

Proceedings of the Linux Symposium

Volume One

July 23rd–26th, 2008
Ottawa, Ontario
Canada

Contents

x86 Network Booting: Integrating gPXE and PXELINUX <i>H. Peter Anvin</i>	9
Keeping the Linux Kernel Honest <i>Kamalesh Babulal & Balbir Singh</i>	19
Korset: Automated, Zero False-Alarm Intrusion Detection for Linux <i>Ohad Ben-Cohen & Avishai Wool</i>	31
Suspend-to-RAM in Linux <i>Len Brown & Rafael J. Wysocki</i>	39
Systems Monitoring Shootout <i>K. Buytaert, T. De Cooman, F. Descamps, & B. Verwilst</i>	53
Virtualization of Linux servers <i>F.L. Camargos, G. Girard, & B. des Ligneris</i>	63
MondoRescue: a GPL Disaster Recovery Solution <i>Bruno Cornec</i>	77
The Corosync Cluster Engine <i>Steven C. Dake</i>	85
LTTng: Tracing across execution layers, from the Hypervisor to user-space <i>Mathieu Desnoyers</i>	101
Getting the Bits Out: Fedora MirrorManager <i>Matt Domsch</i>	107
Applying Green Computing to clusters and the data center <i>Andre Kerstens & Steven A. DuChene</i>	113
Introduction to Web Application Security Flaws <i>Jake Edge</i>	123
Around the Linux File System World in 45 minutes <i>Steve French</i>	129

Peace, Love, and Rockets!	135
<i>Bdale Garbee</i>	
Secondary Arches, enabling Fedora to run everywhere	137
<i>Dennis Gilmore</i>	
Application Testing under Realtime Linux	143
<i>Luis Claudio R. Gonçalves & Arnaldo Carvalho de Melo</i>	
IO Containment	151
<i>Naveen Gupta</i>	
Linux Capabilities: making them work	163
<i>Serge E. Hallyn & Andrew G. Morgan</i>	
Issues in Linux Mirroring: Or, BitTorrent Considered Harmful	173
<i>John Hawley</i>	
Linux, Open Source, and System Bring-up Tools	183
<i>Tim Hockin</i>	
Audio streaming over Bluetooth	193
<i>Marcel Holtmann</i>	
Cloud Computing: Coming out of the fog	197
<i>Gerrit Huizenga</i>	
Introducing the Advanced XIP File System	211
<i>Jared Hulbert</i>	
Low Power MPEG4 Player	219
<i>J.-Y. Hwang, J.-H. Kim, & J.-H. Kim</i>	
VESPER (Virtual Embraced Space ProBER)	229
<i>S. Kim, S. Moriya, & S. Oshima</i>	
Camcorder multimedia framework with Linux and GStreamer	239
<i>W. Lee, E. Kim, J. Lee, S. Kim & S. Park</i>	

On submitting kernel patches	253
<i>Andi Kleen</i>	
Ext4 block and inode allocator improvements	263
<i>A. Kumar, M. Cao, J. Santos, & A. Dilger</i>	
Bazillions of Pages	275
<i>Christoph Lameter</i>	

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium,
Thin Lines Mountaineering*

C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium,
Thin Lines Mountaineering*

Dirk Hohndel, *Intel*

Gerrit Huizenga, *IBM*

Dave Jones, *Red Hat, Inc.*

Matthew Wilson, *rPath*

C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

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

Gurhan Ozen, *Red Hat, Inc.*

Eugene Teo, *Red Hat, Inc.*

Kyle McMartin, *Red Hat, Inc.*

Jake Edge, *LWN.net*

Robyn Bergeron

Dave Boutilier, *IBM*

Mats Wichmann, *Intel*

Authors retain copyright to all submitted papers, but have granted unlimited redistribution rights to all as a condition of submission.

x86 Network Booting: Integrating gPXE and PXELINUX

H. Peter Anvin

rPath, Inc.

<hpa@zytor.com>

Marty Connor

Etherboot Project

<mdc@etherboot.org>

Abstract

On the x86 PC platform, network booting is most commonly done using software that follows the Preboot Execution Environment (PXE) specification. PXELINUX from the SYSLINUX Project and gPXE from the Etherboot Project are popular Open Source implementations of key PXE components.

In this presentation, we will describe how these two projects were able to jointly develop an integrated PXE-compatible product that provides additional network booting capabilities that go well beyond the PXE specification. We will also discuss some of the organizational challenges encountered during this collaboration between two Open Source projects with different priorities, design goals, and development strategies.

1 Motivation

Open Source software development by definition allows and encourages code sharing and collaboration between projects. There are, however, costs associated with these endeavors, and these costs must be weighed against potential benefits associated with using code developed by another project.

In the case of improving integration between gPXE and PXELINUX, developers from SYSLINUX and the Etherboot Project were motivated to collaborate because they believed there might be significant benefits to leveraging work already done by the other project. Although the process of creating better interoperability between products required significant communication and effort, it was a useful and rewarding exercise for both development teams.

To understand how these two Open Source projects reached this point of collaboration we will examine the history of network booting on the x86 PC platform, as well as the development journey each project took prior to this collaborative effort.

2 The PC Platform: Ancient History

Network booting has been implemented in various forms for many years. To appreciate how it evolved it is instructive to examine the early days of PC computing when standards were few, and achieving consensus between vendors on any technical innovation was even more difficult than it is today.

The x86 PC platform has a direct lineage to the original IBM PC 5150 released in 1981. This machine, an open platform, and its successors, the 1983 IBM XT and the 1984 IBM AT, were widely copied by a number of manufacturers. The PC industry largely followed IBM's technical lead until the disastrous 1987 attempt at reclaiming their initial monopoly position with the closed platform PS/2 line erased their technical and marketplace leadership positions in the industry.

As a result, for the first half of the 1990s there was no clear path for new standards to become accepted on the PC platform, and PCs were becoming little more than massively sped-up versions of the IBM AT. Thus, even as new media such as networking and CD-ROMs became available to the platform, there was little support for booting from anything other than the initially supported floppy and hard disk, although PCs could optionally use an expansion card carrying proprietary booting firmware.

TCP/IP networking, as something other than a niche product, came late to the x86 PC platform. For many years, memory limitations when running MS-DOS and its derivatives meant that simpler, proprietary network stacks were used; the primary ones being NetBIOS from IBM and Microsoft, and IPX from Novell.

The response of early network card manufacturers, to the extent they supported booting from networks at all, was simply to provide a socket into which a ROM (usually an EPROM) could be inserted by the end user. This ROM had to be preprogrammed with firmware specific

both to the network card and the network protocol used; as a result, these ROMs were frequently expensive, and few PCs were ever so equipped. To further complicate the situation, the size of ROMs varied widely depending on the card. Some cards supported as little as 8K of ROM space, greatly limiting the amount and complexity of boot software that could be supplied.

In the early 1990s, as the use of TCP/IP became more prevalent, some manufacturers provided TCP/IP-based booting solutions using the then-standard BOOTP and TFTP protocols, often based on downloading a floppy image to high memory (above the 1 MB point addressable by DOS.) These solutions generally did not provide any form of post-download access to the firmware; the downloaded floppy image was expected to contain a software driver for a specific network card and to access the hardware directly.

By the mid 1990s, the PC industry, including IBM, was seriously suffering from having outgrown IBM AT standards. In January 1995, Phoenix and IBM published the “El Torito” standard [3] for booting PCs from CD-ROM media. Support for this standard was initially poor, and for the rest of the decade most operating systems that were distributed on CD-ROM media generally shipped with install floppies for booting PCs that lacked functional CD-ROM booting support.

3 NBI and PXE: Network Booting Strategies

As the cost of network interface cards (NICs) dropped dramatically in the early 1990s, the cost of a network boot ROM could be a substantial fraction of the cost of the NIC itself. The use of OS-dependent, user-installed ROMs as the only method for network booting had become a significant limitation. In the Open Source world, this issue was further complicated by the need to supply a physical piece of hardware, since a software-only distribution would require end users to have access to an expensive EPROM burner to program the software into an EPROM.

In 1993, Jamie Honan authored a document titled “Net Boot Image Proposal” [4] which defined image formats and methods for downloading executable images to a client computer from a server. A key feature of Jamie’s proposal was the specification of *Network Boot Image* (NBI) file format, “a vendor independent format for boot images.” This may have been the first attempt in the

Open Source world at specifying an OS-independent method for network booting on the PC platform.

To keep NBI loader code uncomplicated, a utility called `mknbi` was used to convert OS specific files such as kernels and initrds into NBI format for loading into memory. This simplified loader code because it only needed to load a single, uncomplicated image format. It did, however, require an extra step to convert OS images to NBI format prior to booting.

NBI was never used in the closed-source OS world. Instead, vendors continued to offer incompatible solutions, usually based on their respective proprietary network stacks.

In 1997, Intel *et al.* published the Wired for Management Specification (WfM) [5]. It included as an appendix a specification for the *Preboot Execution Environment* (PXE), a TCP/IP-based, vendor-neutral network booting protocol for PCs, which included an application programming interface (API) for post-download access to the firmware driver. The specification, as well as its current successor [6], both have numerous technical shortcomings, but it finally made it possible for NIC and motherboard vendors to ship generic network booting firmware. Higher-end NICs began to have onboard flash memory preprogrammed with PXE from the factory instead of providing a socket for a user-installable ROM. Motherboards began to have PXE firmware integrated into their BIOSes.

The PXE specification defines a set of software components (Figure 1) including a PXE Base Code (BC) stack, a Universal Network Driver Interface (UNDI) driver—both in ROM—and a Network Boot Program (NBP), which loads from a server. Of these components, only the UNDI is specific to a certain NIC. Just as with the Open System Interconnect (OSI) networking model, this model does not completely reflect the actual division into components, but is nevertheless useful as a basis for discussion.

The PXE approach is significantly different from the one taken by NBI. NBI simply loads a `mknbi`-prepared bootable OS image into memory with no further access to the ROM-resident network routines.¹ In contrast, PXE BC first loads an NBP, which then loads a target

¹Some NBI-compliant ROMs would provide APIs to request additional services, but those APIs were never standardized.

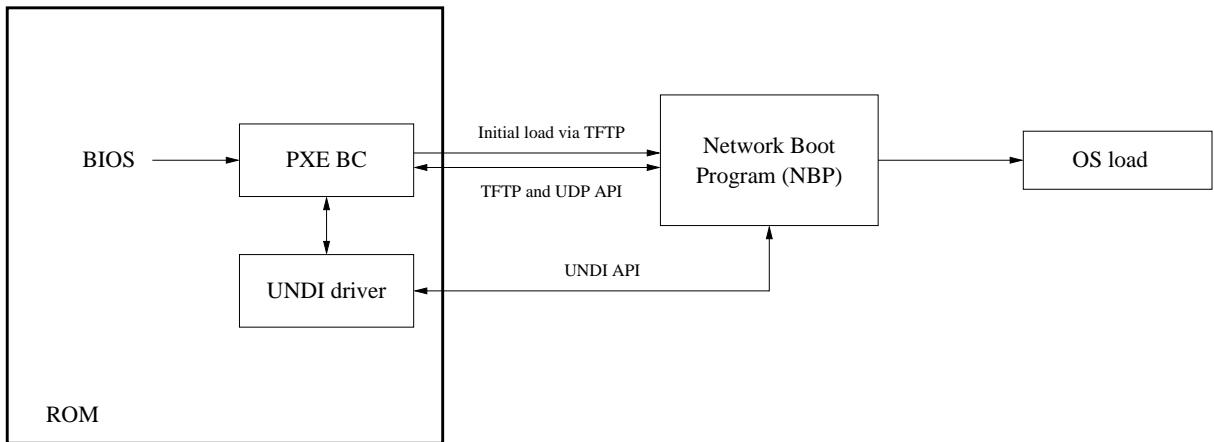


Figure 1: The PXE concept model

OS using still-resident BC and/or UNDI stacks from the ROM using an API defined in the PXE specification.

These differences in approach allow PXE ROMs to contain simple, compact loader code while enabling more specific and capable second-stage loader code to load native OS image formats without pre-processing. Further, NBP code resident on a network server can be upgraded centrally to fix bugs or add capabilities. Added costs to this approach versus NBI include extra time and server load to transfer an NBP before the target OS image is loaded, and the need to maintain multiple components on the boot server. These costs have minimal impact when using modern servers and LAN hardware.

4 The SYSLINUX Project

The SYSLINUX [1] project was started in 1994 by H. Peter Anvin as a way to allow creation of Linux boot floppies without requiring Linux-specific tools. Until that point Linux boot floppies, universally required to install Linux on the PC platform, had been distributed as raw images to be written to a bare floppy disk. On an MS-DOS machine this meant using a tool called RAWRITE. The resulting floppy was an opaque object and was considered unformatted by most non-Linux operating systems.

The SYSLINUX installer (named by analogy to the MS-DOS SYS command) ran directly under MS-DOS and all data was stored in conventional files on an MS-DOS FAT filesystem. Since it was designed for running on floppies it had to be small (the 18K overhead from the FAT

filesystem itself was a point of criticism in the early days): as of version 1.30 (November 3, 1996) the SYSLINUX binary was 4.5K in size. Even so, it contained a reasonably flexible configuration system and support for displaying online help; the latter was particularly important for install disks.

In 1999 Chris DiBona, then of VA Linux Systems, provided an early PXE-equipped system as a development platform for a PXE loader. Since the Intel PXE specification at the time specified that only 32K was available to the NBP it was decided that basing the PXE loader code on SYSLINUX—an existing, compact, configurable loader—would make sense. Thus, SYSLINUX 1.46, released September 17, 1999, included PXELINUX, a PXE network bootstrap program with a SYSLINUX-based user interface. Subsequently SYSLINUX acquired support for other media, specifically ISO 9660 CD-ROMs in El Torito “native mode” and hard disks with standard Linux ext2/ext3 filesystems.

As support for CD-ROM booting—and later, USB booting—on PCs became more universal, pressure to keep code size minimal waned since storage capacities were less restrictive. At the same time network administrators in particular requested a more configurable user interface. To avoid burdening the SYSLINUX core (written in assembly language and challenging to maintain) with additional user interface features, an API was developed to allow user interfaces to be implemented as independent, loadable modules. The first such interface was a very sophisticated system written by Murali Krishna Ganapathy, based on a text-mode windowing interface. Though very advanced and capable of almost



Figure 2: The SYSLINUX simple menu system

infinite customization, it turned out to be too difficult for most users to configure. To address this issue, the “simple menu system” (Figure 2) was implemented and is now used by most SYSLINUX users.

The API also allows support for new binary formats to be written as well as “decision modules” (boot selection based on non-user input, such as hardware detection). An 88,000-line library, derived from `klibc`, is available to developers to make module development easier and as similar to standard applications-level C programming as possible.

Historically SYSLINUX has focused on the PC BIOS platform, but as the bulk of the code has been migrated from the core into modules and from assembly language into C, the feature set of the core has become bounded. The intent is for the core to become a “microkernel” with all essential functionality in (possibly integrated) modules; this would permit the core to be rewritten to support other platforms such as EFI.

PXELINUX has not, however, implemented any protocols other than TFTP. The PXE APIs only permit access at one of three levels: TFTP file download, UDP, or raw link layer frames. No method to access the firmware stack at the IP layer is provided. This means that to support TCP-based protocols, such as HTTP, a full replacement IP stack is required.

It is worth noting that although a large number of people have contributed to SYSLINUX over the years, it has largely remained a one-person project. As of this writing there is a serious effort underway to grow the SYSLINUX project developer base. To facilitate this process

the SYSLINUX project will participate in Google Summer of Code for the first time in 2008.

5 The Etherboot Project

In 1995 Markus Gutschke ported a network bootloader, Netboot, from FreeBSD. Netboot followed Jamie Honan’s 1993 “Net Boot Image Proposal.” Since the first OS targeted for loading was Linux, `mknbi` was used to combine kernel and `initrd` images into a single NBI file before loading.

Since Netboot did not support his network card and Netboot drivers at the time had to be written in assembly language, Markus implemented a new driver interface allowing drivers to be written in C. He called his code Etherboot.

Markus released Etherboot 1.0 in 1995 and it proved to be popular enough that a small community called the “Etherboot Project” [2] formed to support and improve it with additional functionality and drivers. In late 1996 one of the group’s more active contributors, Ken Yap, took over leadership of the project.

In 1997, when Intel published the PXE specification, Ken began work on NILO, a first attempt at an Open Source PXE implementation. In 1998 Rob Savoye, with funding from NLnet Foundation, took over NILO development. For various reasons the project was unsuccessful, and development of NILO officially ceased in 2000 [7].

Etherboot development continued, however, and in 1999 Marty Connor became involved with the project, having discovered it through conversation with Jim McQuillan of LTSP (Linux Terminal Server Project). Marty ported several of Donald Becker’s Linux NIC drivers to Etherboot to provide support in Etherboot for popular cards of the day.

In 2000, Marty created `rom-o-matic.net` [8], a web-based Etherboot image generator that created customized Etherboot images on demand. This made it much easier for people to create and test Etherboot because no specific build environment or command line expertise was required. Usage and testing of Etherboot increased dramatically.

Another boost to Etherboot use came in 2001 when the Etherboot Project first exhibited in the .ORG Pavilion

at the IDG LinuxWorld Expo and invited LTSP to share their booth. Live demos of Etherboot network booting and LTSP thin clients sparked the interest of many potential users.

In 2002 Michael Brown first encountered Etherboot while trying to find a solution for booting wireless thin clients. He developed and submitted an Etherboot driver to support Prism II-based wireless cards, and became a regular contributor to the project.

About this time Marty became concerned that PXE was fast becoming a de facto standard for network booting since it was being included in a significant number of motherboards and mid-to-high-end NICs. Although there was strong opposition within the project to supporting PXE for technical reasons, he felt that unless Etherboot supported the PXE specification Etherboot would quickly become irrelevant for most users.

Added incentive to support PXE came in 2004 when Marty and H. Peter Anvin spoke about creating a complete, compliant Open Source PXE implementation to support PXELINUX. Later in 2004 Michael added partial PXE support to Etherboot, which was then capable of supporting PXELINUX though it lacked full PXE functionality.

In 2005 Marty and Michael created gPXE, a major rewrite of Etherboot with PXE compatibility as a key design goal. Soon after, Marty became the third Etherboot Project Leader and Michael became the project's Lead Developer. Primary development energy was then redirected from Etherboot to gPXE.

In 2006 Michael, with help from Google Summer of Code student Nikhil C. Rao, added more robust and compliant TCP support to gPXE. This enabled Michael to add TCP-based protocols such as iSCSI, which in turn allowed gPXE to network-boot exotic operating systems such as Windows Server 2003.

In 2007 the Etherboot Project exhibited in the LinuxWorld Expo .ORG Pavilion for the 12th time, this time demonstrating gPXE booting of various OSes via HTTP, iSCSI, AoE, and other protocols. Michael and Peter created, coded, and demonstrated a first API for gPXE to PXELINUX integration.

As of 2008 rom-o-matic.net had generated over two million Etherboot and gPXE images, with a typical size of 40K for a ROM image containing support for

DHCP, DNS, TFTP, HTTP, iSCSI, AoE, and multiple image formats including PXE, bzImage, Multiboot, and gPXE scripts.

A large number of people have generously contributed to the success of the Etherboot Project over the years. Many of their names can be found on the project's acknowledgments web page [9]. There are also many users who contribute on the project's mailing lists and IRC channel. Their help with documentation, testing, and support greatly contributes to the quality and popularity of Etherboot and gPXE.

6 The Strengths of Each Project

Given the primary focuses of the projects it is not surprising that each brings different strengths to the collaboration. gPXE supports a wide range of protocols, can be integrated in ROM rather than relying on shipped firmware, and supports other architectures; however, its user interface is limited. PXELINUX has advanced user interfaces and, because it is a part of the SYSLINUX suite, has cross-media support, but its protocol support is limited to TFTP.

Within the PXE concept model PXELINUX strictly acts as the NBP, whereas gPXE can act either as NBP, BC, or BC and UNDI combined depending on how it is configured and compiled. gPXE configured to function as BC and UNDI is most common when it is used in ROM or loaded from disk.

gPXE is also able to be loaded from a server as an NBP and then take over the functions of either the BC only, or the BC and UNDI combined, and then load another NBP such as PXELINUX to perform a target OS load. This configuration, referred to as "chainloading," can be used either to substitute functionality from a partially working PXE stack or to get the enhanced capabilities of gPXE, either way without having to actually modify the ROM on the device.

7 Choosing a Strategy for Collaboration

Collaborative projects carry significant risks and rewards over single-team development. Because of this, potential costs and benefits should to be considered carefully before embarking on such a journey.

Rather than seeking to collaborate with the Etherboot Project, the SYSLINUX project could have implemented its own TCP/IP stack, HTTP client, and iSCSI and AoE initiators for PXELINUX. Alternatively, it could have re-used the code from gPXE or used another Open Source TCP/IP stack, such as lwIP [10].

Collaboration, though requiring some additional development effort, had several potential advantages:

- Using gPXE’s protocol support would mean that SYSLINUX maintainers would not have to integrate and support additional code to support new protocols.
- Code improvements to either project could be of benefit to users of both projects.
- Users of both gPXE and PXELINUX could share a single user interface for accessing features.

In light of these potential advantages, the developers of both projects decided to explore ways of working together.

Popular strategies for collaboration between Open Source projects differ primarily based on whether it is the intention of one project to take over maintenance of code produced by another project or whether the projects intend to maintain separate code bases which interoperate based on well-defined interfaces.

Some common strategies for collaboration between projects are:

- *Componentization*, where one project’s code and development team simply becomes part of another project. The second project then ceases to exist as an independent project.
- *Aggregation*, where one project includes the other’s code as a component, possibly in modified form, but the second project’s code continues to be developed as a separately maintained project. In this model, the first project can be considered a consumer of the second project. This is particularly common with application programs that depend on libraries that are not widely available.

- *Cooperation*, where the two projects mutually agree on a set of APIs and independently implement their respective parts. The projects are maintained separately, and aggregation into a combined product is performed by the distributor or end user.
- *Stacking*, in which one project independently defines an interface available to all potential users of the code, which is completely sufficient (without modification) for the needs of the second project. In this case, the combination is strictly a case of the second project being a consumer of the first, and final aggregation is typically performed by the distributor or end user; this strategy is typified by widely used libraries.

Each of these strategies has advantages and pitfalls, based on the nature of the projects, the development teams, and any corporate entities that may be involved. The tradeoffs between these strategies can be quite different in the Open Source world over what they might be in analogous corporate environments.

8 Integration, so Far

Initial steps toward integrating gPXE and PXELINUX were taken in 2004 when Etherboot first became capable of acting as a PXE ROM (BC and UNDI combined). This allowed Etherboot to replace defective vendor PXE implementations, either by replacing the ROM or by chainloading, but did not provide any additional capabilities. Nevertheless, this approach has been widely used, especially with SiS900 series NICs, a briefly popular NIC with a notoriously buggy vendor PXE stack.

PXELINUX users had been requesting additional protocol support for quite some time, especially the ability to download via HTTP. Not only is HTTP a TCP-based protocol, faster and more reliable than TFTP (based on UDP), but HTTP servers have better support for dynamic content, which is frequently desired for generating configuration files.

At LinuxWorld Expo San Francisco in 2006, SYSLINUX and Etherboot Project developers met to discuss the situation. At that meeting, the following constraints were established:

- The primary focus of gPXE is as ROM firmware. The continued utility of gPXE in ROM must be maintained.

- Extended protocol support must work in PXE-LINUX when it is loaded from a vendor PXE stack. Supporting extended protocols only with gPXE in ROM is not acceptable.
- Although gPXE already had support for extended protocols by accepting a URL via the PXE API's TFTP (PXENV_TFTP_OPEN) routine, the PXE TFTP interface is inadequate for PXELINUX; a new API is necessary for PXELINUX to access functionality beyond what standard PXE APIs permit.

In electronic discussions afterwards, the following high-level plan was agreed to by both development teams:

- An extended PXE API for PXELINUX will be developed.
- A technique will be developed to aggregate gPXE and PXELINUX into a single binary to simplify deployment in existing PXE environments.

Unfortunately around this time both projects became distracted by other priorities: the Etherboot Project focused on providing new and improved network drivers, support for SAN protocols (iSCSI and AoE), and completing an initial gPXE release; the SYSLINUX project on user interface and module API improvements. A test version without the aggregate binary (and certainly not stable enough to be deployed in a real environment) was demonstrated at IDG LinuxWorld Expo 2007, but after that the collaboration languished for months.

Toward the end of 2007 improved protocol support was becoming a high priority for the SYSLINUX project, while Etherboot developers were pushing toward a mid-February initial release (gPXE 0.9.3). Over the holidays developers from both projects conducted a sizable joint development and debugging effort, implementing binary encapsulation support and tracking down a number of issues that occurred when gPXE was chainloaded from a network server, as opposed to running from ROM. Having more developers testing the code helped find bugs that only manifested on certain hardware and particular PXE implementations. Fortunately (for some meaning thereof), the combined team had access to a large and eclectic collection of troublesome hardware.

After the initial beta release of gPXE 0.9.3 on February 14, 2008, the original plan was reviewed by both development teams. As there had been significant changes in both code bases, the plan was revised as follows:

```

PXE 0.9.3 -- Open Source Boot Firmware -- http://etherboot.org
Features: HTTP DNS TFTP iSCSI AoE bzImage Multiboot PXE PXEXT
net0: 52:54:00:12:34:56 on UNDI (open) TX:0 RX:0 ERE:0
DHCP (net0 52:54:00:12:34:56)... ok
net0: 172.27.0.120/255.255.254.0 gw 172.27.0.1
Embedded Image: 15162 bytes at 0xfadc

PXELINK 3.70 0x12eacca Copyright (C) 1994-2008 H. Peter Anvin
UNDI data segment at: 00098110
UNDI data segment size: 2E88
UNDI code segment at: 00099C40
UNDI code segment size: 05B8
PXE entry point found (we hope) at 9AC4:022D
Getting cached packet @1
Getting cached packet @2
Getting cached packet @3
IP address seems to be 0C1B0078 172.27.0.120
ip=172.27.0.120:172.27.0.17:172.27.0.1:255.255.254.0
TFTP prefix: http://raidtest.hns.anvin.org/tftpboot/
Trying to load: pxelinux.cfg/01-52-54-00-12-34-56
*** test configuration file ***
Press F1 for help, or ENTER to boot: -

```

Figure 3: gpxelinux.0 loading its config file via HTTP

- An initial implementation will be based on the already implemented extended PXE API, plus any additions necessary.
- This API will be considered private and not guaranteed to be stable between revisions. Thus, the only supported use will be between gPXE and an embedded PXELINUX; if gPXE is used in ROM it should still chain-load the combined image.
- As SYSLINUX code has increasingly moved toward having most of its code in modules, with a well-defined application binary interface (ABI), the projects will eventually migrate to a model where gPXE implements the SYSLINUX module ABI directly; at that time the private API will be deprecated.

The third item on this list was accepted as a Google Summer of Code project under Etherboot Project mentorship for 2008.

More powerful functionality is possible when gPXE is also used in ROM (or provided on CD-ROM, USB stick, or floppy). gPXE can either be used as the native PXE stack on the system using its own network device drivers, or it can use the UNDI driver from the vendor PXE stack. With suitable configuration gPXE can download an initial NBP image from a specific URL set at compile time or saved in nonvolatile storage. This capability can be used to invoke a service facility with a very small investment in ROM; the only local network resources required are working DHCP and DNS. If the downloaded image is gpxelinux.0, a full range of PXELINUX modular functionality becomes available.

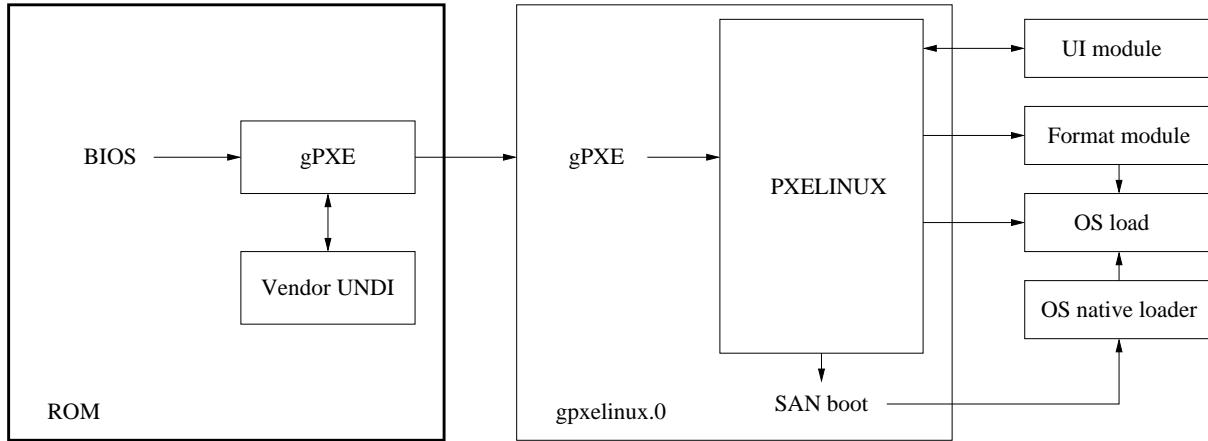


Figure 4: gPXE loading `gpxelinux.0`, using a vendor UNDI driver

Ultimately SYSLINUX and the Etherboot Project decided not to combine code or development teams but rather to modify both of their code bases to support a jointly developed API (*a cooperation*). However, to make it easier for end users, the SYSLINUX distribution now contains a snapshot of gPXE sources so that the combined image can be built from a single download; in this sense, it is also an *aggregation*. However, the intent or the projects to emphasize the cooperative aspects of supporting a common API and to have the SYSLINUX combined source code tree have minimal differences from the primary gPXE distribution.

It is the desire of both projects that this will permit each project to retain its particular focus and identity, while giving end users access to functionally contained in both code bases.

As of April 15, 2008, the combined product is available in beta form as part of SYSLINUX 3.70-pre9. This distribution includes a slightly modified snapshot of the gPXE git repository (containing a few changes necessary for the usage model, but in need of cleanup before being fed upstream into the gPXE tree). When built, this produces `gpxelinux.0`, a combined binary, in addition to the conventional `pxelinux.0`. If loaded from a standard vendor PXE stack, `gpxelinux.0` can be redirected to non-TFTP protocols via the PXELINUX Path Prefix DHCP option [11] or via explicit URL syntax in the configuration file. A DHCP option, in particular, allows even the PXELINUX configuration file to be acquired through non-TFTP protocols such as HTTP, making it much easier to generate configuration files dynamically.

As of the SYSLINUX 3.70-pre9 release, `gpxelinux.0` is not yet a drop-in replacement for `pxelinux.0` in all situations because some issues relating to chainloading another NBP remain. It is expected that these issues will be relatively easy to resolve.

9 Next Steps

At the time of this writing the primary collaborative development focus of the projects is to resolve a few remaining interoperability issues and to clean up SYSLINUX-local modifications to gPXE so that they may be integrated into the official gPXE source base.

The combined `gpxelinux.0` image using the current approach is expected to be released with SYSLINUX 3.70. Over the summer of 2008 considerable progress on implementing the SYSLINUX module API in gPXE will hopefully be made. This effort will also serve as a trailblazing project for the “microkernelized” rewrite of the SYSLINUX core across all media.

10 Lessons Learned Along the Way

When integrating Open Source projects, especially ones developed outside the influence of a corporate or sponsorship structure, one must consider at least the following pitfalls and concerns:

- *Motivation:* For collaboration between two projects to succeed it is important that there be incentives for both of their user communities and de-

velopment teams. Without a shared sense of purpose, enthusiasm for the project may quickly wane on the part of one or both projects.

