read` state behaves similarly. This design allows maximal list concurrency while preserving correctness.

A second implementation challenge for linked lists is that an object may be speculatively moved from one list to another. This requires a record of membership in both the original list (marked as speculatively deleted) and the new list (marked as speculatively added). Ideally, one would simply embed a second `list_head` in each object for speculatively adding an entry to a new list; however, if multiple transactions are contending for a list entry, it is difficult to coordinate reclaiming the second embedded entry from an aborted transaction. For this reason, if a transaction needs to speculatively add an object to a list, it dynamically allocates a second `list_head`, along with some additional bookkeeping. Dynamic allocation of speculative list entries allows a transaction to defer clean-up of speculatively added entries from an aborted transaction until a more convenient time (i.e., one that does not further complicate the locking discipline for lists).

Although TxOS dynamically allocates `list_head` structures for transactions, the primary `list_head` for an object is still embedded in the object. During commit, a transaction replaces any dynamically allocated, speculative entries with the embedded list head. Thus, non-transactional code never allocates or frees memory for list traversal or manipulation.

A final issue with lists and transactional scalability is that most lists in the Linux kernel are protected by coarse locks, such as the `dcache_lock`. Ideally, two transactions that touch disjoint data should be able to commit concurrently, yet acquiring a coarse lock will cause needless performance loss. Thus, we implemented fine-grained locking on lists, at the granularity of a list. This improves scalability (§ 5), but complicates the locking discipline. Locks in TxOS are ordered by kernel virtual address, except that list locks must be acquired after other object locks. This discipline roughly matches the paradigm in the directory traversal code.

4 Evaluation

This section evaluates the overhead of system transactions in TxOS, as well as its behavior for several case studies, including a transactional software installation and a transactional LDAP server. We perform all of our experiments on a server with 1 or 2 quad-core Intel X5355 processors (for a total of 4 or 8 cores) running at 2.66 GHz with 4 GB of memory. All single-threaded experiments use the 4-core machine, and scalability measurements were taken using the 8 core machine. We compare TxOS to an unmodified Linux kernel, version 2.6.22.6—the same version extended to create TxOS.

4.1 Single-thread system call overheads

A key goal of TxOS is to make transaction support efficient, taking special care to minimize the overhead non-transactional applications incur. To evaluate performance overheads for substantial applications, we measured the average compilation time across three non-transactional builds of the Linux 2.6.22 kernel on unmodified Linux (3 minutes, 24 seconds), and on TxOS (3 minutes, 28 seconds). This slowdown of less than 2% indicates that for most applications, the non-transactional overheads will be negligible. At the scale of a single system call, however, the average overhead is currently 29%, and could be cut to 14% with improved compiler support.

Table 4 shows the performance of common file system system calls on TxOS. We ran each system call 1 million times, discarding the first and last 100,000 measurements and averaging the remaining times. The elapsed cycles were measured using the `rdtsc` instruction. The purpose of the table is to analyze transaction overheads in TxOS, but it does not reflect how a programmer would use system transactions because most system calls are already atomic and isolated. Wrapping a single system call in a transaction is the worst case for TxOS performance because there is very little work across which to amortize the cost of creating shadow objects and commit.

The **Base** column shows the base overhead from adding transactions to Linux. These overheads have a geometric mean of $\sim 3\%$, and are all below 20%, including a

Call	Linux	Base		Static		NoTx		Bgnd Tx		In Tx		Tx	
access	2.4	2.4	1.0×	2.6	1.1×	3.2	1.4×	3.2	1.4×	11.3	4.7×	18.6	7.8×
stat	2.6	2.6	1.0×	2.8	1.1×	3.4	1.3×	3.4	1.3×	11.5	4.1×	20.3	7.3×
open	2.9	3.1	1.1×	3.2	1.2×	3.9	1.4×	3.7	1.3×	16.5	5.2×	25.7	8.0×
unlink	6.1	7.2	1.2×	8.1	1.3×	9.4	1.5×	10.8	1.7×	18.1	3.0×	31.9	7.3×
link	7.7	9.1	1.2×	12.3	1.6×	11.0	1.4×	17.0	2.2×	57.1	7.4×	82.6	10.7×
mkdir	64.7	71.4	1.1×	73.6	1.1×	79.7	1.2×	84.1	1.3×	297.1	4.6×	315.3	4.9×
read	2.6	2.8	1.1×	2.8	1.1×	3.6	1.3×	3.6	1.3×	11.4	4.3×	18.3	7.0×
write	12.8	9.9	0.7×	10.0	0.8×	11.7	0.9×	13.8	1.1×	16.4	1.3×	39.0	3.0×
geomean			1.03×		1.14×		1.29×		1.42×		3.93×		6.61×

Table 4: Execution time in thousands of processor cycles of common system calls on TxOS and performance relative to Linux. **Base** is the basic overhead introduced by data structure and code modifications moving from Linux to TxOS, without the overhead of transactional lists. **Static** emulates compiling two versions of kernel functions, one for transactional code and one for non-transactional code, and includes transactional list overheads. These overheads are possible with compiler support. **NoTX** indicates the current speed of non-transactional system calls on TxOS. **Bgnd Tx** indicates the speed of non-transactional system calls when another process is running a transaction in the background. **In Tx** is the cost of a system call inside a transaction, excluding `sys_xbegin()` and `sys_xend()`, and **Tx** includes these system calls.

performance improvement for `write`. Overheads are incurred mostly by increased locking in TxOS and the extra indirection necessitated by data structure reorganization (e.g., separation of header and data objects). Transaction support in the kernel does not significantly slow down non-transactional activity.

TxOS replaces simple linked lists with a more complex transactional list (§3.3). The transactional list allows more concurrency, both by eliminating transactional conflicts and by introducing fine-grained locking on lists, at the expense of higher single-thread latency. The **Static** column adds the latencies due to transactional lists to the base overheads (roughly 10%, though more for `link`).

A key overhead in the TxOS prototype is dynamic checks whether a system call is executing inside a transaction or not. An alternative implementation might provide two versions of each function, one transactional and one non-transactional, and convert the dynamic checks into compile-time checks. This optimization would require installing a second system call table for transactions and more sophisticated compilation support. We capture the benefits in the **Static** column, which reduces the average non-transactional system call overhead to 14% over Linux.

The **NoTx** column presents measurements of the current TxOS prototype, with dynamic checks to determine if a thread is executing a transaction. The **Bgnd Tx** column are non-transactional system call overheads for TxOS while there is an active system transaction in a different thread. Non-transactional system calls need to perform extra work to detect conflicts with background transactions. The **In Tx** column shows the overhead of the system call in a system transaction. This overhead is high, but represents a rare use case. The **Tx** column includes the overheads of the `sys_xbegin()` and `sys_xend()` system calls.

4.2 Applications and micro-benchmarks

Table 5 shows the performance of TxOS on a range of applications and micro-benchmarks. Each measurement is the average of three runs. The slowdown relative to Linux is also listed. Postmark is a file system benchmark that simulates the behavior of an email, network news, and e-commerce client. We use version 1.51 with the same transaction boundaries as Amino [18]. The LFS small file benchmark operates on 10,000 1024 bytes files, and the large file benchmark reads and writes a 100MB file. The Reimplemented Andrew Benchmark (RAB) is a reimplementation of the Modified Andrew Benchmark, scaled for modern computers. Initially, RAB creates 500 files, each containing 1000 bytes of pseudo-random printable-ASCII content. Next, the benchmark measures execution time of four distinct phases: the `mkdir` phase creates 20,000 directories; the `cp` phase copies the 500 generated files into 500 of these directories, resulting in 250,000 copied files; the `du` phase calculates the disk usage of the files and directories with the `du` command; and the `grep`/`sum` phase searches the files for a short string that is not found and

Bench	**Linux ext2**	**TxOS ACI**		**Linux ext3**	**TxOS ACID**	
postmark	38.0	7.6	0.2×	180.9	154.6	0.9×
lfs small						
create	4.6	0.6	0.1×	10.1	1.4	0.1×
read	1.7	2.2	1.2×	1.7	2.1	1.3×
delete	0.2	0.4	2.0×	0.2	0.5	2.4×
lfs large						
write seq	1.4	0.3	0.2×	3.4	2.0	0.6×
read seq	1.3	1.4	1.1×	1.5	1.6	1.1×
write rnd	77.3	2.6	0.03×	84.3	4.2	0.05×
read rnd	75.8	71.8	0.9×	70.1	70.2	1.0×
RAB						
mkdir	8.7	2.3	0.3×	9.4	2.2	0.2×
cp	14.2	2.5	0.2×	13.8	2.6	0.2×
du	0.3	0.3	1.0×	0.4	0.3	0.8×
grep/sum	2.7	3.9	1.4×	4.2	3.8	0.9×
dpkg	.8	.9	1.1×	.8	.9	1.1×
make	3.2	3.3	1.0×	3.1	3.3	1.1×
install	1.9	2.7	1.4×	1.7	2.9	1.7×

Table 5: Execution time in seconds for several transactional benchmarks on TxOS and slowdown relative to Linux. ACI represents non-durable transactions, with a baseline of ext2, and ACID represents durable transactions with a baseline of ext3 with full data journaling.

checksums their contents. The sizes of the `mkdir` and `cp` phases are chosen to take roughly similar amounts of time on our test machines. In the transactional version, each phase is wrapped in a transaction. Make wraps a software compilation in a transaction. Dpkg and Install are software installation benchmarks that wrap the entire installation in a transaction, as discussed below (§ 4.3).

The overhead of system transactions for most workloads is quite reasonable (1–2×), and often system transactions speed up the workload (e.g., postmark, LFS small file create, RAB `mkdir` and `cp` phases). Benchmarks that repeatedly write files in a transaction, such as the LFS large file sequential write phase or the LFS small file create phase, are more efficient than Linux. Transaction commit groups the writes and presents them to the I/O scheduler all at once, improving disk arm scheduling and, on `ext2` and `ext3`, increasing locality in the block allocations. Write-intensive workloads outperform non-transactional writers by as much as 29.7×.

TxOS requires extra memory to buffer updates. We surveyed several applications' memory overheads, and focus here on the LFS small and large benchmarks as two representative samples. Because the utilization patterns vary across different portions of physical memory, we consider low memory, which is used for kernel data structures, separately from high memory, which can be allocated to applications or to the page cache (which buffers file contents in memory). High memory overheads are proportional to the amount data written. For LFS large, which writes a large stream of data, TxOS uses 13% more high memory than Linux, whereas LFS small, which writes many small files, introduced less than 1% space consumption overhead. Looking at the page cache in isolation, TxOS allocates 1.2–1.9× as many pages as unmodified Linux. The pressure on the kernel's reserved portion of physical memory, or low memory, is 5% higher for transactions across all benchmarks. This overhead comes primarily from the kernel slab allocator, which allocates 2.4× as much memory. The slab allocator is used for general allocation (via `kmalloc`) and for common kernel objects, like inodes. TxOS's memory use indicates that buffering transactional updates in memory is practical, especially considering the trend in newer systems toward larger DRAM and 64-bit addresses.

4.3 Software installation

By wrapping system commands in a transaction, we extend `make`, `make install`, and `dpkg`, the Debian package manager, to provide ACID properties to software installation. We test `make` with a build of the text editor nano, version 2.0.6. Nano consists of 82 source files totaling over 25,000 lines of code. Next, we test `make install` with an installation of the Subversion revision control system, version 1.4.4. Finally, we test `dpkg` by installing the package for OpenSSH version 4.6. The OpenSSH package was modified not to restart the daemon, as the script responsible sends a signal and waits for the running daemon to exit, but TxOS defers the signal until commit. A production system could rewrite the script to match the TxOS signal API.

As Table 5 shows, the overhead for adding transactions is quite reasonable (1.1–1.7×), especially considering the qualitative benefits. For instance, by checking the return code of dpkg, our transactional wrapper automatically rolled back a broken Ubuntu build of OpenSSH (4.6p1-5ubuntu0.3), and no concurrent tasks were able to access the invalid package files during the installation.

4.4 Transactional LDAP server

Many applications have fairly modest concurrency control requirements for their stable data storage, yet use heavyweight solutions, such as a database server. An example is Lightweight Directory Access Protocol (LDAP) servers, which are commonly used to authenticate users and maintain contact information for large organizations. System transactions provide a simple, lightweight storage solution for such applications.

To demonstrate that system transactions can provide lightweight concurrency control for server applications, we modified the `slapd` server in OpenLDAP 2.3.35's flat file storage module (called LDIF) to use system transactions. The OpenLDAP server supports a number of storage modules; the default is Berkeley DB (BDB). We used the `SLAMD` distributed load generation engine[1] to exercise the server, running in single-thread mode. Table 6 shows throughput for the unmodified Berkeley DB storage module, the LDIF storage module augmented with a simple cache, and LDIF using system transactions. The "Search Single" experiment exercises the server with single item read requests, whereas the "Search Subtree" column submits requests for all entries in a given directory subtree. The "Add" test measures throughput of adding entries, and "Del" measures the throughput of deletions.

The read performance (search single and search subtree) of each storage module is within 3%, as most reads are served from an in-memory cache. LDIF has 5–14× the throughput of BDB for requests that modify the LDAP database (add and delete). However, the LDIF module does not use file locking, synchronous writes or any other mechanism to ensure consistency. LDIF-TxOS provides ACID guarantees for updates. Compared to BDB, the read performance is similar, but workloads that update LDAP records using system transactions outperform BDB by 2–4×. LDIF-TxOS provides the same guarantees as the BDB storage module with respect to concurrency and recoverability after a crash.

4.5 Transactional ext3

In addition to measuring the overheads of durable transactions, we validate the correctness of our transactional

[1] http://www.slamd.com/

Back end	**Search Single**	**Search Subtree**	**Add**	**Del**
BDB	3229	2076	203	172
LDIF	3171	2107	1032 (5.1×)	2458 (14.3×)
LDIF-TxOS	3124	2042	413 (2.0×)	714 (4.2×)

Table 6: Throughput in queries per second of OpenLDAP's slapd server (higher is better) for a read-only and write-mostly workload. For the **Add** and **Del** workloads, the increase in throughput over BDB is listed in parentheses. The BDB storage module uses Berkeley DB, LDIF uses a flat file with no consistency for updates, and LDIF-TxOS augments the LDIF storage module use system transactions on a flat file. LDIF-TxOS provides the same crash consistency guarantees as BDB with more than double the write throughput.

`ext3` implementation by powering off the machine during a series of transactions. After the machine is powered back on, we mount the disk to replay any operations in the ext3 journal and run `fsck` on the disk to validate that it is in a consistent state. We then verify that all results from committed transactions are present on the disk, and that no partial results from uncommitted transactions are visible. To facilitate scripting, we perform these checks using Simics. Our system successfully passes over 1,000 trials, giving us a high degree of confidence that TxOS transactions correctly provide atomic, durable updates to stable storage.

4.6 Eliminating race attacks

System transactions provide a simple, deterministic method for eliminating races on system resources. To qualitatively validate this claim, we reproduce several race attacks from recent literature on Linux and validate that TxOS prevents the exploit.

We downloaded the symlink TOCTTOU attacker code used by Borisov et al. [1] to defeat Dean and Hu's probabilistic countermeasure [2]. This attack code creates memory pressure on the file system cache to force the victim to deschedule for disk I/O, thereby lengthening the amount of time spent between checking the path name and using it. This additional time allows the attacker to win nearly every time on Linux.

On TxOS, the victim successfully resists the attacker by reading a consistent view of the directory structure and opening the correct file. The attacker's attempt to interpose a symbolic link creates a conflict with the transactional `access` check, which TxOS resolves by putting

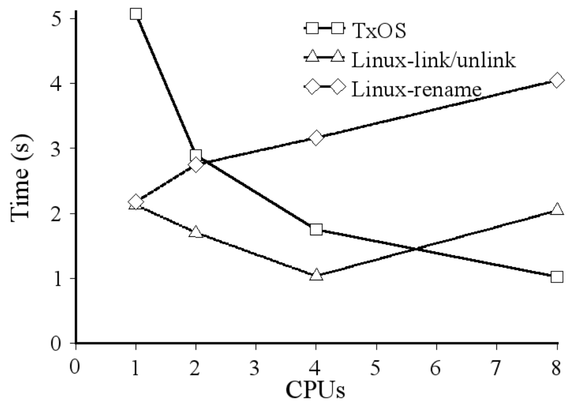


Figure 4: Time to perform 500,000 renames divided across a number of threads (lower is better). TxOS implements its renames as calls to `sys_xbegin()`, `link`, `unlink`, and `sys_xend()`, using 4 system calls for every Linux `rename` call. Despite higher single-threaded overhead, TxOS provides better scalability, outperforming Linux by 3.9× at 8 CPUs. At 8 CPUs, TxOS also outperforms a simple, non-atomic `link`/`unlink` combination on Linux by 1.9×.

the attacker to sleep until the victim commits. The performance of the safe victim code on TxOS is statistically indistinguishable from the vulnerable victim on Linux.

To demonstrate that TxOS improves robustness while preserving simplicity for signal handlers, we reproduced two of the attacks described by Zalewksi [19]. The first attack is representative of a vulnerability present in `sendmail` up to 8.11.3 and 8.12.0.Beta7, in which an attacker induces a double-free in a signal handler. The second attack, representative of a vulnerability in the `screen` utility, exploits lack of signal handler atomicity. Both attacks lead to root compromise; the first can be fixed by using the `sigaction` API rather than `signal`, while the second cannot. We modified the signal handlers in these attacks by wrapping handler code in a `sys_xbegin`, `sys_xend` pair, which provides signal handler atomicity without requiring the programmer to change the code to use `sigaction`. In our experiments, TxOS serializes handler code with respect to other system operations, preventing both attacks.

5 Toward simpler, scalable system calls

System calls like `rename` and `open` have been used as *ad hoc* solutions for the lack of general-purpose atomic actions. These system calls have strong semantics (a rename is atomic within a file system), resulting in complex implementations whose performance does not scale. As an example in Linux, `rename` has to serialize all cross-directory renames on a single file-system-wide mutex because finer-grained locking would risk deadlock. The problem is not that performance tuning `rename` is difficult, but it would substantially increase the implementation complexity of the entire file system, including unrelated system calls.

Transactions allow the programmer to combine simpler system calls to perform more complex operations, yielding better performance scalability and a simpler implementation. Figure 4 compares the unmodified Linux implementation of `rename` to calling `sys_xbegin()`, `link`, `unlink`, and `sys_xend()` in TxOS. In this micro-benchmark, we divide 500,000 cross-directory renames across a number of threads.

TxOS has worse single-thread performance because it makes four system calls for each Linux system call. TxOS quickly recovers, performing within 6% at 2 CPUs and out-performing `rename` by 3.9× at 8 CPUs. The difference in scalability is directly due to implementing transactions with fine-grained locking, whereas Linux must use coarse-grained locks to maintain the fast path for `rename` and keep its implementation complexity reasonable. While this experiment is not representative of real workloads, it shows that solving consistency problems with modestly complex system calls like `rename` will either harm performance scalability or introduce substantial implementation complexity. Because of Linux's coarse-grained locks, TxOS' atomic `link`/`unlink` pair outperforms the Linux non-atomic `link`/`unlink` pair by 1.9× at 8 CPUs.

A kernel that provides a smaller set of simple calls as well as a facility to compose them into more complex operations will be more maintainable than a kernel that supports a wide array of point solutions. Moreover, the complexity of managing fine-grained locking inside of transactions is encapsulated inside a small code base. In TxOS, the locking code inside a given system call is generally not complicated by transaction support. While adding transactions to the kernel may seem to increase the complexity of the system at first blush, TxOS demonstrates that the complexity can be tightly encapsulated and transactions can obviate the need for other complex or poor-performing code.

6 Design alternatives

The problems that system transactions solve are currently addressed to some degree by file locking and transactional file systems. However, neither approach is a complete solution, as explained in this section.

File locking File locking in Linux takes many forms: mandatory locking, advisory locking, and lock files. Lock files and advisory locking both provide concurrency control at the system level when all programs respect the locks; however, one cannot prevent buggy or malicious applications from ignoring these locks and accessing the data concurrently.

The reason advisory locking is popular is that mandatory locking can lead to denial of service on the system. The OS can revoke a mandatory lock, but likely at the cost of corrupting the underlying file.

File locking in Linux is associated with a file inode; this means that file locking cannot protect the file system namespace against TOCTTOU attacks. Finally, it is worth emphasizing that file locking only addresses concurrency control for system resources. File locking does not provide the ability to recover from a failed operation, which is useful for problems like software installation.

Transactional file systems The examples in the paper introduction focus on the file system, which is the source of the largest pain points. A transactional file system, such as TxF adopted by Windows Vista [14], can address some of the key issues. Unfortunately, implementing transactions within a particular file system (below the virtual filesystem (VFS) layer) undermines API simplicity and leads to usability problems.

When transactions are implemented in a specific file system, the key problem is file system state propagating into other volatile resources which cannot be rolled back by the file system if a transaction fails. State flowing from the file system to other resources leads to either conservative restrictions on transactions or speculative state leaking from an aborted transaction. For example, memory mappings are not under the control of the filesystem, and therefore most transactional file systems cannot allow a transactionally written file to also be memory mapped and executed. Linux software installers commonly unpack a set of files and then configure the software with post-installation scripts included in the software package. The conservative prohibition against executing transactionally written binaries, common in transactional file systems, prevents rolling back an install that can't be configured properly. As a second example, file handles are not visible to a file system and they do not roll back even if the backing store rolls back. For this reason, the Windows transactional file system requires that file handles used in a transaction be closed when a transaction ends and then subsequently re-opened when they are again needed. Finally, running processes can observe transactional file system data and propagate it (or results computed using it as inputs) through a pipe to new child process. Returning again to the post-installation script example, rolling back modifications to the file system does not kill the running post-installation script nor does it stop any daemons it may have launched or undo requests it sent to another running service. Compensating for these actions in userspace is difficult; the simplest programming model requires more robust kernel support.

Implementing transactions as a first-class kernel primitive simplifies the programming model for developers; treating transactions as a core kernel abstraction better encapsulates implementation details. For example, most transactional file systems expose locking details to users, often in the form of forcing them to reason about the risk of deadlock between completely unrelated programs. The TxOS prototype encapsulates all locking details within the kernel, guaranteeing the user that transactions cannot deadlock with each other. Transaction isolation in TxOS can still lead to starvation in some pathological cases, such as a long-running transaction denying access to a file or short transactions repeatedly aborting a longer transaction before it can commit. Rather than expose locking or other low-level system details to users as a way to enforce transaction scheduling policy, TxOS allows system administrators to set high-level policies through a file in `/proc`. For instance, the default policy is to favor the transaction with the highest scheduling priority, but this can be replaced with a policy that favors the oldest transaction. Selecting a high-level contention management policy is much less error-prone than managing kernel locks in userspace.

A final argument for generalized transaction support in the kernel is that transactions are a useful feature for all Linux file systems. The TxOS design implements file system transactions primarily in the VFS layer, leaving minimal adoption work for a specific file system. The main onus on a specific file system is ensuring atomic commit of data to disk, which is already provided by common techniques such as journaling. As a proof point, we implemented a transactional `ext3` file system in one developer month on TxOS. Transaction support can be generalized in the VFS layer while imposing minimal development effort on individual file systems.

7 Limitations and future work

TxOS does not yet provide transactional semantics for several classes of OS resources. Currently, TxOS either logs a warning or aborts a transaction that attempts to access an unsupported resource: the programmer specifies the behavior via a flag to `sys_xbegin()`. Among the unsupported resources are the network, certain classes of inter-process communication, and user interfaces.

Ongoing work on TxOS is focused in three directions. First, we plan to study additional applications that can benefit from transactions. Second, we plan to add support for additional kernel abstractions and resources. Third, we would like to find additional optimizations to improve the performance of the TxOS prototype.

8 Conclusion

TxOS demonstrates that transactions are a practical abstraction a widely-deployed, commodity OS. The code changes required are substantial, but so are the benefits. The source code for TxOS is available at `http://txos.code.csres.utexas.edu`.

References

[1] N. Borisov, R. Johnson, N. Sastry, and D. Wagner. Fixing races for fun and profit: How to abuse atime. In *USENIX Security*, 2005.

[2] D. Dean and A. J. Hu. Fixing races for fun and profit: how to use access(2). In *USENIX Security*, pages 14–26, 2004.

[3] M. Herlihy and E. Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *PPoPP*, 2008.

[4] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI*, 2007.

[5] J. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[6] P. E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy Update Techniques in Operating System Kernels*. PhD thesis, 2004.

[7] W. S. McPhee. Operating system integrity in OS-/VS2. *IBM Systems Journal*, 13(3):230–252, 1974.

[8] G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *ICCC*, pages 213–228, 2002.

[9] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. In *SOSP*, 2005.

[10] NIST. National Vulnerability Database. `http://nvd.nist.gov/`, 2010.

[11] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating system transactions. In *SOSP*, 2009.

[12] D. E. Porter and E. Witchel. Operating systems should provide transactions. In *HotOS*, 2009.

[13] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel. MetaTM/TxLinux: Transactional memory for an operating system. In *ISCA*, 2007.

[14] M. Russinovich and D. Solomon. *Windows Internals*. Microsoft Press, 2009.

[15] F. Schmuck and J. Wylie. Experience with transactions in QuickSilver. In *SOSP*. ACM, 1991.

[16] D. Tsafrir, T. Hertz, D. Wagner, and D. D. Silva. Portably preventing file race attacks with user-mode path resolution. Technical report, IBM Research Report, 2008.

[17] M. J. Weinstein, J. Thomas W. Page, B. K. Livezey, and G. J. Popek. Transactions and synchronization in a distributed operating system. In *SOSP*, 1985.

[18] C. P. Wright, R. Spillane, G. Sivathanu, and E. Zadok. Extending ACID semantics to the file system. *Trans. Storage*, 3(2):4, 2007.

[19] M. Zalewski. Delivering signals for fun and profit. 2001.

CPU bandwidth control for CFS

Paul Turner
Google
pjt@google.com

Bharata B Rao
IBM India Software Labs, Bangalore
bharata@linux.vnet.ibm.com

Nikhil Rao
Google
ncrao@google.com

Abstract

Over the past few years there has been an increasing focus on the development of features for resource management within the Linux kernel. The addition of the fair group scheduler has enabled the provisioning of proportional CPU time through the specification of group weights. Since the scheduler is inherently work-conserving in nature, a task or a group can consume excess CPU share in an otherwise idle system. There are many scenarios where this extra CPU share can cause unacceptable utilization or latency. CPU bandwidth provisioning or limiting approaches this problem by providing an explicit upper bound on usage in addition to the lower bound already provided by shares.

There are many enterprise scenarios where this functionality is useful. In particular are the cases of pay-per-use environments, and latency provisioning within non-homogeneous environments.

This paper details the requirements behind this feature, the challenges involved in incorporating into CFS (Completely Fair Scheduler), and the future development road map for this feature.

1 CPU as a manageable resource

Before considering the aspect of bandwidth provisioning let us first review some of the basic existing concepts currently arbitrating entity management within the scheduler.

There are two major scheduling classes within the Linux CPU scheduler, `SCHED_RT` and `SCHED_NORMAL`. When runnable, entities from the former, the *real-time* scheduling class, will always be elected to run over those from the *normal* scheduling class.

Prior to *v2.6.24*, the scheduler had no notion of any entity larger than that of single task[1]. The available management APIs reflected this and the primary control of bandwidth available was `nice(2)`.

In *v2.6.24*, the *completely fair scheduler* (CFS) was merged, replacing the existing `SCHED_NORMAL` scheduling class. This new design delivered *weight* based scheduling of CPU bandwidth, enabling arbitrary partitioning. This allowed support for *group scheduling* to be added, managed using *cgroups* through the *CPU controller* sub-system.

This support allows for the flexible creation of scheduling groups, allowing the fraction of CPU resources received by a group of tasks to be arbitrated as a whole. The addition of this support has been a major step in scheduler development, enabling Linux to align more closely with enterprise requirements for managing this resouce.

The hierarchies supported by this model are flexible, and groups may be nested within groups. Each group entity's bandwidth is provisioned using a corresponding `shares` attribute which defines its weight. Similarly, the `nice(2)` API was subsumed to control the weight of an individual task entity.