In the case of the SYSLINUX-Etherboot collaboration, both projects recognized the opportunity to leverage each others work and both were motivated to explore how they might productively work together. Understanding the motivations of other project participants was an important part of keeping the collaboration moving forward.

- *Focus:* The primary reason for combining two projects is that each brings different strengths to the table. It is likely that each project has development goals aimed toward improving its respective strengths. A goal related to facilitating code combination might therefore be relatively low on the priority of either one or *both* the parent projects! A good working relationship is likely to improve joint focus, but other driving forces may still prevail, such as funding issues.

Focus differences were a significant issue early in the SYSLINUX-Etherboot collaboration. Rather than completely executing the original project plan, each project ended up working more on other priorities, especially SAN support for gPXE and improved user interfaces for SYSLINUX. Not until late 2007 discussions did both projects agree on the priority of the joint development effort and commit to the shared goal of producing a test release in the March 2008 timeframe.

- *Culture:* Every Open Source project has a unique culture that generally depends on the preferences of the original or principal developers or administrators. Just as in a corporate collaboration or merger, culture clashes can manifest themselves as subtle but significant roadblocks to progress. In Open Source projects, such issues may include frequency and style of developer communication, review and commit policies, and coding style. In large projects, these processes are often more formalized than in smaller projects. Nevertheless, it is important to recognize, respect, and address differences between collaborative partners as they can significantly affect the success of the joint effort.

Whereas the SYSLINUX project has a single central maintainer responsible for all technical direction, Etherboot Project decision making is somewhat more distributed. This difference complicated

some early discussions until it became clear that for actionable agreement to be achieved, all relevant Etherboot Project team members needed to be included in technical discussions.

- *Credit where credit is due:* These days many, if not most Open Source software developers are deriving their income either directly or indirectly from Open Source work. Others, such as students, may be concerned about future marketability. Still others may consider recognition a major driver for their Open Source involvement. Accordingly, *recognition is very valuable currency in the Open Source world*. A perception, true or not, that one project is trying to usurp credit for another project's work is likely to create ill will and poor relations.

Discussions of credit can be sensitive, as some people may feel their concerns aren't appropriate or valid. In the particular case of the SYSLINUX-Etherboot Project collaboration, both sides had concerns, but some went unvoiced for a long time. Although both sides had good intentions, these unresolved concerns slowed the collaboration considerably, until they were discussed and addressed as legitimate issues.

By recognizing that with the best intentions such issues can and do occur—even in a collaboration involving relatively small projects—one can significantly improve the chances for a successful and timely joint project.

Learning to work together benefited the code bases and development teams of both projects. Code changes needed to support jointly developed interfaces required optimization and auditing of critical code sections. In addition, communication, confidence, and trust between developers significantly improved during the process of working together to achieve a shared goal.

References

- [1] SYSLINUX project web site,
<http://syslinux.zytor.com/>
- [2] Etherboot Project web site,
<http://www.etherboot.org/>
- [3] C. Stevens and S. Merkin, “*El Torito*” Bootable CD-ROM Format Specification, Version 1.0,

January 25, 1995,

<http://www.phoenix.com/NR/rdonlyres/98D3219C-9CC9-4DF5-B496-A286D893E36A/0/specscdrom.pdf> or
<http://tinyurl.com/99c5f>

- [4] J. Honan and G. Kuhlmann, “Draft Net Boot Image Proposal,” Version 0.3, June 1997, <http://www.nilo.org/docs/netboot.html>
- [5] Intel Corporation *et al.*, *Network PC System Design Guidelines*, Version 1.0b, August 5, 1997, <http://www.intel.com/design/archives/wfm/>
- [6] Intel Corporation, *Preboot Execution Environment (PXE) Specification*, Version 2.1, September 20, 1999, <http://download.intel.com/design/archives/wfm/downloads/pxespec.pdf>
- [7] K. Yap, “NILO; organization and status,” May 2000, <http://www.nlnet.nl/project/nilo/how.html>
- [8] Etherboot project, [rom-o-matic.net](http://www.rom-o-matic.net/),
- [9] Etherboot project, Acknowledgements, <http://etherboot.org/wiki/acknowledgements>
- [10] A. Dunkels *et al.*, *The lwIP TCP/IP Stack*, <http://lwip.scribblewiki.com/>
- [11] D. Hankins, *Dynamic Host Configuration Protocol Options Used by PXELINUX* (RFC 5071), December 2007, <http://www.ietf.org/rfc/rfc5071.txt>

Keeping the Linux Kernel Honest

Testing Kernel.org kernels

Kamalesh Babulal

IBM

kamalesh@linux.vnet.ibm.com

Balbir Singh

IBM

balbir@linux.vnet.ibm.com

Abstract

The LinuxTM Kernel release cycle has been short with various intermediate releases and with the lack of a separate kernel development tree. There have been many challenges with this rapid development such as early bug reporting, regression tracking, functional/performance testing, and test coverage by different individuals and projects. Many kernel developers/testers have been working to keep the quality of the kernel high, by testing as many possible subsystems as they can.

In this paper, we present our kernel testing methodology, infrastructure, and results used for the v2.6 kernels. We summarize the bug reporting statistics based on the different kernel subsystems, trends, and observations. We will also present code coverage analysis by subsystem for different test suites.

1 Introduction

The Linux kernel has been growing with every release as a result of the sheer number of new features being merged. These changes are being released at a very high rate, with each release containing a very large number of changes and new features. The time latency at which these changes are made is very short between every release cycle. All of this is occurring across many disparate hardware architectures.

This presents unique problems for a tester who must take all of the following into account:

- Lines of code added,
- Time interval between each release,
- Number of intermediate releases,

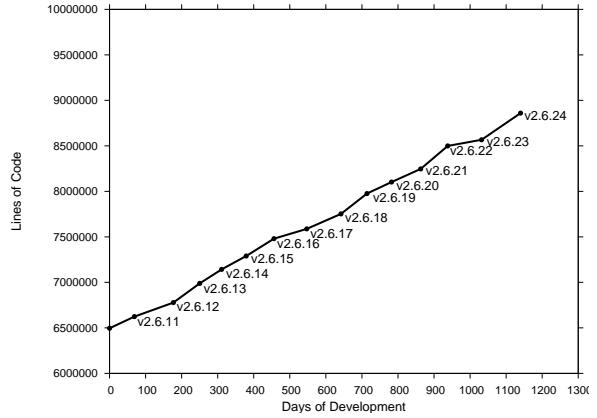


Figure 1: Size of the kernel with every release

- Testing on wide range of architectures and platforms,
- Regressions carried from previous release,
- Different development trees, and
- Various configurations and boot options to be tested.

In Section 2 we briefly explain the challenges. Section 3 outlines our methodology to meet the challenges. Section 4 presents the results of code coverage analysis, including the fault injection coverage results for the standard test cases and we discuss our future plans in Section 5.

2 Challenges

2.1 Different Trees

In the past, Linux kernel development had separate development and stable trees. The stable trees had an even number and the development trees had an odd number

as their second release number. As an example 2.5 was development release, while 2.4 was stable release. With the 2.6 kernel development, there are no separate *development* and *stable* trees. All the development is based upon the current stable tree and once the current development tree is marked as stable, it is released as the next mainline release.

There are intermediate development releases between major releases of a 2.6 kernels, each having its own significance. Figure 2 shows the different development trees and the flow of patches from development trees to the mainline kernel.

The -stable tree contains critical fixes for security problems or significant regressions identified for the mainline tree it is based upon. Once the mainline tree is released, the developers start contributing to the new features to be included in the next release. The new features are accepted for the inclusion during the first two weeks of development following the mainline release. After two weeks the merge window closes and the kernel is in feature freeze; no further features will be accepted into this release. This is released as -rc1. After -rc1 the features are tested well and stabilized. During this period roughly weekly release candidates, -rc, are produced, as well as intermediate snapshots of Linus's git tree.

The `-rc` releases are a set of patches for the bug fixes and other important security fixes, based on the previous `-rc` release. For example, 2.6.25-rc3 will have the fixes for the bugs identified in 2.6.25-rc2.

In addition to the mainline releases, we have a number of testing trees for less stable features, as well as sub-system specific trees. The `-mm` tree has experimental patches and critical fixes that are planned to be pushed to the mainline kernel. For a new feature it is recommended that it be tested in `-mm`, since that tree undergoes rigorous testing, which in-turn helps in stabilizing the feature. `-mm` is rebased often to development releases to test the patch set against the development tree.

The `-next` release was introduced with the 2.6.24 development series. The `-next` tree has the patch changesets from different maintainers, intended to be merged into the next release. Changesets are rebased to the current development tree, which helps to resolve the merge conflicts and bugs before they get introduced in the next release. Resolving the conflicts and bugs on

a regular basis would allow the development and stable releases to be based on the -next tree in the future (refer to [5] for more information on kernel trees).

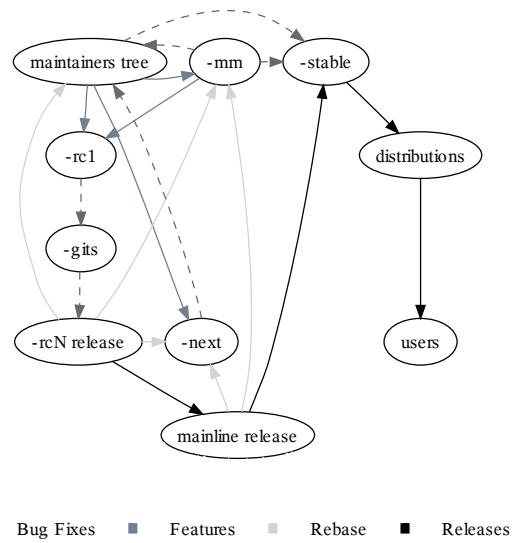


Figure 2: Linux Kernel Development Cycle

2.2 Release Frequency

Testing is essential to deliver high quality software and testing every phase of development is important if we are to catch bugs early. This is especially true for projects like Linux where the source is growing by 2.32% [1] with every release, as represented in Figure 1. These numbers are pretty high for any project. Starting from 2.6 kernel development series, there is no separate stable and development tree, which means that more code is being added and it needs to be tested thoroughly.

Bugs caught and fixed early in the cycle helps to maintain the kernel quality by:

- Providing a pleasant user experience,
 - Avoiding building on top of buggy code, and
 - Easier debugging, since the code is fresh in the authors mind.

The stable kernel releases are made approximately every 2-3 months, during which time an average of 3.18

versions	.20	.21	.22	.23	.24	Total
stable	22	8	20	18	5	73
stable-git	16	18	18	26	22	100
stable-mm	2	2	0	1	1	4
stable-mm +hotfixe(s)	0	0	0	3	6	9
rc	7	7	11	8	8	41
rc-git	59	52	57	70	47	285
rc-mm	12	6	10	7	4	41
rc-mm +hotfixe(s)	0	10	18	14	3	45
next	0	0	0	0	35	35
Total	118	103	134	147	131	633

Table 1: Summary of releases between kernel versions

changes are being accepted every hour. This has been the trend for the past $1\frac{1}{2}$ years.

Table 1 shows the data of various intermediate releases made for past $1\frac{1}{2}$ years. These incremental changes to the kernel source ensure that any code changes made to the kernel are well tested before they accepted into mainline.

With the average of 126 kernels (refer to Table 1) being released between two sequential major releases, we have at least one kernel per day being released. Developers end up limiting their testing to unit testing for the changes they make and start concentrating on new development. It is not practical for most of the developers to test their changes across all architectures supported by Linux. A significant amount of testing is being done by others from the community, which includes testers, developers, and vendors (including distribution companies).

2.3 Test Case Availability

There is a significant testing effort in the community by various individual testers, developers, distribution companies, and Independent Software Vendors. They contribute to the effort with combinations of testing methods such as:

- Compile and Boot test (including the cross compiler tests),

- Regression Testing,
- Stress Testing,
- Performance Testing, and
- Functional Testing.

These efforts are not being captured completely because most of these efforts are not visible. We will not be able to account for any testing done unless it is being published or shared. Many features are accepted into the mainline after stringent code reviews and ample testing in the development releases. But not many developers provide the test cases or guidelines to test their functionality even though it is in the interest of developers to keep the quality of their code high.

Over the years there have been many test projects and suites with a primary focus to improve the quality of Linux. They are constantly undergoing lots of changes between every release. They have been adding new test cases to test the new features getting merged into the kernel, *but the updating is not fast enough to catch those early bugs and for some features we do not have test cases available.*

Any person who is interested in testing the kernel has no single, nor even a small number, of test projects which can cover most of the kernel subsystems. We do have a large number of test projects, but each is independent requiring installation and configuration; the test setup is a huge effort for a tester. Not all the testers have the harness infrastructure in place for testing due to the limitations of the hardware they own.

Most of the test projects act at best as regression, stress, performance test suites, or combinations thereof, but we do not have tests which can be used for functional testing of the new features being merged into the kernel. It is very critical for new code to be tested thoroughly, even before it gets merged in to the mainline.

In the existing scenario there are many valuable test cases/scripts available from individuals, which could expose bugs on other environments untested by them. *Sharing these test cases and scripts with the community* through one of the test projects will help in improving the testing much more by:

- Enabling the code to be tested on a variety of hardware,

- Improving the test coverage on executing all possible code paths,
- Avoiding duplication of test case development, and
- Making reproduction of bugs easier.

2.4 Kernel options

The Linux kernel is highly configurable, which allows specific features to be enabled as required. This means that to test the kernel fully we would have to test with each combination of these options. Testing kernels compiled only with the best known¹ configurations cannot expose any bugs hidden under the untried combinations. As an example the kernel can be configured to use any one of the following different memory models:

```
CONFIG_FLATMEM
CONFIG_DISCONTIGMEM
CONFIG_SPARSEMEM
CONFIG_SPARSEMEM_EXTREME
CONFIG_SPARSEMEM_VMEMMAP
```

They are mutually exclusive, which means the kernel can be compiled with only one of these options. Testing all combinations of memory models would require five different build and test cycles. Some of the combinations should be tested to improve the testing coverage of the kernel.

Many of the new features getting merged into the kernel are not tested by all individuals because their existence is not known to the tester. The example above shows how quickly permutations and combinations can grow. Usually what gets tested is the defaults on each architecture and the defaults that depend on the machine (testers do not deviate from a configuration that works for them).

It is important to test the responsiveness of the kernel with different boot options (refer to [10] for more available kernel parameters). For example, booting with less memory by passing `mem=<less memory>` as a boot parameter could test kernel behaviour when booted on a system with less memory. An extensive testing of this kind could take lots of kernel testing cycles, with total number of test combinations = number of boot parameter combinations × number of kernel configurations × number of releases.

¹the configuration with which the kernel builds and boots without any issues

2.5 Existing Test Projects

There are many existing test projects used by the individual test contributors. We summarize some of them along with their key features:

LTP (Linux Test Project)² is a Regression/Functional test suite. It contains 3000+ test cases to test the basic functionality of the kernel. It is capable of testing/stressing filesystem, memory, scheduler, disk I/O, network, and system calls. It also provides some additional test suites such as pounder, kdump, open-hpi, open-posix, code coverage, and others.

It is ideal for running the basic functionality verification, with sufficient stress generated from the test cases. LTP does not support the kernel build test. LTP results can be formatted as HTML pages. It lacks the support for machine parseable logs. The test case results are either PASS or FAIL, which makes it complex for a tester to understand the reason behind test failure.

IBM autobench is a client harness project which supports setting up the test execution environment, execution of the test suites, and capturing the logs with environmental statics. It supports a kernel build test along with support for profiling. It is capable of executing test cases in parallel. The job control support is basic, allowing user to have minimal control over the way the tests are executed. The tool is written using bash/perl scripts.

Autotest³ is an open source based client/server harness capable of running as a standalone client or is easily plugged into an existing server harness. Test cases included are capable of regression, functionality, stress, performance, kernel build tests, and they support various profilers. Autotest is written in python which allows the user to take more control of job execution by including python syntax in the job control file. Its object oriented and has a cleaner design.

Autotest has built-in error handling support. The logs are machine parseable with consistent exit status of the test executed as well as providing a descriptive message of the status. Parse⁴ is built into the server harness. It summarizes the job execution results from different

²<http://ltp.sourceforge.net/>

³<http://test.kernel.org/autotest/>

⁴Parser used by the autotest to parse the test results <http://test.kernel.org/autotest/Parse>

testers⁵ and formats them in query-able fashion for the tester to interpret them better.

3 How is the kernel being tested?

3.1 Methodology

Release early, release often[7] is the Linux kernel development philosophy. The development branches are released very frequently. As an example we have two -git releases per day, typically one -next release along with regular -rc and -mm releases (Different development releases have been explained in section 2.1). Testing development releases earlier and more often helps in identifying the patches that break the kernel. This allows fixing them earlier in the cycle. Before merging the patches, they should have been tested across all supported architectures, but it is not practical to expect all developers to test their patches that widely before merging.

Ideally we do the **Build test** on all releases on various hardware⁶, with different configuration options. Build test is focused on build errors and warnings while building the kernel. This is followed by the **Regression test** suite. This helps uncover bugs introduced as side effects of new kernel changes. In order to ensure a regression free kernel, the suite is bundled with test suites from different test projects to test the filesystems, disk I/O, memory, scheduler, IPC, commands functional verification, and system calls with little stress.

The more thoroughly the development releases are tested, the better the Linux kernel quality is. Testing thoroughly requires two or more machine days based on the tests run on the releases. We selectively pick up development releases for a complete round of testing. Testing all releases with more than just build and regression tests would take too long and the important releases would be tested much later, after the release.

We do build and regression testing on all of the Linux kernel releases, which includes -stable, -rc, -git, -mm, -next. The focus is on certain development releases, which are tested more with stress tests and functionality tests along with some profile information extraction. As an example we focus more

on -majorrelease, -rc1, and -mm releases. The debug options are tested on the major releases. -mm is one of the important development releases with bleeding edge features incorporated in it, so we test rounds of -mm + hotfixes if available. The Filesystem stress tests are executed over:

- Ext2/3 Filesystem
- Reiserfs Filesystem
- XFS Filesystem
- JFS Filesystem
- CIFS Filesystem
- NFS3/4 Filesystem

Comparing kernel performance under certain workloads on the machine to historical measures verifies the performance improvement or degradation. **Performance Testing** results are captured for almost all of the kernel releases. Results of workloads such as dbench, kernbench, and tbench are captured on the same machines, consistently validating the performance with every kernel release.

3.2 Infrastructure

Human hours are costlier in comparison to machine hours. Given the frequency of kernel releases, machine hours can be better used for setting up test environments and execution. Human hours can be best utilized in analyzing test results and debugging any bugs found. Figure 3 explains how the infrastructure works.

Mirror/Trigger: kernels are rsync'd to the local mirror, within a few minutes of the releases. Once the mirroring is complete, it acts as a trigger to test the newly downloaded kernel image. The trigger is initiated by any of the kernel releases mentioned in Section 2.1.

Test Selection/Job Queues: based on the kernel release, the predefined test cases are queued to the server for test execution. Section 3.1 explains the selection criteria of different predefined set of test cases to be queued, based upon the kernel release.

IBM's ABAT⁷ server schedules the queued jobs from users based upon the availability of machine and does

⁵TKO (test.kernel.org) database is used to populate the results

⁶we cover the x86, Power™ and s390x architectures

⁷Automated Build And Test

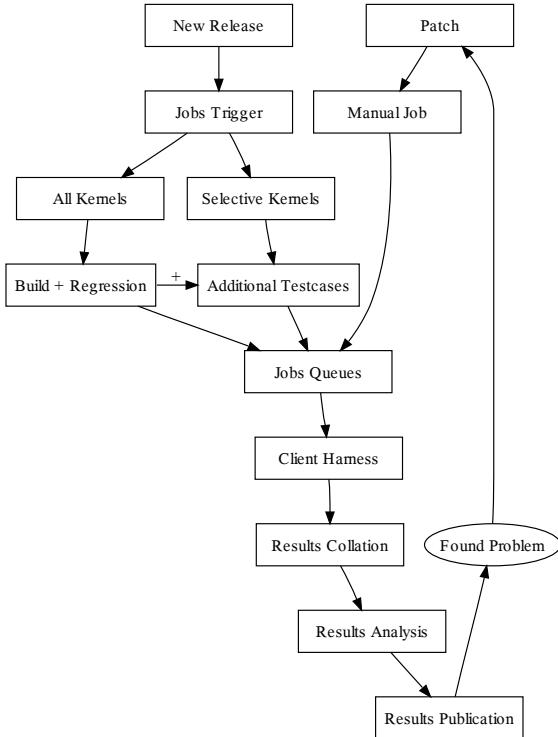


Figure 3: Infrastructure of kernel.org testing

more than just a simple queuing system. ABAT test framework is not open source, but the results are published to the community through <http://test.kernel.org>⁸

Client harness: both IBM autobench and autotest are supported as the client harness tools though their control file syntax is different. The client starts the test execution reading the job control file that is passed over when the job gets scheduled. It is responsible for building the appropriate kernel, running the tests, capturing the logs and other information, and making the results available. Section 2.5 explains more about these clients.

Results Collation: results are gathered asynchronously as the jobs complete and are pushed to test.kernel.org. They are grouped relevantly by TKO before publishing them. Kernel binaries and other system information dumps are stripped off the results. Results published at TKO are of a standard set of test cases that are used for testing.

⁸In collaboration between development and test team at IBM Linux Technology Center

Results Analysis: tests produce large amount of logs and other information. Analyzing the test information collected is time consuming. Relevant information is extracted and displayed as status using colour combinations each representing the percentage of test cases completed successfully. Results can be viewed in different layouts.⁹ Performance data is analysed on selective benchmarks to provide historical performance graphs.

Results Publication: after automated analysis, the results are made available on the TKO website. Human monitoring is needed to take action on test failures or performance regressions. The problems are reported to the community (mostly via email) with the links to the test results.

Found Problem: when a test failure or performance regression is noticed, it is reported back to the community (mostly via email) by the person monitoring the results. Depending on the kernel release, another round of jobs are queued with the additional patches received for the problem reported. Currently only IBM engineers can manually submit the jobs on ABAT though the results and performance graphs are published as is done for regular jobs.

4 Results Analysis

The test procedure attempts to execute as many as possible code paths in the Linux kernel using different test projects. When combined together, the various tests tend to cover most of the kernel, but there has always been a gap between *tested and untested code*. Code coverage helps us to quantify this gap.

In this section we compare the code coverage results of 2.6.20, 2.6.21, 2.6.22, 2.6.23 and 2.6.24 for x86 and PowerTM architecture. We also look at the results of executing some of these tests using the fault injection framework.

4.1 Code Coverage Setup

The gcov kernel patch¹⁰ and lcov package¹¹ from LTP were used for the code coverage. Table 2 shows

⁹user selects the row and column heads. Condition based views are available.

¹⁰<http://ltp.sourceforge.net/coverage/gcov.php>

¹¹<http://ltp.sourceforge.net/coverage/lcov.php>

Benchmarks	Description
runltp	A collection of tools for testing the Linux kernel and related features.
dbench	Filesystem benchmark that generates good filesystem load.
aio-stress	Filesystem benchmark that generates asynchronous I/O stress load.
aio-cp	Testing tool that copies files by using async I/O state machine.
hackbench	A benchmark for measuring the performance, overhead, and scalability of the Linux scheduler.
vmmstress	Performs general stress with memory race conditions between simultaneous read fault write fault, copy on write (COW) fault.
kernbench	A CPU throughput benchmark. It is designed to compare kernels on the same machine, or to compare hardware.
ltp-stress	Stresses the system using the LTP test suite.
hugepage-tests	Perform basic functional and stress tests for large pages
ramsnake	Allocate 1/8 of the system RAM and kick off threads to use them for 3600 seconds.
random_syscall	Pounds on syscall interface and does random syscalls
reaim	A multiuser benchmark that tests and measures the performance of open system multiuser computers.
sdet	Workload created by parallel execution of common UNIX commands.
libhugetlbfs	Interacts with the Linux hugetlbfs to make large pages available to applications in a transparent manner.
Others	Tested NFS, CIFS and the autofs filesystem.

Table 2: Benchmarks used for Code Coverage

the benchmarks used. Coverage was run on x86 and Power™ architectures, with following configurations:

8P Intel® XEON™, 10GB Memory
2P IBM® POWER5+, 7GB Memory

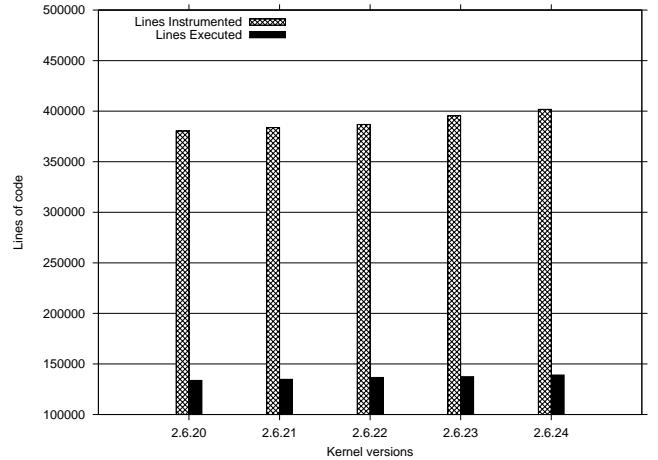


Figure 4: Code coverage on x86 architecture

Kernels	Lines Instrumented	Lines Executed
2.6.20	380,333	133,718
2.6.21	383,800	134,800
2.6.22	386,868	136,653
2.6.23	395,520	137,472
2.6.24	401,802	139,052

Table 3: Lines Instrumented Vs Executed on x86 architecture

Figure 4 and Table 3 show number of lines instrumented and the code covered of the various kernels mentioned for the x86 architecture. The figure shows the following trends:

- The number of lines instrumented shows a modest increase of 5.64%. When compared to the rate of change of the kernel, it does not seem significant. The code coverage data above fails to show that changed lines are also covered as the kernel version changes.
- The code coverage shows a modest increase of 3.98%. It is very encouraging to see code coverage increase as the number of lines in the kernel increase.
- Versions 2.6.23 and 2.6.24 show a trend of decline in code coverage percentage. The coverage

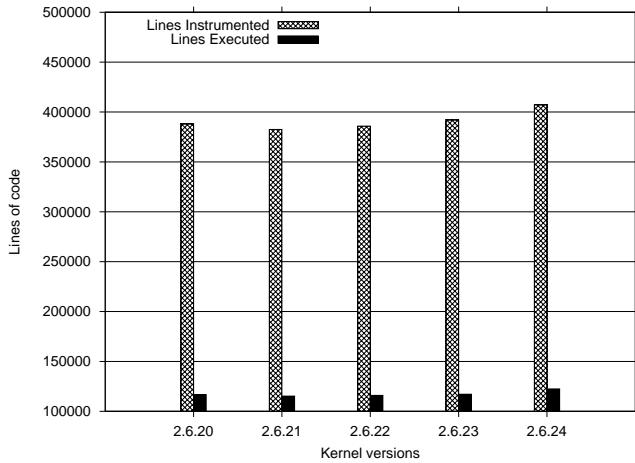


Figure 5: Code coverage on Power™ architecture

Kernels	Lines Instrumented	Lines Executed
2.6.20	388,019	116,736
2.6.21	382,552	115,242
2.6.22	385,853	116,010
2.6.23	391,993	117,168
2.6.24	407,194	122,431

Table 4: Lines Instrumented Vs Executed on Power™ architecture

of 2.6.20 kernel was 35.15%, where as for 2.6.24 it is 34.6%.

Figure 5 and Table 4 show the coverage of the various kernels mentioned for the Power™ architecture. The figure shows the following trends:

- The number of lines of instrumented code show a trend similar to the ones for the x86 architecture.
- The code coverage has increased with increasing kernel versions.
- The code coverage percentage however is less than that of the x86 architecture, close to 30%.
- There is no decline in the code coverage percentage as seen on the x86 architecture, indicating that x86 is growing rapidly.¹²

Figures 6,7 and Tables 5,6 show the component-wise break up for the code coverage obtained for the 2.6.24

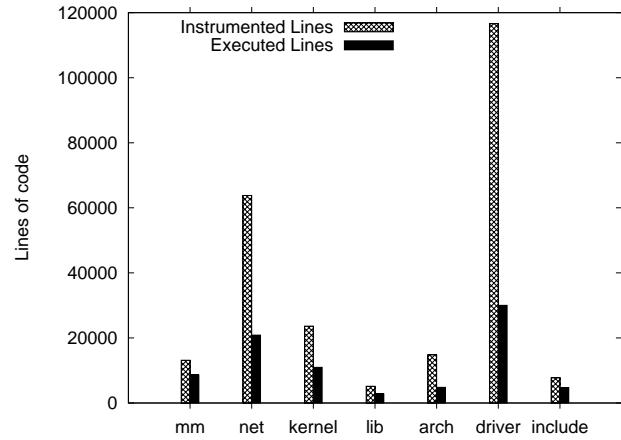


Figure 6: Component-wise code coverage on 2.6.24 kernel with x86 architecture

Directories	Lines Instrumented	Lines Executed
mm	14,585	9,133
net	65,441	20,749
kernel	23,407	10,916
lib	5,062	2,799
arch	24,954	5,700
driver	88,195	8,930
include	7,638	4,434

Table 5: Lines Instrumented Vs Executed on 2.6.24 kernel with x86 architecture

Directories	Lines Instrumented	Lines Executed
mm	14,585	9,133
net	65,441	20,749
kernel	23,407	10,916
lib	5,062	2,799
arch	24,954	5,700
driver	88,195	8,930
include	7,638	4,434

Table 6: Lines Instrumented Vs Executed on 2.6.24 kernel with Power™ architecture

¹²which might be true, due to the x86 and x86-64 merge.

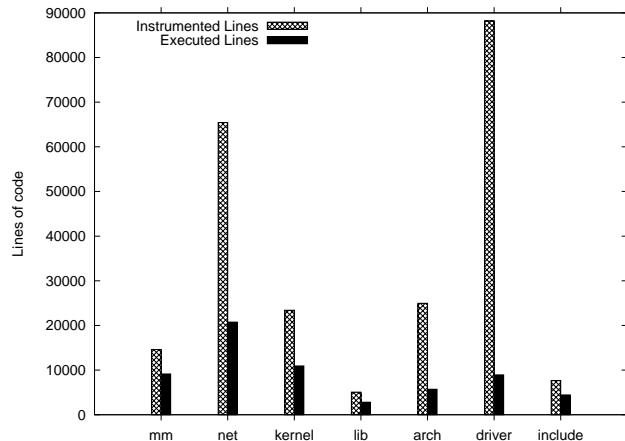


Figure 7: Component-wise code coverage on 2.6.24 kernel with Power™ architecture

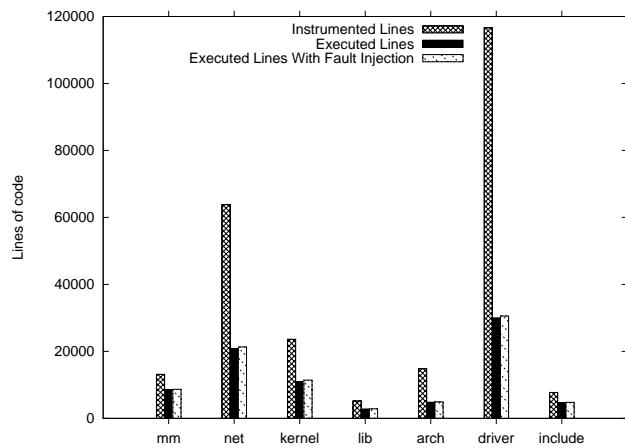


Figure 8: Fault injection code coverage on 2.6.24 kernel with x86 architecture

kernel on the x86 and Power™ architectures respectively. The component-wise break up shows some interesting trends as well:

- The mm, kernel, lib, and include subdirectories are among those that have the highest code coverage.
- The subsystem with highest coverage is mm, with close to 65% coverage
- drivers and arch subdirectories are among those that have the least code coverage. The main focus of our testing is not to test architecture or platform-specific code.