Figure 1 shows the hierarchical groups that might be created in a typical university server to differentiate CPU bandwidth between users such as professors, students, and different departments.

One way to think about *shares* is that it provides lower-bound provisioning. When CPU bandwidth is scheduled at capacity, all runnable entities will receive bandwidth in accordance with the ratio of their share weight. It's key to observe here that not all entities may be runnable

[1]Recall that under Linux any kernel-backed thread is considered individual task entity, there is no typical notion of a *process* in scheduling context.

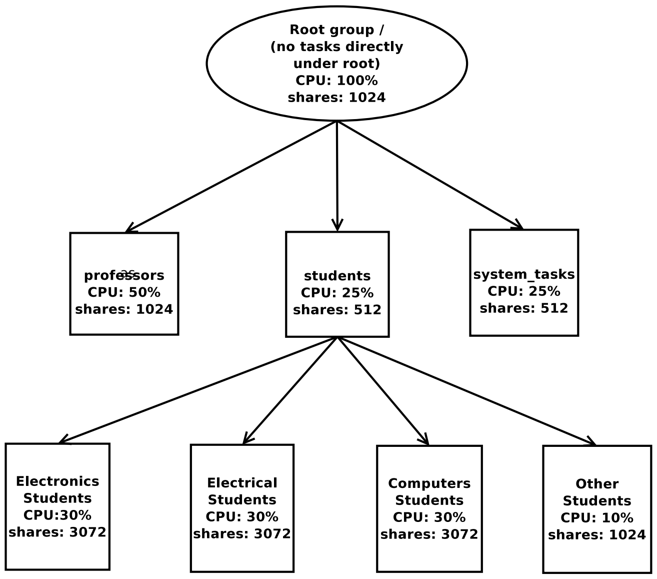

Root group / (no tasks directly under root) CPU: 100% shares: 1024
professors CPU: 50% shares: 1024
students CPU: 25% shares: 512
system_tasks CPU: 25% shares: 512
Electronics Students CPU:30% shares: 3072
Electrical Students CPU: 30% shares: 3072
Computers Students CPU: 30% shares: 3072
Other Students CPU: 10% shares: 1024

Figure 1: An example hierarchy applicable to a university server.

in this situation; this means that CPU bandwidth is comparatively available in abundance and the entities that are running will be able to consume bandwidth at a higher rate than their weight would permit were more entities runnable.

It should be noted that the concept of proportional shares is different from a guarantee. Assigning share to a group doesn't guarantee that it will get a particular amount of CPU. It only means that the available CPU bandwidth will be divided as per the shares. Hence depending on the number of groups present, the actual amount of CPU time obtained by groups can vary. [2]

For example: If there were 3 groups with 1024 shares each, then each would receive $\frac{1024}{1024+1024+1024} = 33.3\%$ of the CPU when all were runnable[3]. If a 4th group on the same level were to become active with a share of 2048, then the other groups would now receive only 20% of available bandwidth. The CPU bandwidth available to a group (by weight) is *always* relative.

[2]Also recall: These ratios are only relative to the time avai lable to SCHED_NORMAL. Time spent in SCHED_RT execution is independent of this mo del.

[3]Since group entities are containers, for a group entity to be runnable it must have an active child entity, the leaves of this tree must thusly all be task entities.

2 Motivation for bandwidth control

As discussed above, the scheduler is work conserving by nature; when idle cycles are available in the system, it is because there were no runnable entities available (on that cpu) to consume them. While for many use-cases, efficient use of idle CPU cycles like this might be considered optimal, there are two key side effects that must be considered:

1. The actual amount of CPU time available to a group is highly variable as it is dependent on the presence and execution patterns of other groups, a machine can the not be *predictably partitioned* without intimately understanding the behaviors of all co-scheduled applications.

2. The maximum amount of CPU time available to a group is not predictable. While this is closely related to the first point, the distinction is worth noting as this directly affects capacity planning.

While not of concern to most desktop users, these are key requirements in certain enterprise scenarios. Bandwidth control aims to address this by allowing upper limits on group bandwidths to be set. This allows both capacity and the maximal effect on other groups to be predicted.

This feature is already available for the *real-time* group scheduler (SCHED_RT). The first attempt to add this functionality to CFS (SCHED_NORMAL) was posted in June 2009 by the RFC post [2] to the Linux Kernel Mailing List (LKML). In the subsequent sections of this paper, we discuss this initial approach and how it has evolved into CFS Bandwidth Control below.

3 Example use cases

Bandwidth provisioning is commonly found useful in the following scenarios:

- Pay-per-use:

  In enterprise systems that cater to multiple clients/customers, a customer pays for, and is provisioned with a specific share of CPU resources. In such systems, customers would object should they receive less and its in the provider's interest that they not provided more. In this case CPU bandwidth provisioning could be used directly to constrain the customers usage and provide soft bandwidth to interval guarantees. Such pay-per-use scenarios are frequently seen in *cloud* systems where service is priced by the required CPU capacity.

- Virtual Machines

  For (integrated) Linux based hypervisers such as KVM, bandwidth limits at the scheduler level may be useful to control the CPU entitlements of hosted VMs.

- Latency provisioning

  The explicit provisioning of containers within the machine allows for expectations to be set with respect to latency and worst-case access to CPU time. This becomes particularly important with non-homogenous collections of latency sensitive tasks where it is difficult to restrict co-scheduling.

- Guarantees

  In addition to maximum CPU bandwidth, in many situations applications may need CPU bandwidth guarantees. Currently this is not directly supported by the scheduler. In such cases, hard limits settings of different groups may be derived to reach a minimum (soft) guarantee for every group. An example of how to obtain guarantees for groups by using hard limit settings is provided in the OpenVZ wiki [1].

4 Interfaces

As discussed above the *cgroups* interface has been leveraged for managing the CPU controller subsystem. Our work extends these interfaces.

In case of `SCHED_RT`, bandwidth is specified using two control parameters: the enforcement interval (`cpu.rt_period_us`) and allowable consumption (`cpu.rt_runtime_us`) within that interval. Accounting is performed on a *per-CPU* basis. For example if there were 8 CPUs in the system [4], the group would be allowed to consume 8 times the `cpu.rt_runtime_us` within an interval of `cpu.rt_period_us`. This is enabled by allowing unconsumed time to be transferred from CPUs present in the `root_domain` span that have unconsumed bandwidth available.

In our initial approach [3], the bandwidth specification exposed for `SCHED_NORMAL` class was based on this model. However for the reasons described in the subsequent sections, we have now opted for global specifcation of both enforcement interval (`cpu.cfs_period_us`) and allowable bandwidth (`cpu.cfs_quota_us`). By specifying this, the group as a whole will be limited to `cpu.cfs_quota_us` units of CPU time within the period of `cpu.cfs_period_us`.

Of note is that these limits are hierarchical, unlike `SCHED_RT` we do not currently perform feasibility evaluaion regarding the defined limits. If a child has a more permissive bandwidth allowance than its parent, it will be indirectly throttled when the parent's quota is exhausted.

Additionally, there is the global control: `/proc/sys/kernel/sched_cfs_bandwidth_slice_us`

This *sysctl* interface manages how many units are transferred from the global pool each time a local pool requires additional quota. The current default is 10ms. The details of this transfer process are discussed in later sections.

5 Existing Approaches

5.1 CFS hard limits

This was the first approach [3] at implementing bandwidth controls for CFS and was modelled on the existing bandwidth control scheme in use by the *real-time* scheduling class. The mechanism employed here is quite direct. Each group entity (specifically `cfs_rq` here) is provisioned locally with `cfs_runtime_us` units of time. CPUs are then allowed to borrow from one another within a given `root_domain`. This means that the *externally* visible bandwidth of the group is effectively the weight of the `root_domain` CPU mask multiplied by `cfs_runtime_us` (per `cfs_period_us`). When a CFS group consumes all its runtime and when there is nothing left to borrow from the other CPUs, the group is then throttled. At the end of the enforcement interval, the bandwidth gets replenished and the throttled group becomes eligible to run once again.

[4]Assuming the `root_domain` has not been partitioned via *cpusets*

A typical time line for a process that runs as part of a bandwidth controlled group under *CFS Hard Limits* appears as shown in Figure 2.

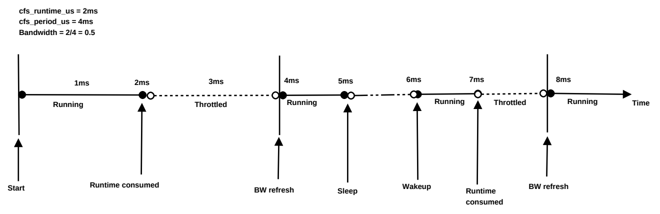


Figure 2: Progress of a task in bandwidth controlled group

Figure 3 shows the same situation with borrowing of quota from other cpus.

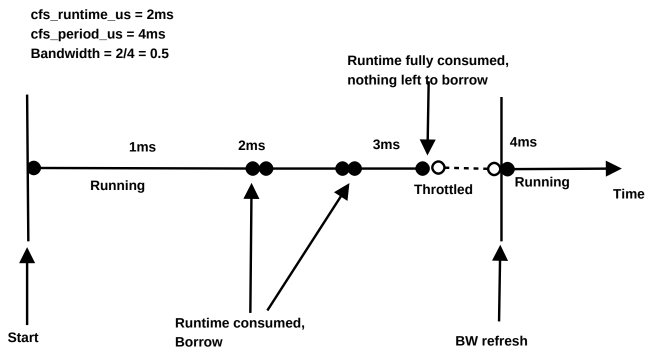


Figure 3: Progress of a task in bandwidth controlled group with runtime borrowing

The strategy for time-redistribution is to visit all neighbor CPUs and transfer $\frac{1}{n}$ of their remaining run-time, where n is the weight of the root_domain span. There is an implicit assumption made within this scheme that we will be able to converge quickly above to `cfs_period_us` while borrowing. This is indeed true for the *real-time* class as, by default, `SCHED_RT` is provisioned with 95% of total system time, allowing an individual CPU to reach to `rt_period_us` with only a single partial iteration of borrowing in the default configuration. This can also be expected to hold more generally as the nature of entities requiring this scheduling class is to be well-provisioned.

In the more general case, where the reservation may represent only a small-to-medium fraction of system resources, this convergence breaks down. Each iteration is able to maximally consume $\frac{n-1}{n}$ additional time, at the expense of taking every `rq->lock`. Moreover, as most cpus will not be allowed to reach the upper bound of the period we will have an extremely long tail of redistribution.

5.2 Hybrid global pool:

The primary scalability issue with the local pool approach is that there there is a many-to-many relationship in the computation and storage of remaining quota. This is acceptable provided either the existence of a strong convergence condition or 'small' SMP systems.

Tracking quota globally is also not a solution that scales with machine size due to the large contention the global store then experiences. One of the advantages of the local quota model above is that, when within quota, consumption is very efficient since it can potentially be accounted locklessly and involves no 'remote' queries.

Our design for the distribution of quota is a hybrid model which attempts to combine both local and global quota tracking. To each `task_group` a new `cfs_bandwidth` structure has been added. This tracks (globally) the allocated and consumed quota within a period. However, consumption does not occur against this pool directly; as in the local pool approach above there is a local, per `cfs_rq`, store of granted and consumed quota. This quota is acquired from the global pool in a (user configurable) batch size. When there is no quota available to re-provision a running `cfs_rq`, it is locally throttled until the next quota refresh. Bandwidth refresh is a periodic operation that occurs once per quota period within which all throttled run-queues are unthrottled and the global bandwidth pool is replenished.

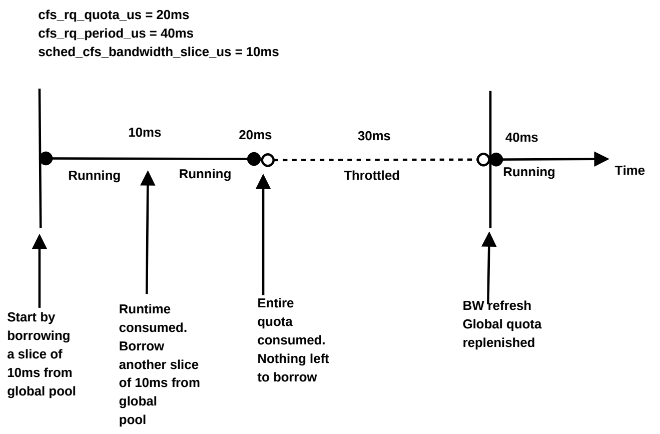

cfs_rq_quota_us = 20ms
cfs_rq_period_us = 40ms
sched_cfs_bandwidth_slice_us = 10ms
10ms
20ms
30ms
40ms
Running
Running
Throttled
Running
Time
Start by borrowing a slice of 10ms from global pool
Runtime consumed. Borrow another slice of 10ms from global pool
Entire quota consumed. Nothing left to borrow
BW refresh Global quota replenished

Figure 4: progress of a task in bandwidth controlled group with global quota

6 Design

CFS bandwidth control implements the above discussed hybrid approach to bandwidth accounting. The global quota is distributed in 'slices' to local per-CPU caches, where it is then consumed. The local accounting for these quota slices closely resembles the `SCHED_RT` case, however, the many-to-many CPU interactions on refresh and expiration are avoided. The batching of quota distribution also allows for linear convergence to quota within a provisioned period.

A summary of specific key changes is provided below.

6.1 Data-structures

`struct cfs_bandwidth`:

This is the top-level group representation of bandwidth, encapsulated within the corresponding `task_group` structure. The remaining `quota` within each period, as well as the total `runtime` assigned per period, and quota `period` length are managed here.

`struct cfs_rq`:

This is the per-group runqueue of all runnable entities present in `SCHED_NORMAL` group. Each group has one such representative runqueue on every CPU. The entities within a group are arranged in a time-ordered RB tree. Time ordering is done by *vruntime* or virutal runtime, which is a rough indication of the amount of CPU time obtained by an entity. We have annotated this structure with the new variables `quota_assigned` and `quota_used`, which track the total bandwidth allocated from the global pool to this runqueue and the amount consumed respectively.

6.2 Bandwidth distribution and constraint

`update_curr()`:

This function is periodically called from scheduler ticks as well as during other scheduler events like enqueue and dequeue to update the amount of time the currently running entity has received. Time since the last invocation is charged against the entity. The weight-normalized vruntime which forms the basis for fair-share scheduling is also updated here.

The accounting here is extended to call the new function `account_cfs_rq_quota`. This ensures that quota accounting will occur at the same instance at which execution time is charged.

`account_cfs_rq_quota()`:

This function forms the basis for quota distribution and tracking. The rough control flow is as shown in Figure 5:

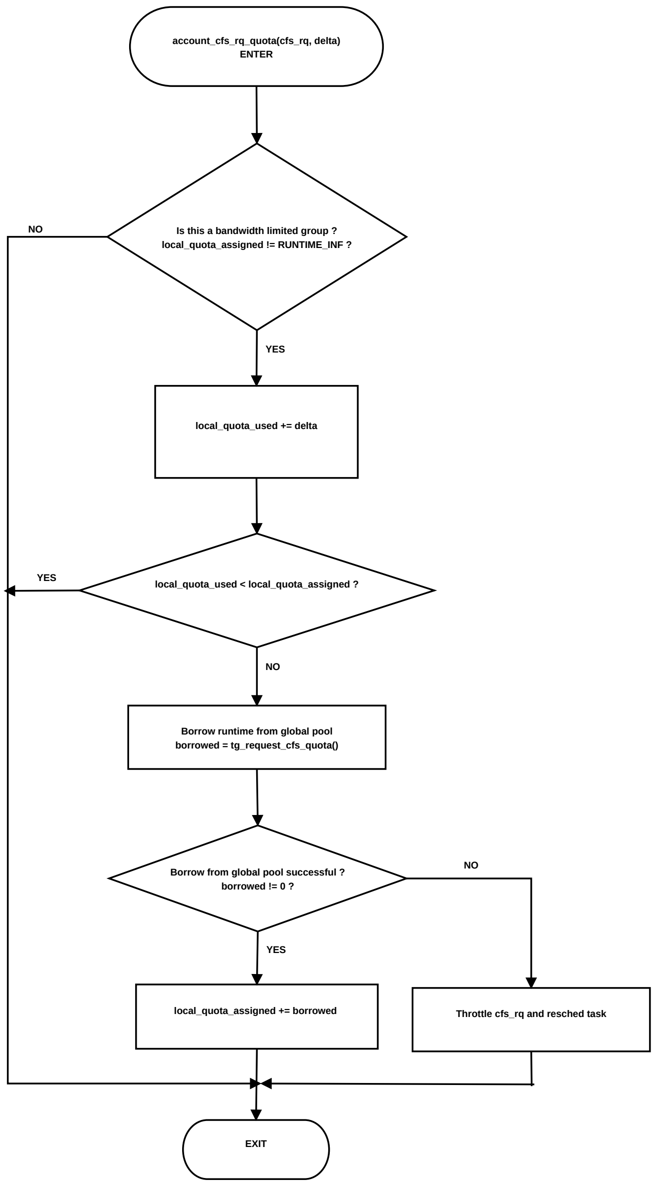

account_cfs_rq_quota(cfs_rq, delta)
ENTER
NO
Is this a bandwidth limited group ?
local_quota_assigned != RUNTIME_INF ?
YES
local_quota_used += delta
YES
local_quota_used < local_quota_assigned ?
NO
Borrow runtime from global pool
borrowed = tg_request_cfs_quota()
Borrow from global pool successful ?
borrowed != 0 ?
NO
YES
local_quota_assigned += borrowed
Throttle cfs_rq and resched task
EXIT

Figure 5: Control flow diagram for account_cfs_rq_quota()

Entity throttling

A throttled `cfs_rq` is one that has run out of local bandwidth (specifically, `local_quota_used` ≥

local_quota_assigned). By extension this means there was no global bandwidth availble to 'top-up' or refresh the pool. At this point the entity is no longer schedulable in the current quota period as it has reached its bandwidth limit. [5]

Such a cfs_rq is considered *throttled.* This state is tracked explicitly using the equivalently named attribute. Issuing a throttle operation (throttle_cfs_rq()) will dequeue the sched_entity from its parent cfs_rq and set the above throttled flag. Note that for an entity to cross this threshold it must be running, and thus, be the current entity. This means that the dequeue operation on the throttled entity will just update the accounting since the currently running entity is already dequeued form RB tree as per CFS design.

We must however, ensure that it is not able to reenter the tree due to either a *put* operaton, or thread wake-up. The first case is handled naturally as the decision to return an entity to the RB tree is based off the entity->on_rq flag, which has been unset as a consequence of accounting during dequeue. The task wakeup case is handled directly by ceasing to enqueue past a throttled entity within enqueue_task_fair().

When the lone throttled entity of a parent is dequeued, the parent entity will suddenly become non-runnable, since it no longer has any runnable child entities. Even though the parent is not throttled, it need not remain on the RB tree. The natural solution to this is to continue dequeuing past the throttled entity until we reach an entity with load weight remaining. This is analogus to the ancestor dequeue that may occur when a nested task sleeps.

The unthrottle case is symmetric, the entity is reenqueued and the throttled flag is cleared. Parenting entities must then potentially be enqueued up the tree hierarchy until we reach either the root, or an ancestor undergoing its own throttling operations.

Quota Refresh

task_group quota is refreshed periodically using *hrtimers* by programming the hrtimer to expire every cfs_period_us seconds. A struct hrtimer period_timer is embedded in cfs_bandwidth structure for this purpose. If there is a quota interval within which no bandwidth is consumed then the timer will not be re-programmed on expiration.

The refresh operation first refreshes the global quota pool, stored in the cfs_bandwidth structure. We then iterate over the per-CPU cfs_rq structures to determine whether any of them have been throttled. In the case of a throttled cfs_rq, we attempt to assign more bandwidth to it and – if successful – unthrottle it. Unlike SCHED_RT case, we could do this check for throttled cfs_rqs speculatively to reduce the contention on rq->lock. Future development here could involve refresh timer consolidation to further reduce overhead in the many *cgroup* case.

Locking Considerations

Since the locally assigned bandwidth is maintained on the cfs_rq, we are able to nest tracking and modification under the parent rq->lock. Since this lock is already held for existing CFS accounting, this allows the local tracking of quota to be performed with no additional locking.

Explicit locking is required to synchronize modification to the assigned or remaining bandwidth available in the global pool. Such an operation occurs when a local pool exceeds its (locally) assigned quota or through configuration change.

6.3 Challenges

- **Slack time handling**

  One caveat of our chosen approach is that the time locally assigned may have been allocated from the global pool in a previous quota interval. This represents potential local over-commit when bandwidth is expressed versus reservation. The maximum outstanding over-subscription within a given set of consecutive intervals is constant at $num\_cpus \cdot batch\_slice$. It is of note that this holds true for any number of consecutive observed interval since the input rate of the system is bounded, the over-commits occurs from remaining *slack* time that may be left over from the interval immediately prior to the first measured period. One potential approach to mitigate this for environments where

[5]Since bandwidth is defined on at the group level it should be noted that an individual task entity will never be throttled, only its parent.

harder limits are required is to use generation counters during the allocation and distribution of quota.

- **Fairness issues**

  The waterfall distribution of quota from global to local pools is also potentially accompanied by risks involving the fairness of consumption. Consider for example the case of a multi-CPU machine. Since the allocation of quota is currently in batch sized amounts it is possible for a multi-threaded application to experience reduced parallelism. It is also possible for quota to be 'stranded' as load-balancing leaves local quota unavailable for consumption due to no runnable entities. The latter can also potentially be addressed by a generational model as it would then be possible to return quota to the global pool on a voluntary sleep. However as systems scale the former potentially becomes a real problem. A short term mitigation strategy could be to reduce the batch-sizing used for the propagation of quota from global to local pools. Longer term strategies for resolving this issue might include a graduated scheme for batch slice sizing and subdividing the global pool at the `sched_domain` level to allow for finer granularity of control on distribution.

- **Load-balancer interactions**

  CFS bandwidth control currently supports only a very primitive model for load balancer interactions. Throttled run-queues are excised from load-balancing decisions; it turns out that it is hard to improve this model without undesirable emergent behaviors. At first inspection it may appear that migrating threads from a locally throttled run-queue to an unthrottled one would be a sensible decision. This quickly breaks down when the case of insufficient quota is considered. Here the last run-queue to have quota available will resemble that last seat in the game 'musical chairs', inadvertantly creating a 'herd' of executing threads. Likewise, it does not make sense to migrate a thread to a run-queue that has already been throttled as it being runnable indicates there is local quota still available.

  While improving the actual mechanics of load balancing in these conditions may be a large technical challenge, there may be an easier case to make for improvement in the distribution of share weight. Currently, when a run-queue is throttled its participation in group share distribution is also halted. This may result in undesirable rq weight fluctuations. One avenue that has been considered in this area is to continue allowing throttled entities to participate in weight calculations for re-distribution. This will allow entities to re-wake with their correct weight and prevent large swings in distribution. For this to work effectively however some stronger guarantees that quota will expire relatively concurrently are desired to avoid skews.

7 Results

This section describes the experiments we have undertaken to validate CFS bandwidth control. We describe our test setup and the benchmarks run. We then present our results and discuss some limitations in the current approach. Finally we explore the effects of changing parameters like `sched_slice` and enforcement periods in the system.

We performed all our tests on a 16-core AMD machine with no restrictions to affinity. We ran our tests with our patches based on top of the *v2.6.34* kernel, which as of writing this paper was the most recent stable kernel release. Outside the default x86 Kconfig, we enabled `CONFIG_FAIR_GROUP_SCHED` and `CONFIG_CFS_BANDWIDTH_CONTROL`. We configured our test machine with the following *cgroup* configuration for all experiments.

Container	Shares	Notes
system	1024	Contains system tasks such as sshd, measurement tasks, etc.
protag	65536	Benchmark runs in this container. The protag container is large enough to mitigate system interference.

We monitored system utilization in a monitoring thread that was part of the system container. This thread woke up once a second and read `/proc/stat` for busy usage and idle time for all CPUs in the system. We used busy time measured as a fraction of the total system time as a metric to measure the effectiveness of CPU bandwidth control.

Two simple benchmarks were used to validate the bandwidth control mechanism - while-1 soakers and sysbench. Both these benchmarks are multi-threaded applications and can completely saturate the machine in the absence of bandwidth control. We would also like to note that system daemon/thread interference was minimal and can be ignored for the purposes of this discussion.

When the runtime allocated to a cgroup is the same as its period, it can be expected to receive one CPU's worth of wall time. When the runtime is twice the period it gets about two CPUs worth, and so on. The same bandwidth limits were explicitly attained using the `cpuset` subsystem and CPU affinities as a control group. It should be noted that due to the lack of affinity and that time was observed on all runnable CPUs in the first case there is nothing special about the integral CPU case for *CFS Bandwidth Control*, it merely enables the use of cpusets as an `OPT` control parameter.

Each benchmark was run with three different bandwidth enforcement periods - 100ms, 250ms, and 500ms. In each set of runs, we allocated an integer core worth of runtime to the protag cgroup ranging from 1 core worth upto the full 16 cores worth of runtime. We compare these results with the baseline measurement for the respective benchmark.

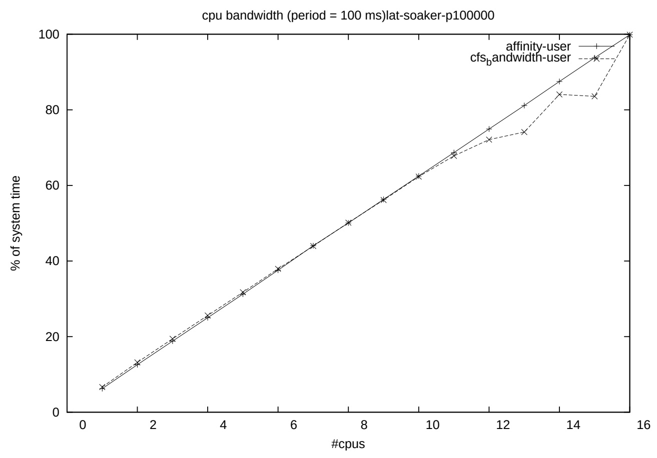


Figure 6: while-1 soakers, CFS Bandwidth Control, 100ms

Figures 6, 7 and 8 show the comparision of CPU times obtained by while-1 soakers when run with affinities and when run with bandwidth control. We see that the average deviation from the baseline benchmark is very small in each of these cases. We also notice that the average deviation from baseline decreases as the enforcement period increases.

Figures 9, 10 and 11 show the results for sysbench. The benchmark was to run the CPU sysbench test with 16

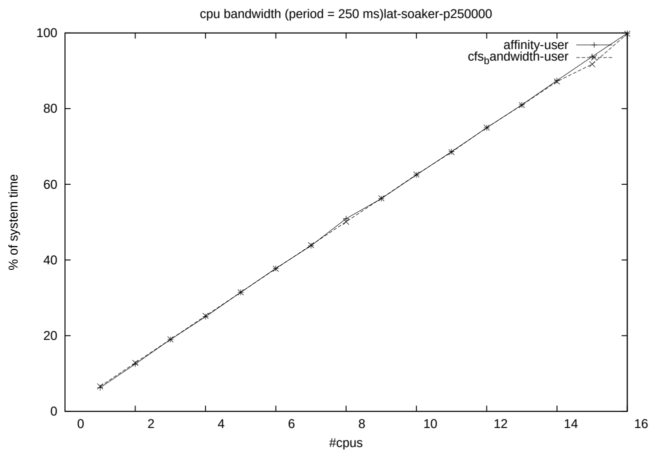


Figure 7: while-1 soakers CFS Bandwidth Control, 250ms

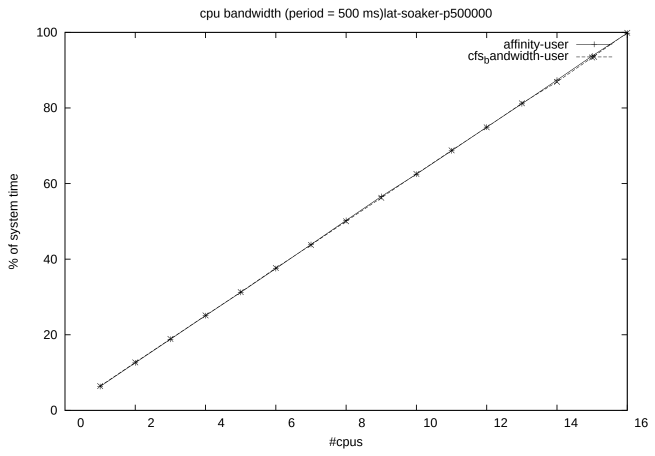


Figure 8: while-1 soakers, CFS Bandwidth Control, 500ms

worker threads. Each thread computed all prime numbers lesser than 100000. Again, we see that the deviation from the baseline benchmark is very small in most cases and improves as the enforcement period increases.