Directories	Lines Instrumented	Lines Executed	
		Disabled	Forced
mm	13,147	8,597	8,688
net	63,802	20,855	21,306
kernel	23,633	10,962	11,431
lib	5,152	2,785	2,853
arch	14,836	4,810	4,918
driver	116,623	30,031	30,552
include	7,751	4,707	4,777

Table 7: Fault Injection code coverage on 2.6.24 kernel with x86 architecture

We used the **Fault Injection framework**¹³ to get more coverage of the error handling path. The kernel was configured with:

```
N > /debug/fail_page_alloc/task-filter
20 > /debug/fail_page_alloc/probability
2000 > /debug/fail_page_alloc/interval
-1 > /debug/fail_page_alloc/times
0 > /debug/fail_page_alloc/space
1 > /debug/fail_page_alloc/verbose
N > /debug/fail_page_alloc/ignore-gfp-wait

N > /debug/fail_make_request/task-filter
20 > /debug/fail_make_request/probability
2000 > /debug/fail_make_request/interval
-1 > /debug/fail_make_request/times
0 > /debug/fail_make_request/space
1 > /debug/fail_make_request/verbose
N > /debug/fail_make_request/ignore-gfp-wait
```

(refer to [9] for more configuration details) The results of the coverage are shown in Figure 8. Getting coverage results with fault injection enabled turned out to be very challenging, since most applications are not ready to deal with failures. The test applications we saw would fail and abort their operation on error. We had to manually make changes to get coverage data with fault injection and we were forced to keep the failure rate very low. Due to these factors, we did not see a significant improvement in code coverage with the fault injection framework enabled, it was just 0.6% improvement. The test cases and infrastructure need to be enhanced to deal with the failures forced from the fault injection framework.

¹³injects errors at various kernel layers, helping to test error handling

5 Future Plans

We plan to extend our testing by adopting and updating test cases from various test projects and test scripts published on the mailing list to improve the coverage of untested code. We intend to test the practically possible permutations of kernel configurations and boot options with selective releases. We intend to build tests with the a cross-compiler setup will help us in finding the build errors over various platforms. Testing the kernel error path is very critical to avoiding surprises under certain situations. We could perform error handling path testing on selective kernels using the fault injection framework available in the kernel.

6 Conclusion

Testing kernel releases *earlier and often*, helps in fixing the bugs earlier in the cycle. The earlier the problem gets fixed, the lower the costs involved in fixing it. Our infrastructure tests various kernels across different hardware, using many benchmarks and test suites performing build, regression, stress, functional, and performance testing. Benchmarks are run selectively depending upon the kernel release. Testing results are contributed back to the community through test.kernel.org. We discussed some of the harnesses commonly used by the projects.

Code coverage results explain the gap between the lines of code being added and tested. For better and more complete testing of the kernel, we need test cases that can help us better test existing and new features. For these we require developers to share their tests and testing methodology. The fault injection framework helps testing the error handling part of the kernel, so improvements made to the framework could result in better kernel coverage.

7 Acknowledgments

We would like to thank Andy Whitcroft for his input to and review of drafts of this paper.

We also owe lot of thanks to Sudarshan Rao, Premalatha Nair, and our teammates for their active support and enthusiasm.

8 Legal Statement

©International Business Machines Corporation 2008. Permission to redistribute in accordance with Linux Symposium submission guidelines is granted; all other rights reserved.

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

IBM, IBM logo and ibm.com are trademarks of International Business Machines Corporation in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

References in this publication to IBM products or services do not imply that IBM intends to make them available in all countries in which IBM operates.

INTERNATIONAL BUSINESSMACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you. This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

References

- [1] Greg Kroah-Hartman, Jonathan Corbet, and Amanda McPherson, *How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It*, <http://www.linux-foundation.org/publications/linuxkerneldevelopment.php>, April 2008.
- [2] Fully Automated Testing of the Linux Kernel, Martin Bligh and Andy P. Whitcroft. In *Proceedings of the Linux Symposium 2006*.
- [3] Linux Test Project, <http://ltp.sourceforge.net/>

- [4] Autotest,
<http://test.kernel.org/autotest>
- [5] Kernelsource/Documentation/HOWTO.
- [6] Linux Kernel Mailing List.
linux-kernel@vger.kernel.org,
<http://lkml.org/>
- [7] Linux Test Project - Test Tools Matrix. <http://ltp.sourceforge.net/tooltable.php>
- [8] The Cathedral and the Bazaar, Eric Steven
Raymond.
- [9] kernelsource/Documentation/
fault-injection/fault-injection.
txt
- [10] kernelsource/Documentation/
kernel-parameters.txt

Korset: Automated, Zero False-Alarm Intrusion Detection for Linux

Ohad Ben-Cohen
Tel-Aviv University
ohad@bencohen.org

Avishai Wool
Tel-Aviv University
yash@acm.org

Abstract

Host-based Intrusion Detection Systems traditionally compare observable data to pre-constructed models of normal behavior. Such models can either be automatically learnt during a training session, or manually written by the user. Alas, the former technique suffers from false positives, and therefore repeatedly requires user intervention, while the latter technique is tedious and demanding.

In this paper we discuss how static analysis can be used to automatically construct a model of application behavior. We show that the derived model can prevent future or unknown code injection attacks (such as buffer overflows) with guaranteed zero false alarms. We present Korset, a Linux prototype that implements this approach, and focus on its Kernel implementation and performance.

1 Motivation

The battle between attackers and defenders has been ongoing throughout the course of computer history. Attackers expose and exploit application vulnerabilities on a daily basis, and as a result, software vendors regularly apply fixes to close breaches and mitigate attacks. Alas, it's a seemingly endless cycle that has attackers on the upper hand, as defenders are mostly responding.

Since even fully patched applications may have unknown security flaws, defenders commonly use a Host-based Intrusion Detection System (HIDS), or even multiple HIDSs, in order to augment security. The advantage of using HIDSs relies on the fact that they may stop attacks which exploits an application vulnerability that is still publicly unknown (a.k.a. zero-day exploit) or against which a patch is not yet provided.

HIDSs identify malicious activity typically by comparing a variety of observable data to a pre-constructed

model of normal application behavior. When a running process deviates from its model of behavior, it is assumed to be subverted by an attacker. In such an event the HIDS can take actions to prevent the attacker from damaging the system, e.g., by terminating the hijacked process.

There are two classic methodologies for constructing an application's model of normal behavior. One prevalent methodology infers the model from statistical data: the model is constructed over a period of time, called "training," which is assumed to be attack-free (and hopefully typical). During the training period, the behavior of the application is observed, collected and transformed into a representative model. After the model is constructed and the training period is over, the HIDS monitors the process, and any deviation from the constructed model is considered an attack and may result in the termination of the process. This methodology is highly automated and capable of detecting a wide range of attacks, but since it is based on statistical data, it has the inherent problem of false positives. Recently developed methods have yielded lower rates of false positives; however, in practice, it is still a major problem.

A second common HIDS methodology for constructing an application model of normal behavior is based on generating application policies. Such policies define the allowed behavior of the program, using rules that precisely specify which system resources a process can access and in what way. The policies can be written either by the developer himself or by a knowledgeable user, because writing them requires a precise understanding of the expected behavior of the application. The advantage of using program policies relies on the fact that they can describe the program's behavior as accurately as the program code itself, and thus can completely eliminate false alarms. Alas, manually writing accurate program policies is not for the faint of heart as it is a tedious and demanding task.

A HIDS capable of automatically deriving accurate pro-

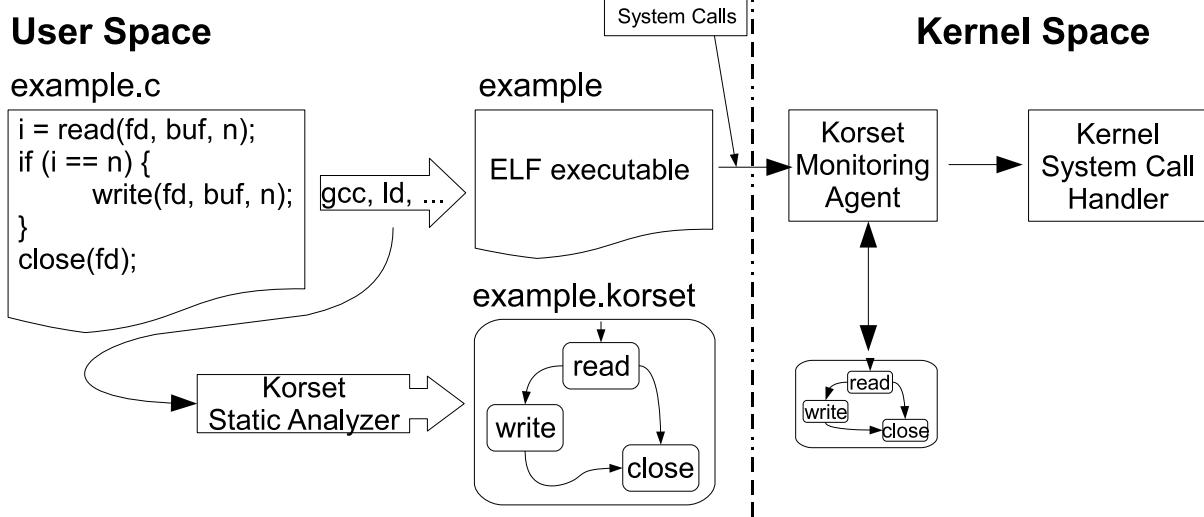


Figure 1: Korset’s system architecture. On the left an application is compiled. Korset’s static analyzer observe the building process, and creates a corresponding Korset graph. On the right, Korset’s in-kernel monitoring agent loads the application’s graph when it is executed, and then monitor the issued system calls by simulating the automaton. For simplicity, there is no notion of a process in the figure.

gram policies would enjoy both worlds of zero false positives and automation. This can be achieved using static analysis methods as explained in Section 2.

2 The General Idea

Korset’s model of application behavior is Control Flow Graphs (CFG) induced from the source code and object files of the program. Assuming that the most practical ways for an attacker to inflict damage involve system calls, Korset prunes the CFGs from the nodes that do not represent system calls. The resulting model is an automaton that represents the legitimate order of system calls that an application may issue. This automaton is then enforced by Korset’s monitoring agent, which is built into the Linux kernel, by simulating every emitted system call. When a divergence from the automaton is encountered, the running process is terminated.

Assuming that the program was written with benign intent, and isn’t self-modifying, then its source code reflects the full extent of the legitimate application behavior, and nothing else. Every possible path of execution is obviously represented in the source, and as a result, also in the induced automaton. Every sequence of system calls not matching the derived automaton couldn’t have been issued by the program itself, and therefore can

be safely regarded as an intrusion. This leads to Korset guaranteeing zero false positives.

3 Architecture

Korset has two main subsystems (see Figure 1), described in the following subsections.

3.1 The static analyzer

The static analyzer is a user space subsystem that is responsible for creating the final application CFG. When the static analyzer is enabled, the application CFG is automatically created as part of the compilation process. When the user builds an application (by running make, or compiling a random source file), the static analyzer also creates the CFG. This is achieved by wrapping the GNU build tools (gcc, ld, as, ar) in a way that is transparent to the build system. As a result, a CFG is constructed for every built object file, executable or library (currently only static libraries are supported). The CFGs of C programs are initially derived using GCC’s capability to dump a representation of the program’s CFG during compilation, and the CFGs of Assembly programs are derived by analysing their object code (see Figure 2).

After the CFGs of the application’s functions are created, they are linked together to create a unified CFG

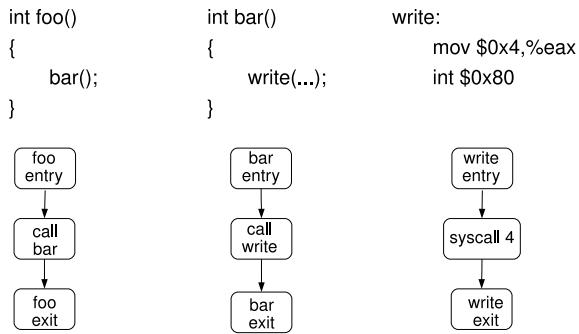


Figure 2: Three simple functions and their representative control flow graphs, as constructed by Korset. Note that `write`, which belongs to glibc, is a simple wrapper around a direct system call (Korset does not yet support the x86's sysenter facility).

that corresponds with the executable of the application (see Figure 3). Along the way, the graphs are simplified by tossing away CFG nodes that don't add relevant information, and as a result the final CFG consists exclusively of system calls nodes (see Figure 4).

The CFG simplification is a lengthy process which includes numerous steps of graph determinizing and automaton manipulations. The complete theory and algorithms behind this process is described in length in our companion paper [BW08]. The end result, shown in Figures 5 and 6, is designed to achieve minimum runtime overhead: the final CFG is consisted of only system calls nodes and is actually a completely deterministic automaton. It is then transformed to a binary representation, which is designed to achieve maximum performance by placing the possible emitted system calls of each graph node closely after the location of the node itself (see Figure 7).

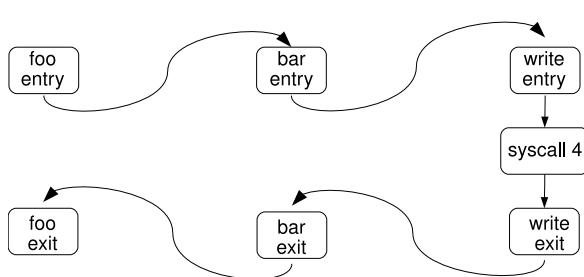


Figure 3: The CFGs are linked together in order to construct the final CFG of the program's executable.

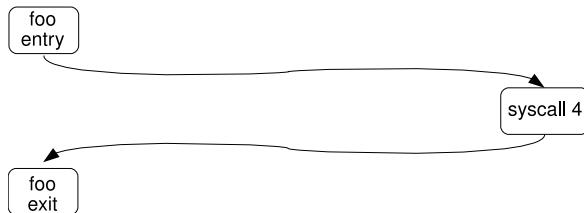


Figure 4: The resulting CFG is very simple since Korset prunes away all graph nodes that do not add relevant information.

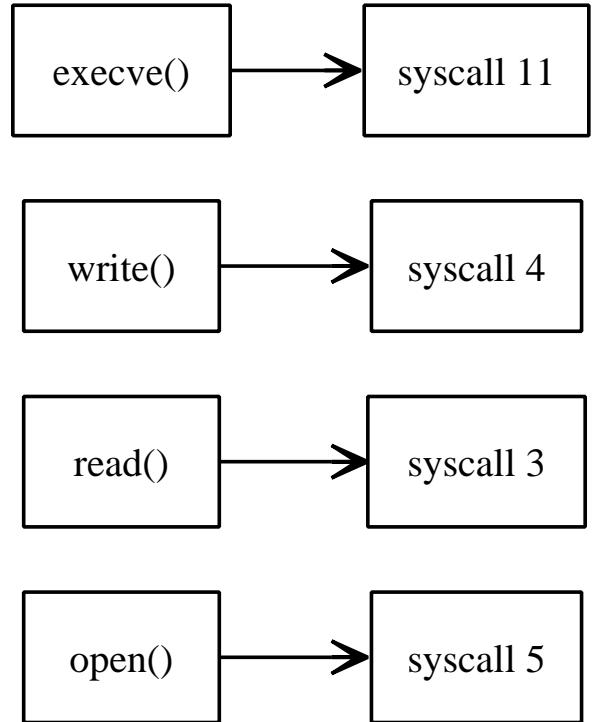


Figure 5: The CFGs of glibc's `execve()`, `write()`, `read()` and `open()`. These library routines are simple wrappers around the relevant system call, as reflected in the CFGs. The system calls involved are `read(3)`, `write(4)`, `open(5)` and `execve(11)`.

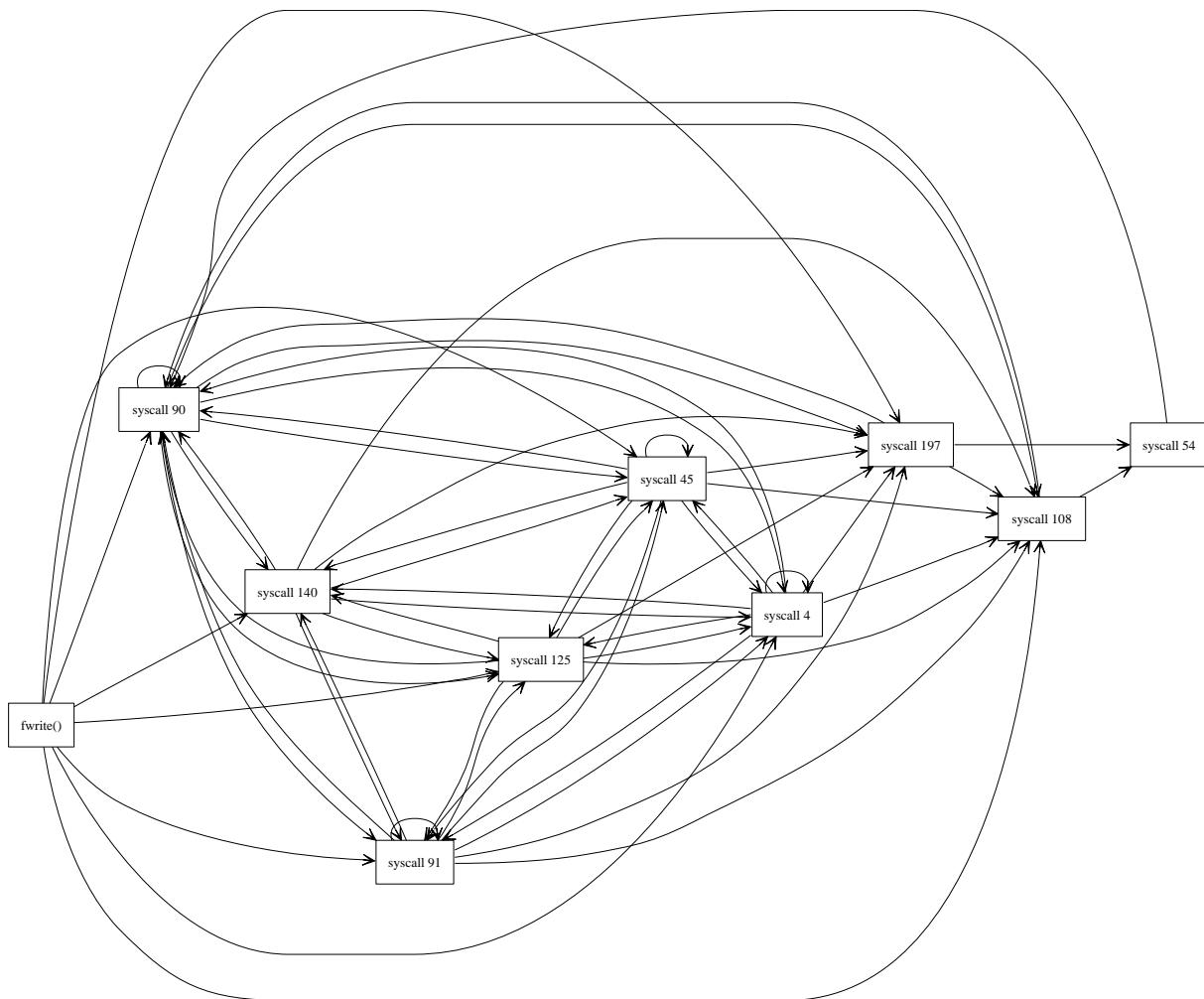


Figure 6: The CFG of glibc's `fwrite()`. The system calls involved are `write(4)`, `brk(45)`, `ioctl(54)`, `mmap(90)`, `munmap(91)`, `fstat(108)`, `mprotect(125)`, `_lseek(140)` and `fstat64(197)`.

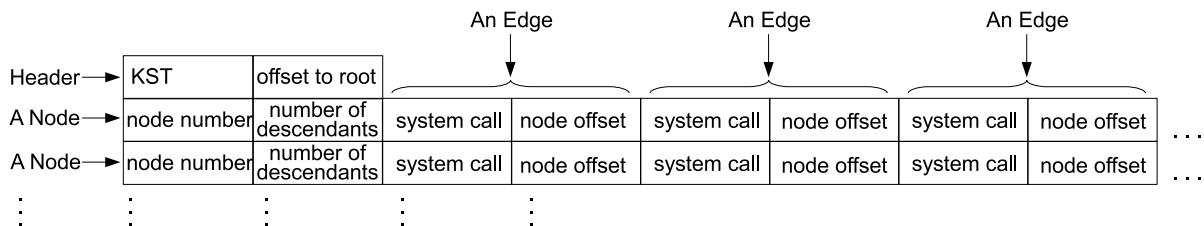


Figure 7: The binary representation of Korset's graphs is geared towards run-time efficiency.

3.2 The Monitoring Agent

Korset's monitoring agent is built natively into the Linux kernel. When a monitored program is executed, the monitoring agent loads its CFG, observes the issued system calls by the process, and then validates their legitimacy by simulating them on the induced automaton.

Following is a description of the components that the monitoring agent is built of.

Note: Support for the changes described in Subsection 3.2.1 is added only if the kernel's build variable `CONFIG_SECURITY_SYSCALL` is enabled. Likewise, support for all other changes described in Subsections 3.2.2-3.2.5 is added only if the kernel's build variable `CONFIG_SECURITY_KORSET` is enabled.

3.2.1 Linux Security Modules changes

Linux Security Modules (LSM) is a Linux kernel security framework that provides, among other things, a set of security hooks that are used to perform access control. These hooks can be used by security modules to implement any desired model of security. In order to support the monitoring of system calls, we have added a new LSM hook, called `security_system_call`, into the `security_operations` structure (defined by `include/linux/security.h`). This new hook function is called every time user space makes a request to execute a system call, by a handful of assembly instructions that we have added to the `system_call` handler (no support yet for the x86's sysenter facility and in general only the x86 architecture is currently supported). The `security_system_call` hook function is given two arguments:

1. The location of the `struct thread_info` of the current process (which have just made a request to execute a system call), retrieved from the process' kernel stack using the `GET_THREAD_INFO` macro. The `struct thread_info` can be used to access the process' `task_struct` structure (the Linux process descriptor), where process security state is maintained by Korset.
2. The requested system call number, as given in `%eax` from user space.

Any security module that registers the `security_system_call` hook should use these two arguments to decide whether or not to allow the execution of the desired system call. The hook function should return zero if permission to execute the system call is granted. If zero is returned, the system call handler continues with normal execution of the system call. Otherwise, the system call handler immediately returns to user space with an `EACCES` error (permission denied).

As with other LSM hooks, a security module who wishes to use the `security_system_call` hook should call `register_security` to set `security_ops` to refer to its own hook function.

3.2.2 task_struct changes

In order to maintain a per-process automaton in the kernel, we have added the following new fields to the `task_struct` structure:

- `korset_graph`: The location, in memory, of the automaton. When a process is not monitored, this field holds a NULL.
- `korset_node`: An offset that, together with `korset_graph`, yields the location in memory of the automaton node that the current process is at.
- `korset_size`: The size, in bytes, of the automaton that is pointed to by `korset_graph`. Used by automaton sanity checks.

3.2.3 CFG Loader

When app is executed, Korset looks for the file `app.korset`, which, if it exists, holds the CFG of app. This is done in `fs/exec.c` by the `do_execve` function, in almost exactly the same manner as the kernel looks for app's executable itself (with the exception that only READ permissions/flags are needed instead of the usual EXEC ones). If the file `app.korset` exists, Korset loads it from disk. This is done in `fs/binfmt_elf.c` by the `load_elf_binary` function (currently only ELF executables are supported by Korset), again in a very similar way to how the kernel loads the executable itself. After app's graph is successfully read from disk,

its location in memory is put in the `korset_graph` field of the current process, the `korset_size` field is updated with the graph's size and the `korset_node` field is set to point to the graph's root node. (The offset to the root node is taken from `app.korset`, as seen in Figure 7.)

3.2.4 CFG Enforcer

Whenever a process issues a system call a , Korset's `security_system_call` LSM hook function is called. If `korset_graph == NULL` then the current process is not monitored, and a is immediately approved. Otherwise, if `korset_graph != NULL`, then the current process is monitored, and then Korset validates the legitimacy of a by simulating one step of the automaton. This is accomplished by checking the outgoing edges of the current node, one by one, looking for an edge that is carrying a . If such an edge is found, the `korset_node` field of the current process is updated to point to the destination node of the found edge, and a is executed. If such an edge is not found, the current process is terminated, and `security_system_call` returns -1 to the `system_call` handler. As a result, the `system_call` handler does not execute the requested system call. Instead, it immediately returns to user space as explained in 3.2.1.

In some cases, after an edge carrying the requested system call is found, it is also desired to manipulate its location in the graph for performance reasons. This manipulation is done within the enforcing code. For more information, see Section 4.

3.2.5 CFG Dumper

As mentioned in the previous subsection, sometimes the application CFG is manipulated while it is enforced. This may happen for run-time optimization reasons, as described in section 4. In those cases, it might be desired that the graph file `app.korset` itself will be updated to reflect the changes. This is where Korset's CFG Dumper kicks in. When a process terminates, and the conditions described in Section 4 are met, the updated CFG is dumped to disk. This is done in the `do_exit` function, defined by `kernel/exit.c`, in a manner that resembles the way the kernel dumps a core file.

4 Runtime Optimization via Frequent Edges First

Validating a system call a involves matching a against each of the outgoing edges of the current node v , until either a match is found or all outgoing edges have been traversed. The time complexity of this operation is obviously $O(d)$, where d is the number of outgoing edges v has. The actual overhead is obviously dependent on the position, in memory, of the more frequent edges. If a system call a is carried by the first edge in memory, only one matching iteration will be performed to locate it, and the run-time overhead will be minimal. Based on that simple observation, we added the following Frequent Edges First (FEF) algorithm to Korset's monitoring agent:

1. Every time a system call match is made to an outgoing edge e of node v , e is moved-to-front, i.e., it is removed from its i -th location in memory, the $i-1$ edges that are located in positions $1..i-1$ are shifted to positions $2..i$, and e is placed in position 1.
2. When a monitored process finishes its execution, its updated CFG is dumped to disk.

This means that when the FEF is enabled, the CFG dynamically changes during the execution of the process. After the process terminates, the most frequent edges of that specific execution are positioned before the less frequent ones. If the next execution of that application would have a similar system call sequences, Korset's overhead will be substantially smaller. The FEF can be used in two modes:

1. Always Enabled: In this mode, the CFG will always be updated during process execution according to the FEF algorithm. The main benefit of this mode is that a recurring sequence of system calls will incur minimal overhead. However, this mode introduces additional computation that is performed per observed system call, which may actually increase Korset's run-time overhead. Therefore, we do not use this mode.
2. Only Once: In this mode, the FEF is used as the final step of CFG construction—the relevant application is executed once, in a common workload. Upon termination, Korset dumps the updated

CFG to disk, which is then used as the application’s newly constructed CFG. This way, when a new CFG is constructed, the position of its outgoing edges reflect a real execution of its program rather than a random order. This mode is obviously cheap—it has no run-time cost. Its only cost is in the initial phase of CFG construction. Despite the low cost and obvious limitation of this mode, it is quite effective. We have found that even a single FEF adjustment of a CFG during a single execution of a program significantly improves Korset’s performance in future executions.

The ‘Always Enabled’ mode can be further tuned to reduce run time cost (e.g., an outgoing edge can be advanced only if it is not in the first t locations, where t is a tunable variable that might be different for each application). This is a topic for further research. In contrast, the ‘Only Once’ mode is always used, since it has no run time cost, and was found to be notably effective.

5 Run-Time Micro Benchmarks

Figure 8 demonstrates the percentage of overhead imposed by Korset’s monitoring agent via micro benchmarks for four system calls with different speeds (write is the slowest while setuid is the fastest). For a single system call a , the actual overhead depends on the position in memory of the edge carrying a . We measured three different scenarios for each of the system calls:

1. Best Case: The matching edge is in the first position. While this is the best run-time performance that can be achieved, high precision models (i.e. with low branching factors), should produce similar results.
2. Bad Case: The matching edge is in the 50th position. This overhead level can only be achieved with models that have bad precision since a node with 50 or more outgoing edges is very limited with its constraining effectiveness.
3. Worst Case: The matching edge is in the 325th position (there are 325 different system calls in Linux 2.6.24). Measured for comparison.

The figure shows two things:

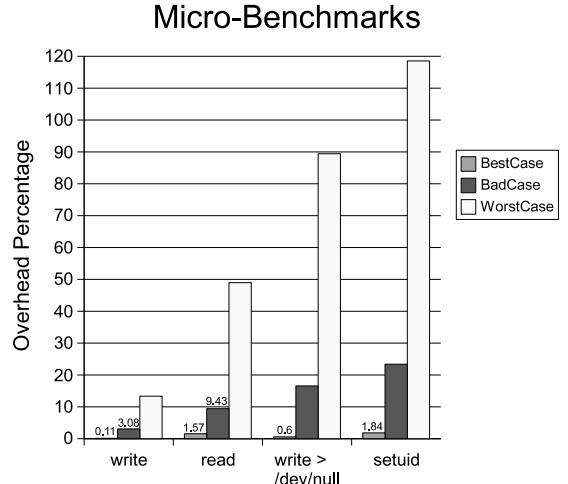


Figure 8: Micro-benchmarks of Korset’s overhead on four system calls with variable speed. setuid is the fastest system call and thus has the biggest percentage of run-time overhead incurred by Korset.

1. The overhead imposed by a small branching factor is negligible. Achieving a small branch factor is obviously a factor of precision, but the same performance can be achieved also by placing the more common graph edges first (which is what our Frequent Edges First process does).
2. As long as the system call itself is slow, Korset’s run-time penalty is small (in percentage) even if the branching factor is big. As the system call is faster, Korset’s run-time penalty is becoming more significant (in percentage). Measuring Korset’s performance on simple I/O-centric applications yielded results similar to the best case (around 1% overhead) [BW08].

6 Related Work

The idea of applying code-based static analysis techniques to automatically construct models with zero false alarms was introduced in a seminal work by Wagner and Dean [WD01]. Their work covers the theory behind the idea, and is a recommended read. Wagner and Dean used a non-deterministic model which resulted in big run-time overheads, and their prototype was implemented in Java. Continuing their work, Giffin et al. demonstrated static analysis of SPARC binary code to generate system calls CFGs of Solaris application [GJM02, HGH⁺04, GJM04, GDJ⁺05]. Giffin

et al. have introduced numerous automaton manipulation techniques and binary modification techniques to increase model precision and reduce non-determinism. Methods to increase the model precision were introduced, based on additional observable data. More specifically, it was suggested to add system call arguments [WD01, GJM04], program counter and call stack information [HGH⁺04, GJM04] and environment information [GDJ⁺05]. A completely deterministic model, which results in better run-time performance, was suggested [BW08]. Lastly, the code-based static analysis model is not foolproof. An attacker can intentionally issue system calls in order to arrive to a desired automaton node. This type of attack, called *mimicry attack*, was introduced by Wagner et al. at [WD01, WS02].