7.1 Overhead

CFS bandwidth control adds a minimal overhead to the enqueue/dequeue fast paths. We used tbench to measure overhead on these paths as it exercises these paths very frequently (450K times a second). We used on a vanilla 2.6.34 kernel with affinity masks as the baseline. The tabular data below shows the overhead with enforcement period at 100ms.

As described earlier, quota is distributed in `batch_slice` amounts of runtime. This is set to 10ms by

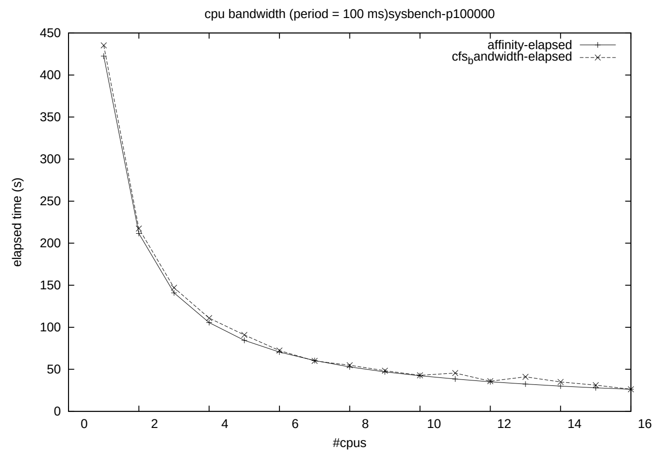


Figure 9: sysbench bandwidth control, 100ms

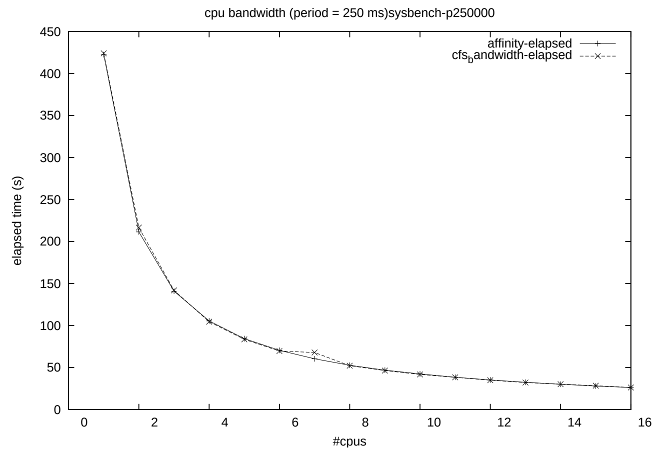


Figure 10: sysbench bandwidth control, 250ms

default in our current system, but we expose this as a tunable that can be set at runtime via a procfs tunable. While decreasing this value increases the frequency at which CPUs request for quota from the global pool, we did not notice any measurable impact on performance.

8 Conclusions and Futures

CFS Bandwidth Control is a light-weight and flexibile mechanism for bandwidth control and specification. We

cputime	baseline	bandwidth control
1	219.022	213.415
8	1668.12	1653.7
16	2451.82	2421.36

Table 1: Overhead of CFS bandwidth control, tbench 10 procs

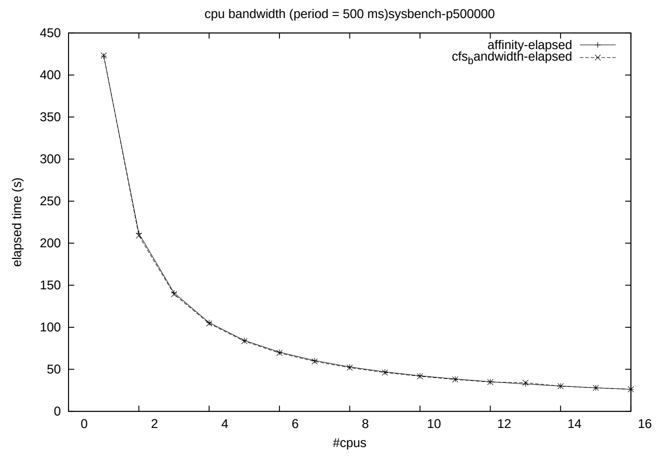


Figure 11: sysbench bandwidth control, 500ms

are currently in the process of attempting to collect test-data on a wider variety of both proprietary and open-source workloads and hope to publish this data as it becomes available.

Our patchset is in a stable state and we encourage any customers interested in this requirement to evaluate whether it meets their needs and provide feedback. The current posting is version 2, available at [4].

We are currently pursuing peer review with the hopes of merging this feature into the mainline scheduler tree. Looking forwards we are attempting to deliver improvements such as generational quota to mitigate potential slack time issues and improve fairness. For simplicity's sake given the review process however, formal consideration of this should be post-poned until the original approach reaches maturation within the community.

9 Acknowledgements

We would like to thank Ken Chen, Dhaval Giani, Balbir Singh and Srivatsa Vaddagiri for discussion related to the development of this work. Much credit is also due to the original bandwidth mechanisms present in `SCHED_RT` as they have both inspired and formed the basis for much of this work. We would also like to thank Google and IBM for funding this development.

10 Legal Statements

References

[1] Guarantees in OpenVZ. `http://wiki.openvz.org/Containers/Guarantees_for_resources`.

[2] Bharata B Rao. CFS hard limits - v0, June 2009. `http://lkml.org/lkml/2009/6/4/24`.

[3] Bharata B Rao. CFS hard limits - v5, January 2010. `http://lkml.org/lkml/2010/1/5/44`.

[4] Paul Turner. CFS bandwidth control - v2, April 2010. `http://lkml.org/lkml/2010/4/28/88`.

Page/slab cache control in a virtualized environment

Balbir Singh
Linux Technology Center, IBM,
balbir@linux.vnet.ibm.com

Abstract

The Linux page/slab cache subsystems are one of the most useful subsystems in the Linux kernel. Any attempts to limit its usage have been discouraged and frowned upon in the past. However, virtualization is changing the role of the kernel running on the system, specifically when the kernel is running as a guest. Assumptions about using all available memory as cache and optimizations will need need to be re-looked in an environment where resources are not fully owned by one guest OS.

In this paper, we discuss some of the pain points of page cache in a virtualized environment; like double caching of data in both the host and guest and its impact on memory utilization. We look at the current page cache behavior of Linux running as a guest and when multiple instances of guest operating systems are running. We look at current practices and propose new solutions to the solving the double caching problem in the kernel.

1 Introduction

The cache systems are typically designed to grow, they tend to use as much memory is required for caching key data that can be reused later. They also provide a reclaim system that can quickly reclaim back memory used for caching. An often asked question on the Linux Kernel Mailing List (LKML) [5] relates to why the free memory on the system is very low, even though the system is mostly idle or even when the system has few applications that do not take up a lot of memory. Typically we distinguish between free memory and freeable memory. The cache (unless dirty) falls in the category of freeable memory. We use memory to optimize the cost of otherwise reading from a slow device. The ability to reclaim from the cache when needed is a good design trade-off.

The scenario is quite different in a virtualized environment. The entire guest kernel memory is mapped into the hypervisor address space. The memory cached in the kernel, shows up as mapped memory in the hypervisor. Beyond the change of the way memory is visible, caching policies in both the guest and host can lead to double caching. Double caching is not very good in a virtualized environment, where resources are scarce and heavily shared. In the sections to follow, we look at page cache in a virtualized environment, some basic data about page cache in a virtualized environment, our approaches to solving the problem, future work and we finally conclude with recommendations.

NOTE: We've used the terms host and hypervisor interchangeably in this paper.

2 I/O in a Virtualized Environment

Our focus in this paper is on the KVM hypervisor [3] and the Linux Operating System running as the guest operating system. The KVM hypervisor configuration can be very complex. Lets look at the various ways of carrying out I/O.

1. **Direct Assignment**: In this mode, the IOMMU [2] creates one or more unique address spaces which can be used for DMA operations. With IOMMU's and direct assignment, a device can be assigned to a virtual machine directly. This speeds up I/O immensely. The drawback of such a scheme is scalability. There are standards that allow to solve the scalability problem by virtualizing the workqueues, interrupts, registers on a per VM basis while using the same device. The guest drivers need to support these devices to make full use of the capabilities.

2. **Paravirtualized I/O**: The hypervisor uses the virt I/O [6] subsystem to paravirtualize the I/O and makes as efficient as possible. The data exchanged

between the guest and the host is done via a zero-copy mechanism, with efficient notification mechanism for availability of data. This mode requires support from the guest operating system to have paravirtualized drivers.

3. **Emulated I/O**: In this mode, the hypervisor emulates a storage device. Guest drivers do I/O to the emulated device and the emulated device in-turn does I/O to the actual physical device.

Modes 2 and 3 above need support from the hypervisor to carry out the complete I/O. Beyond the I/O modes listed above, virtual machines themselves can be configured in

1. **Dedicated Partition Mode**: In this mode, the virtual machine is installed the file system on a partition. This could be an entire disk, a virtual partition spanning multiple disks, an LVM partition or a disk partition.
2. **Virtual Machine Image Mode**: In this mode, the virtual machine is installed in an image file. A set of Virtual Machine Images (VMI) are kept together in a virtual machine repository

Understanding the details of the various image formats is essential to identify the cost of doing I/O operations and hence the levels of caching and the cost of caching data in memory. In this paper, we don't focus on any specific image file format. The focus is on common strategies.

3 Page Caching Strategies

There are various strategies that one can employ for page cache. The strategies are examined here

3.1 Guest Only Caching

In the guest only caching strategy, the host page cache is bypassed. This is done by passing the `cache=none` argument to the hypervisor during guest startup. This option enables direct I/O and directly writes the data to disk, bypassing the host page cache. This strategy works well for cases where the filesystem of the VM is dedicated to the guest using direct assignment for I/O or if the guest works in dedicated partition mode. Guest only caching mode can be used with VMI's, but it can be an ineffective strategy if the hypervisor is doing I/O. Several VM's running in parallel, have their own I/O scheduler and if the host does not merge the I/O's, it can cause excessive head movement in seeking devices. If there are several VM's running in parallel, they could cache memory proportional to their size. The total consumption of memory in each of the guests for caching can be very high. This consumption shows up as mapped memory in the hypervisor. The most effective way to free the memory cached in the guests is through ballooning. This requires that we have an auto ballooning daemon running in the background and a cooperative guest. [1]

3.2 Host Only Caching

In the host only caching strategy, the guest cache is not used for caching. All the caching is delegated to the host. This works well for VMI's, the host page cache optimizes disk I/O. The host is able to optimize I/O from all VM's and provides higher throughput. KVM supports `writethrough` and `writeback` caching. In writethrough caching, the I/O is blocked till the data hits the disk. In writeback mode, the I/O returns as soon as the data hits the host page cache. The big advantage of the writeback mode is the throughput, the biggest disadvantage is potential of data loss if the hypervisor crashes. True host only caching is not possible, each guest maintains its own cache, which leads to mixed caching. Typical recommendations to reduce guest caching include changing the setting of `vm.swappiness` to 0. In section 4 we look at the results from the various modes mentioned in this section, including results when `vm.swappiness` is set to 0.

3.3 Mixed Caching

In this mode, both the host and the guests cache I/O data. This happens in a typical VM setup. The disadvantages listed in section 3.2 apply to this strategy. Beyond that a system with caching both on the host and the guest(s) incurs a penalty of double memory usage for caching the same data. While there are several ways to deal with page duplication problem [1], none of them deal with the duplication of page cache between the host and the guest.

[1]A guest is considered cooperative if it has a balloon driver enabled and running

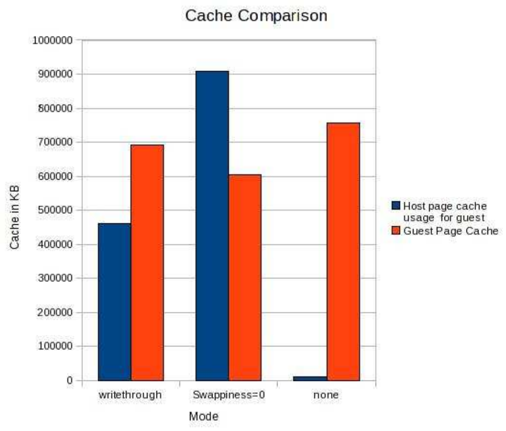


Figure 1: Cache Usage in various modes

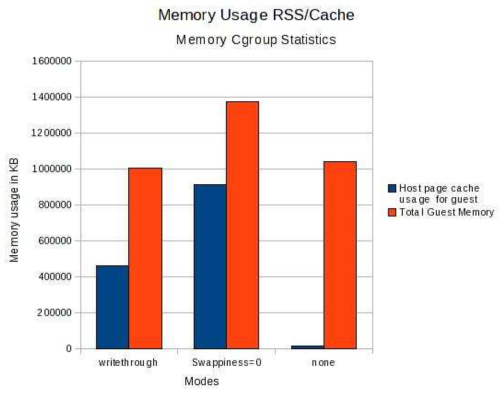


Figure 2: Host Page Cache and Guest RSS Usage in various modes

4 Page Cache Control

The double caching behaviour of was studied using memory cgroups [7]. A new cgroup was created for the virtual machine being executed. The virtual machine(s) ran the kernbench [8] benchmark. Memory cgroups can provide information about the RSS and page cache (mapped and unmapped) usage of the process running inside the cgroup (in this case the virtual machine comprises of the processes running as a part of hypervisor). Each VM was allocated 1 gigabyte of RAM and 2 Virtual CPUs (VCPUs)

Figure 1 shows the unmapped cache usage in three modes.

1. The first mode is the writethrough mode, which was described earlier in section 3.2. The guest and host both consume memory for page cache simultaneously and independently. The guest usage is however larger than the host usage. The data showed 60% of the data was duplicated over the entire run of the benchmark.

2. The second mode is the writeback with swappiness in the guest set to 0. The results showed that the guest page cache usage was lower than the host page cache usage and also lower than the usage in writethrough mode. The usage however was not close to 0, it was close to 50% of the host page cache usage. The host page cache usage was quite high.

3. The last mode is the direct I/O or the `cache=none` mode. In this mode, the hypervisor uses direct I/O to write out the pages from the guest block device to the disk. The data shows that the host page cache usage for the virtual machine is almost 0, all the caching is done in the guest. The size of the cache in the guest is high and higher than the other modes experimented with.

Figure 2 shows the page cache usage on behalf of the guest versus the RSS of the virtual machine. The results show that in addition to the memory being occupied by as cache in the guest (which shows up under RSS usage in the figure), the host is also caching page cache data. The key observations are

1. Host side caching for `cache=none` is almost 0 as expected.

2. With `cache=writethrough`, there is still double caching. The host uses close to 40% of the guest memory for caching data

3. When swappiness is set to 0 and the mode is `cache=writeback`, the host uses additional memory to cache guest data.

5 Proposed Approach

The proposed approach consists of two mechanisms to reduce the double caching of page cache data. The approaches are discussed

5.1 Mixed Caching With Host Emphasis

In this mechanism, both the guest and host use memory for page cache, but the cache is primarily pushed towards the host page cache. The guest page cache is monitored and shrunk frequently. The kernel has a partial implementation of this approach for NUMA systems [4] when the `zone_reclaim_distance` is greater than 0, implying that the cost of allocation from different nodes is high, the code does local reclaim of easy to free pages before allocating from a distant node. The algorithm reuses this behaviour and exploits the `min_unmapped_ratio` to keep the unmapped page cache usage under control.

Algorithm 1 Modified VM algorithm for page cache control

```
get_page_from_freelist()
...
determine zone to allocate from
if zone is below watermark then
  if should_balance_unmapped_cache() then
    wakeup kswapd
  end if
end if
```

Algorithm 2 Check if page cache should be controlled

```
should_balance_unmapped_cache()
if unmapped pages for zone > min_unmapped_ratio * number of zone pages then
  return TRUE
else
  return FALSE
end if
```

Algorithm 3 Kswapd changes

```
balance_pgdat()
...
on wakeup check if zone is below watermark or should_balance_unmapped_cache()
if unmapped pages need balancing then
  Reuse zone_reclaim logic
  for various reclaim priorities do
    Invoke reclaim targeting only unmapped pages and with swapping out of pages disabled
  end for
end if
```

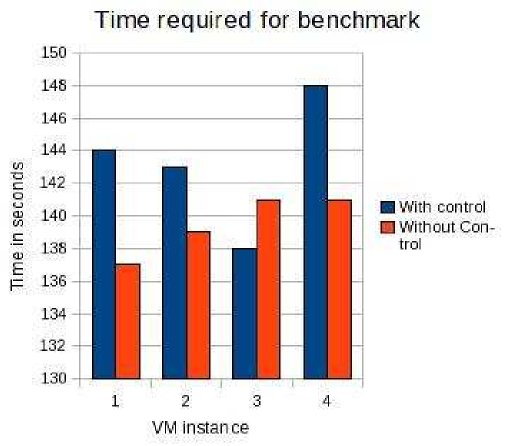


Figure 3: Time comparison of kernbench for with and without changes

Algorithms 1, 2, 3 show the changes made to control unmapped pages in the page cache. The code provides control over unmapped page cache via a boot parameter called `unmapped_page_control`. This boot parameter selectively activates the page cache control feature. By default 1% of the memory can be used for unmapped page cache. There is a sysctl `vm.min_unmapped_ratio` that can be tuned in the guest to control the amount of unmapped page cache.

5.1.1 Experiments and Results

The approach was tested by running four VM's in parallel, each running kernbench. Each VM had 1 GB of memory and 2 VCPUs.

The figure shows an overhead of close to 5% when the feature is enabled with `min_unmapped_ratio` set to 1%.

Figure 4 shows the free and cached memory usage of the benchmark running in four VM's without any changes to support control of unmapped pages. As can be seen, the free memory is low and the unmapped page cache memory usage is high. Figure 5 shows the free and cached memory usage of the same benchmark running in four VM's with the `unmapped_page_control` boot parameter specified during bootup. The figure shows a higher free memory and lower unmapped page cache utilization.

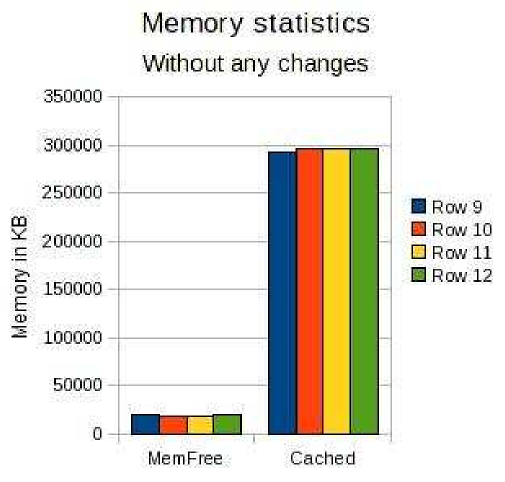


Figure 4: Free and cached memory inside the guest without any changes

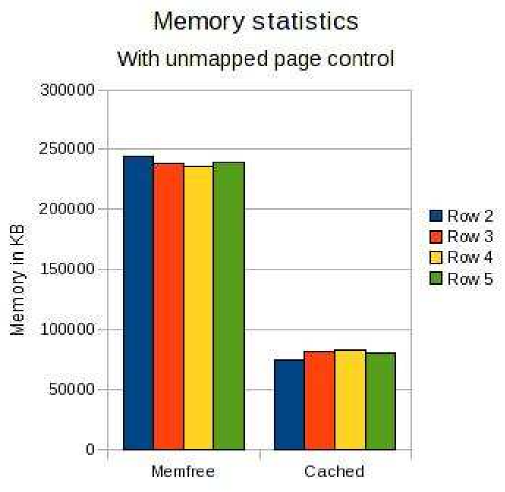


Figure 5: Free and cached memory inside the guest with changes

5.2 Cooperative Unmapped Page Cache Control

In this mechanism, the ballooning driver is used to cooperatively control page cache. In contrast to the previous approach, this approach is activated selectively on memory pressure within the hypervisor. The code changes in this approach are quite simple and consists of the following:

1. Create a new GFP flag, called `__GFP_FREE_CACHE`
2. Use `__GFP_FREE_CACHE` from the balloon driver, when it allocates pages under pressure.
3. The virtual memory subsystem honours the `__GFP_FREE_CACHE` flags by reusing code from `zone_reclaim` and the approach above to free both unmapped page cache and slab cache pages when the guest operating system is ballooned.

The key challenge with this approach is that the cache usage is externally controlled when ballooning occurs. It is important to make the correct decisions on when to balloon a particular guest and by how much. Typically a hypervisor would have a automatic tuning daemon whose job is to monitor memory usage in the host, the free memory, memory pressure in the host, the guest memory usage, various entitlements and makes smart decisions on which guests to balloon [2]. For the experiments and results obtained using this approach, we used a similar tool to monitor and automatically balloon the guests as required.

5.2.1 Experiments and Results

The test setup involved four VM's all running kernbench with 4 VCPUs and 6GB of memory. Four guest VM's ran this test in parallel. The test was run under a memory monitor as described in section 5.2, which means it was subjected to auto ballooning based on host memory pressure, the size, usage and entitlement of each of the VM's. As stated earlier, the ballooning operation could increase or decrease the memory footprint of the guest VM.

[2]A ballooning operation can either reduce the memory footprint of the guest or give it additional memory to use

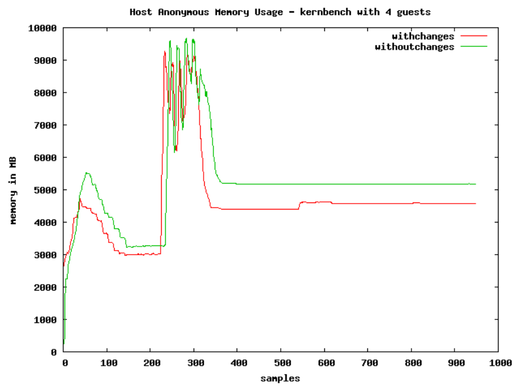


Figure 6: Host Anonymous Memory, running kernbench four VM's

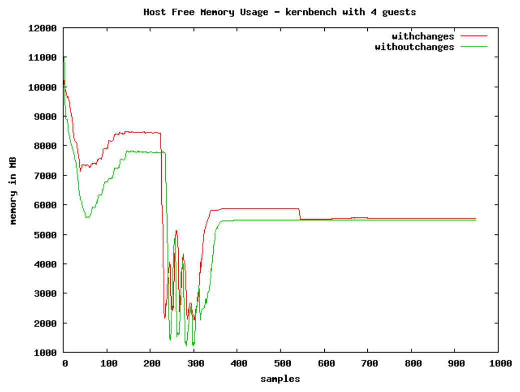


Figure 7: Host Free Memory, running kernbench four VM's

	VM	With Changes	Without Changes
make -j3	1	88.83	87.582
make -j16	1	76.786	76.686
make -j3	2	88.124	87.463
make -j16	2	77.264	76.704
make -j3	3	88.808	87.544
make -j16	3	76.748	75.522
make -j3	4	88.128	87.436
make -j16	4	76.828	75.63

Table 1: Elapsed time four VM's running kernbench

Figure 6 shows the anonymous memory usage in the host and correspondingly figure 7 shows the in the graph show the usage and free memory with and without changes to the operating system for cooperative ballooning. Figure 6 shows that the anonymous memory usage after the changes is lower as expected. This indicates that the cooperative ballooning technique, reduces the cache size and in turn the RSS size of each guest VM[3]. Similarly figure 7 shows that the free memory in the host is higher with changes.

Table 1 shows the results for the kernbench run in each VM with and without the ballooning changes for cooperative page cache management. The results show no significant overhead of the patches, but as the graphs earlier depict, it results in higher free memory in the hypervisor.

6 Future Work

The approaches listed in the paper are by no means complete. There are several additional possibilities to reduce page cache deduplication. One of them is to extend KSM [1] to deal with page cache data between host and guest operating systems. There is also additional scope in paravirtualizing hints such as `madvise(2)`, so that the hypervisor is aware of the hints and can appropriately handle the hints and manipulate its page cache usage in line with the hints coming from the applications running in the guest operating system.

7 Conclusion

Our results show that there is definitely double caching of page cache data between the hypervisor and guests.

[3]As seen from the host

Our approach pushes the caching of page cache data (more specifically unmapped page cache) to the hypervisor. The approaches listed above *unmapped page control* and *cooperative page cache control*.

The unmapped page control approach provides the best control over double caching and it also provides the flexibility to the user on what percentage of memory can be used for unmapped pages. This approach however, shows a noticeable overhead on the run time, due to the control introduced. We noticed in our experiments that the overheads came from the time required to scan and remove unmapped cached pages, rather than the lack of memory for caching.

In the cooperative approach provides noticeable benefit in terms of free memory available when the technique is used. The technique however, requires a daemon that continuously monitors all the guest operating systems and invokes ballooning operations when necessary. The really good aspect of this approach was minimal to no overhead in implementing this feature.

We believe that both the approaches listed above have an important role to play. The invocation and usage of these approaches is best left to the system administrator/user or a higher level software making decisions for virtualization environments. The key advantage these approaches provide is that they allow more free memory in the hypervisor, which allows additional work to be executed in the hypervisor.

8 Acknowledgements

The author would like to thank the following people for their help and support throughout this effort. Their names appear in no particular order below. Naren Devaiah, Premalatha Nair, Dipankar Sarma, Vaidyanathan Srinivasan, Adam Litke, Joel Schopp, Mike Day, Paul Mckenney, Karl Rister, Avi Kivity, Dave Hansen, Tim Pepper, Anthony Liguory, Rayan Harper, Rik van Riel, Larry Kessler, Ankita Garg, Supriya Kannery, Venkata R Jagana and many others who've been helped me via discussion/suggestions.

9 Legal Statement

References

[1] Andrea Arcangeli, Izik Eidusa, and Chris Wright. Increasing memory density using ksm. In *OLS '09: The 2009 Linux Symposium*, pages 19–28, 2009.

[2] Muli Ben-Yehuda, Jimi Xenidis, Michal Ostrowski, Karl Rister, Alexis Bruemmer, and Leendert van Doorn. The price of safety: Evaluating iommu performance. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 9–20, July 2007.

[3] Avi Kivity. kvm: the linux virtual machine monitor. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 225–230, July 2007.

[4] Christoph Lameter. Local and remote memory. In *Memory in a Linux/NUMA System*, pages 1–25, July 2006.

[5] Linux Kernel Mailing List. http://lkml.org, Last viewed in May 2010.

[6] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, 2008.

[7] Balbir Singh and Vaidayanathan Srinivasan. Containers: Challenges with memory resource controller and its performance. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 209–222, 2007.

[8] Kernbench version 0.42. http://www.kernel.org/pub/linux/kernel/people/ck/apps/kernbench/, Last viewed in May 2010.

The Benefits of More Procrastination in the Kernel

Geoff T Smith
Your affiliation
`your-address@example.com`

Abstract

Scaling Beyond 10 Gigabit Ethernet

Peter P. Waskiewicz Jr.
LAN Access Division, Intel Corp.
peter.p.waskiewicz.jr@intel.com

Abstract

10 Gigabit Ethernet is a fast-growing market in today's networking industry. Faster platforms, lower infrastructure costs, and a growing number of advanced features have driven this migration to higher bandwidth needs. However, as the platforms keep getting faster and keep growing in number of CPU cores and NUMA memory nodes, scaling networking becomes more and more challenging.