7 Current Status and Future Work

Korset is still a very young prototype. It blocks real shellcodes, and it was found to incur negligible overhead on simple I/O-centric applications (see our companion paper [BW08] for a detailed evaluation and analysis), but it is still very far from a mature HIDS. It does not yet support dynamically linked applications, multi-threaded applications, signals, `setjmp`/`longjmp`, etc. In addition, there is still a lot of work required to increase the precision of the automatons, e.g., better assembly analysis, better indirect calls analysis, add additional observable data to the model, improve the automaton manipulation algorithms, etc.

Korset is hosted at <http://www.korset.org>.

8 Conclusion

Korset is an HIDS prototype that automatically constructs models of application behavior, and enforces them from the Linux kernel with guaranteed zero false positives. Korset is capable of stopping future, or publicly unknown code injection attacks (e.g., buffer overflows). Although Korset is still a prototype, it demonstrates a viable HIDS methodology with promising intrusion detection properties.

References

- [BW08] Ohad Ben-Cohen and Avishai Wool. Korset: Making intrusion detection via static analysis practical. Submitted for publication, 2008.
- [GDJ⁺05] Jonathon T. Giffin, David Dagon, Somesh Jha, Wenke Lee, and Barton P. Miller. Environment-sensitive intrusion detection. In *Recent Advances in Intrusion Detection*, pages 185–206, 2005.
- [GJM02] Jonathon T. Giffin, Somesh Jha, and Barton P. Miller. Detecting manipulated remote call streams. In *Proceedings of the 11th USENIX Security Symposium*, pages 61–79. USENIX Association, 2002.
- [GJM04] J. Giffin, S. Jha, and B. Miller. Efficient context-sensitive intrusion detection. In *Proc. 11th Annual Network and Distributed Systems Security Symposium (NDSS)*, 2004.
- [HGH⁺04] H.H. Feng H.H., J.T. Giffin, Yong Huang, S. Jha, Wenke Lee, and B.P. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *Proceedings IEEE Symposium on Security and Privacy*, pages 194–208, 9-12 May 2004.
- [WD01] David Wagner and Drew Dean. Intrusion detection via static analysis. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, page 156, Washington, DC, USA, 2001. IEEE Computer Society.
- [WS02] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 255–264, New York, NY, USA, 2002. ACM.

Suspend-to-RAM in Linux®

A. Leonard Brown

Intel Open Source Technology Center

len.brown@intel.com

Rafael J. Wysocki

Institute of Theoretical Physics, University of Warsaw

rjw@sisk.pl

Abstract

Mobile Linux users demand system suspend-to-RAM (STR) capability due to its combination of low latency and high energy savings.

Here we survey the design and operation of STR in Linux, focusing on its implementation on high-volume x86 ACPI-compliant systems. We point out significant weaknesses in the current design, and propose future enhancements.

This paper will be of interest primarily to a technical audience of kernel and device driver developers, but others in the community who deploy, support, or use system sleep states may also find it useful.

1 Introduction

When a computer is powered on, it enters the *working state* and runs applications.

When a computer powered off, it consumes (almost) no energy, but it doesn't run applications.

As illustrated in Figure 1, several power-saving system *sleep states* are available between the working and power-off states. System sleep states share several properties:

- Energy is saved.
- The CPUs do not execute code.
- The I/O devices are in low-power states.
- Application state is preserved.

However, system sleep states differ from each other in two major ways:

- The size of the energy-saving benefit.

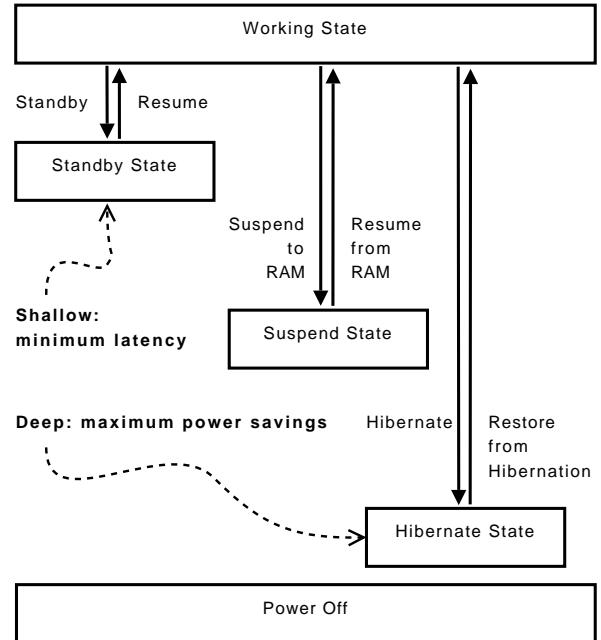


Figure 1: System States

- The *wake-up latency* cost required to return to the working state.

Linux supports three types of system-sleep states: standby, suspend-to-RAM, and hibernate-to-disk.

Standby is the most *shallow* system-sleep state. This means that it enjoys minimal wake-up latency. However, standby also delivers the least energy savings of the sleep states.

Suspend-to-RAM is a *deeper* sleep state than standby. STR saves more energy, generally by disabling all of the motherboard components except those necessary to refresh main memory and handle wake-up events.

In practice, STR generally enjoys the same latency as standby, yet saves more energy. Thus standby support is typically of little benefit, and computer systems often do not provide it.

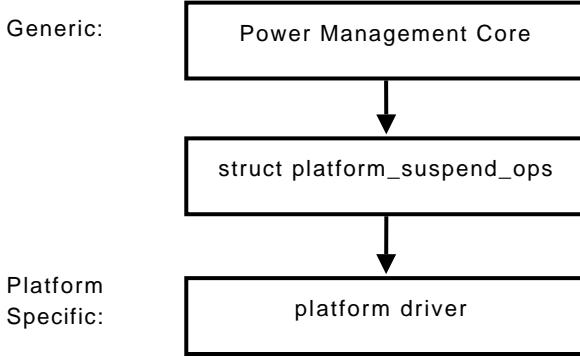


Figure 2: Linux PM infrastructure

Hibernate-to-disk is deeper than suspend-to-RAM. Indeed, it shares the same (almost zero) energy consumption as the power-off state. This makes hibernate valuable when application state needs to be preserved for a long period. Unfortunately hibernate resume latency is quite high, so it generally is not as useful as STR.

Here we focus on suspend-to-RAM, popular due to its combination of relatively low latency and relatively high energy savings. We first introduce the Linux Power Management (PM) Core, which is responsible for controlling suspend and resume operations at the kernel level. Next we describe the role of the platform firmware and ACPI in STR, and we provide an overview of the operations performed by the Linux kernel during suspend and resume. We examine some parts of the kernel’s STR infrastructure in more detail, focusing on the known issues with the current implementation and planned improvements. Finally, we consider some problems related to the handling of hardware, especially graphics adapters, their current workarounds, and future solutions.

2 The Linux Power Management Core

STR and standby are available only on systems providing adequate hardware support for them: The system main memory has to be powered and refreshed as appropriate so that its contents are preserved in a sleep state. Moreover, the hardware must provide a mechanism making it possible to wake the system up from that state and pass control back to the operating system (OS) kernel. It is also necessary that power be at least partially removed from devices before putting the system into a sleep state, so that they do not drain energy

```

struct platform_suspend_ops {
    int (*valid)(suspend_state_t state);
    int (*begin)(suspend_state_t state);
    int (*prepare)(void);
    int (*enter)(suspend_state_t state);
    void (*finish)(void);
    void (*end)(void);
};

```

Figure 3: struct platform_suspend_ops

in vain. The part of the system providing this functionality is often referred to as *the platform*. It defines the foundation of the system—the motherboard hardware as well as the firmware that ships with it.¹

To carry out STR and standby power transitions, the Linux kernel has to interact with the platform through a well-defined interface. For this purpose, the kernel includes *platform drivers* responsible for carrying out low-level suspend and resume operations required by particular platforms. They supply a set of callbacks via struct platform_suspend_ops shown in Figure 3. The .valid() and .enter() callbacks are mandatory, while the others are optional.

The platform drivers are used by the PM Core. The PM core is generic; it runs on a variety of platforms for which appropriate platform drivers are available, including ACPI-compatible personal computers (PCs) and ARM platforms.

This paper focuses primarily on ACPI-compatible platforms. The next section describes how ACPI fits into the system architecture and describes some of the specific capabilities that ACPI provides for suspend/resume. Then we combine the PM core and ACPI discussions by stepping through the suspend and resume sequences on an ACPI platform, in which all six of the platform_suspend_ops are invoked.

3 ACPI and Platform System Architecture

Platform hardware defines the programming model seen by software layers above. Platforms often augment this programming model by including an embedded controller (EC) running motherboard firmware. The EC off-loads the main processors by monitoring sensors for

¹The motherboard hardware includes not only the major processor, memory, and I/O sub-systems, but also interrupt controllers, timers, and other logic that is visible to the software layers above.

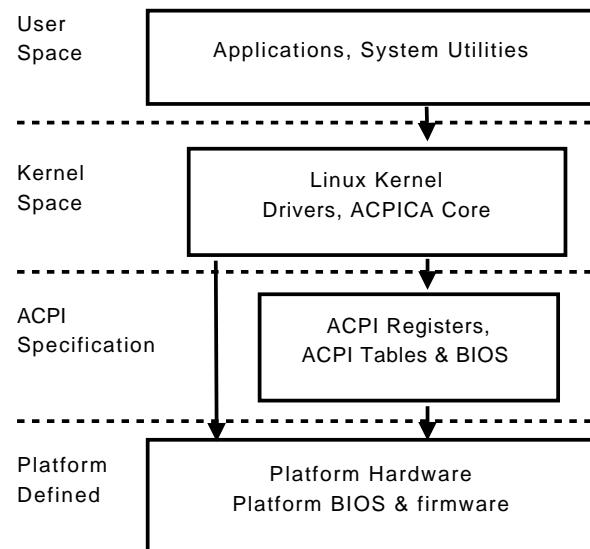


Figure 4: Platform Abstraction Layers

buttons, batteries, temperature, fans, etc. In addition, personal computer (PC) motherboards include a PC-compatible BIOS (Basic Input Output System) responsible for initializing the system and presenting some standard services, such as booting, to the OS kernel.

The ACPI specification [ACPI-SPEC] defines a layer of abstraction that sits above the platform-defined layer. ACPI specifies that the platform hardware must support certain standard registers, and that the BIOS be extended to export tables in memory to guide the OS kernel. Figure 4 shows how ACPI fits into the system architecture.

Some ACPI tables are simply static data structures that the ACPI BIOS uses to describe the machine to the OS. Other ACPI tables contain functions, known as *methods*, to be executed in the kernel, encoded in the ACPI Machine Language (AML).

AML methods are first expressed in ACPI Source Language (ASL) and then translated into the AML byte code with the help of an ASL compiler. The compiled AML tables are then burned into PROM when the motherboard is manufactured.

AML is executed by the ACPI Component Architecture [ACPI] Core's AML interpreter residing in the next layer up—the Linux kernel. This design allows AML, which is effectively ACPI BIOS code, to run in kernel context.² That, in turn, allows platform designers to use many different types of hardware components

and to tailor the interfaces between those components and the OS kernel to their needs.

However, since the AML code is obtained by compiling source code written in ASL, the preparation of an ACPI platform involves software development that is prone to human error. There are additional problems related to the ACPI's abstraction capability, some of which are discussed in Section 11.

4 ACPI and Suspend to RAM

Aside from its configuration and run-time power-management responsibilities, ACPI also standardizes several key hooks for suspend and resume:

1. Device power states.
2. Device wake-up control.
3. Standard mechanism to enter system sleep states.
4. Firmware wake-up vector.

When discussing ACPI and devices, it is important to realize that ACPI firmware is stored into EEPROM when a motherboard is created. Thus ACPI can be aware of all logic and devices that are permanently attached to the motherboard, including on-board legacy and PCI devices, device interrupt routing, and even PCI slot hot-plug control. But ACPI has no knowledge of the devices that may later be plugged into I/O slots or expansion buses.

ACPI defines “Device Power States” (D-states) in terms of power consumption, device context retention, device-driver restore responsibilities, and restore latency. The PCI Power Management specification [PCI-PM] similarly defines the D-states used by PCI devices; ACPI extends the notion of D-states to non-PCI motherboard devices.

D-states can be used independently of system sleep states, for run-time device power management. Note, however, that D-states are not performance states; that is, they do not describe reduced levels of performance. A device is non-functional in any D-state other than D0.

²Before ACPI, the choices to support run-time BIOS code were

to call the BIOS directly in real mode, which required a real-mode OS, or to invisibly invoke the System Management Mode (SMM).

System sleep states, such as STR, mandate the use of D-states. Before D-states were implemented in Linux, we found several systems that would suspend and resume from RAM successfully, but they had devices which would continue to drain energy while the system was suspended—which severely shortened the the system’s ability to sleep on battery power.

ACPI also defines a mechanism to enable and disable device wake-up capability. When the system is in the working state, this mechanism can be used to selectively wake up a sleeping device from a D-state. When the whole system is suspended, this capability may be used to enable automatic system resume.

Many devices export native wake-up control. In particular, modern Ethernet Network Interface Cards (NIC) support Wake-on-LAN (WOL) and their drivers export that function via `ethtool(1)`. Note that this has to be a native capability, because ACPI firmware can not provide wake-up support for add-on adapters.

Today the wake-up support in Linux is in flux. There is a legacy hook for ACPI wake-up capability in `/proc/acpi/wakeup`, but that interface is nearly unusable, as each entry refers to an arbitrary 4-letter ASL name for a device that the user (or an application) cannot reliably associate with a physical device. The device core provides its own wakeup API in `/sys/devices/.../power/wakeup`; this is not yet fully integrated with the ACPI wake-up mechanism.

The ACPI specification carefully describes the sequence of events that should take place to implement both suspend and resume. Certain platform ACPI hooks must be invoked at various stages in order for the platform firmware to correctly handle suspend and resume from RAM. These hooks are mentioned in later sections that detail the suspend and resume sequence.

Finally, ACPI provides a standard mechanism to tell the platform what address in the kernel to return to upon resume.

More information about ACPI in Linux can be found in previous Linux Symposium presentations [ACPI-OLS], as well as on the Linux/ACPI project home page [ACPI-URL].

5 Suspend Overview

There are two ways to invoke the Linux kernel’s suspend capability. First, by writing `mem` into `/sys/power/`

state.³ Second, with the `SNAPSHOT_S2RAM ioctl` on `/dev/snapshot`, a device provided by the user-space hibernation driver. This second method is provided only as a means to implement the mixed suspend-hibernation feature⁴ and will not be discussed here.

Once `mem` has been written to `/sys/power/state`, the PM core utilizes the `platform_suspend_ops` in the steps shown in Figure 5. First, it invokes the platform driver’s global `.valid()` method, in order to check whether the platform supports suspend-to-RAM.

The `.valid()` callback takes one argument representing the intended system-sleep state. Two values may be passed by the PM core, `PM_SUSPEND_STANDBY` and `PM_SUSPEND_MEM`, representing the standby and the suspend sleep states, respectively. The `.valid()` callback returns `true` if the platform driver can associate the state requested by the PM core with one of the system-sleep states supported by the platform. Note, however, that on non-ACPI systems the choice of the actual sleep state is up to the platform. The state should reflect the characteristics requested by the core (e.g., the STR state characteristics if `PM_SUSPEND_MEM` is used), but the platform may support more than two such sleep states. In that case, the platform driver is free to choose whichever sleep state it considers appropriate. But the choice is made later, not during `.valid()`.

If the `.valid()` platform callback returns `true`, the PM core attempts to acquire `pm_mutex` to prevent concurrent system-wide power transitions from being started. If that succeeds, it goes on to execute `sys_sync()` to help prevent unwritten file system data from being lost in case the power transition fails in an unrecoverable manner. It switches the system console to a text terminal in order to prevent the X server from interfering with device power-down. It invokes suspend notifiers to let registered kernel subsystems know about the impending suspend, as described in Section 7. It *freezes the tasks*, as detailed in Section 8. This puts all user processes into a safe, static state: They do not hold any semaphores or mutexes, and they cannot run until the PM core allows them to.

³The old STR user interface, based on `/proc/acpi/sleep`, is deprecated.

⁴This feature first creates a hibernation image and then suspends to RAM. If the battery lasts, then the system can resume from RAM, but if the battery fails, then the hibernation image is used. An experimental implementation is provided by the `s2both` utility included in the user-land suspend package at <http://suspend.sf.net>.

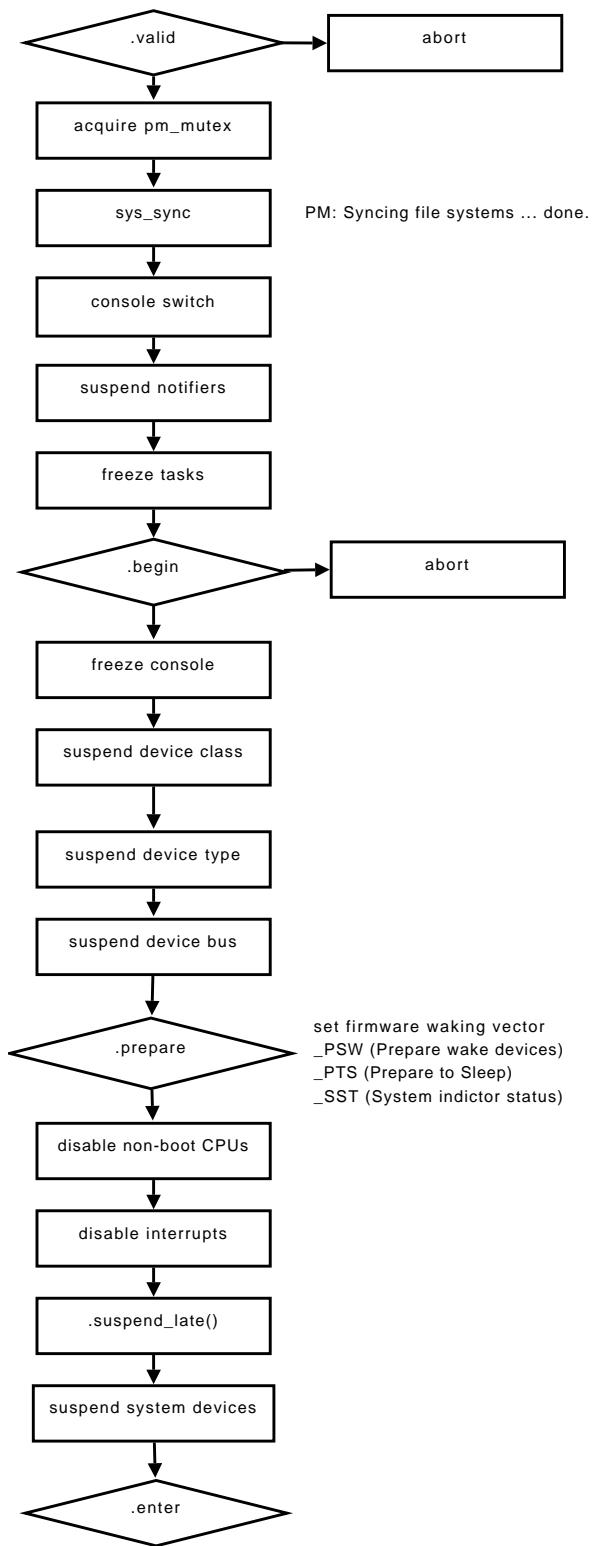


Figure 5: Suspend Sequence

Next the PM core invokes the `.begin()` platform-suspend callback to notify the platform driver of the desired power transition. The `.begin()` callback takes one argument representing the system-sleep state requested by the PM core. The interpretation as well as the possible values of its argument are the same as for the `.valid()` callback. At this point, however, the platform driver chooses the sleep state in which to place the system and stores the information for future reference. The choice made by the platform driver is not directly conveyed to the PM core, which does not have the means to represent various sleep states that may be supported by different platforms and does not really need that information. Still, the sleep state chosen by the platform driver, the *target state* of the transition, may determine the low-power states (D-states on ACPI systems) into which devices should be put. For this reason, the platform driver provides device drivers with information on the low-power states in which they are supposed to place their devices. The `.begin()` callback returns 0 on success or an error code on failure, in which case the PM core aborts the transition.

After `.begin()` succeeds, the PM core blocks system console messages in order to protect the console devices from being accessed while they are suspended,⁵ and starts suspending devices (i.e., putting them in low-power states). Devices are suspended in reverse order of registration; in this way the kernel should never find itself stuck in a situation where it needs to access a suspended device or where a suspended parent has an active child. To suspend a device, the PM core invokes the suspend callbacks provided by the device-class driver, device-type driver, and bus-type driver associated with it, in that order.

Device-class, device-type, and bus-type drivers can each implement one device-suspend method, called `.suspend()`, and one corresponding device-resume method, called `.resume()`. Bus-type drivers can define extra device-suspend and -resume methods to be executed with interrupts disabled, called `.suspend_late()` and `.resume_early()`, respectively. These additional methods are invoked after the non-boot CPUs have been disabled, as described below.

Each of the suspend callbacks takes two arguments,

⁵Since this mechanism makes debugging difficult, there is a `no_console_suspend` kernel command-line parameter which prevents it from triggering.

a pointer to the appropriate device structure and a `pm_message_t` argument, representing the transition being carried out. Currently five values of this argument are recognized: `PMSG_ON`, `PMSG_FREEZE`, `PMSG_SUSPEND`, `PMSG_HIBERNATE`, and `PMSG_PRETHAW`. The first represents the transition back to the working state, while `PMSG_FREEZE`, `PMSG_HIBERNATE`, and `PMSG_PRETHAW` are specific to hibernation. Thus only `PMSG_SUSPEND` is used for standby and STR. Since the same callbacks are invoked for both suspend and hibernation, they must determine the proper actions to perform on the basis of the `pm_message_t` argument.

The device-class, device-type, and bus-type suspend callbacks are responsible for invoking the suspend callbacks implemented by individual device drivers. In principle the names of those suspend callbacks may depend on the device class, device type, or bus type the device belongs to, but traditionally drivers' suspend callbacks are called `.suspend()` (or `.suspend_late()` if they are to be executed with interrupts disabled). Also, all of them take two arguments, the first of which is a pointer to the device structure and the second of which is as described above.

The framework for suspending and resuming devices is going to be changed. The current framework has some deficiencies: It is inflexible and quite inconvenient to use from a device-driver author's point of view, and it is not adequate for suspending and resuming devices without the freezing of tasks. As stated in Section 8, the freezing of tasks is planned to be phased out in the future, so a new framework for suspending and resuming devices (described in Section 9) is being introduced.

After executing all of the device-suspend callbacks, the PM core invokes the `.prepare()` platform suspend method to prepare the platform for the upcoming transition to a sleep state. For the ACPI platform, the `_PTS` global-control method is executed at this point.

Next the PM core disables non-boot CPUs, with the help of the CPU hot-plug infrastructure. We will not discuss this infrastructure in detail, but it seems important to point out that the CPU hot-plug notifiers are called with special values of their second argument while non-boot CPUs are being disabled during suspend and enabled during resume. Specifically, the second argument is bitwise OR'ed with `CPU_TASKS_FROZEN`, so that the notifier code can avoid doing things that might

lead to a deadlock or cause other problems at these times. The notifier routines should also avoid doing things that are not necessary during suspend or resume, such as un-registering device objects associated with a CPU being disabled—these objects would just have to be re-registered during the subsequent resume, an overall waste of time.⁶

After disabling the non-boot CPUs, the PM core disables hardware interrupts on the only remaining functional CPU and invokes the `.suspend_late()` methods implemented by bus-type drivers which, in turn, invoke the corresponding callbacks provided by device drivers. Then the PM core suspends the so-called system devices (also known as `sysdevs`) by executing their drivers' `.suspend()` methods. These take two arguments just like the regular `.suspend()` methods implemented by normal (i.e., not `sysdev`) device drivers, and the meaning of the arguments is the same.

To complete the suspend, the PM core invokes the `.enter()` platform-suspend method, which puts the system into the requested sleep state. If the `.begin()` method is implemented for given platform, the state chosen while it was executed is used and the argument passed to `.enter()` is ignored. Otherwise, the platform driver uses the argument passed to `.enter()` to determine the state in which to place the system.

6 Resume Overview

The resume sequence is the reverse of the suspend sequence, but some details are noteworthy.

Resume is initiated by a wake-up event—a hardware event handled by the platform firmware. This event may be opening a laptop lid, pressing the power button or a special keyboard key, or receiving a *magic* WOL network packet.

Devices must be enabled for wake-up before the suspend occurs. On ACPI platforms, the power button is always enabled as a wake-up device. The sleep button and lid switches are optional, but if present they too are enabled as wake-up devices. Any platform device may be configured as a wake-up device, but the power, sleep, and lid buttons are standard.

⁶It also would mess up the PM core's internal lists, since the objects would be re-registered while they were still suspended.

When a wake-up event occurs, the platform firmware initializes as much of the system as necessary and passes control to the Linux kernel by performing a jump to a memory location provided during the suspend. The kernel code executed at this point is responsible for switching the CPU to the appropriate mode of operation.⁷ This sequence is similar to early boot, so it is generally possible to reuse some pieces of the early initialization code for performing the resume CPU-initialization operations.

Once the CPU has been successfully reinitialized, control is passed to the point it would have reached if the system had not been put into the sleep state during the suspend. Consequently the PM core sees the platform-suspend callback `.enter()` return zero. When that happens, the PM core assumes the system has just woken up from a sleep state and starts to reverse the actions of the suspend operations described in Section 5.

It resumes `sysdevs` by executing their `.resume()` callbacks, and then it invokes the device-resume callbacks to be executed with interrupts disabled. That is, it executes the `.resume_early()` callbacks provided by bus-type drivers; they are responsible for invoking the corresponding callbacks implemented by device drivers. All of these callbacks take a pointer to the device object as their only argument.

Subsequently the non-boot CPUs are enabled with the help of the CPU hot-plug code. As mentioned above, all of the CPU hot-plug notifiers executed at this time are called with their second argument OR-ed with `CPU_TASKS_FROZEN`, so that they will avoid registering new device objects or doing things that might result in a deadlock with a frozen task.

After enabling the non-boot CPUs, the PM core calls the `.finish()` platform-suspend method to prepare the platform for resuming devices. In the case of an ACPI platform, the `_WAK` global-control method is executed at this point.

Next the PM core resumes devices by executing the device-resume methods provided by bus-type, device-type, and device-class drivers, in that order. All of these methods are called `.resume()` and take a device pointer as their only argument. They are responsible for invoking the corresponding methods provided by device

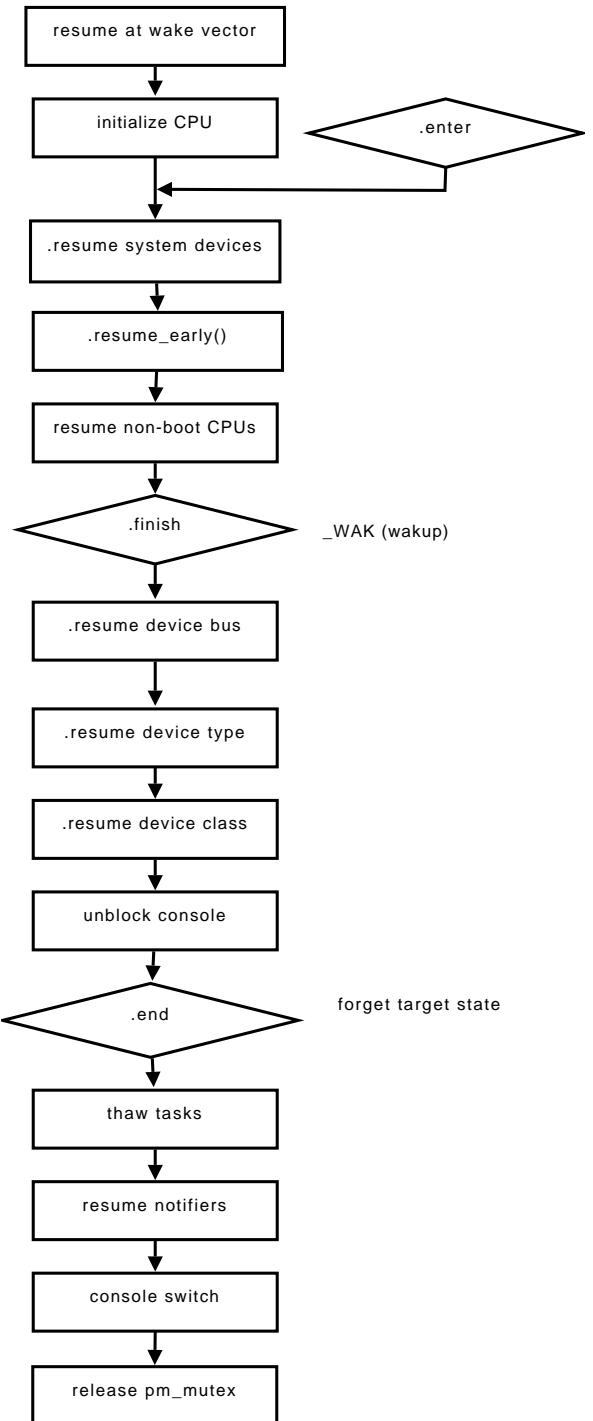


Figure 6: Resume Sequence

⁷For example, protected mode on an *i386* PC or 64-bit mode on an *x86-64* system.

drivers. Although these methods may return error codes, the PM core cannot really do anything about resume errors; the codes are used for debugging purposes only. Devices are resumed in order of registration, the reverse of the order in which they were suspended.

Once devices have been resumed, the PM core unblocks the system console so that diagnostic messages can be printed, and calls the `.end()` platform method. This method is responsible for doing the final platform-resume cleanup. In particular, it assures that the information about the target sleep state of the system stored by `.begin()` has been discarded by the platform driver.

The last three steps of resume are the thawing of tasks, invoking suspend notifiers with the appropriate argument (`PM_POST_SUSPEND`), and switching the system console back to whatever terminal it had been set to before the suspend started. Finally, the PM core releases `pm_mutex`.

7 Suspend and Hibernation Notifiers

Suspend and hibernation notifiers are available for subsystems that need to perform some preparations before tasks are frozen (see Section 8). For example, if a device driver needs to call `request_firmware()` before a suspend, that should be done from within a suspend notifier.

The notifiers are registered and un-registered using the `register_pm_notifier()` and `unregister_pm_notifier()` functions, respectively. Both these functions take one argument, a pointer to an appropriately populated `struct notifier_block`. If there is no need to un-register a suspend notifier, it can be registered with the help of the simplifying macro `pm_notifier()`, which takes only a function name and a priority as arguments.

The notifiers are called just prior to freezing tasks during both suspend and hibernation with their second argument set to `PM_SUSPEND_PREPARE` or `PM_HIBERNATION_PREPARE`, respectively,⁸ as well as during resume from a sleep state or hibernation with the second argument equal to `PM_POST_SUSPEND` or `PM_POST_HIBERNATION`, respectively. In addition,

⁸They also are called during resume from hibernation with `PM_RESTORE_PREPARE`, but we will not discuss that here.

they are called if suspend or hibernation fails. The PM core does not distinguish these invocations from the calls made during a successful resume; for this reason, the notifier code should be prepared to detect and handle any potential errors resulting from a suspend failure. Regardless, the rule is that if the notifiers were called with `PM_SUSPEND_PREPARE` during suspend, then they are called with `PM_POST_SUSPEND` to undo the changes introduced by the previous invocation, either during resume or in a suspend error path.

Notifiers return zero on success; otherwise they return appropriate error codes. However, while an error code returned by a notifier called during suspend causes the entire suspend to fail, error codes returned by notifiers called during resume are ignored by the PM core, since it is not able to act on them in any significant way.