This paper will illustrate the challenges facing high I/O networking. It will focus on what has been done in the past year to help scale, including better NUMA locality, cacheline-aligned structures, and more efficient interrupt handling. Looking to the future, the paper will highlight ongoing areas for improvement, such as interrupt affinity policies with MSI-X, better NUMA scalability, working with increased CPU counts, all of which are necessary to efficiently drive speeds beyond 10 Gigabit.

1 Introduction

Technology marches on. Huge growth in datacenter and other computing environments have been made in the previous year, with more innovations continuing to push technology forward. With the shift towards Cloud computing, large virtualized clusters, and storage systems running over standard Ethernet networks, the demand for massive amounts of network bandwidth has never been higher. 10 Gigabit Ethernet just isn't enough in many cases, and the demand for multiple ports of 10 Gigabit Ethernet has become the status quo for these larger datacenter installations. To meet the need for higher bandwidth, the industry is also moving towards the next tier of networking technology, 40 Gigabit Ethernet. The question is: can today's Linux kernel handle the performance demand?

To better understand what's required to scale beyond 10 Gigabit networking, a better understanding of the current challenges at 10 Gigabit networking is requiremented.

This paper will dive into various hotspots in today's kernel that have large impacts on network performance scaling. As the throughput demands increase towards 40 Gigabit, these hotspots become more pronounced in their impact on overall performance scaling. Using examples of today's existing platforms, an illustration of each performance point will be made, and how to tune it for better scaling and efficiency.

2 The Evolution of Networking

2.1 Today's 10 Gigabit Ethernet scaling issues

10 Gigabit Ethernet still has a way to go before reaching optimum performance. Many of the performance tuning problems with 10GbE are masked by how fast today's platforms are. However, these issues can be amplified by reducing the size of the packet payload, and watching packets per second (pps) dropping dramatically. They can also be amplified by increasing the number of 10GbE ports in a machine, or by running other CPU and memory-intensive applications on the same machine. The out-of-the-box performance still has many bottlenecks and scalability issues.

2.2 The speed evolution: Why 40G and not 100G?

The next evolution of Ethernet technology has made a somewhat odd jump. Up until now, each generation has made a factor-of-ten jump in speed, such as 10Mbit to 100Mbit, 100Mbit to 1Gbit, and 1Gbit to 10Gbit. So why not 10Gbit to 100Gbit? There are two primary reasons. The first reason is the optics technology used for 40Gbit is the same technology used in 10Gbit. This makes 40Gbit a more cost-effective evolution of Ethernet technology, especially in a cost-conscious environment such as corporate IT shops.

The bigger reason for 40Gbit, however, is that 10Gbit is still highly demanding on today's platforms. At smaller packet sizes, such as in routers and bridges, 10Gbit is still able to consume a decently powerful platform. 40Gbit is a massive evolution of I/O demand, which planned future platforms will be strained to drive. 100Gbit devices would be starved due to the sheer amount of data throughput required of the memory bus and processors. Custom ASICs may be able to drive 100Gbit on single-port switch uplinks, but general purpose deployment of such devices will require much more platform-level horsepower to be viable solutions.

3 NUMA

3.1 NUMA in use today

NUMA (Non Uniformed Memory Architecture) is a memory topology concept. In older platform designs, a front-side bus was utilized that had a single memory controller. In the larger multi-proc systems, multiple front-side busses were present, also having a memory controller per front-side bus. However, the bandwidth among CPUs was still partitioned across CPU sockets. Also, the front-side bus frequency to memory was typically a few orders of magnitude slower than the CPUs, thus causing memory-intensive applications to be bottlenecked by memory access.

With a NUMA topology, memory controllers control a bank of memory that is preferred for a specific set of computational resources. In a recent AMD® or Intel® platform, the memory controllers are embedded in the CPU socket itself, so in a dual-socket configuration, there would be two memory controllers, and therefore two NUMA domains. Also, today's NUMA memory controller implementations interleave memory accesses, bringing the memory access frequency close to or equal with the CPU frequency. This nets a large boost in performance for those memory-intensive applications, which high throughput networking is a part of.

3.2 Scaling networking with NUMA

While interconnects between CPU sockets in NUMA-enabled platforms are incredibly fast (e.g. 6.4 GT/sec on Intel's QuickPath Interconnect™ - QPI[1], 6.4 GT/sec on AMD's HyperTransport™ [2]), making memory accesses across these interconnects increases memory latency and can drive down I/O throughput. This becomes an even larger problem when the number of sockets increases, since process scheduling becomes very non-deterministic. The amount of inter-socket memory accesses will increase dramatically, and in a heavy workload scenario, this will drive memory latencies very high.

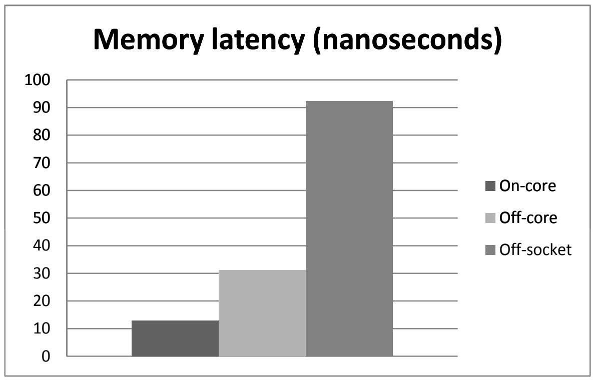


Figure 1: Memory latencies intra-socket and inter-socket

As seen from Figure 1, keeping memory accesses local to a CPU socket yields the best performance[3]. Memory latency on the same CPU socket is very fast, but once the memory access leaves for another CPU socket, the latency increases dramatically. As workloads increase, the amount of bandwidth flowing between CPU sockets increases almost exponentially. This is due to traffic from the original memory requests, plus the requests being forwarded to other CPU sockets and memory controllers, plus all the replies from each sets of requests. Even with today's CPUs and platform topologies, this perfect storm of NUMA node cross-talk at high network workloads will bring a high-end platform to its knees. Better attention to tuning and memory allocation configuration is required to make the platform more efficient.

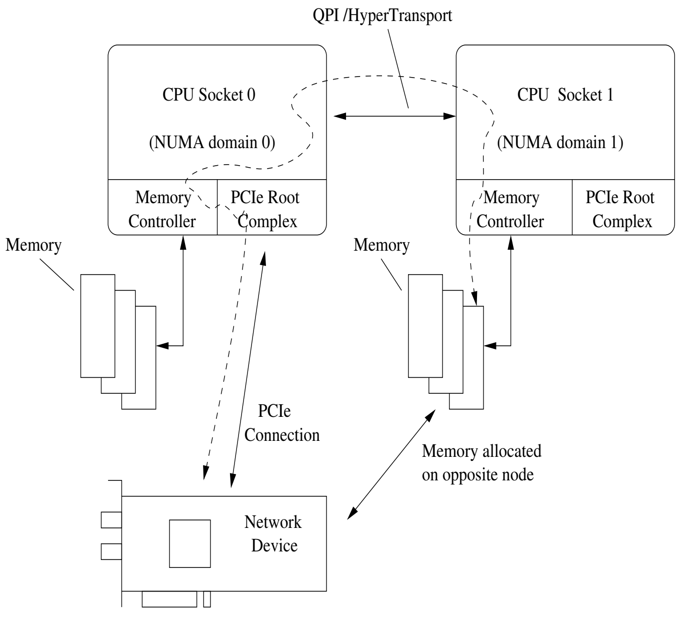


Figure 2: Example PCIe layout with NUMA cross-talk

3.3 NUMA vs. PCI Express root complexes

One of the many scaling challenges that networking faces is the location of the PCIe root complex. On older platforms (front-side bus architecture), the North Bridge chipset handled memory accesses and also owned the PCIe lanes allocated to onboard devices. This allowed device drivers to operate without caring where the PCIe lanes were with respect to a CPU affinity.

In today's platform topologies, PCIe root complexes are being driven into the CPU socket itself. This means that certain PCIe slots on a motherboard have hard-defined CPU affinities since the PCIe lanes attached to that slot are solely owned by a particular CPU socket. Now this topology can help with I/O latency for that device, since the PCIe to memory controller physical distance and access time are now reduced. However, this can be the source of additional problems, where processes are running on a different CPU socket, but need the I/O devices attached to another socket. This now creates additional NUMA node crosstalk, as well as crosstalk to access the PCIe device.

In Figure 2, the network device might have all memory structures and DMA regions allocated on NUMA node 1. But the PCIe uplink is connected to the root complex on CPU socket 0. Therefore, all memory access and DMA operations must cross the CPU socket interconnect, causing massive amounts of NUMA crosstalk. The question is: is it better to allow this crosstalk to occur, or is it better to only allow access to an I/O device from the native CPUs associated with its socket? This is a very difficult question to answer in a non-partitioned (non-virtualized) system. Artificial partitions placed on the process scheduler can have unwanted side effects (cache-thrashing, unbalanced CPU loads, etc.), not to mention wanting to use all available computing resources within a platform. The best way to approach this issue is to align network flows with particular CPUs and network device queues. This way, most of the network flows will be aligned to the local CPUs, since outgoing flows will return to the same CPUs. This will help reduce NUMA cross talk, since many of the network flows will be deterministic. More discussion on network flow alignment is found in section 4.

4 Network Flow CPU Alignment

4.1 What thrashed my cache?

Benchmarks are always easy to manipulate to yield desired results. For example, to demonstrate how well a 10 Gigabit device can scale, the test used can be tuned to have perfect CPU-to-flow alignment, interrupt alignment, and optimal payload buffer sizes. Then the exact number of threads can be run to allow the device to hit line rate, while keeping the CPUs at the lowest CPU utilization possible. This is usually a great graph to look at, but is completely uninteresting and unrealistic in real-world computing scenarios.

A more typical model has a mix of traffic types, payload buffer sizes, and flow/interrupt alignment. In many cases, the selection of network queues is random both on transmit (done in the kernel) and on receive (typically done in the underlying hardware). Plus it's pretty much guaranteed the Rx and Tx queue selections will not line up; in other words, network flows will be spattered across the CPUs.

This spattering of network flows causes CPU cache-thrash, and also causes the operating system's process scheduler to work harder by rescheduling processes to other CPUs. The overall effect is more time is spent with the process accounting versus getting data moved. This is inefficient. This problem becomes even worse as more threads are added, and more network ports are added.

4.2 Flow Steering to the rescue

Flow Steering is a solution that can resolve the cache-thrash and overworked process scheduler. The general concept is to identify a new flow on transmit, usually by extracting certain fields from the packet (source/destination address and ports), and computing a hash using those values. Then store this hash value along with the CPU the flow originated from into a table. Then on the receive side, do the same packet inspection and hash calculation, and then look for a match. If a flow matches an entry in the table, then direct that packet at the cached CPU in the table.

At the time of writing this paper, a software-based proposal is still being worked on to solve the problem of Flow Steering. This work is from Google®, and is called RFS, or Receive Flow Steering. This will help devices that have either single or multiple receive queues. The driver will simply hand the packet to the operating system, and the network stack will compute the hash and issue an IPI (inter-processor interrupt) to schedule the processing of that packet on the intended CPU.

Another approach is to have this support natively in the networking hardware. Intel's® 82599 10GbE controller, run by the ixgbe driver, has an implementation of Flow Steering in silicon. It's called Flow Director, with limited driver interaction to program the filters. Other hardware that is known to have these filtering capabilities is Sun's® Neptune 10GbE adapter, which the niu driver runs. An effort is underway to try to use any available hardware offloads for Flow Steering with the kernel's software layer, thus creating a common Flow Steering interface.

In Figure 3, specific flows are aligned to a specific CPU core. The alignment occurs when the transmit queue and receive queue return to the same CPU. In this figure though, it can be seen that there isn't a perfect vertical alignment; flows on CPU 2 are going to Rx and Tx queues 1, where flows on CPU 1 are going to Rx and Tx queues 2. In a vertical alignment, CPU 1's flows would go through Rx and Tx queues 1, and CPU 2's flows would go through Rx and Tx queues 2. This model works better with the in-kernel Tx queue selection engine using the netdevice's `->select_queue` mechanism.

Flow Steering still isn't the end-all solution to the scaling problem though. Even with network flows aligned

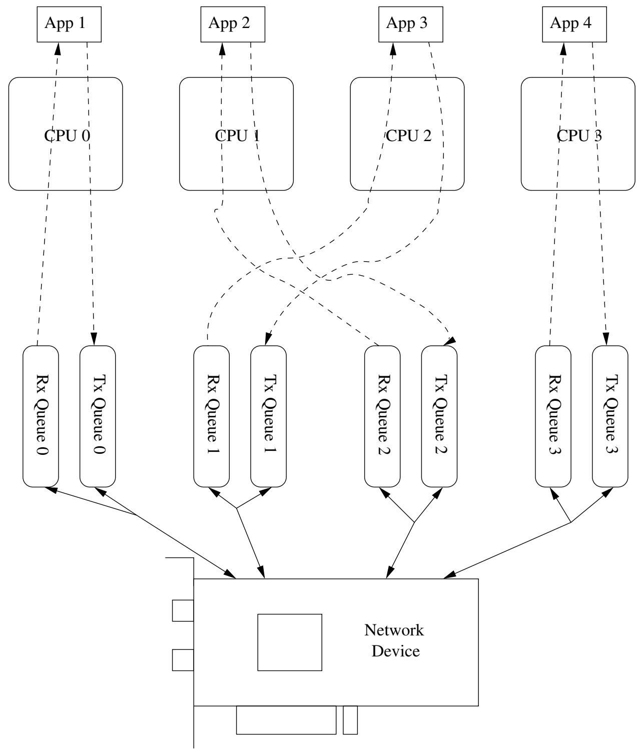


Figure 3: Example of Flow Steering in action

to CPUs, there is still one more variable to the problem: interrupts.

5 Interrupt Alignment and MSI-X

5.1 MSI-X interrupts in networking

As network devices increase in speed, the only way to scale these workloads is to spread the load across multiple processing engines, both in the network device and the host CPUs. Network devices also come with multiple queues in hardware that help partition traffic flows for easier processing. The traditional INT_* pin interrupts can't be used to take advantage of these multiple queues. Also, newer Message Signaled Interrupts (MSI) can't be used, since they still only provide a single interrupt source, which doesn't map well into multiple producers and consumers.