8 Freezing Tasks

In both suspend and hibernation, tasks are frozen before devices are suspended. This assures that all user-space processes are in a stable state in which they do not hold any semaphores or mutexes, and they will not continue running until the PM core allows them. This mechanism was introduced with hibernation in mind, to prevent data from being written to disks after the hibernation image was created. Otherwise the on-disk data would not reflect the information preserved within the hibernation image, leading to corruption when the system resumed. Historically, Linux's support for hibernation has been much more robust than its support for STR, and drivers' suspend and resume callbacks were designed and tested with hibernation in mind. They generally expected tasks to be frozen before they were executed. Since the same callbacks were (and still are) used for both STR and hibernation, it became necessary to freeze tasks before STR as well as before hibernation.

The piece of code that freezes tasks is called the *frozen*. It is invoked by the PM core after the suspend notifiers are called (see Section 7) and just before the `.begin()` platform-suspend method is executed. It works by traversing the list of all tasks in the system and setting the `TIF_FREEZE` flag for the ones marked as freezable (i.e., those without the `PF_NOFREEZE` flag set).

It does this first for user-space tasks, calling `signal_`

`wake_up()` on each of them.⁹ The code uses a busy loop in which the freezer checks if there still are any tasks with `TIF_FREEZE` set. The loop finishes when there are none left, or the only remaining ones also have the `PF_FREEZER_SKIP` flag set.¹⁰

The tasks for which `TIF_FREEZE` has been set are forced by the signal handler to call `refrigerator()`. This function unsets `TIF_FREEZE`, sets the `PF_FROZEN` flag for the current task, and puts it into the `TASK_UNINTERRUPTIBLE` state. The function will keep the task in this state as long as the `PF_FROZEN` flag is set; the PM core has to reset that flag before the task can do any more useful work. Thus, the tasks that have `PF_FROZEN` set and are inside the `refrigerator()` function are regarded as “frozen.” As a result of the way in which `refrigerator()` is entered, the frozen tasks cannot hold any semaphores or mutexes, so it is generally safe to leave them in this state before suspending devices.

When all of the user-space tasks have been frozen, the freezer sets `TIF_FREEZE` for the remaining freezable tasks (i.e., freezable kernel threads). They also are supposed to enter `refrigerator()`. But while user-space tasks are made call `refrigerator()` by the generic signal-handling code, kernel threads have to call it explicitly in order to be frozen. Specifically, they must call the `try_to_freeze()` function in suitable places. Moreover, the freezer does not call `fake_signal_wake_up()` on them, since we do not want to send a fake signal to a kernel thread. Instead the freezer calls `wake_up_state(p, TASK_INTERRUPTIBLE)` on those tasks (where `p` is a pointer to the task’s `struct task_struct` object). This causes the tasks to be woken up in case they are sleeping—but it also means that kernel threads in the `TASK_UNINTERRUPTIBLE` state cannot be frozen.¹¹

Although freezing tasks may seem to be a simple mechanism, it has several problems. First of all, the main limitation of the freezer (inability to handle uninterruptible tasks) causes it to fail in many cases where we would like it to succeed. For example, if there

⁹The freezer distinguishes user-space tasks from kernel threads on the basis of the task’s `mm` pointer. If this pointer is `NULL` or has only been set temporarily, the task is regarded as a kernel thread; otherwise it is assumed to belong to user-space.

¹⁰This allows the freezer to handle some corner cases, such as the `vfork()` system call.

¹¹This also applies to user-space processes in that state.

is a task waiting on a filesystem lock in the `TASK_UNINTERRUPTIBLE` state and the lock cannot be released for a relatively long time due to a network error, the freezer will fail and the entire suspend will fail as a result. Second, the freezer does not work well with device drivers having a user-space component, because they may not be able to suspend devices after their user-space parts have been frozen. Third, freezing tasks occasionally takes too much time. It usually does not take more than several milliseconds, but in extreme cases (i.e., under a heavy load) it may take up to several seconds, which is way too much for various important usage scenarios. Finally, the approach used by the freezer to distinguish user-space processes from kernel threads is not optimal. It turns out that there are kernel threads which in fact behave like user-space processes and therefore should be frozen in the same way, by sending fake signals to them with `signal_wake_up()`. These threads often fail to call `refrigerator()` in a timely manner, causing the freezer to fail.

For these reasons, the kernel developers generally agree that the freezing of tasks should not be used during suspend. Whether it should be used during hibernation is not entirely clear, but some implementations of hibernation without freezing tasks are being actively discussed. In any case, there ought to be an alternative mechanism preventing user-space processes and kernel threads from accessing devices that are in a low-power state (i.e., after they have been suspended and before they are resumed). It is generally believed that device drivers should handle this, and for this purpose it will be necessary to rework the suspend and resume framework.

9 Proposed Framework for Suspending and Resuming Devices

As stated in Section 5, the current framework for suspending and resuming devices does not seem to be adequate. It is considered inflexible and generally difficult to use in some situations. It does not include any mechanisms allowing the PM core to protect its internal data structures from damage caused by inappropriate driver implementations.¹² It does not provide enough context information to resume callbacks. Finally, it may not be

¹²For example, if a callback or notifier routine registers a new device object below a suspended parent, the ordering of the device list used by the PM core will be incorrect and the next suspend may fail as a result.

suitable when the freezing of tasks is removed and device drivers are made responsible for preventing access to suspended devices. Consequently a new framework for suspending and resuming devices is now being introduced [PATCHES].

The first problem solved by the new framework is the lack of separation between the suspend and hibernation callbacks, especially where the resume part is concerned. Within the current framework the same device-resume callbacks are used for both hibernation and suspend, and since they take only one argument (a pointer to the device structure), it is nearly impossible for them to determine the context in which they are invoked. This is a serious limitation leading to unnecessary complications in some cases, and it is going to be fixed by introducing separate device-resume callbacks for suspend and hibernation.¹³ Likewise, separate device-suspend callbacks for suspend and hibernation will be introduced, so that the `pm_message_t` argument (used for determining the type of transition being carried out, see Section 5) will not be necessary any more.

```
struct pm_ops {
    int (*prepare)(struct device *dev);
    void (*complete)(struct device *dev);
    int (*suspend)(struct device *dev);
    int (*resume)(struct device *dev);
    int (*freeze)(struct device *dev);
    int (*thaw)(struct device *dev);
    int (*poweroff)(struct device *dev);
    int (*restore)(struct device *dev);
};
```

Figure 7: Proposed `struct pm_ops`

`struct pm_ops`, representing a set of device-suspend and -resume callbacks (including hibernation-specific callbacks), is defined as shown in Figure 7. Each device-class or device-type driver implementing these callbacks will provide the PM core with a pointer to one of these structures. Since bus-type drivers generally need to define special device-suspend and -resume callbacks to be executed with interrupts disabled, the extended `struct pm_ext_ops` structure detailed in Figure 8 is provided for their benefit.

Although the implementation of suspend and resume callbacks in device drivers may generally depend on the

¹³In fact, two separate device-resume callbacks are necessary for hibernation: one to be called after creating an image and one to be called during the actual resume.

```
struct pm_ext_ops {
    struct pm_ops base;
    int (*suspend_noirq)(struct device *dev);
    int (*resume_noirq)(struct device *dev);
    int (*freeze_noirq)(struct device *dev);
    int (*thaw_noirq)(struct device *dev);
    int (*poweroff_noirq)(struct device *dev);
    int (*restore_noirq)(struct device *dev);
};
```

Figure 8: Proposed `struct pm_ext_ops`

bus type, device type, and device class their devices belong to, it is strongly recommended to use `struct pm_ops` or `struct pm_ext_ops` objects. However, the legacy callback method pointers will remain available for the time being.

The majority of callbacks provided by `struct pm_ops` and `struct pm_ext_ops` objects are hibernation-specific and we will not discuss them. We will focus on the callbacks that are STR-specific or common to both suspend and hibernation.

The `.prepare()` callback is intended for initial preparation of the driver for a power transition, without changing the hardware state of the device. Among other things, `.prepare()` should ensure that after it returns, no new children will be registered below the device. (*Un*-registering children is allowed at any time.) It is also recommended that `.prepare()` take steps to prevent potential race conditions between the suspend thread and any other threads. The `.prepare()` callbacks will be executed by the PM core for all devices before the `.suspend()` callback is invoked for any of them, so device drivers may generally assume that the other devices are functional while `.prepare()` is being run.¹⁴ In particular, `GFP_KERNEL` memory allocations can safely be made. The `.prepare()` callbacks will be executed during suspend as well as during hibernation.¹⁵

The `.suspend()` callback is suspend-specific. It will be executed before the platform `.prepare()` method is called (see Section 5) and the non-boot CPUs are disabled. In this callback the device should be put

¹⁴However, user-space tasks will already be frozen, meaning that things like `request_firmware()` cannot be used. This limitation may be lifted in the future.

¹⁵During hibernation they will be executed before the image is created, and during resume from hibernation they will be executed before the contents of system memory are restored from the image.

into the appropriate low-power state and the device's wake-up mechanism should be enabled if necessary. Tasks must be prevented from accessing the device after `.suspend()` has run; attempts to do so must block until `.resume()` is called.

Some drivers will need to implement the `.suspend_noirq()` callback and its resume counterpart, `.resume_noirq()`. The role of these callbacks is to switch off and on, respectively, devices that are necessary for executing the platform methods `.prepare()` and `.finish()` or for disabling and re-enabling the non-boot CPUs. They should also be used for devices that cannot be suspended with interrupts enabled, such as APICs.¹⁶

The `.resume()` callback is the counterpart of `.suspend()`. It should put the device back into an operational state, according to the information saved in memory by the preceding `.suspend()`. After `.resume()` has run, the device driver starts working again, responding to hardware events and software requests.

The role of `.complete()` is to undo the changes made by the preceding `.prepare()`. In particular, new child devices that were plugged in while the system was suspended and detected during `.resume()` should not be registered until `.complete()` is called. It will be executed for all kinds of resume transitions, including resume-from-hibernation, as well as in cases when a suspend or hibernation transition fails. After `.complete()` has run, the device is regarded as fully functional by the PM core and its driver should handle all requests as appropriate. The `.complete()` callbacks for all devices will be executed after the last `.resume()` callback has returned, so drivers may generally assume the other devices to be functional while `.complete()` is being executed.

All of the callbacks described above except for `.complete()` return zero on success or a nonzero error code on failure. If `.prepare()`, `.suspend()`, or `.suspend_noirq()` returns an error code, the entire transition will be aborted. However the PM core is not able to handle errors returned by `.resume()` or `.resume_noirq()` in any meaningful manner, so

¹⁶At present, APICs are represented by `sysdev` objects and are suspended after the regular devices. It is possible, however, that they will be represented by platform device objects in the future.

they will only be printed to the system logs.¹⁷

It is expected that the addition of the `.prepare()` and `.complete()` callbacks will improve the flexibility of the suspend and resume framework. Most importantly, these callbacks will make it possible to separate preliminary actions that may depend on the other devices being accessible from the actions needed to stop the device and put it into a low-power state. They will also help to avoid some synchronization-related problems that can arise when the freezing of tasks is removed from the suspend code path. For example, drivers may use `.prepare()` to disable their user-space interfaces, such as `iocfts` and `sysfs` attributes, or put them into a degraded mode of operation, so that processes accessing the device cannot disturb the suspend thread.

Moreover, we expect that the introduction of the hibernation-specific callbacks and the elimination of the `pm_message_t` parameter will help driver authors to write more efficient power-management code. Since all of the callbacks related to suspend and hibernation are now going to be more specialized and the context in which they are invoked is going to be clearly defined, it should be easier to decide what operations are to be performed by a given callback and to avoid doing unnecessary things (such as putting a device into a low-power state before the hibernation image is created).

10 Suspend to RAM and Graphics Adapters

One of the most visible weaknesses of Linux's current implementation of suspend-to-RAM is the handling of graphics adapters. On many systems, after resume-from-RAM, the computer's graphics adapter is not functional or does not behave correctly. In the most extreme cases this may lead to system hangs during resume and to the appearance of many unusual failure modes. It is related to the fact that the way Linux handles graphics does not meet the expectations of hardware manufacturers.¹⁸

For a long time graphics has been handled entirely by the X server, which from the kernel's point of view is

¹⁷To change this, the resume callbacks would have to be required to return error codes *only* in case of a critical failure. This currently is not possible, since some drivers return noncritical errors from their legacy resume callbacks. In any event, drivers have a better idea of what recovery options are feasible than the PM core does.

¹⁸This mostly applies to the vendors of notebooks.

simply a user-space process. Usually the X server uses its own graphics driver and accesses the registers of the graphics adapter directly. In such cases the kernel does not need to provide its own driver as well, and the X server is left in control. Normally this does not lead to any problems, but unfortunately with suspend-to-RAM it does.

When the system is put into STR, power is usually removed from the graphics adapter, causing it to forget its pre-suspend settings. Hence during the subsequent resume, it is necessary to reinitialize the graphics adapter so that it can operate normally. This may be done by the computer's BIOS (which gets control over the system when a wake-up event occurs) and it often is done that way on desktop systems. However, many laptop vendors tend to simplify their BIOSes by not implementing this reinitialization, because they expect the graphics driver provided by the operating system to take care of it. Of course this is not going to work on Linux systems where the kernel does not provide a graphics driver, because the X server is activated after devices have been resumed and that may be too late for it to reinitialize the graphics adapter, let alone restore its pre-suspend state. Furthermore X may not even have been running when the system was suspended.

The ultimate solution to this problem is to implement graphics drivers in the Linux kernel. At a minimum, graphics drivers should be split into two parts, one of which will reside in the kernel and will be responsible for interacting with the other kernel subsystems and for handling events such as a system-wide power transition. The other part of the driver may still live in the X server and may communicate with the first part via a well-defined interface. Although this idea is not new, it was difficult to realize owing to the lack of documentation for the most popular graphics adapters. Recently this situation has started to change, with first Intel and then AMD making their adapters' technical documentation available to kernel and X developers. As a result, `.suspend()` and `.resume()` callbacks have been implemented in the `i915` driver for Intel adapters, and it is now supposed to correctly reinitialize the adapter during resume-from-RAM.¹⁹ It is expected that this ability will also be added to the graphics drivers for AMD/ATI adapters in the near future.

Still, there are many systems for which the graphics

adapters are not reinitialized correctly during resume-from-RAM. Fortunately it was observed that the reinitialization could often be handled by a user-space wrapper executing some special BIOS code in 16-bit emulation mode. It turned out that even more things could be done in user-space to bring graphics adapters back to life during resume, and a utility called `vbetool` was created for this purpose.²⁰ At the same time, a utility for manipulating backlight on systems using ATI²¹ graphics adapters, called `radeontool`, was created. These two programs were merged into a single utility called `s2ram`, which is a wrapper around the Linux kernel's `/sys/power/state` interface incorporating the graphics reinitialization schemes.²²

Although `s2ram` is a very useful tool, it has one drawback: Different systems usually require different operations to be carried out in order to reinitialize their graphics adapters, and it is necessary to instruct `s2ram` what to do by means of command-line options. Moreover, every user has to figure out which options will work on her or his system, which often is tedious and can involve several failed suspend/resume cycles. For this reason `s2ram` contains a list of systems for which a working set of options is known, and the users of these systems should be able to suspend and resume their computers successfully using `s2ram` without any additional effort.²³

11 Problems with Platforms

The flexibility given to platform designers by ACPI can be a source of serious problems. For example, if the ACPI Machine Language (AML) routines invoked before suspend are not implemented correctly or make unreasonable assumptions, their execution may fail and leave the system in an inconsistent state. Unfortunately the kernel has no choice but to execute the AML code, trusting that it will not do any harm. Of course if given platform is known to have problems, it can be blacklisted on the basis of its DMI²⁴ identification. Still, be-

²⁰`vbetool` was written by Matthew Garrett.

²¹ATI was not a part of AMD at that time.

²²The creator of `s2ram` is Pavel Machek, but many people have contributed to it since the first version was put together. `s2ram` is available from <http://suspend.sf.net>.

²³`s2ram` matches computers against its list of known working systems based on the DMI information in the BIOS. The list is built from feedback provided by the `s2ram` users and is maintained by Stefan Seyfried.

²⁴Desktop Management Interface.

¹⁹This driver is included in the 2.6.25 kernel.

fore blacklisting a system, the kernel developers have to know what kind of problems it experiences and what exactly to do to prevent them from happening. That, in turn, requires the users of those systems to report the problems and to take part in finding appropriate workarounds.

Moreover, problems may arise even if there is nothing wrong with the AML code. This happens, for example, if the kernel provides native drivers for devices that are also accessed from the AML routines, because the kernel has no means to determine which registers of a given device will be accessed by the AML code before actually executing that code. Thus, if the native driver accesses the device concurrently with an AML routine, some synchronization issues are likely to appear. It is difficult to solve those issues in a way general enough to be applicable to all systems and, again, blacklisting is necessary to make things work on the affected platforms.

Another major inconvenience related to ACPI platforms is that the requirements regarding STR changed between revisions of the ACPI specification. The suggested code ordering for suspend changed between ACPI 1.0 and ACPI 2.0, and there are systems for which only one of them is appropriate. Some of these systems fail to suspend or even crash if the code ordering implemented by the kernel is not the one they expect. Again, the kernel has no choice but to use one suspend code ordering by default and blacklist the systems that require the other one.²⁵

Last but not least, testing the interactions between the kernel and a particular platform is problematic because it can be carried out on only a limited number of systems. Even if the kernel follows the ACPI specification and works on the systems available to its developers, it may very well fail to work on other systems having different implementations of the AML code in question. For this reason, it is very important that the users of STR test development kernels and immediately report any new STR-related problems, so that the developers can investigate and fix them before the new code is officially released.

12 Future Work

We've reached a level of stability where STR is useful on a large number of systems. We need to continue these

²⁵At present, the ACPI 2.0 ordering is used by default.

stability efforts with the goal of broad, and ultimately universal, deployment.

But to make STR even more useful, we need to increase our focus on performance. We need tools to track STR performance such that performance is easily and widely measured, issues are easily identified, regressions are prevented, and benefits of tuning are permanent.

Linux needs a stable user/system API for device D-state control. D-states should be widely available to run-time device power management, which must inter-operate well with system sleep states. (Some progress in this area has already been made; a few subsystems, such as USB, can power down devices when they are not in active use.)

The wake-up API available in `sysfs` needs to be more widely used and better integrated with the platform wake-up mechanisms.

It is unclear that the existing CPU hot-plug infrastructure is ideally suited to system suspend, and alternatives should be sought.

We need to think about the work required for device drivers to properly implement suspend and make sure that the burden on driver authors is minimized. This applies to the API seen by individual device drivers as well as to the infrastructure provided by the upper-level device-class drivers.

13 How to Participate

STR in Linux is approaching a point where a critical mass of developers routinely use it, and thus test it. These developers often run the development and release-candidate kernels and thus immediately notice and report regressions. With tools such as `git-bisect` [GIT-OLS, GIT-URL], these developers are empowered to do an excellent job isolating issues, even if they never read or modify a single line of suspend-related code.

Please join them! For STR on Linux to reach the next level of success, it is critical that the Linux community assert that STR work properly on the systems that they have, and actively file bugs and help isolate regressions when STR does not meet expectations. The more active testers we have, the easier it will be for the community to successfully deploy STR on a broad range of systems.

Also, note that there is now a dedicated kernel Bugzilla category for STR related issues: <http://bugzilla.kernel.org>, product *Power Management*, and Component *Hibernation/Suspend*.

Still another group in the community must be mobilized—driver authors. At this point, drivers are expected to include full suspend/resume support if they are used on systems that support suspend. A new driver should not be considered complete enough for inclusion in the kernel if it does not include suspend support.

14 Acknowledgments

Linux’s suspend-to-RAM support is not a new idea; it has been evolving for years. We must all acknowledge that we are standing on the shoulders of the giants who came before us, and thank them for all they have done.

In particular, the authors would like to single out Pavel Machek of Novell/SuSE, co-maintainer of suspend and hibernation, who has been key to development and adoption. Also Andy Grover and Patrick Mochel, who set a lot of the foundation starting way back in Linux-2.4.

We thank Alan Stern for many valuable suggestions and for helping us to prepare the manuscript. We also thank Pavel Machek for valuable comments.

Finally, we thank the communities on the mailing lists `linux-pm@lists.linux-foundation.org` and `linux-acpi@vger.kernel.org`, where the actual work gets done.

References

[ACPI-SPEC] *Advanced Configuration and Power Interface Specification*
[\(http://www.acpi.info\).](http://www.acpi.info)

[ACPICA] ACPICA project home page
[\(http://acpica.org\).](http://acpica.org)

[PCI-PM] *PCI Bus Power Management Interface Specification* (<http://www.pcisig.com/specifications/conventional/>).

[ACPI-OLS] Brown, Keshavamurthy, Li, Moore, Pallipadi, Yu; *ACPI in Linux—Architecture, Advances, and Challenges*; In *Proceedings of the Linux Symposium* (Ottawa, Ontario, Canada, July 2005).

[ACPI-URL] Linux/ACPI project home page
[\(http://www.lesswatts.org/projects/acpi\).](http://www.lesswatts.org/projects/acpi)

[GIT-URL] GIT project home page
[\(http://git.or.cz\).](http://git.or.cz)

[GIT-OLS] J.C. Hamano, *GIT—A Stupid Content Tracker*, In *Proceedings of the Linux Symposium* (Ottawa, Ontario, Canada, July 2006).

[REPORT] R.J. Wysocki, *Suspend and Hibernation Status Report*
[\(http://lwn.net/Articles/243404\).](http://lwn.net/Articles/243404)

[PATCHES] R.J. Wysocki, *Separating Suspend and Hibernation* (http://kerneltrap.org/Linux/Separating_Suspend_and_Hibernation).

Systems Monitoring Shootout

Finding your way in the Maze of Monitoring tools

Kris Buytaert

Inuits

Kris.Buytaert@inuits.be

Frederic Descamps

Inuits

Frederic.Descamps@inuits.be

Tom De Cooman

Inuits

Tom.DeCooman@inuits.be

Bart Verwilst

Inuits

Bart.Verwilst@inuits.be

Abstract

The open source market is getting overcrowded with different Network Monitoring solutions, and not without reason: monitoring your infrastructure is becoming more important each day. You have to know what's going on for your boss, your customers, and for yourself.

Nagios started the evolution, but today OpenNMS, Zabbix, Zenoss, GroundWorks, Hyperic, and different others are showing up in the market.

Do you want light-weight, or feature-full? How far do you want to go with your monitoring, just on an OS level, or do you want to dig into your applications, do you want to know how many query per seconds your MySQL database is serving, or do you want to know about the internal state of your JBoss, or be alerted if the OOM killer will start working soon?

This paper will provide guidance on the different alternatives, based on our experiences in the field. We will be looking both at alerting and trending and how easy or difficult it is to deploy such an environment.

1 Some Definitions

The monitoring business has its own set of terms, which we will gladly explain.

How do you want your monitoring system being “served?” A light version? Full-blown and feature-full? The most important question is: how far do you want to go with your monitoring? Just on the OS level, or do you want to dig into your applications, do you want to know how many query per seconds your MySQL database is

serving, do you want to know about the internal state of your JBoss, or be triggered if the OOM killer will start working soon? ... As you see, there are several ways of monitoring depending on the level of detail.

In our monitoring tool, we add hosts. This host can be any device we would like to monitor. Next we need to define what parameter on the host we would like to check, how we are going to get the data, and at which point we'd consider the values not within normal limits anymore. The result is called a *check*. There are several ways to ‘get’ the required data. Most monitoring tools can use SNMP as a way to gather the required data. Either the tool itself performs an SNMP-get, or it receives data via an SNMP-trap. A lot of tools can also work with external scripts that can be ‘plugged in.’ Most of the time you can use a script in whichever language you like (bash, perl, C, java, etc.), as long as it sticks to certain rules set by the monitoring tool. These rules make sure the tool can understand the data returned. Checks that are performed by the monitoring tool itself are called active checks. The monitoring tool ‘polls’ another device to get some data out of it. Checks performed and submitted by external applications are called passive checks. Passive checks are useful for monitoring hosts and services on that host that are, e.g., not directly reachable for the monitoring server or where no direct check is possible or available. An SNMP Trap can be implemented in a monitoring solution via a passive check.

Alerting is the way to send a warning signal. Usually this is an automatic signal warning of a danger. In monitoring services, alerts can be sent via different methods when available: an email is sent to one or many users with the warning message and the result of the check

having a problem. A message is sent on the mobile of one or many users with the description of the problem. The monitoring interface highlights the problem in a user-friendly way, usually with colors depending on the severity level of the problem. We obviously also want Instant Messaging to be used. All the alerts are related to services that the monitoring system must check. The level and the signal of the alerts can be setup independently by services. All good monitoring systems are able to send different notifications depending on the time frame. So for example during the night only the on-call person will be notified via SMS of the problem.

The data the monitoring tool is gathering is also stored for historical reference. This can be used to generate reports, so one can view the status detail for a specific host or service over a certain period. General availability can be checked and reviewed. Custom availability reports can be generated depending on the monitoring tools capabilities. E.g., the uptime of a certain host or service. How many times it went down, for how long, also providing a time related percentage of downtime or uptime. We then can report a summary of all events and status linked to the services we are monitoring.

Next to reports themselves, some tools also provide a way to chart the gathered data. Although monitoring and trending are actually two different businesses, some tools are combining them into a single interface. The capabilities of the monitoring tool might be enough to suite your historical data trending needs, still there are other specific tools available which are pretty good in performing only the trending task. These standalone tools mostly provide more options, or might provide an easier way to accomplish certain tasks than the monitoring tools.

Trending and reporting can definitely help in perfecting/fine-tuning the monitoring task. Using trends we can adjust already implemented monitoring rules to situations noticed in the trends/reports. Checks can be adjusted to certain situations. We can disable a check during a certain time period, or we could loosen the limits a check is set to trigger on during this time interval.... In other words, it can help to make a check more accurate. So in the end to establish a more accurate monitoring system, which also gives us the ability to notice upcoming issues and react on them, making the business of system administration more proactive.

Trending is reading of the variation in the measure-

ment of data over several intervals. In your environment you are interested in how many queries your databases parses per second, and how this evolves over time. You are also interested in figuring out how many traffic passes over a certain switch port and if you have enough bandwidth capacity left. Trending therefore is not so much related to the uptime of certain services but to the behavior of the service itself.

2 What are we looking for?

We started out asking ourselves what we want in a network and infrastructure monitoring. Some interesting topics came up. We look at an NMS from different viewpoints, from a user viewpoint, from a sysadmin viewpoint, and also from a developer viewpoint.

Note that we are experienced systems people, but often new to the tools, and that's probably the way you want it. We want to know how fast you can get up to speed with a tool while not having to spent hours and days tuning the platform, or even read manuals.

Monitoring tools are typically set up by System Administrators or Infrastructure Architects and then handed over to either Junior staff or Operations people. That means that once it is up and running, everybody should be able to use it. A good and clean GUI is a requirement.

However, as you are going to add over 100 servers at once to a monitoring application, you do not want to use a web-interface for that process. You want to be able to write a simple for-loop for a variety of servers to create config-files, database-scripts, adjust them, and add them. In a later stage, you want some administrators to add hosts to the system through some GUI, but with a template-based system so they can reuse what you prepared.

When introducing a monitoring tool in a large environment, that also means that you are rather reluctant to install a daemon on all these hosts—unless you can fully automate that installation.

SNMP is a great tool to manage/monitor just about anything in your network. You can call this an advantage because SNMP is a package available in every OS, does not have many dependencies prior to installing, it is easy to configure for simple setups, and most of all: quick result! (The ability to add SNMP-scripts of course is a big plus.) However, unlike the name tells us, SNMP

is everything but simple and requires in-depth thinking of how to provision our SNMP configuration at all the hosts, so configuring SNMP might be as difficult as adding a client specific to your monitoring, too.

We want to be able to automate as much as possible, so we prefer a config-file-based approach. It doesn't matter whether it has its own syntax, as long as it is still human-readable. A web interface of course is also a must; however, it should be supportive, not enforcing. You should be able to create a configuration that the webinterface doesn't overrule. We have to realize that performing an action on 3 hosts via GUI is feasible, but on 300, it isn't. So we would rather settle for less GUI features that we never use.

We all have to please managers and customers, so we want a tool that is capable of creating very clear and complete reports and graphs. Ideally graphs that can be mapped against each other to pinpoint concurrent events, etc.

It goes without saying that a monitoring tool should be stable. As the tool is monitoring events in our infrastructure, it needs to be up and running all the time in order to be able to escalate issues. It also needs to be able to scale; infrastructures grow at a terrific speed these days, so the monitoring tool you are starting out with today must be capable of either delegating groups of machines or manage them itself.

When looking at the resources used in your infrastructure, you are interested in memory usage, IO performance, etc. So you want your monitoring tool to be almost invisible. We mentioned before that ideally no plugins have to be installed, but if they have to be installed, we want them to be lightweight. A JVM requiring a big chunk of your precious RAM just to see if a process is still running might be overkill. A monitoring tool should blend in with the infrastructure, not add more requirements to it.

A good Monitoring Platform has capabilities of both telling us how a machine performed in the past, how much time a certain service was available and how long it wasn't. We want to see trends in usage—e.g., whether disk and memory usage have been changing over time.

Ideally you also get a separate status page, no technical data, but a clean status page along with persons to contact in case of problems.

A good monitoring tool has plenty of alternative notification methods. SMS and mail are primary methods, but Instant Messaging solutions are more and more a standard requirement, too. And you want to be able to configure these notifications, select different methods based on the time, how critical an incident is, and so on. Also you don't want to be spammed by the tool, you want relevant escalation when appropriate, but you don't want a message flooding your inbox that you will filter away anyhow.

A great framework is a good start, but without an active community it is probably as bad as a commercial tool that also requires you to buy a lot of extra features. You need a plugin model that allows you to add checks and functionalities in different languages: PHP, bash, perl, C, etc. So having a clear and powerful API is a must. That way your community will be able to write new checks for services that you either don't know yet or haven't heard about yet.

3 Nagios

Nagios is considered by many to be the godfather of host and service monitoring on Linux. It is without doubt the best known monitoring tool out there, and is available by default in all major Linux distributions. Nagios version 3.0 was released in March 2008, but will only be picked up by most distributions in the next 6-12 months. In the meantime, version 2.11 is the most common in today's distributions. Nagios was created by Ethan Galstad and is licensed under the GPL v2. Many others contributed to Nagios since its conception through plugins, add-ons, bugfixes, suggestions, testing, bug reporting, ideas, and feature requests. Where as Nagios was incepted in 1999, Ethan only recently started Nagios Enterprises to commercially support the Open Source project.

One of the greatest features of Nagios is the great scriptability of its configuration. Nagios is fully configured with text files, which can easily be generated on the fly to perfectly fit into your network. Everything from hosts, services, contacts to groups and alert escalation plans can be configured in this way. Configuring Nagios comes with a pretty steep learning curve, though, for first-time users. At frequent intervals the Nagios monitoring deamon will run checks on hosts and services you specify via a mechanism of external plugins that then return status information.

Problems can be reported to the administrators by means of SMS, email, instant messages, or a variety of other ways. Escalation is also supported, so that when the first contact doesn't acknowledge the problem within a predefined time frame, another person or group can be notified in order to get the problem resolved. Nagios works by running predefined checks on a configurable interval through a plethora of external plugins that return status information to the Nagios service itself. Since Nagios is solely an alerting tool, and lacks a fancy GUI with graphs, trending, and other monitoring features, it is usually accompanied by a separate tool that handles those features, such as Cacti. The Nagios/Cacti combo is the most popular one. Several independent projects also try to improve and extend the pretty rudimentary and boring looks of the Nagios web interface by implementing a new web-based frontend on top of the Nagios data and configuration, thus merging the reporting and monitoring/trending features into one single rich web interface.

Nagios by itself is pretty lightweight, with a C-based backend and a cgi-based frontend (web GUI), and requires hardly any dependencies to get it in a working state, and can be used on a very modest (virtual) machine.

4 GroundWorks Open Source

A couple of months ago we decided to testrun the GroundWorks Open Source 1.6 RPM. We didn't get far as we immediately had to with our efforts, because GroundWorks decided that their RPM should modify our httpd initscripts and point to its own httpd installation in `/usr/local/groundworks`, hence breaking other tools we had on the platform.

So this was our second encounter with the GroundWorks platform. GroundWorks expects you to untar a tarball, then run its installation script which comes to ask you if it can install a bunch of RPMs. Or you can just try to install these RPMs manually, which fails as the RPM expects you to have set your JAVA home. I've never seen an RPM that failed in the preinstall because of lacking environment settings, and this obviously shouldn't happen as people expect to be able to install RPMs from a repository during boot time, obviously lacking those environment variables.

So we opted for the installer. It requires SELinux to be disabled, complains about lack of memory (it expects

one GB and I only have 256MB). It also tells me I need 40GB of disk space rather than telling me how much disk space I really need, it explains on failing me with 200MB short on my rootfs. The installer failed multiple times on me, first telling me I need a different java-1.5.06 java version than the java-1.5.06 version I had installed.

The installer knows that mysql isn't started, but doesn't try starting it itself—it tells me to start it before continuing, which I do in another console, and then it aborts because mysql isn't started. Error handling seems to be a feature for the next installer version as the installer today fails on you without errors. Although the underlying RPM installation process gave clean and clear errors, the GroundWorks Monitor 5.2 failed to capture them, not even in its logfile.

So we were pretty disappointed with the install process. Sadly GroundWorks still didn't fix the Apache problem mentioned before, either.

As we are setting up different test platforms in a remote isolated lab, we often tunnel port 80 of the monitoring tool to another port on our own machine. GroundWorks was the first tool that complained about having tunneled its port 80 to my local port 8888, and insisted on refreshing itself to `localhost : 80`—hence a totally different environment, although a problem I could solve. In this case, you are often forced to tunnel your monitoring tool and map it on different ports, and a web application should be unaware of that.

The next problem was the documentation. The installer process points you to a Bookshelf in the application, a Bookshelf you can't access unless you are logged on to the system, which doesn't work, as you never got any information about a default username or password combination.

After 3 days, the forum came back with the obvious `admin:admin` answer that didn't work for me at first. It worked now.

The auto discovery seemed to work only partly, it found some hosts, but it seems it doesn't update the Nagios config files, so I couldn't figure out how to get them in the Nagios overview.

For a Nagios-based tool, you would expect a good and easy installation procedure and great documentation. You'd expect to build around a great tool and make it better.

5 Zenoss

Bill Karpovich, Mark Hinkle, and Erik Dahl are starting to become regular names in the Open Management industry; they bring us Zenoss. Zenoss gives you a single, integrated solution for monitoring your entire infrastructure—network, servers, and applications. They claim to support inventory, configuration, availability, performance, and events of your services. Zenoss comes in a Free community edition and a different commercially supported version. Its Free version includes Availability Monitoring, Performance Monitoring, Event Management, and core reporting functionality.

Zenoss likes to compare itself to both proprietary and open source tools, claiming that unlike the others it is both easy to install and configure, and it's affordable. It is more open, brings no vendor lock-in and has better community collaboration.

The Zenoss architecture breaks down into three parts. A user part with the WebConsole/Portal, a Data Layer (where all the data lives), and the Process Layer that collects the data via standard protocols. Zenoss is one integrated package, not some different packages glued together into a bigger whole. You can configure templates and map instances to those templates.

In the data layer, Zenoss uses three places to store its data, its CMDB (Configuration Management DataBase) is an object model stored in Zope (ZODB). It is obvious that for historical data they use RRDTTool, and the events are being stored in a MySQL database. A nice mix-and-match to store everything they need.

The actual work is done by a series of daemons and control services that provide node discovery, configuration modeling, availability monitoring, performance monitoring, event collection, and automated responses. Each of these services can run as a single local instance or be distributed, hence providing a scalable solution. Zendisc stands for the device discovery; Zenmodeller is used to get configuration details and map resources to the resource model; whereas ZenWinmodeler uses WMI to discover windows-based services. Different plugins such as ZenPin, ZenStatus, ZenPerfSNMP, and ZenProcess are used to check services. ZenSyslog, ZenEventlog, and ZenTrap are used to collect events, and as the names already show, they respectively collect syslog events, WMI events, and SNMP traps. And let's not

forget ZenActions, which is responsible for notifications and automated action scripts.

Zenoss is heavily based on Zope and Python for its web framework, which classifies it as a rather lightweight platform.

Zenoss has a prebuild VMware Appliance; RHEL, Fedora, and CentOS packages; and source tarballs available for SuSE, Debian/Ubuntu, Gentoo, OSX, FreeBSD, and Solaris.

We downloaded the zenoss-2.1.3-0.el5 RPM from SF.net.

Upon initial startup, it populates the database and starts building parts of itself. It almost performs a clean install in /opt, although different files in its directory don't belong to the package ;)

From there it is on to the webbrowser to use the web-GUI to configure everything. Its autodiscovery functionality uses SNMP. So any host with SNMP will be autodetected. Others will remain unknown. Autodiscovery seems to work, but only partially; it detects all of the devices that use SNMP, but fails to recognize details as configured in our snmpd config (process lists, diskspace usage, etc.). We flooded our root partition on one of the machines on purpose; snmpd was configured to start alerting at 10% left. Zenoss failed to recognize that. Shouldn't we expect this to be working? In the Main Views, Zenoss has a nice Ajax host relation diagram, it realizes hosts are connected to different networks and maps them out—really nice feature.

6 OpenNMS

‘OpenNMS: professional software, amateur marketing.’ That's their slogan, and it is a good reflection of what they stand for. Their site is mostly technical documentation, no fluff on how good they are how many features they have, just plain and correct facts.

OpenNMS is one of the older Open NMS platforms around. Back when they started out, Nagios was the lean and mean monitoring tool and OpenNMS was the Enterprise Grade platform that would take on HP OpenView, IBM Tivoli, and other proprietary monitoring tools.

In 2001 the choice was easy, you either had Nagios or OpenNMS, if you had SNMP and weren't afraid of deploying a J2EE appserver, you went for OpenNMS; otherwise, Nagios. Today things have changed :)

An OpenNMS instance can watch a large number of nodes from a single server, with a minimal amount of configuration and reconfiguration work needed. In an OpenNMS platform, you can define flexible rules to specify how often and when certain devices are polled, to whom different alerts are sent, and so on.

The basic element that OpenNMS monitors—an interface—is uniquely identified by an IP address. Services are mapped to those interfaces, and a number of interfaces on the same devices can be mapped together to a *node*, in OpenNMS terminology. OpenNMS was one of the first tools around to support autodiscovery. OpenNMS first polls a device; it tries to connect to an IP address as defined in the range, then uses SNMP to collect data.

Events are the core of a Network Monitoring system. OpenNMS has a daemon running called `eventd`. It makes the distinction between two types of events: those that are generated by OpenNMS itself, and those that are generated via SNMP traps. Events trigger actions such as logging a message, triggering an automatic action via an external script, or triggering the notification system.

Notifications can be sent to users or groups as defined in OpenNMS. One can configure delays, escalations, and email addresses to send the alerts to.

Our CentOS testbed was fairly pleased with some good installation documentation on how to install OpenNMS using yum. OpenNMS has its own repository, [yum.opennms.org](http://opennms.org).

7 Zabbix

Zabbix is a network management platform created by Alexei Vladishev. It is designed to monitor and track the status of various network services, servers, and other network hardware. Zabbix has a mission: “To create a superior monitoring solution available and affordable for all.” Zabbix was released for the first time in 2001, and the Zabbix company was founded in 2005 in Riga, Latvia.

Zabbix has three main parts: the daemon, the agent, and the web interface. The daemon collects all data from the agents and populates the database. The independent web interface then parses that data from the database and provides the users with an overview of what’s happening.

It uses MySQL, PostgreSQL, SQLite, or Oracle to store data. Its web-based frontend is written in PHP. Zabbix offers several monitoring options. Simple checks can verify the availability and responsiveness of standard services such as SMTP or HTTP without installing any software on the monitored host. A Zabbix agent can also be installed on UNIX and Windows hosts to monitor statistics such as CPU load, network utilization, disk space, etc. As an alternative to installing an agent on hosts, Zabbix also includes support for monitoring via SNMP.

As Zabbix uses a database to store its data, this also involves some extra configuration steps. Installing via the system’s package manager will be straightforward, as dependencies will be resolved, too. The Zabbix package contains the MySQL (or other) tables to be imported in the database. As you install Zabbix on a server, you’ll probably want the Server and the Webserver installed. Note that you don’t really need to place them on the same machine, but of course, you can. The default user is `admin`, with no password assigned.

On the Zabbix-server side, all configuration is done using the web interface. For initial setup, connect as user `admin`. The most important part is *Configuration* in the top-most menu.

In the configuration part, a lot of things can be configured or customized—from adding hosts to customizing the look and feel of the web interface itself.

We’ll start with some general Zabbix-info. Simple example: we have a host running an FTP server; when the FTP service goes down, we’d like to be alerted. As with probably all kinds of monitoring applications, you first need to add a host. (More on templates later, let’s just assume we defined a host with a name and an IP address). When this is done, an *item* for this host can be created. An item has all the data to define how a check is to be performed on the host. (Important ones: a name for the item, a check type: info about what data we want and how to get it, a check interval.) The result is that a *key* is stored for a certain host (e.g., FTP-key being 0 or 1, off or on). A simple check is used by Zabbix to monitor agentless hosts. They include ICMP, HTTP, and others. The next thing is to make sure the system notices when the FTP service goes down and does send us a notification. In Zabbix terms, this means we need to define a *trigger* and an *action*. The trigger is quite simple in this case: when the FTP is down, trigger to

YES. The trigger uses an *expression*, in which a key is evaluated. Now that we have a key and we have configured a trigger on it, we want the system to send us an email when the trigger's state changes. In Zabbix, this is done by adding an *Action*. An action has an *Event Source* (in this case, it is a trigger), a *condition*, and an *operation*. In the condition, we make the link with the correct trigger; in the operation, we simply say to send an email to a certain user. A Zabbix-template is made of several items and triggers. All templates are actually just some kind of special host-definitions, which are put into the templates-group. It does not contain any actions. So when linking a host with a template, a lot of items and triggers will be added for that host, but you will still need to define actions yourself. Most of the default templates in Zabbix are using items based on keys grabbed from the Zabbix agent. So if you want to get up and running as soon as possible, just install the Zabbix-agent on the host to be monitored. One way to add hosts is by using *Discovery*. Zabbix can *scan* a network, or a part of it, and add hosts it finds. The scan can be defined. E.g., Zabbix can check a network range for hosts that are running the Zabbix agent, or you can set other requirements like the presence of a ssh-server, etc. Based on the output of this scan, Zabbix can use an *Action* to do something with the host it has found. E.g., a discovered Linux host that is running the Zabbix-agent can directly be put in the Linux-group and the Linux Template can be assigned right away.

As mentioned earlier, Zabbix will use an action to act on a trigger. Zabbix can use different methods to send out alerts, such as email, Jabber messages, or text messages to a mobile phone. One can also define the days and hours on which a person can receive alerts.

Zabbix uses items that have a key containing data; this data can easily used to produce graphs. And Zabbix seems to be built keeping this in mind. By default, the data of items can be seen in a graph. Just go to the top-most *Monitoring* tab and select *Latest data*. Graphs viewed here are called *Simple Graphs*. Data is stored during one year by default, but this can be changed when creating an item. Zabbix can also monitor Applications. For example, an application MySQL Server may contain all items which are related to the MySQL server: availability of MySQL, disk space, processor load, transactions per second, number of slow queries, etc. Apart from the default graphs, you can also create custom graphs. With the default ones, there is always

just one item present in the graph. When you create a graph yourself, you can group multiple items in one graph.

Zabbix also provides an *Overview* page. Several of these pages can be created; they are called *Screens* in Zabbix terminology. A screen is a combination of several types of information—e.g., Simple graphs, custom created graphs, host info, trigger info, or event info. One can define what is being displayed in a screen. It could be set up in a way as to create a perfect status-page for the monitoring system.

8 Hyperic-HQ

Hyperic claims to be Open Source Web Infrastructure Monitoring and Management Software; it aims at automating your operations. Hyperic HQ is GPL, but they also offer a Silver Support package that includes low-cost support.

Hyperic has Auto-Discovery, it understands a lot of technologies over 9 different operating systems, and it is within their goals to manage everything centrally and quickly, allowing their customers to focus on serious issues. HQ collects both real-time and historical metrics from production environments including hardware, network, and application layers of your infrastructure with what they claim is no need for invasive instrumentation. Hyperic does performance tracking, alerting upon performance problems or inventory changes and even diagnoses errors to issue corrective actions remotely. They claim to be able to correlate events, config changes, or even security events in your environment.

The list of tools and platforms that Hyperic HQ knows about is growing every day. Not only does Hyperic manage these products, it does so by talking to the native APIs that these products provide. Unlike different other tools we've ran into, Hyperic goes IN the application you are monitoring.

Looking at Hyperic HQ from an architectural point of view, they isolate different layers. They start out with a platform which is a machine / operating system combination or a network or storage devices. Hyperic HQ likes to look at components such as the CPU, the Network interfaces, or the Filesystems. One step further is the server. The server is the actual piece of software installed on the machine—it could be a web server, a

database, or a messaging server. Next up is the service. An example might be the vhost that is configured within a web server. The bigger picture for HypericHQ is the Application, which is usually a combination of different components that need to be working, the combination of an Apache virtual host, filesystem, and a MySQL database.

Hyperic is a typical Agent Server setup. The Hyperic agent is a “lightweight Java-based client” that consumes between 10 and 70MB of memory, depending on the number of plugins enabled. Its kernel is capable of processing commands from its agent subsystem like measuring, controlling, autodiscovery, and event tracking, and it will be acting as a listener to delegate requests to the appropriate system. On top of that kernel a plugin layer is available that allows different subsystems to interact with a particular product. The Agent also acts as a local cache for data when it can’t reach the HQ server.

For the HQ-server, at least 1GB of RAM is advised—but for deployments with more than 25 agents, however, 4GB is recommended. Hyperic HQ is running its own Jboss Application server which might be an unwanted overhead for some people. Hyperic HQ also ships with its own PostgreSQL database. Hyperic HQ server is in charge of processing all incoming monitoring data, detecting alert conditions and triggering sending out messages, managing the inventory, scheduling auto-discovery scans, and last but not least, processing all the commands initiated by the end user. One of the features not often seen in a monitoring tool is the availability to cluster the framework, meaning that one can spread the load of different aspects over more nodes of the monitoring framework.

Hyperic HQ has an active community and even its own HyperForge where people can find all kind of different plugins.

Back in the early days they only had a couple of tarballs, but those were fine. However, when you want to automate the agent installation, a package such as an RPM would be better. Seems like recently indeed Hyperic figured out that an RPM would be interesting...

The RPM does almost everything for you. Manually starting the HQ Server, however, should be done using the Hyperic user. The problem with this RPM, however, is that it unpacks a tarball. From here it is a similar action with your clients. After configuring your clients

(you can copy a prepared `agent.properties` file around), you’ll see a list of autodetected services in your Dashboard. And Hyperic proposes you to add this to its inventory. Now, Hyperic isn’t flawless—it detects a lot of services, only to fail, finding different versions of that same service. For example, it finds the local HQ JBoss that it requires for its own workings, but fails to find the JBoss 4.0.3 installed on another server that it is supposed to monitor. However, modifying the parameters in the `agent.properties` file should solve that.

Most other Monitoring systems just detect your MySQL being up or down. Hyperic tells you how fast your indexes and tables are growing, and how many QPS you have on a specific table. The information is in there; however, we have to admit that sometimes it takes a while to find it. :)

Apart from looking at your database, Hyperic also gives you a GUI to optimize and check your tables. So Hyperic goes beyond monitoring, alerting, and trending into fully managing your infrastructure.

Also, when working with different JBoss versions, Hyperic knows about JMX, giving you more fine-grained information.

The big disadvantage of Hyperic is that it requires an awful lot of resources. Hyperic might call a 70MB-eating JVM lightweight, but we call it fat and overweight (especially when compared to other alternatives). When you are trying to measure performance of an ill-behaving server, adding the Hyperic client to that server will, for sure, have an impact on your system. On the other hand, however, you can really drill into an application with Hyperic, you can look deep inside your databases and application servers. And that with almost no configuration efforts.

9 Conclusion

We covered a lot in just 3.5 weeks; getting a full-blown monitoring tools shootout done is a lot of work. We tried our best to test as much as possible and to get a good idea of what worked where and how things worked out of the box. We spent equal time on most of the solutions.

With such little time we had little data for trending analysis, but we haven’t shut down our boxen, they’ll be gathering more data, and we’ll update our findings as we go.

Nagios is still one of the most-deployed tools around, lots of people prefer it because they are used to it and it does what they expect. But do we want more? Do we want easier installations, trending, and so on?

Looking at the installation effort, GroundWorks is obviously the poorest performer in the class. Other tools have prepackaged builds in mainstream Linux distributions that simply work. Where as GroundWorks makes the process more difficult and fails to deliver clear documentation.

On the Auto discovery part, both Agent-based tools score fairly good. Compared to Zabbix, Zenoss doesn't even come close in discovering services, or at least isn't that intuitive.

Zabbix also positively surprised us regarding trending and quick graph correlation, combined with a good set of default templates that get you up to speed in no time.

Part of being a good tool is the capability of not having to look at a manual, of not having to redefine a set of terms so your users understand what you mean. GroundWorks and Zenoss fail there.

We've seen GroundWorks trying to add its own features to Nagios. Because they wanted to build on the shoulders of giants, not reinventing the wheel. The question, however, is: did they make it better? We're not convinced.

On the other hand, we see the OpenNMS and Hyperic people with different monitoring approaches (agent-based vs. agent-less). The OpenNMS folks have written a plugin for Hyperic-HQ that interacts with OpenNMS. This way you really get the best of both worlds. You get the good features of OpenNMS network device integration within one view of Hyperic's application overview.

Hyperic gives you a lot more than a regular monitoring tool, as it goes deep into the applications itself. On the negative side, you might call HypericHQ bloated.

Mark Hinkle, however, mentioned an interesting point in his blog:

“Nagios that has been around longer than any of the monitoring solutions mentioned here they have a large base of plugins and tests used to check status. Hyperic, Groundwork, OpenNMS, and Zenoss all support Nagios

plugins as it is the most utilitarian approach to expanding their products rather than create new standards that might prevent users from using previous customizations and gives flexibility to try new solutions. This adherence to standards enforced (or at least motivated) by users rather than vendors is a bit of a novelty.”

As mentioned, these are our initial findings. We plan to continue our evaluation and keep you posted on our site.

Virtualization of Linux servers

a comparative study

Fernando Laudares Camargos

Revolution Linux

fernando.laudares@revolutionlinux.com

Gabriel Girard

Université de Sherbrooke

gabriel.girard@usherbrooke.ca

Benoit des Ligneris

Revolution Linux

benoit.des.ligneris@revolutionlinux.com

Abstract

Virtualization of x86 servers has been a hot topic in the last decade, culminating in changes in the architecture’s design and the development of new technologies based in different approaches to provide server virtualization.

In this paper, we present a comparative study of six virtualization technologies released under the GPL license. Our analysis is done in two steps. First we evaluate the overhead imposed by the virtualization layers by executing a set of open source benchmarks in the Linux host system and inside the virtual machines. Secondly we analyze the scalability of those technologies by executing a benchmark suite concurrently in multiple virtual machines. Our findings may help users choose the technology that better suits their needs.

1 Introduction

Virtualization is not a new idea [17, 18]. What has changed is its practical purpose. In the begining, virtualization was used as a way of providing multiple access to mainframe systems by running multiple operating systems on the same machine concurrently [31, 30], as well as a safe environment for software development. At that time operating systems were designed for a single user, so the solution to provide access to several users was to run many operating systems in separate virtual machines.

Today, the actual operating systems can provide multiple access to the same machine. There is no more need for running multiple operating systems on the same machine—unless we want them for other reasons. Development is still one use for virtualization, but now the

main focus has changed to other applications such as server virtualization. The idea behind server virtualization has always been to make a better use of the available resources—this is being achieved today through a technique called server consolidation. Studies have shown that the majority of data centers found in today’s enterprises are organized around a silo-oriented architecture [24], in which each application has its own dedicated physical server. This may have been the right design in the past but today the computational resources of the average physical server exceeds the needs of most server applications, which means a share of those resources is being wasted, only around 15% of them being actually used on average [21, 25]. The solution to avoid this waste of physical resources is then to consolidate a group of those servers in one physical machine. Doing this by virtualizing the underlying infrastructure warrants a greater level of isolation between the servers as well as providing other advantages inherent to server virtualization besides server consolidation, which are covered by many other studies [12, 23, 32]. This concept is illustrated by Figure 1. In this scenario four under-utilized physical servers were consolidated in one.

The virtualization layer that allows the hosting of guest operating systems may be provided by different virtualization solutions. Each solution implements this layer in a different way. One negative side-effect of this model is that the existence of such a layer implies a possible overhead that can affect the performance of applications running inside a virtual machine[26]. Ideally this overhead is minimized.

The main objective of this study is to evaluate six virtualization solutions for Linux released under the GPL

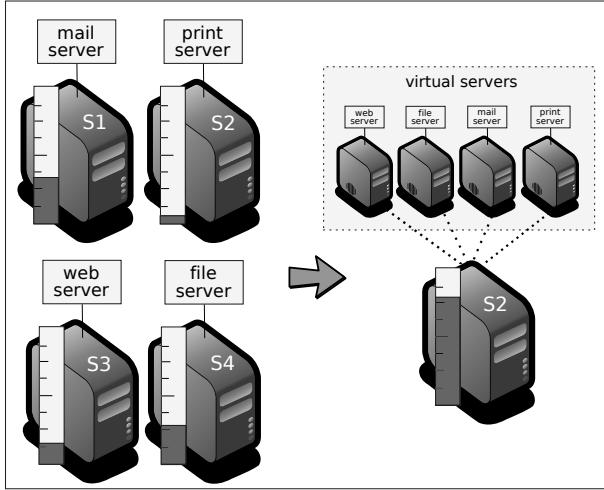


Figure 1: Virtualization as a way to provide server consolidation

license and measure their efficiency. We borrow the definition of *efficiency* from [33], which refers to it as being the combination of *performance* and *scalability*. The stated characteristics of virtualization solutions may lead us to make bad choices if we base ourselves solely on them. This study aims to be practical. We hope our findings may help users to choose the virtualization solutions that better suits their needs.

The rest of this article is structured as follows. Section 2 presents the main points that have motivated us to conduct this study. In Section 3 we present our way of categorizing the different virtualization solutions in groups of technologies that share similar characteristics. Section 4 presents the methodology used to conduct our evaluation. The obtained results are presented and discussed in Section 5. Finally, our conclusions and suggestions for future work are presented in Section 6.

2 Motivation

A number of studies evaluating different virtualization technologies have been published in the past. Table 1 presents some of them chronologically, indicating the year of publication as well as the respective virtualization technologies that were covered by each study. While those studies have been of great value for our understanding of this article’s subject, none has covered the main open source virtualization solutions side-by-side and just a few of them were published by independent sources. The main reason for this lack of coverage

is that such a work requires a great amount of time. Still we were intrigued by this possibility and we believed many others would be as well.

We have chosen to limit the scope of our study to open source technologies because of the friendly nature of the software user licence, which didn’t prevent us publishing our findings nor required them to be scrutinized before a publication permission would eventually be issued. The access to the source code of the software also provides the freedom to adapt it as we need to, which proved to be useful during our experiments.

As users of some of the virtualization technologies that were subjected to our evaluation, our intention with this project was to gain a better view of how they relate to each other. We believe and hope that our feedback may contribute to their continuous development in a positive way.

We have not evaluated the features of the different virtualization technologies on an individual basis. The main characteristics of the majority of those technologies have already been presented in other studies [20, 33, 6].

3 Overview of virtualization technologies

Virtualization technologies differ in the way the virtualization layer is implemented. In 1974, Popek & Goldberg [26] published a paper that presented one way of doing it, which was later referred to as *classic virtualization* [1]. This paper became a reference in the subject and presented a series of requirements for a control program (which is known today as operating system, or *supervisor*) to work as a virtual machine monitor (VMM, also known today as *hypervisor*). Such requirements, organized as a set of properties, limited the construction of such VMMs to a specific machine generation or processor instruction set architecture (ISA) “(. . .) in which sensitive instructions¹ are a subset of privileged instructions.” This characteristic allows the VMM to selectively trap only the privileged instructions issued inside the virtual machines by the guest operating systems and let the remaining instructions be executed directly by the processor. This procedure thus correlates with the stated efficiency property, which is directly associated to a low virtualization overhead.

¹Instructions that may change the current state of the physical machine (IO, for instance).

Study	Year	Evaluated technologies	Version	Kernel version
[2]	2003	Xen	-	2.4.21
		VMware Workstation	3.2	2.4.21
		UML	-	2.4.21
		VMware ESX Server	-	2.4.21
[13]	2004	Xen	-	2.4.26
[27]	2005	Linux-VServer	1.29	2.4.27
		UML	-	2.4.26 / 2.6.7
		Xen	-	2.6.9
[5]	2006	Linux-VServer	1.29	2.4.27
		UML	-	2.4.26 / 2.6.7
		Xen	2.0	2.6.9
		VMware Workstation	3.2	2.4.28
		VMware GSX	-	-
[14]	2006	Linux-VServer	2.0.0.2rc9 / 2.1.0	2.6.15.4 / 2.6.14.4
		MCR	2.5.1	2.6.15
		OpenVZ	022stab064	2.6.8
		Xen	3.0.1	2.6.12.6
[1]	2006	VMware Player	1.0.1	-
[10]	2006	VMWare ESX Server	3.0	-
[35]	2007	VMware ESX Server	3.0.1 GA	-
		Xen	3.0.3-0	-
[36]	2007	VMware ESX Server	3.0.1	2.6.9
		XenEnterprise	3.2	2.6.9
[11]	2007	Xen	3.0.3 (unstable)	-
		OpenVZ	stable	2.6
[33]	2007	Linux-VServer	2.0.1	2.6.17
		Xen	3.0.2-testing	2.6.16
[16]	2007	Linux-VServer	2.2	2.6.20
[22]	2007	Xen	-	-
		KVM	24	-

Table 1: Summary of studies evaluating virtualization technologies published between 2003 and 2007

Even then the researchers were aware of the fact that not all architectures may be “virtualizable” this way. This realization is specially true for today’s most popular architecture, commonly known as x86. The original design of the x86 architecture did not include virtualization [8]. A detailed account on this issue is reported in [28]. In summary, not all sensitive instructions in the x86’s architecture are a subset of its privileged instructions. In practice, this prevents the implementation of a VMM capable of selectively trapping only the sensitive instructions executed by the guest operating systems. Instead, it would have to trap all instructions, incurring a considerable system overhead.

Other virtualization technologies have been developed to avoid this issue and implement alternative ways to virtualize the x86 architecture. Some of them rely on the complete or partial emulation of the underlying hard-

ware [36] and provide what is called *full-virtualization*. *QEMU* [4] is a software emulator that emulates the entire hardware stack of a machine and is used as the base for various virtualization projects in this category, like *KQEMU* (an “accelerator” for QEMU that may be used to transform the latter in a virtualization solution [3]), *KVM* and *VirtualBox*. Those solutions are extremely flexible, meaning they can theoretically support any guest operating system developed for the x86 architecture, but are not among the most efficient ones due to the hardware emulation.

A second category contains virtualization solutions that implement a technique called *para-virtualization*. The most important consideration when running more than one operating system concurrently on the same machine is that this class of software is designed to control the machine exclusively. That is why the use of emulation

works so well—in this case, the guest operating system is still the only one controlling the machine, but the machine in question is not the physical machine, but a virtual machine. The key for para-virtualizing a system is to make the guest operating systems aware of the fact that they are being virtualized [12]—and ask them to collaborate. In exchange, the VMM provides an almost direct access to some of the physical resources of the machine. This approach provides an efficient virtualization technology but it is also an extremely invasive technique since it requires important modifications to the guest OS kernel structure. *Xen* is the most well known para-virtualization solution.

The third and last of our categories focus on the *virtualization at the operating system level*. The virtualization layer in this particular implementation is set above the operating system [23]. “Virtual machines” are soft partitions [15], or containers [14], that replicate the environment of the host operating system. This is in theory the most efficient kind of virtualization—yet the less flexible. The efficiency comes from the fact that there is only one kernel in execution at any time, and thus the absence of hypervisor overhead. The lack of flexibility comes from the same reason—one may even run different flavors of Linux, but they will share the same kernel. Linux-VServers and OpenVZ are two examples of OS-level virtualization solutions. Both are available as a patch that can be applied to the Linux kernel.

Recently, the x86 architecture received additional extensions [34, 37] that allows the implementation of a *classic* VMM that responds to Popek & Goldberg’s criteria. Whether the use of those extensions will assure the development of more efficient virtualization solutions is still arguable [1]. Nonetheless, since version 2.6.20 the Linux kernel comes with KVM, a simple yet robust piece of software that uses those additional extensions to transform Linux into a hypervisor [29, 19]. More recent versions of Xen also use those extensions, as well as a little dose of emulation, to allow for the virtualization of “closed source” operating systems.

4 Methodology

There is no consensus in the scientific community nor in the industry about what would be the best way to evaluate virtualization solutions. The *Standard Performance Evaluation Corporation* (SPEC) created a committee at the end of 2006 that is studying this matter

[7]. VMware and Intel, both members of the committee, stepped ahead and released two different benchmark suites (or workloads, since both suites are composed of already existent and independent benchmarks), named respectively *VMmark* [10] and *vConsolidate* [9]. The workloads that compose both suites are very similar, the difference is in the way each company combines the obtained results to define a scoring system. Neither of these benchmark suites have been considered in this study because they both rely on a benchmark for mail servers called *LoadSim*, which works only with Microsoft’s Exchange Server.

The majority of the studies in Table 1 have used benchmarks that target different parts of the system. We have adopted a similar approach. Our analysis is done in two steps. In the first step we evaluate the overhead imposed by the virtualization layers by executing a set of open source benchmarks in the Linux host system and inside the virtual machines. In the second step, we analyze the scalability of those technologies by executing *Sys-Bench*, one of the benchmarks used in the first step, concurrently in multiple virtual machines.

We decided to use Ubuntu 7.10 as the operating system for the host system, as it comes with kernel version 2.6.22-14, which is well supported among all the evaluated virtualization solutions. For the virtual machines we have chosen Ubuntu’s Long Time Supported version, which at the time of this study was Ubuntu 6.10. This proved to be a bad decision for a comparative study and one of the major weaknesses of this work, since the majority of the OS utilities, like Rsync and Bzip2, have different versions in each release, not to mention the difference in kernel versions.

The name and version of the benchmarks used in our evaluation are shown in Table 2. Table 3 presents a list of the evaluated virtualization solutions, showing the kernel versions used in the host and guest operating systems.