MSI-X is an extension of MSI, where multiple interrupt vectors are allocated. Put simply, MSI-X allows a device to have multiple interrupt sources, each of which can fire independently from one another. However, these are delivered as messages on PCI-Express, rather than a hard interrupt line to a processor. This also

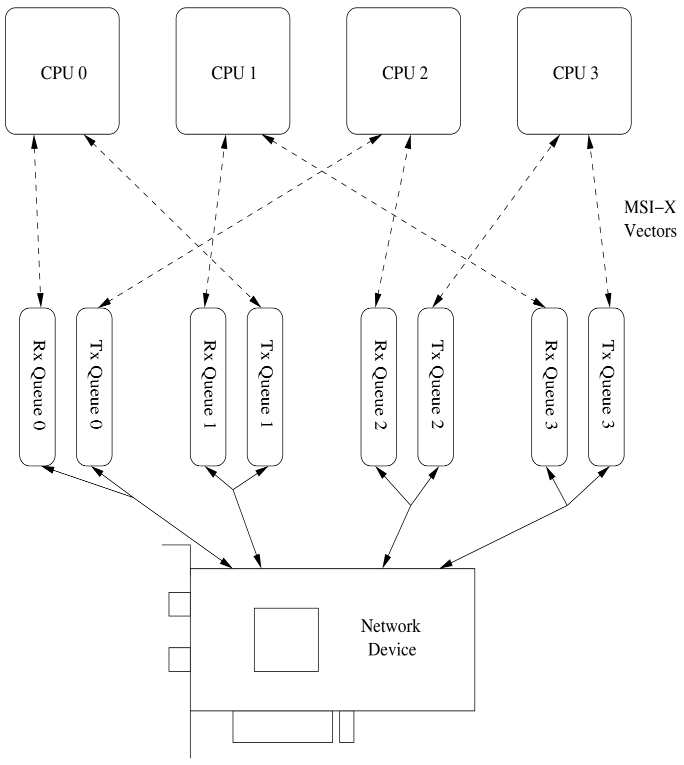


Figure 4: Example of unbalanced MSI-X interrupts

allows the MSI-X message to be directed at a specific CPU or group of CPUs if desired.

In networking, MSI-X can be used very effectively with multiple queues. A single MSI-X vector can be assigned to each individual Rx or Tx queue, giving a one-to-one mapping of queues to interrupt sources. Now these queues can partition the traffic load, plus they can notify an individual CPU that work is available, without impacting other CPUs or queues on the device. This interrupt model is required in order to scale to 10 Gigabit and beyond.

Figure 4 illustrates an example of MSI-X vectors being assigned to network resources, and mapped to a CPU. However, this illustration shows how MSI-X vectors can be completely sprayed across CPUs in a random fashion. This prevents deterministic behavior of the interrupts, and will severely impact the scalability of the networking load.

5.2 What thrashed my cache, again?

Even with perfect network flow alignment with CPUs, the CPU caches can still be thrashed, and the operating system process scheduler can still be overworked. Interrupts are used in devices to both indicate when hardware resources can be reclaimed by the driver, and on the receive path when data arrives and needs to be passed up the network stack. When the interrupt fires, it has a CPU affinity associated with it. So if the network flow alignment is currently bound to one CPU, and its associated MSI-X vector is bound to a different CPU, the cache and process scheduler will be thrashed, again.

MSI-X interrupt affinity is also essential for proper network scaling. But this is not true if the Tx queue selection mechanism doesn't match the Rx queue selection mechanism. In other words, if the CPU generating the transmit session doesn't match the CPU that would process the receive for that data stream, then interrupt CPU affinity is useless. At that point, the CPU cache would be compromised, and process scheduling would need rescheduling.

Once a Flow Steering mechanism is in place, whether it's in hardware or software, interrupt affinity for those flows is needed. Figure 5 shows what a balanced system would look like, when Flow Steering aligns flows, and the associated MSI-X interrupts are affinitized to the same CPU as the flows.

5.3 Automatic interrupt affinity control

Development has been ongoing to allow device drivers to control their own affinity for an interrupt vector. Many proposals have been made, ranging from direct affinity control from a driver, to extending the userspace daemon irqbalance to have more intelligence of network device behavior. None of these seemed to be a complete solution, mainly because intelligence of interrupts and network flows couldn't be fully shared with these proposals, and the correct decision couldn't be made.

A new approach was recently merged into the kernel, and into the irqbalance daemon. The new interface is present in the 2.6.35 kernel (yet to be released at the time of this writing). It allows a device driver to provide a "hint" to irqbalance, where this hint is a CPU mask of preferred CPUs for this interrupt to be linked. Then irqbalance can take this information, and further balance interrupts on the platform using these hints. This way the driver gets the affinity behavior of its interrupts, and the rest of the platform is properly balanced based on interrupt load across the system.

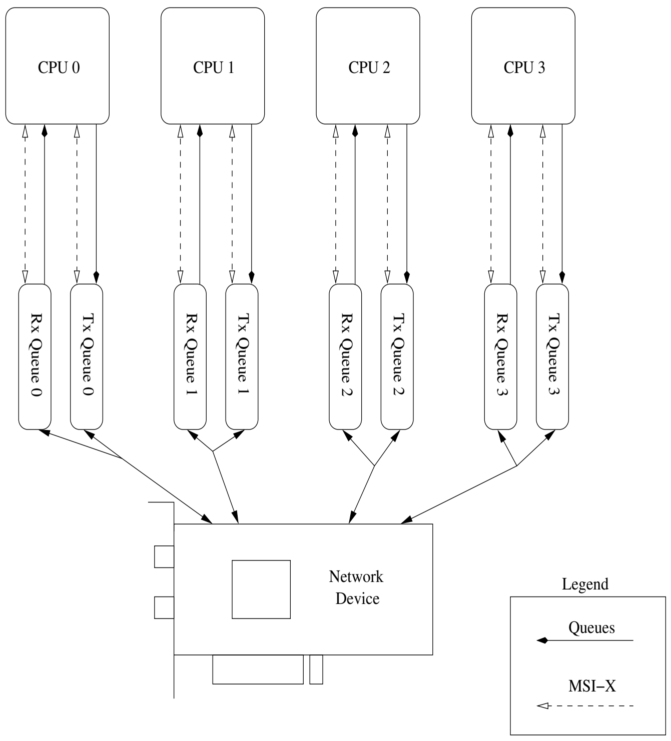


Figure 5: Example of balanced MSI-X interrupts with Flow Steering

6 Scaling to 40 Gigabit

6.1 Hiding behind the platform?

40 Gigabit is the next evolution for Ethernet devices. As showcased earlier in this paper, significant gains in performance can be seen on 10 Gigabit Ethernet devices just by tuning the kernel in certain ways. While these tunings are not absolutely required for 10 Gigabit Ethernet, they will be necessary for 40 Gigabit Ethernet.

The platforms that exist today, and the platforms that are planned for the future, will not be able to scale 40 Gigabit Ethernet with the random resource layout that occurs in today's kernel and device drivers. Even with the shift towards larger PCIe busses like PCIe 3.0, which boasts 8 GT/sec of bandwidth per PCIe lane, platforms will not have enough bandwidth to handle the amount of memory I/O required if it's not tuned. In Figure 2, one sees a layout that can cause extreme NUMA cross-talk to occur. In a 40 Gigabit Ethernet setup, if all memory accesses needed to cross the CPU interconnects to access a remote memory node, the projected interconnects won't even be able to keep up.

One problem that doesn't have a solution yet is the PCIe affinity to a specific CPU socket, and how a single 40 Gigabit device will scale being bound directly to a CPU socket. The best way to scale is to make use of all computing resources present in a platform. That means using all available CPU cores in each CPU socket, and using all available memory bandwidth of the other memory controllers in the platform. However, this also means that CPU interconnect traffic will increase, which is not optimal, as previously pointed out.

6.2 New technologies in a 40 Gigabit world

40 Gigabit Ethernet isn't just about a faster link. Features and technology advancements are needed to utilize the large pipes. Different methods to carve up the pipe are the first step. Advances in Single Root I/O Virtualization, SR-IOV, have enabled support in various ecosystems. VMWare, Xen, and KVM all have SR-IOV support. With this technology, bandwidth can be allocated to different Virtual Machines, and make better use of the full 40 Gigabit pipe. Other virtualization technologies evolving include the Virtual Edge Bridging technology, VEB, which has been getting some traction recently in the Linux kernel.

Other advancements and refinements with Data Center Bridging (DCB) in IEEE have also created more opportunities to use this technology in 40 Gigabit networks. Large network-based storage deployments, such as Fiber Channel over Ethernet and iSCSI, can use DCB to produce reliable, cost-effective solutions by using a 40 Gigabit device.

As more of these technologies emerge that take advantage of larger network pipes, 10 Gigabit and 40 Gigabit networks become even more practical for large-scale deployment.

6.3 How a 40 Gigabit layout should look

Pulling all the tuning requirements together, a scalable 40 Gigabit Ethernet layout looks fairly rigid. This does imply that a system administrator will require knowledge of the network device layout, or a system administrator will need to influence the layout of the networking resources in the underlying drivers. Applications need to be placed on specific CPU cores to make the best use of memory controller bandwidth, while trying to spread the load across the majority of the platform.

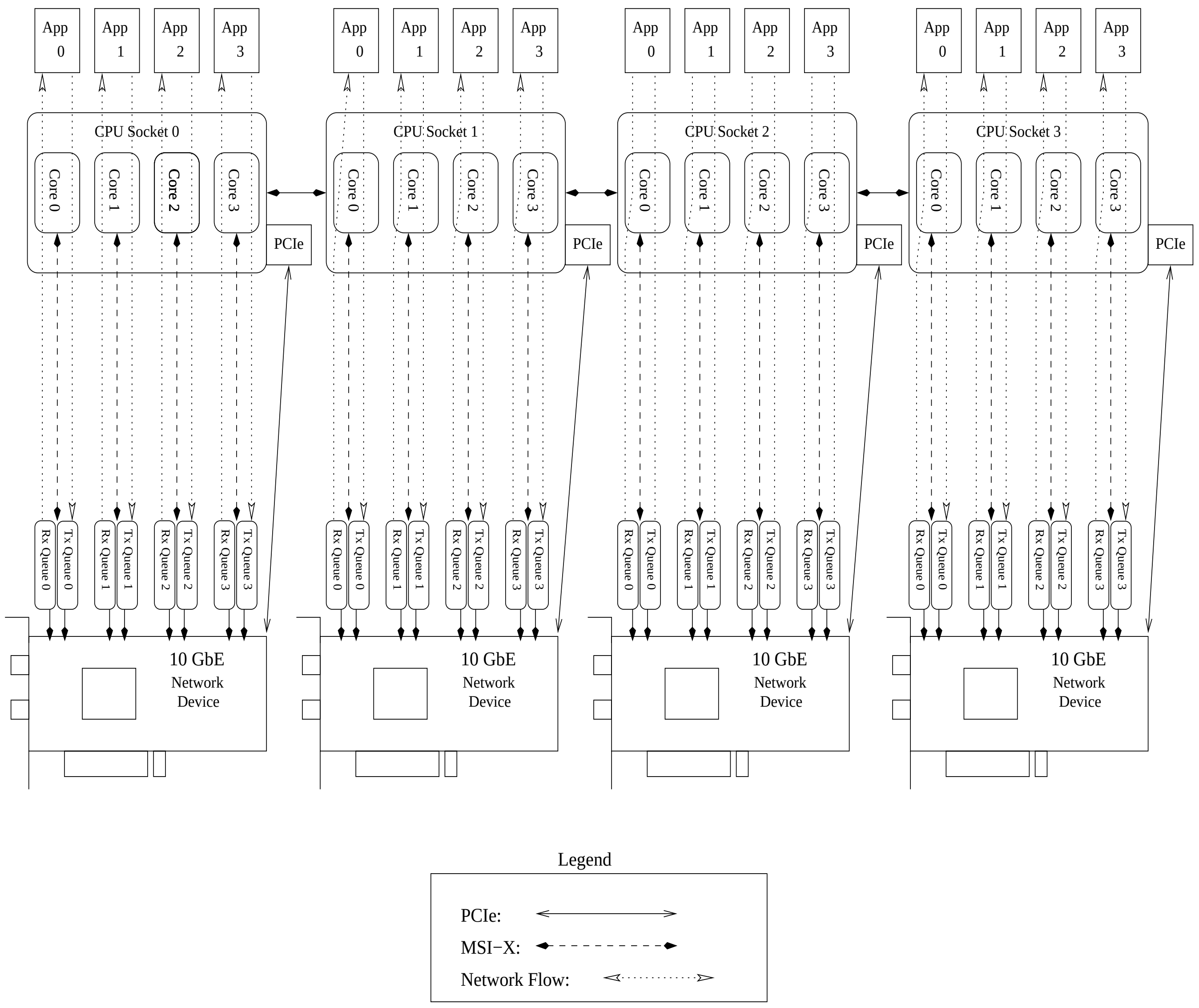


Figure 6: Four 10GbE Devices with MSI-X and Flow Affinity

Figure 6 shows the whole picture of a 40 Gigabit layout using four 10 Gigabit devices. This is what today's model would look like. Applications running on each CPU socket would be scheduled on specific CPU cores, with their flows being steered into the attached 10 Gigabit device and back. The associated MSI-X interrupts for the queues are also routed to the CPU cores the network flows are being steered to. This provides full vertical affinitization of network flows and interrupts, and will achieve the best model for scaling 40 Gigabit networking.

7 Conclusion

Ethernet networks are still evolving to higher speeds such as 10 Gigabit. As costs continue to drop, more demand for 10 Gigabit and 40 Gigabit networks will emerge. And while today's platforms are capable of handling the load required of these devices, it's obvious the Linux kernel still needs help to be tuned for best performance. In the years to come, work must be completed in the areas discussed in this paper to improve the overall out-of-the-box experience for users.

References

[1] Intel Corp. QuickPath Specification *http://www.intel.com/technology/quickpath*

[2] AMD Corp. HyperTransport Specification *http://www.amd.com/us/products/technologies/hypertransport-technology/Pages/hypertransport-technology.aspx*

[3] Paul E. McKenney, NetConf 2009 *RCU and Breakage* *http://vger.kernel.org/netconf2009_slides/NetConf.2009.07.13b.odp*