The compilation of the results is done as follows. All experiments are repeated four times. The first sample was discarded, the presented results for each set of experiments being the median of the second, third, and fourth samples. The results of the experiments conducted inside the virtual machines are normalized using the results of the respective experiments conducted in the non-virtualized environment as a base. The evaluation was done in two steps. The first step focused on the overhead

Benchmark	Version (Host)	Version (VMs)	Unit of measure
Bzip2	1.0.3-0ubuntu2.1	1.0.4-0ubuntu2.1	time
Dbench	3.04	3.04	throughput
Dd (coreutils)	5.93-5ubuntu4	5.97-5.3ubuntu3	throughput
Kernel (build)	linux-2.6.22.14	linux-2.6.22.14	time
Netperf	2.4.4	2.4.4	throughput
Rsync	2.6.6-1ubuntu2.1	2.6.9-5ubuntu1	time
SysBench	0.4.8	0.4.8	throughput

Table 2: List of the benchmarks used in the first step of our evaluation and respective metrics

Virtualization solution	Version	Host Kernel	Guest Kernel
KQEMU	1.3.0 pre11-6	2.6.22.14-kqemu	2.6.15-26-amd64
KVM	58	2.6.22-14-server	2.6.15-26-amd64
Linux-VServer	2.2.0.5	2.6.22.14	2.6.22.14
OpenVZ	5.1	2.6.22-ovz005	2.6.22-ovz005
VirtualBox	1.5.4_OSE/1.5.51_OSE	2.6.22-14-server	2.6.22.14
Xen	3.1.0	2.6.22-14-xen	2.6.22-14-xen

Table 3: Evaluated virtualization solutions and the respective kernel versions

of a single virtual machine. In the second step the experiments are repeated concurrently in n virtual machines, with $n=1,2,4,8,16$, and 32.

The experiments were conducted using an IBM/Lenovo desktop configured with an Intel Core 2 Duo 6300 processor, 4G of RAM, an 80GB SATA hard drive and two gigabit network interfaces. The main network interface is connected to a LAN and the other is attached with a cross-over cable to a second machine, which was used as a client for the experiments involving a network. This second machine is a Dell Optiflex GX configured with an Intel Pentium 2 processor, 123M of RAM, a 250GB IDE hard drive and one gigabit interface.

The main machine was rebooted before the beginning of each new set of tests. Before moving on to test the next virtualization solution, the disk was re-formatted and the operational system was re-installed from scratch.

In the first step of the evaluation, the virtual machines were configured with 2G of RAM. Table 4 shows the memory allocation used in the second step. The top limit was set to 2039M of RAM because this was the maximum amount supported by QEMU-based virtualization solutions during preliminary tests.

To run the scalability tests we have used Konsole's *Send input to all sessions* function to log in the n virtual ma-

Number of VMs	Memory/VM (in M)
n=1	2039
n=2	1622
n=4	811
n=8	405
n=16	202
n=32	101

Table 4: Memory distribution used in the second step of the evaluation

chines simultaneously from a third machine, connected to the test machine through the LAN interface, and start the benchmark in all VMs simultaneously.

Figure 2 presents the commands and respective parameters used to execute each one of the benchmarks.

5 Results and discussion

This section presents the results of our experiments. For all charts but the ones representing our scalability evaluations, the results for the different virtualization solutions have been normalized against the results when running without virtualization. Higher bars represent better

Kernel Build

```
$ make defconfig
$ date +%s.%N && make && date +%s.%N
$ make clean
```

Dbench

```
$ /usr/local/bin/dbench -t 300 -D /var/tmp 100
```

Netperf

```
$ netserver          # server side
$ netperf -H <server> # client side
```

Rsync

```
Experiment 1:
$ date +%s.%N && rsync -av <server>::kernel /var/tmp && date +%s.%N # client side
# where 'kernel' is the linux-2.6.22.14 file tree (294M)
$ rm -fr /var/tmp/*
Experiment 2:
$ date +%s.%N && rsync -av <server>::iso /var/tmp && date +%s.%N      # client side
# where 'iso' is ubuntu-6.06.1-server-i386.iso (433M)
$ rm -fr /var/tmp/*
```

Dd

```
Experiment 1:
$ dd if=/opt/iso/ubuntu-6.06.1-server-i386.iso of=/var/tmp/out.iso
$ rm -fr /var/tmp/*
Experiment 2:
$ dd if=/dev/zero of=/dev/null count=117187560 # 117187560 = 60G
```

Bzip2

```
$ cp /opt/ubuntu-6.06.1-server-i386.iso .
$ date +%s.%N && bzip2 -9 ubuntu-6.06.1-server-i386.iso && date +%s.%N
$ rm ubuntu-6.06.1-server-i386.iso.bz2
```

SysBench

```
$ mysql> create database sbtest;
$ sysbench --test=oltp --mysql-user=root --mysql-host=localhost --debug=off prepare
$ sysbench --test=oltp --mysql-user=root --mysql-host=localhost --debug=off run
```

Figure 2: Commands and parameters used to execute the benchmarks

performance of the virtualization solution for the respective workload.

VirtualBox is a virtualization solution that allows the user to decide whether or not it should use the virtualization extensions present in the processor. This fact lead us to perform all experiments with this software twice, with (`--hwvirtex on`) and without (`--hwvirtex off`)

the use of such extensions. As previously mentioned, QEMU is the base for a considerable number of the virtualization solutions evaluated in this study. Since the virtual machine image used by KVM and KQEMU is also compatible with QEMU, we have also evaluated the latter in the first step of our practical study and included the results whenever we considered it to be appropriate. Our main reason for doing this is to show how

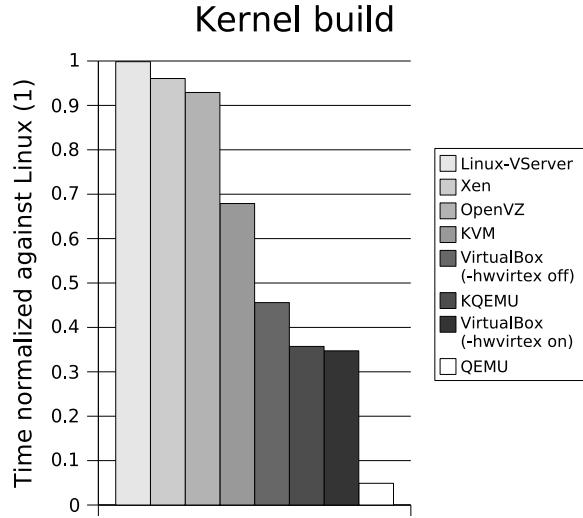


Figure 3: Relative performance of the virtualization solutions for the kernel compilation experiment.

much a virtualization layer below QEMU (like KQEMU and KVM) can benefit the performance of the applications running inside it.

Figure 3 shows the results for the *kernel build* experiments. Kernel compilation is a CPU intensive task which involves multiple threads and stress the filesystem in both reading and writing small files. Those characteristics make it for a good overall system performance indication. As expected, virtualization solutions relying on both OS-level and para-virtualization technologies presented a performance close to Linux's. Among the full-virtualization solutions, KVM's performance is far superior.

This first graphic shows a unique situation in our study in which the non-use of the virtualization extensions by VirtualBox results in performance that is higher than when VirtualBox makes use of such extensions to accomplish the same task. In all the other experiments, such a difference in performance will be less significant.

The next experiment is a file compression using *Bzip2*. This is also a CPU intensive task, but with low I/O requirements. The `-9` option used for maximum compression also demands more memory for the process to execute the compression. For this experiment, we have all virtualization solutions performing close to Linux, except for KQEMU and OpenVZ, as shown in Figure 4. The low performance of OpenVZ was a surprise since we expected it to perform close to Linux for all experi-

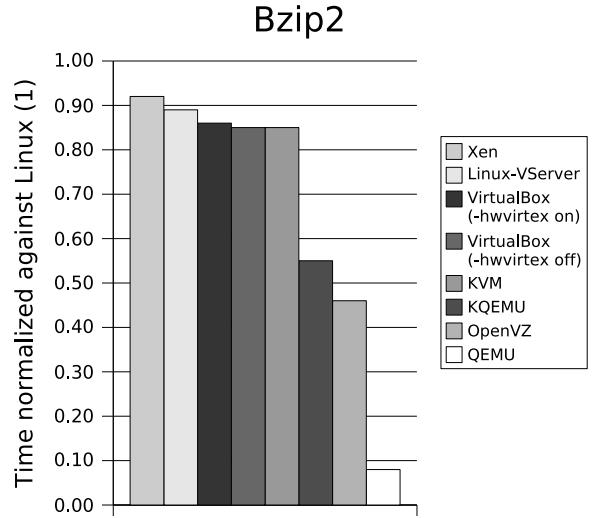


Figure 4: Evaluating the virtualization solutions with Bzip2

ments.

Figure 5 shows the results for the experiments with *Dbench*, a file system benchmark that simulates the load placed on a file server. Here, the sole virtualization solution to match Linux closely was Linux-Vserver, the remaining solutions showing performance less than 30% of Linux. This includes Xen, which has shown better performances in other studies [2, 33] for a similar workload. We were not able to successfully run this benchmark with VirtualBox without it crashing for an unknown reason.

The results for our experiments of disk performance done with *dd* are presented in Figure 6. These experiments do not stress the CPU but focus mainly on disk I/O. For the first experiment, which copies a 433M *iso* image file to the same ext3 partition, Linux-VServer presented performance that considerably surpasses that of Linux. VServer's modifications to the kernel clearly benefit this kind of task. Xen and KVM presented good performance while OpenVZ was significantly slower. In the second experiment, 60G of null characters are read from `/dev/zero` and written to a scratch device (`/dev/null`). Since the data is not actually written to the disk this experiment focuses on the I/O operations of the OS without physically stressing the disk. For this experiment, Xen shows a decrease in performance while OpenVZ performs a lot better than in the first experiment, but still shows a considerable overhead when compared to Vserver and KVM. We were unable

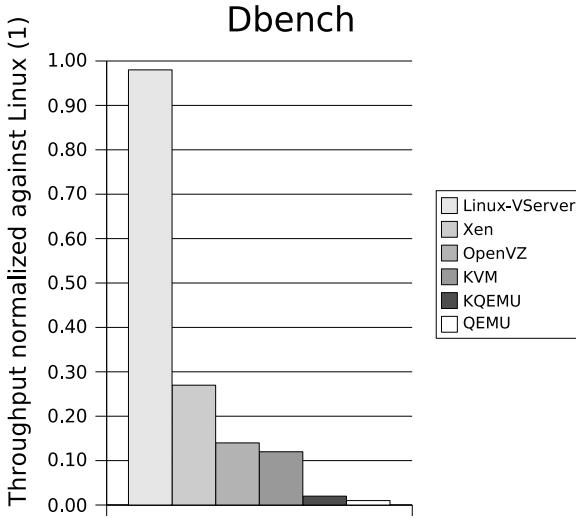


Figure 5: Evaluating the virtualization technologies with Dbench, a file system benchmark

to correctly measure this experiment with KQEMU and VirtualBox, the resultant values for time and throughput being noticeably inaccurate when compared against a wall clock.

Figure 7 shows our results for *Netperf*, a simple network benchmark that uses a stream of TCP packets to evaluate the performance of data exchange. We can clearly differentiate two groups in this experiment: the technologies that are based on QEMU, presenting a poor performance, and the others, which all presented excellent performance. We highlight VirtualBox’s performance, possibly due to the use of a special network driver implementation that communicates closely with the physical network interface.

To complement our network evaluation, we have performed two other experiments using *Rsync* to transfer data from a server to a client machine. The first experiment consisted in the transfer of the entire kernel source tree, which is composed by 23741 small files for a total of 294M. The second experiment consisted in the transfer of the same iso image file used in the compression benchmark. Figure 8 presents the results for both experiments. They confirm the strength of OpenVZ for tasks that include transferring data throughout the network. In the opposite sense, these kinds of task reveal one of the major weaknesses of KVM.

Figure 9 presents our results for the *SysBench* database server performance benchmark (OLTP). The workload

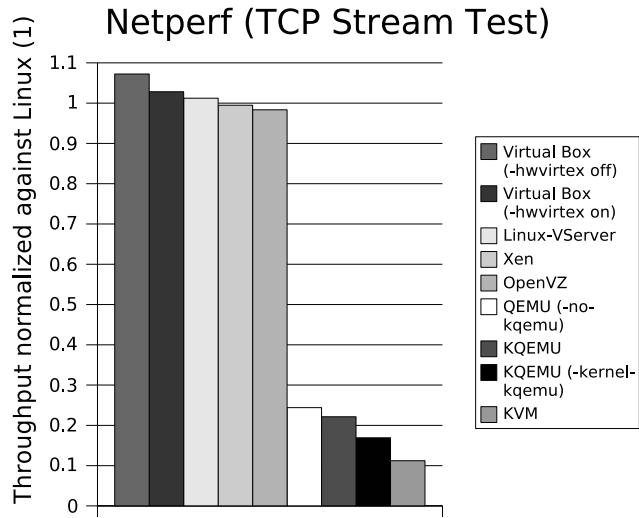


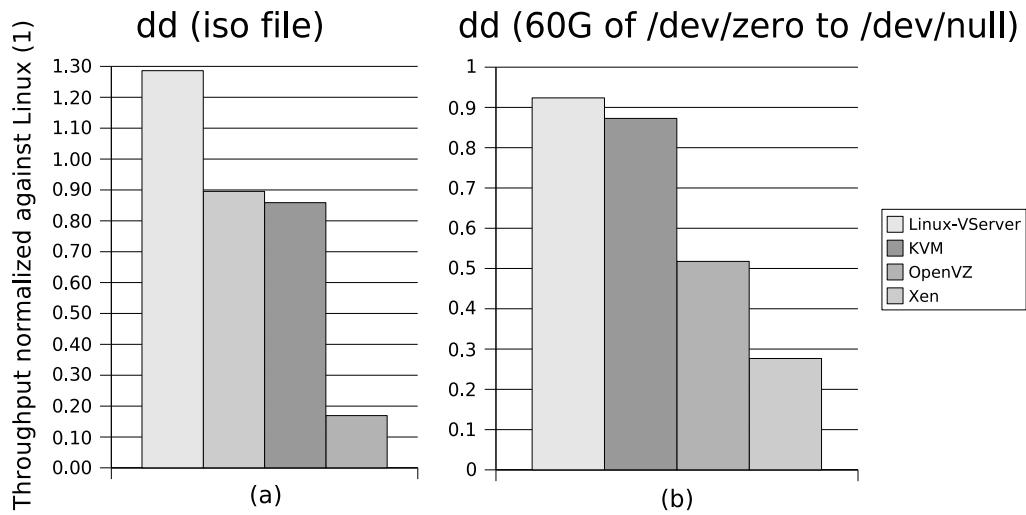
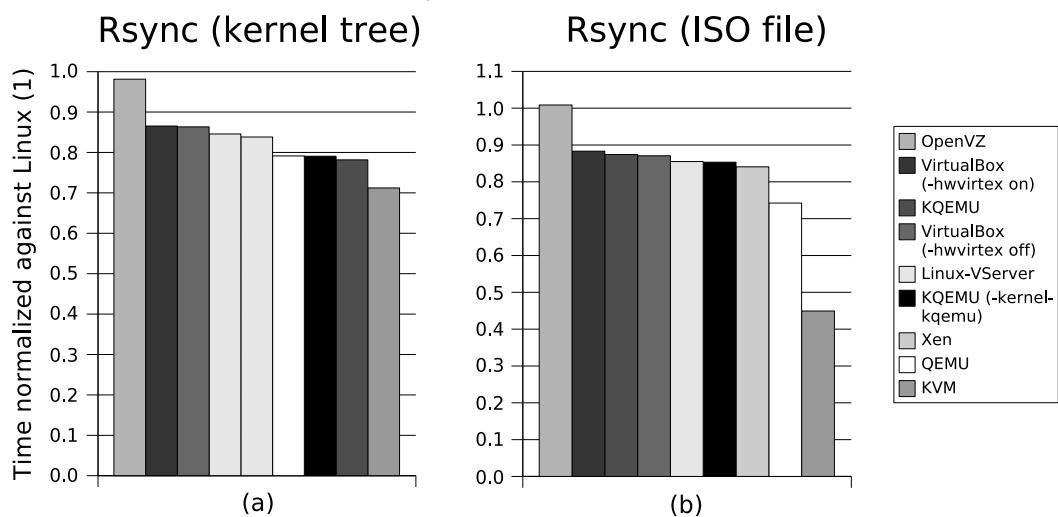
Figure 7: Netperf uses a stream of TCP packets to evaluate the performance of the network

consisted of a set of 10000 transactions performed against a *MySQL* database. For this workload, Linux-VServer and Xen where the only virtualization solutions to perform close to Linux, while KVM and OpenVZ presented performance half as good.

For the second part of our evaluation, we have chosen the SysBench benchmark to evaluate how the virtualization solutions perform when having to manage and share the physical resources of the server between multiple virtual machines executing the same workload. Figure 10 presents our results for this scenario. The left side of the picture (a) shows the aggregate throughput for each set, which was calculated by multiplying the average throughput of the virtual machines, shown in the right side of the picture (b), by the number of virtual machines executing concurrently.

For all virtualization solutions but KVM and VirtualBox, the biggest aggregate throughput appears when 4 VMs were running concurrently. Xen and KQEMU presented a similar behavior, producing an almost constant aggregate throughput, but with opposite performances: while Xen can be considered the most efficient virtualization solution for this particular workload, the inability of KQEMU to make a good use of the available resources was evident.

The two most interesting results were achieved by VirtualBox and Linux-VServer. The aggregate throughput of the first grew smoothly until the number of running

Figure 6: Using *dd* to evaluate disk performanceFigure 8: Evaluating data transfer in the network with *Rsync*

Sysbench (OLTP) at scale

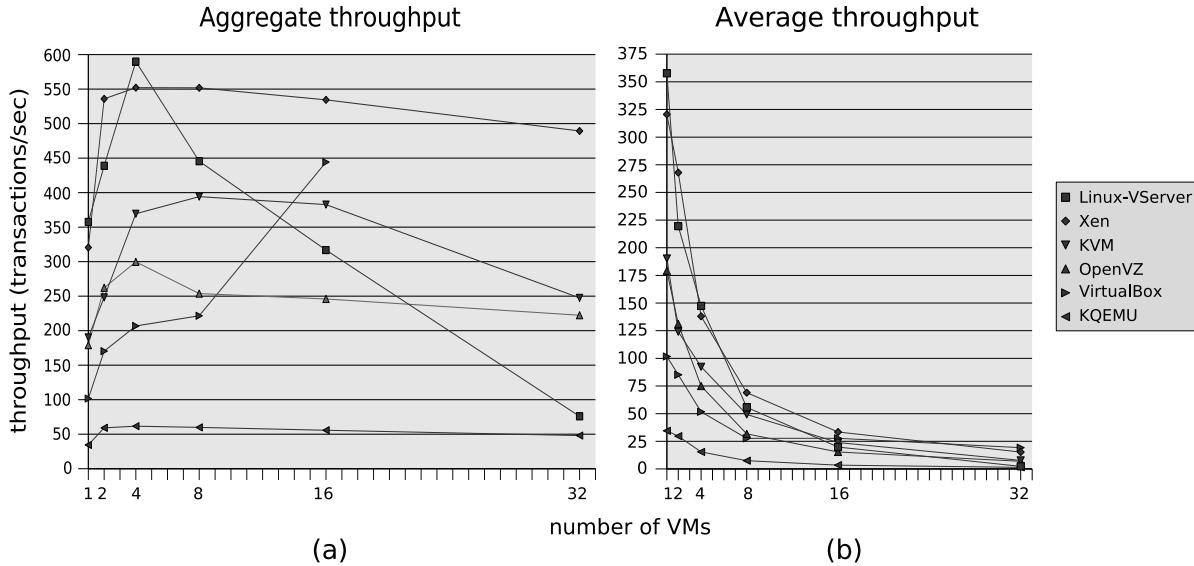


Figure 10: Evaluating the capacity of the virtualization solutions to manage and share the physical available resources with *SysBench*: the left side of the picture (a) shows the aggregate throughput (average throughput per VM x number of VMs) while the right side of the picture (b) shows the average throughput per VM

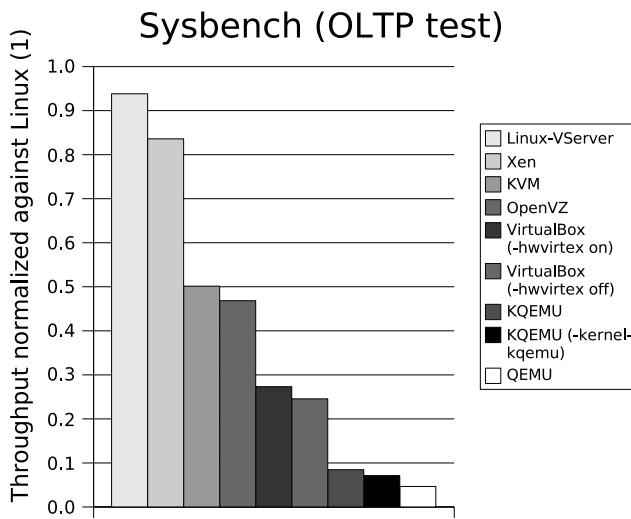


Figure 9: Using the *SysBench* OLTP benchmark to evaluate the performance of a database server

virtual machines reached 8. When we doubled the number of virtual machines to 16, the average throughput per VM remained the same, duplicating the total aggregate throughput produced. We were not able, though, to execute the experiment with 32 VMs, as the available memory per VM was insufficient to run the benchmark. With Linux-VServer, the aggregate throughput obtained for two and eight VMs (or vservers) was almost the same, but it fell considerably when running 16 and 32 vservers concurrently.

This is not the behavior we are used to seeing in production systems running this virtualization solution. We have contacted the authors of [16], who pointed some issues that could help explain Linux-VServer's performance for this particular workload at scale. In summary, they suggested to try different kernel I/O schedulers and timer frequencies, and also executing the same experiment directly in the host OS with and without VServer's kernel patch, and compare the results with those of VServer. Figure 11 summarizes this analysis. In the legend, the data between parenthesis are respectively the Ubuntu version, the kernel timer frequency, and the kernel I/O scheduler used in each experiment.

The Vserver patch used in the experiment did not

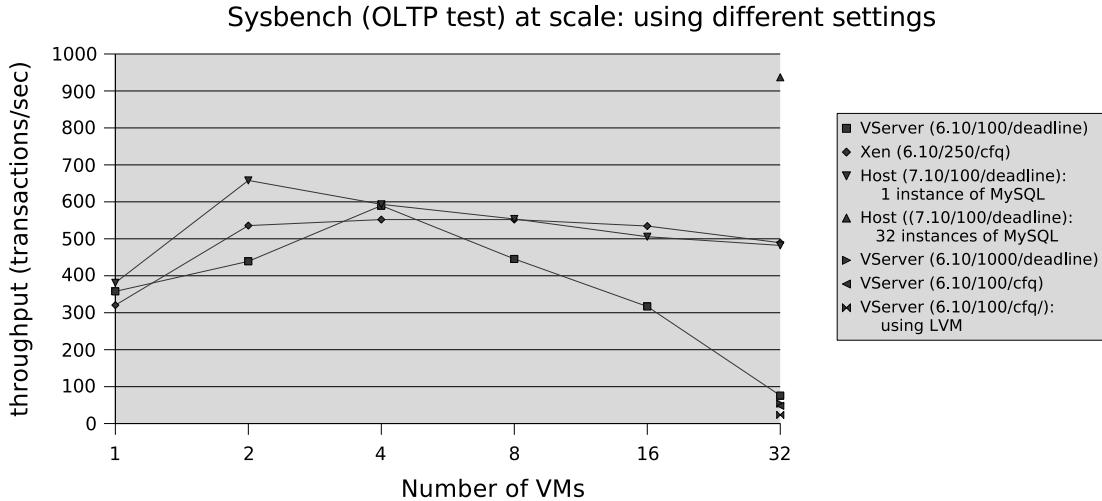


Figure 11: Repeating the scalability evaluation with Linux-VServer, Xen, and Linux using different settings.

change the kernel standard timer frequency, set to 100 Hz, nor the I/O scheduler, *deadline*. Xen uses a different configuration, setting the kernel timer frequency to 250 Hz and selecting a different I/O scheduler, called *Completely Fair Queueing(cfq)*. We have repeated the experiment with VServer for n=32 VMs using different combinations of timer frequency and I/O scheduler. We have also experimented installing each vserver in individual LVM partitions. Neither of those configurations gave better results.

To eliminate the possibility of this being a kernel-related issue we repeated the experiment with 1, 2, 4, 8, and 32 instances of SysBench running in the host system and accessing the same database. The resulting performance curve resembles more the one of Xen. The aggregate throughput decreases slowly when there is more than 4 instances executing concurrently and does not follow the pattern of VServer. We also repeated this experiment running 32 instances of SysBench that connected each to different databases, hosted by 32 *mysqld* servers running on different ports of the host system. This configuration achieved by far the best aggregate throughput in our experiments.

The PlanetLab project uses Linux-VServer to share part of the available resources of their clusters of machines in “slices” [33]. We tried a patch used in the project that fixes a CPU scheduler related bug present in the version of VServer we used in our experiments, with no better results. Finally, we decided to try a different workload, the kernel build benchmark. For VServer and Xen ex-

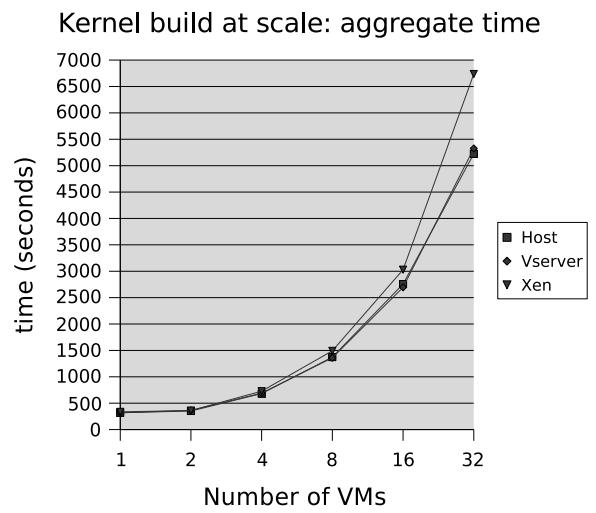


Figure 12: Evaluation of the scalability of Linux-VServer and Xen using the kernel build as benchmark and Linux as reference

periments we used the same configuration as in our previous experiments. For the experiment in the host system we have simply used multiple kernel source trees. Figure 12 summarizes the results, showing the aggregate time for each set of experiments.

For this particular workload, Linux-VServer performs as well as the Linux host system, while Xen follows closely but shows a small overhead that becomes more significant when the experiment is done with 32 VMs.

The results we had with this experiment helps to show how the nature of the workload affects the performance of the virtualization solutions, and how Linux-VServer can deliver the performance we expect from it.

6 Conclusion

This paper evaluated the efficiency of six open source virtualization solutions for Linux. In the first part of our study, we used several benchmarks to analyze the performance of different virtualisation solutions under different types of load and related it to the raw performance of Linux, observing the resulting overhead. In the second part of our study, we evaluated the scalability of those virtualization solutions by running the same benchmark concurrently in multiple virtual machines.

For the first part of our evaluation, Linux-VServer performed close to Linux in all experiments, showing little to no overhead in all situations but one, in which it surpassed Linux's own performance for disk I/O. When executing SysBench at scale, thought, VServer failed to deliver the expected aggregate throughput, specially when the benchmark was running in more than four virtual machines concurrently. A closer look at the problem indicates that it is not directly related to Linux's kernel. We tried different combinations of kernel I/O schedulers and timer frequencies with no better results. To be fair, we repeated the experiment with Linux, Xen, and Linux-VServer using the kernel build benchmark instead of SysBench. This time VServer's performance was comparable to Linux's while Xen showed a small overhead that became more significant when running the experiment concurrently in 32 VMs.

Xen performed fairly well in all experiments but the one using the file system benchmark Dbench. In fact, no other virtualization solution had good results for this benchmark, with the exception of Linux-Vserver. Xen was also the solution that presented the best aggregate throughput when executing SysBench at scale.

KVM performed quite well for a full-virtualization solution. Although it makes for a good development tool, our results indicate it should be avoided for running application services that rely heavily on network I/O. On the other hand, the performance of OpenVZ was disappointing, except when the workload included data transfer throughout the network, which proved to be a strength of this virtualization solution. VirtualBox also

showed good results for experiments that focused on the network but did not perform as well as the others, with the exception of the file compression benchmark. Finally, KQEMU may also be a good candidate in the area of development but for now its use should be avoided in production systems.

For the consolidation of Linux servers, virtualization technologies such as para-virtualization and OS-level virtualization seem to make more efficient use of the available physical resources. However, our findings indicate that the scalability of virtualization solutions may be directly related to the nature of the workload. Except for Linux-VServer and Xen, we have not used different workloads in the second part of our work and we suggest that this should be taken in consideration for future studies. It would also be interesting to repeat the scalability experiments using a mix of different workloads. With the exception of web hosting centers, there are few production systems interested in running multiple instances of the same workload in the same physical server.

References

- [1] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 2–13, New York, NY, USA, 2006. ACM Press.
- [2] 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, New York, NY, USA, 2003. ACM Press.
- [3] Daniel Bartholomew. Qemu: a multihost, multitarget emulator. *Linux J.*, 2006(145):3, May 2006.
- [4] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.
- [5] Franck Cappello Benjamin Quetier, Vincent Neri. Selecting a virtualization system for grid/p2p

- large scale emulation. In *Proc of the Workshop on Experimental Grid testbeds for the assessment of large-scale distributed applications and tools (EXPGRID'06), Paris, France, 19-23 june, 2006.*
- [6] Lucas Bonnet. Etat de l'art des solutions libres de virtualisation pour une petite entreprise. Livre blanc, Bearstech, Decembre 2007.
[http://bearstech.com/files/
LB-virtualisationEntrepriseBearstech.pdf](http://bearstech.com/files/LB-virtualisationEntrepriseBearstech.pdf).
- [7] Standard Performance Evaluation Corporation. Spec virtualization committee, April 2008.
<http://www.spec.org/specvirtualization/>.
- [8] Simon Crosby and David Brown. The virtualization reality. *Queue*, 4(10):34–41, 2007.
- [9] Casazza et al. Redefining server performance characterization for virtualization benchmarking. *Intel Technology Journal*, 10(3):243–252, 2006.
- [10] Makhija et al. Vmmark: A scalable benchmark for virtualized systems. Tech reaport, VMware, Inc., 2006.
- [11] Padala et al. Performance evaluation of virtualization technologies for server consolidation. Tech Reaport HPL-2007-59, HP Laboratories Palo Alto, 2007.
- [12] Justin M. Forbes. Why virtualization fragmentation sucks. In *Proceedings of the Linux Symposium*, volume 1, pages 125–129, Ottawa, ON, Canada, June 2007.
- [13] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the xen vir tual machine monitor. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for On-Demand IT Infrastructure*, Boston, MA, USA, October 2004.
- [14] Cedric Le Goater, Daniel Lezcano, Clement Calmels, Dave Hansen, Serge E. Hallyn, and Hubertus Franke. Making applications mobile under linux. In *Proceedings of the Linux Symposium*, volume 1, pages 347–368, Ottawa, ON, Canada, July 2006.
- [15] Risto Haukioja and Neil Dunbar. Introduction to linux virtualization solutions. Technical report, Hewlett-Packard Development Company, L.P., September 2006.
- [16] Marc E. Fiuczynski Herbert Potzl. Linux-vserver: Resource efficient os-level virtualization. In *Proceedings of the Linux Symposium*, volume 2, pages 151–160, Ottawa, ON, Canada, June 2007.
- [17] IBM. Driving business value with a virtualized infrastructure. Technical report, International Business Machines Corporation, March 2007.
- [18] M. Tim Jones. Virtual linux, December 2006.
<http://www.ibm.com/developerworks/library/l-linuxvirt/index.html>.
- [19] M. Tim Jones. Discover the linux kernel virtual machine, April 2007. <http://www.ibm.com/developerworks/linux/library/l-linux-kvm/>.
- [20] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, Ottawa, ON, Canada, June 2007.
- [21] Tim Kassel and Jeffrey Peck. The rise of the virtual machine and the real impact it will have. Technical report, Thomas Eisel Partners, 2006.
- [22] Jun Nakajima and Asit K. Mallick. Hybrid-virtualization: Enhanced virtualization for linux. In *Proceedings of the Linux Symposium*, volume 2, pages 87–96, Ottawa, ON, Canada, June 2007.
- [23] Susanta Nanda and Tzi cker Chiueh. A Survey on Virtualization Technologies. Rpe report, State University of New York, 2005.
- [24] Pradeep Padala. Adaptive control of virtualized resources in utility computing environments. In *Proc of the EUROSYS (EUROSYS'07), Lisboa, Portugal, March 21-23*, 2007.
- [25] John Pflueger and Sharon Hanson. Data center efficiency in the scalable enterprise. *Dell Power Solutions*, pages 08–14, February 2007.

- [26] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.
- [27] Benjamin Quetier and Vincent Neri. V-meter: Microbenchmark pour evaluer les utilitaires de virtualisation dans la perspective de systemes d'emulation a grande echelle. In *16eme Rencontres Francophones du Parallelisme (RenPar'16)*, Le Croisic, France, Avril 2005.
- [28] J. Robin and C. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor, 2000.
- [29] Michael D. Day Ryan A. Harper and Anthony N. Liguori. Using kvm to run xen guests without xen. In *Proceedings of the Linux Symposium*, volume 1, pages 179–188, Ottawa, ON, Canada, June 2007.
- [30] Love H. Seawright and Richard A. MacKinnon. Vm/370 - a study of multiplicity and usefulness. *IBM Systems Journal*, 18(1):4–17, 1979.
- [31] B. D. Shriver, J. W. Anderson, L. J. Waguestack, D. M. Hyams, and R. A. Bommel. An implementation scheme for a virtual machine monitor to be realized on user - microprogrammable minicomputers. In *ACM 76: Proceedings of the annual conference*, pages 226–232, New York, NY, USA, 1976. ACM Press.
- [32] Amit Singh. An introduction to virtualization, 2006. <http://www.kernelthread.com/publications/virtualization/>.
- [33] Stephen Soltesz, Herbert Potzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proc of the EUROSYS (EUROSYS'07), Lisboa, Portugal, March 21-23*, 2007.
- [34] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.
- [35] VMware. A performance comparison of hypervisors. Technical report, VMware, Inc., 2007.
- [36] XenSource. A performance comparison of commercial hypervisors. Technical report, XenSource, Inc., 2007.
- [37] Alan Zeichick. Processor-based virtualization, amd64 style, part i. Technical report, Advanced Micro Devices, 2006. http://developer.amd.com/article_print.jsp?id=14.

MondoRescue: a GPL Disaster Recovery Solution

Bruno Cornec
Project Lead
bruno.cornec@hp.com

Abstract

MondoRescue is a GPL Disaster Recovery solution. It has existed since 2000, and has now matured to a global solution used both to restore systems in case of emergency as well as to deploy dozens of systems having the same or nearly the same configuration. The main Web site is at <http://www.mondorescue.org> where all the detailed information is contained.

This paper will explain the various functions of the solution: backup image creation, (various filesystems supported, compression schemas, media, distributions, ... *mindi*, the mini-distro creating the environment needed for restoration), restore options, (fully automated, interactive, compare mode, online, ...), some extended usages for which the presenter made patches (LVM v2 support, PXE support...), creation of an image deployment server suited for PXE, restoration of mondo images, cloning techniques with mondo, and so on.

It will also explain the new orientation given to the project since September 2005 with the new maintainership of B. Cornec and the various aspects considered for 3.0.x.

At last, we will also cover some important points around project management (SVN, ML, trac, ...).

1 First encounter with MondoRescue

My initial discovery of the tool was back in 2000, when I was in charge of a project in HP for which I had to find a way to provide an easy installation mechanism of a pre-load of Linux on our servers in EMEA. More over, the idea was to do that with media that the customer would get in the same box as the server.

At that time, only 2 tools existed for this purpose: *mkCDrec* and *MondoRescue*. As I more easily understood the way to use *MondoRescue* for my project,

combined with the fact that the project already covered 90% of my needs, and that the developer (Hugo Rabson) was cooperative and inclined to integrate patches, I then chose it as my tool for this project, and have used it ever since.

Having done a patch to automate the recovery steps to make it easy to both customers and plant people in charge of the installation, I was quickly integrated in the development team of the project (even if not very active). I tried to make an Itanium (ia64) port, but didn't follow up with it at that time, as our HP team wasn't working on the Itanium program anymore. In the other hand, the pre-install Linux project was really successful and gave me a good opportunity to jump on the project.

The goal of *MondoRescue* is thus to provide a GPL Disaster Recodery (DR) or Cloning technology which is distribution-neutral (it runs on Mandriva, Fedora, OpenSUSE, RHEL, CentOS, SLES, Debian, Ubuntu, Gentoo, Slackware), filesystem-neutral (it supports ext2/3, reiserfs, XFS, JFS, LVM v1 and v2, and software & hardware RAID), media-neutral (CD-R/RW, DVD+-R/RW, tape, network, ISO files), kernel-neutral (provides a fail-safe one, or use the one you run or another one, as long as *MondoRescue* requirements are fulfilled).

The interface is either command-line based (via options) or a semi-graphical interface based on *newt*. A Web interface to the CLI is also planned for version 3.x.

1.1 *mindi*

MondoRescue uses *mindi* to create the bootable part of the DR media set, which uses itself *busybox* to provide a light Linux environment at restore time. *mindi* is a shell script. Its role is to create the necessary boot environment needed at restore time to be able to correctly set up your hardware and have access to your recovery images.

Typically, mindi is called through mondoarchive to build that small boot environment. It can also be called separately in order to provide a minimal boot image matching your current installation, so you can use it as a rescue disc.

mindi creates a bootable image that can be used to either create recovery USB devices, using syslinux, or serve as the boot part of an El-Torito ISO9660 CD, using isolinux.

mindi has some interesting options to help you analyze the underlying environment and check that it will be handled correctly.

`mindi --findkernel` finds the image of the running kernel in the file system. Example on a Mandriva 2007.1 distribution:

```
# mindi --findkernel
/boot/vmlinuz-2.6.17.14-mm-desktop-5mdv
```

Example on a RHEL 4 distribution:

```
# mindi --findkernel
/boot/vmlinuz-2.6.9-42.ELsmp
```

Another useful option of mindi for debugging problems is the `--makemountlist`. Example on a MDV 2008.1 distribution (mindi 2.0.2):

```
# mindi --makemountlist /tmp/1
Your mountlist will look like this:
Analyzing LVM...
```

DEVICE	Mountpoint	Format	Size	Label/UUID
/dev/sda1	/	ext3	1239	
/dev/sdb1	/home	ext2	476937	/home
/dev/sda8	/tmp	ext3	1129	
/dev/sda6	/usr	ext3	12033	
/dev/sda7	/var	ext3	4235	
/dev/sda5	swap	swap	980	

This exhibits some news brought by the more recent version (support of UUID for Ubuntu, for instance). If the output of that command doesn't reflect the layout of partitions on your system, then you will experience problems during normal mindi's run.

Most of hardware support issues met with Mondorescue come in fact from mindi. As its responsibility is to create a correct boot environment for the machine, it has to put all the required drivers on the boot part of your rescue media. In order to do that, in current versions, a lot of supported drivers are included in mindi's code for all the required drivers that you may want to use

(CD/DVD drivers, IDE/SATA/SCSI controllers, USB controllers, tape controllers, and network controllers for those doing PXE restore). Code example:

```
FLOPPY_MODS="ide-floppy floppy"
TAPE_MODS="ht st osst ide-tape ide_tape"
SCSI_MODS="3w-xxxx 3w_xxxx 3x_9xxx\\\
3x-9xxx 53c7,8xx a100u2w a320raid\\\
aacraid adpahci advansys aha152x\\\
aha1542 aha1740 aic79xx aic79xx_mod\\\
aic7xxx aic7xxx_mod aic7xxx_old\\\
AM53C974 atp870u BusLogic cciss\\\
cpqfc dmx3191d dpt_i2o dtc eata\\\
eata_dma eata_pio fdomain gdth\\\
g_NCR5380 i2o_block i2o_core\\\
ide-scsi ieee1394 imm in2000\\\
initio ips iscsi isp megaraid\\\
megaraid_mm megaraid_mbox\\\
mptbase mptscsih mptsas mptspi\\\
mptfc mptscsi mptctl NCR53c406a\\\
ncr53c8xx nsp32 pas16 pci2000\\\
pci2220i pcmcia ppa psi240\\\
qla1280 qla2200 qla2300 qlogicfas\\\
qlogicfc qlogicisp raw1394\\\
scsi_debug scsi_mod sd_mod seagate\\\
sg sim710 sr_mod sym53c416 sym53c8xx\\\
sym53c8xx_2 t128 tmscsim u14-34f\\\
ultrastor wd7000 vmhgfs"
```

```
# ide-probe-mod
IDE_MODS="ide ide-generic ide-detect\\\
ide-mod ide-disk ide-cd ide_cd\\\
ide-cs ide-core ide_core edd paride\\\
ata_generic ata_piix libata\\\
via82cxx generic nvidia ahci\\\
sata_nv cmd64x"
PCMCIA_MODS="pcmcia_core ds\\\
yenta_socket"
USB_MODS="usb-storage usb-ohci\\\
usb-uhci usbcore usb_storage input\\\
hid uhci_hcd ehci_hcd uhci-hcd\\\
ehci-hcd ohci-hcd ohci_hcd usbkbd\\\
usbhid keybdev mousedev"
CDROM_MODS="$TAPE_MODS $FLOPPY_MODS\\\
$IDE_MODS af_packet cdrom isocd\\\
isofs inflate_fs nls_iso8859-1\\\
nls_cp437 sg sr_mod zlib_inflate\\\
$USB_MODS $PCMCIA_MODS"
NET_MODS="sunrpc nfs nfs_acl lockd\\\
fscache loop mii 3c59x e100\\\
bcm5700 bnx2 e1000 eopro100\\\
ne2k-pci tg3 pcnet32 8139cp\\\
8139too 8390 forcedeth vmxnet vmnet"
EXTRA_MODS="$CDROM_MODS vfat fat\\\
loop md-mod linear raid0 raid1 xor\\\
raid5 raid456 lvm-mod dm-mod\\\
dm-snapshot dm-zero dm-mirror\\\
jfs xfs xfs_support pagebuf\\\
reiserfs ext2 ext3 minix nfs\\\
nfs_acl nfsvd lockd sunrpc jbd\\\
mbcache"
```

Of course, maintaining that list is an endless story. ; -) That's why an evolution planned for the next stable release will be to automatically include the full list of all drivers to use. Anyway, this is a difficult task when you want to extend Mondorescue in the cloning area. For example, if you want to restore the content of your physical system in a Virtual Machine (VM) for demo purposes, you need to have the drivers for the VM in your image, which may not even be part of your original distribution and that you may want to be able to add. The same is true if you backup on SCSI and restore on SATA controllers...

However, currently you have to deal with a fixed list of drivers, and modify `mindi`'s code by amending those variables in case some driver is missing.

When called from `mondoarchive`, `mindi` has a high number of parameters for the moment passed to him with the `-custom` option, which triggers the fact that it's used in a `mondoarchive` context. This will change in version 3.0.x as instead a configuration file will be used, generated by `mondoarchive` to override the defaults held in `mindi`'s configuration file.

The most important information in trying to understand `mindi`'s issues is contained in the log file `/var/log/mindi.log`. If used alone, you should always attach it to your request on the mailing list (if used from `mondoarchive`, the tool includes it in its own log file, so you don't need it anymore).

Another tool provided by `mindi` is the `/sbin/init` script launched at boot time, whose role is as said to prepare the drivers to support your hardware, but also to configure all the software stacks required to be able to access the DR sets stored on some media somewhere. It goes from just mounting a physical CD-ROM on the right directory to setting up LVM on your hard driver and the network stack, mounting via NFS the remote file system, finding on it the right ISO image to loopback-mount on the same directory.

The steps involved in `init` include:

- Load the mandatory modules from `initrd`:
`InsertEssentialModules`
- Get restore media going: `HandleTape | HandleCDROM | start-nfs | start-usb`

- Get additional stuff from restore media including all other modules: `install-additional-tools` (gets ordinary NFS working)
- Now that all modules are available, load the ones we need: `insert-all-my-modules`.

As all those scripts are currently bash scripts run by `busybox`, one way to debug a problem is to include a `set -x` command inside them at the interesting line where you want to have a detailed view of actions run by the shell. This is, of course, also true for `mindi` in itself. Then, once those tasks have been performed, the command `mondorestore` is launched to deal with the restoration process in itself from the DR set.

1.2 mondoarchive

MondoRescue provides one command, `mondoarchive`, which handles the creation of the DR set. `mondoarchive` is written in C. It accepts multiple options to force it to act specifically (and can be launched that way with `cron`); or without options, you will get a curses-based interface asking you the questions required to make your archive. After having done some sanity checks on your system, `mondoarchive` will first build a catalog of all the files to archive. Then it will call `mindi` to create the boot image used at restore time, passing to it all the parameters required with its `-custom` option. Then `mondoarchive` will call `afio` and your compressor of choice to prepare the package of files to back up. When it estimates that it won't be able to add more files in order to fit within the planned capacity, it will call `mkisofs` on the temporary directory to produce the final ISO, and loop through those steps again up to the last file in the catalog. At the end of that step, it will do the same with the so-called "big files," which are split in order to take the maximum advantage of media size.

It supports multiple compression tools:

- `bzip2`: the default one historically. Best compression ratio, slower backup/restore time, ideal when space constraints are predominant.
- `lzo`: added second to improve performance while retaining a good compression ratio. The `lzo` package has to be added to the distribution as most of them do not include it.

- `gzip`: added recently in 2.2.1. Offers probably the best balanced solution between time and compression ratio. Standard in all distributions.

MondoRescue is based on the `afio` tool to create archive packages. The choice of `afio` may be surprising, but it presents some advantages compare to the more classical `tar` or `cpio`:

- Compatible with `cpio` to allow for easy recovery of content in case of catastrophe where even `afio` is not available.
- Does compression on a file-by-file basis, instead of doing it on the resulting package. This improves the reliability of the solution as even if a media has an error, the user may still be able to recover all the other files; whereas with a single compressed package, he may lose up to the whole package.
- Doesn't compress already compressed files (based on files' suffixes, whose list can be modified).
- Archives are portable between different types of UNIX systems, as they contain only ASCII-formatted header information.
- Supports various block sizes (useful for tape backups).

With `mondoarchive` you can generate files (ISO images that you can burn later or keep on an NFS server for further deployment with PXE) or you can directly create media (CD, DVD, tapes, and USB devices) that you can then use to regenerate your system. The size of the image is a parameter the administrator may choose at will (e.g., 700MB (CD-size), 4GB (DVD-size), 50GB through the LAN, etc.); `mondoarchive` will create the number of media needed automatically and potentially invoke the burning command to make your media.

`mondoarchive` (and `mindi`) can use either your running kernel to boot your system at restore time, or whatever kernel you specify to it. Among those you may want to use in some very particular cases is the failsafe kernel that `mindi` can provide as a replacement.

It has to be noted that `mondoarchive` is working on a live system, so precautions need to be taken when backing up databases, or evolving sets of data that need to remain consistent (you may want to stop or snapshot your

application before launching the archival). A possibility is to run the system in the S run-level to avoid those issues. Also the filesystems are backed up when they are still mounted, so a `fsck` will have to occur at the first boot after restore.

Here is an example of `mondoarchive`'s CLI usage:

```
/usr/sbin/mondoarchive -OV \
-k /boot/vmlinuz-2.6.16 -d /bkp -s 700M \
-E "/usr/share/doc /usr/src /mnt/fs2" \
-N -p machine1 -T /home/mondo/tmp \
-S /home/mondo/scratch \
-n server.hpintelco.org:/writer/nfs
```

- -O: Backup the machine
- -V: Verify the backup
- -k: Do not use the running kernel but a specific one
- -d: Destination path for the images created (ISO files)
- -s: Size of the images created
- -E: List of directories to exclude from backup
- -N: Do not backup network file systems
- -p: Prefix to be used in image name
- -T: Path of the temporary files (useful for NFS mode to have them created locally)
- -S: Path of the scratched files (useful for NFS mode to have them created locally)
- -n: NFS location of the images

Look at the man page of `mondoarchive` for more details on these options.

The usage of a configuration file in v3.0.x will allow to avoid using all those parameters in the future. That same configuration file will be used as a base to fill all the fields of the Web interface in order to generate the right `mondoarchive` command.

1.3 mondorestore

MondoRescue provides one command, `mondorestore`, which handles the restore of the DR set. `mondorestore` is written in C. It will handle the restore process on your system once `/sbin/init` has set things up. One option you can pass to it is the `-p` option to specify the prefix used at backup time, which is only useful when using NFS restore (name doesn't matter for physical media).

`mondorestore` is generally called by `/sbin/init` during the boot of your DR media. It can also be called manually in the case you booted in expert mode (in order to setup your hardware before for whatever reason) or in interactive mode from your running system to restore selectively chosen files.

`mondorestore` will get the parameter passed to `/sbin/init` from the boot prompt, from the `post-init` script, which could be `-Z compare`, `-Z nuke` or (by default) `-Z interactive` (see below for details). Without parameter, `mondorestore` will first ask you whether it should operate in compare, interactive or automatic mode. Then it will ask from which media you want to restore (CD/DVD/Tape/HDD/NFS are among the choices). Then depending on the type of media, it may ask complementary questions before displaying an interface allowing you to change your file system's layout (could be size, type), as long as the mount point remains accessible and of a sufficient size to host the data. When done with that layout, the restore process will start by formatting your hard drive and then restoring your data from the DR media set. At the end it will restore the boot loader (including your modifications if necessary) and relabel if needed.

`mondorestore` supports multiple ways of doing the restoration:

- From bootable media: CD/DVD, USB devices (keys, disks), and OBDR tapes
- From Virtual Media (HP Proliant only): USB CD emulation
- From the network: PXE.

The content of the archives can then be on the same media, or located on a hard disk on the machine, on a tape, or on a network share (NFS). Even if MondoRescue is primarily a DR solution, some derived usage may be envisioned. ; -) One of such usage is as a backup solution. Thus the `mondorestore` command can also be launched from the shell and will provide you with a semi-graphical interface to select the files and directories you want to restore.

Different modes are provided at restore time when you boot from your MondoRescue DR set:

- Automatic: In this mode, no questions will be asked of the user, and the restore will be done fully automatically, as much as possible. This is the option to use in another derived usage of MondoRescue, cloning machines.
- Interactive: This is the standard mode. In this mode, you will be asked a certain number of questions (with generally good defaults coming from your original machine) on how you want to restore your system. This mode allows you to make changes compared to the original machine—for instance, to adapt to a new hardware configuration, etc. Thus the type of the filesystem, its size, and layout can be modified, as well as new hardware supported (if the drivers are available). It is indeed possible to restore on a SATA controller when the backup was made on a SCSI controller.
- Compare: This mode is a non-destructive one, allowing you to compare the content of what is on your DR media with what is currently on the system. A report will be generated at the end to give you the list of all differing files.
- Expert: in this mode, `mondorestore` is not even launched, allowing you to have access to a shell giving you full control before launching it to prepare the hardware the way you want.

Another derived usage is to use MondoRescue as a cloning tool. This generally requires knowledge of both the master and target machine. If those machines are identical, you may use the `-H` option, which will allow you to use the automatic mode at restore time, making it easy for people in charge of the deployment to indeed restore the system (only one keyword to type at boot time). If those machines are different—especially in terms of drivers—you'll have to inform `mindi` that it will have to include additional modules in order to support the target platform correctly.

2 Historical view

Coming back to my history with the MondoRescue project, I have to confess that after that initial work, I wasn't really involved around MondoRescue until 2004, when I worked again on my Itanium port to finish it in April. Patches were committed upstream as usual in the CVS repository where I have had write access

since June, 2003. I also produced (around the end of 2004) some other patches to improve Proliant support (the /dev/cciss device file) but this time was unable to commit them, as we were in a transition phase between CVS and SVN... which never really worked (or during a too-short period of time). :-(The original author of the project then became less interested in it, and questions appeared on the mailing-list and between the developers on how to proceed, without conclusion at that time.

Then there was the “famous” FastServers affair! (look at Google for “FastServers mondo” if you want the history). Fortunately I also had PXE patches I published under the GPL at the same time, so the story wasn’t a very good one indeed. I needed the ability to store a large number of MondoRescue images on a central repository and be able to redeploy them from that same server through the network, without media manipulation. To achieve this, the central server needs to be configured as an NFS server (the only protocol supported for file sharing) and PXE server (using ISC DHCP server, hpa TFTP server, and pxelinux), and some support was added in mindi to handle the loading of network drivers, IP setup in busybox, the NFS stack, and mounting the images from that file share.

But with all that, we were still unable to commit code; patches were floating on the mailing-list without real integration; Hugo published his last version the 3rd of May 2005 (2.04) and then gave nearly no news; developers were discussing from time to time what to do... clearly the project was going to die.

And I was a bit upset by that, as I was using it quite a lot in the HP/Intel Solution Center in which I was (and still am) working, I had patches that would improve some cases that I was unable to commit, so was attaching them to mails on the mailing-list, ... So finally I decided it was time to have a working development environment, and hosted it on BerliOS.de. I chose them as they were offering subversion access, whereas Sourceforge was still using CVS only (and I was convinced, during the few months of availability, of subversion’s advantages over CVS). I proposed to all the people working around MondoRescue to give them access as they were accustomed to, and that’s how Andree Leidenfrost, who was already committing bug fixes and was managing Debian packages, was involved in the new development team. Here is the message which announced the “fork” (with original typos :–):

From: Bruno Cornec <Bruno.Cornec@hp.com>
Date: Fri, 9 Sep 2005 20:48:30 +0200
Subject: Announce: mondo 2.04_berlios/
mindi 1.04_berlios available (was:
[Mondo-devel] Is Mondo still being worked
on?)

Hello,

First some feedback:

I’m working with mondorescue since 2000 to integrate some patches I needed at that time. I’ve never been a very active developper, but provided from time to time some patches to improve things (PXE, cciss support, -H option, ...) and was SVN committer.

After monthes of problems to contribute to the development of mondo (CVS to SVN migration, then SVN repository not accessible for me since nearly one year now), I’m now in the “scratching an itch” phase :–) (Cf: <http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/ar01s02.html>)

So I decide it was time to do something in order to avoid the project to decline. I like it, I use it a lot, it makes my life easier, both at work and at home. So here are the news:

I have registered a mondorescue project on BerliOS, which provides a sourceforge like environement, including SVN support.

I’ve taken the latest tar ball available from mondo (2.04_cvs_20050523) and mindi (1.04_cvs_20050523) on <http://www.mondoressue.org> and initiate an SVN repository with them. I’ve patches the code with what I had locally. The Changelog looks like:

v2.04_berlios (Bruno Cornec <bcornec@users.berlios.de>
- Add -p option to generate ISO images
file names with prefix. The new default
name for ISO images is mondoressue-1.iso
- Mandrake 2005 support
- NFS patches (Yann Aubert <technique@alixen.fr>)

I’d be happy to add new developpers to SVN write access list. You just have to create an account and send it to me so that I can add you. The SVN access is available through https:
[//developer.berlios.de/svn/?group_id=2524](https://developer.berlios.de/svn/?group_id=2524)

I’ve setup a tiny Web page and Wiki. The main access is at https:
[//developer.berlios.de/projects/mondoressue/](https://developer.berlios.de/projects/mondoressue/)

And for those of you who would like to try the new version, it can be downloaded from there or from
<ftp://ftp.berlios.de/pub/mondorescue>. Please register bugs and feature requests as well.

I do not promise anything, as I don't have much time myself, but hope to be able to get help from you so that the project goes on living.

Bruno.

It soon became obvious that it would not just be a transition in tools to manage the project, but also a change in the development team. At that time, I considered myself just as a maintainer of a branch that should be reintegrated in the standard tree, when possible. But in fact, that branch was becoming the standard tree! So after some discussions on the mailing list, and to avoid confusion, it was decided that the next version would be named 2.05 to show it was a better version than the prior one made by Hugo (integration of lots of patches, and a first build process approach).

As in addition to maintaining the code itself (enhancements, bug fixes), it was also necessary to take over all the things related to the project: The subversion repository (September, 2005), the Web site content (fully rewritten and delivered in February, 2006), the domain names (mondorescue.org/.com in January, 2006), the HOWTO (January, 2006), animation of the mailing-list (could be a full-time job ; -), announcements on freshmeat, etc.

Since we took over the maintenance, 11 versions have been published. To sustain the development effort, it also became obvious that being hosted was less convenient and powerful than having a dedicated machine providing those services for the project. So in July of 2006, HP gave me a server dedicated to the project which is hosted by the HP/Intel Solution Center in Grenoble, France, where I work. It has allowed us to manage our own Subversion repository and the Web site (without the unavailability met on hosting platforms), and to add additional functions such as a real ftp site and Trac, an ingenious tool to help manage projects (Wiki, Bug tracking, and SVN browsing). We couldn't decently work without it today. Sympa is probably the next candidate to host mailing-lists.

Three generations of build process have also been developed based on another ingenious tool, QEMU (for Virtual Machine (VM) emulation), and home-made scripts

(which have evolved into a separate project at <http://trac.project-builder.org>) to build for 50+ distributions now from a single set of source files. This allows us to launch a single script to activate all those builds in sequence from the SVN repository automatically. This is a great help in ensuring coherency and consistency across distributions, and makes it easy for users to install it on their distribution of choice. This work will also serve as a base for automated testing in the future.

The future version 2.2.7 is planned for end of July, 2007, and will propose support of the latest distributions (OpenSuSE 11, etc.), various fixes on 2.2.6, etc.

The 2.2.x branch of subversion should then enter into maintenance mode and all our efforts will concentrate on producing the long-promised 3.0.X version. The main changes are internal, with the introduction of a new low-level internal library, with test routines to improve reliability using only dynamic memory allocation. The rest of the code is being parsed to remove as much static memory usage as possible during the 3.0.X timespan. As part of that work, configuration files for both mindi and mondoarchive will be introduced to help users change the way the tool works, without dealing with the code at all. Also a web interface used to generate the options of the mondoarchive will be available, helping with the integration of mondo in another project called [dploy.org](http://trac.dploy.org), made in conjunction with the LinuxCOE team managed by HP fellows (cf. <http://trac.dploy.org>). New features will also include a internationalization support, maybe reactivated FreeBSD support, etc.

3 Getting Help

There are multiple ways to get help when you encounter issues with MondoRescue.

- Look at the documentation available on MondoRescue Web site at <http://www.mondorescue.org/docs.shtml> (includes HOWTO, man pages, migration docs, etc.).
- Look at the FAQ on the Wiki at <http://trac.mondorescue.org/wiki/FAQ>. It contains valuable tips and tricks for some common issues.

- Read the messages on screen, and the log files. 90% of the time they just contain the information which will help you help yourself. They are located under /var/log/mindi.log for mindi, /var/log/mondoarchive.log for mondoarchive, and /var/log/mondorestore.log during the restoration process.
- Look at http://sourceforge.net/mailarchive/forum.php?max_rows=25&style=ultimate&offset=25&forum_name=mondo-devel for archives of the mailing list, which may already contain information about your issue.
- If you think you need external help, you may reach the Mondorescue community via the regular mailing list (<mailto:mondo-devel@lists.sourceforge.net>). But that community can only help you if you provide at least the above-mentioned log files.
- If you think you have found a bug in Mondorescue, please fill a bug report at <http://trac.mondorescue.org/newticket> and also include the log files. Likewise, if you wish Mondorescue had your new shiny dream-function, feel free to create a feature request at the same URL.
- Access to source code is provided as per the license of the software (GPL). So you may want to add/remove/modify whatever feature; access is made available through FTP at <ftp://ftp.mondorescue.org/src> and SVN at `svn co svn://svn.mondorescue.org/mondorescue/branches/stable` (replace stable by the branch you want to get, which could be 2.2.6 for the latest stable 2.2 published—the stable name is for 3.0.x). Patches are more than welcome ;-. All packages provided are available at <ftp://ftp.mondorescue.org/>. Read the instructions on <http://trac.mondorescue.org/wiki/DistributionPackaging> if you want to rebuild either your packages or a version from source of the MondoRescue project, or apply some specific patches.

The Corosync Cluster Engine

Steven C. Dake

Red Hat, Inc.

sdake@redhat.com

Christine Caulfield

Red Hat, Inc.

ccaulfie@redhat.com

Andrew Beekhof

Novell, Inc.

abeekhof@suse.de

Abstract

A common cluster infrastructure called the Corosync Cluster Engine is presented. The rationale for this effort as well as a history of the project are provided. The architecture is described in detail. The internal programming API is presented to provide developers with a basic understanding of the programming model to architecture mapping. Finally, examples of open source projects using the Corosync Cluster Engine are provided.

1 Introduction

The Corosync Cluster Engine [Corosync] Team has designed and implemented the Corosync Cluster Engine to meet logistical needs of the cluster community. Some members of the cluster developer community have strong desires to reduce technology and community fragmentation.

Technology fragmentation results in difficulty with interoperability. Different project clustering systems do not inter-operate well because they each make decisions regarding the state of the cluster in inconsistent ways. Each cluster software may take different approaches to managing failures, communicating, reading configuration files, determining cluster membership, or recovering from failures.

Community fragmentation results in dispersal of developer talent across many different projects. Most projects have a very small set of developers. These developers in the past have not worked on the same infrastructure but instead implement code with similar functionality. This software is then deployed in various cluster systems and must be maintained and developed by individual projects.

The Corosync Cluster Engine resolves these issues by separating the core infrastructure from the cluster services. By making this abstraction, all cluster services

can cooperate on decision making in the cluster. This abstraction also unifies the core code base under one open source group with the purpose to maintain, develop, and direct a reusable cluster infrastructure with an OSI-approved license.

2 History

The Corosync Cluster Engine was founded in January 2008 as a reduction of the OpenAIS project. The cluster infrastructure primitives are reduced from the Service Availability Forum Application Interface Specification APIs into a new project. This effort was spawned by various maintainers of cluster projects to improve interoperability and unify developer talent.

The OpenAIS project was founded in January 2002 to implement Service Availability Forum Application Interface Specification APIs [SaForumAIS]. These APIs are designed to provide an application framework for high availability using clustering techniques to reduce MTTR [Dake05]. During the development of OpenAIS, more development time was spent on the infrastructure than the APIs. As a result of the focus on the infrastructure, a completely reusable plug-in based Cluster Engine was created.

3 Architecture

3.1 Overview

Corosync Cluster Engine clusters are composed of processors connected by an interconnect. This paper defines an interconnect as a physical communication system which allows for multicast or broadcast operation to communicate packets of information. This paper defines a processor as a common computer, including a CPU, memory, network interface chip, physical storage and operating system such as Linux. This type of cluster is commonly referred to as a shared-nothing cluster.

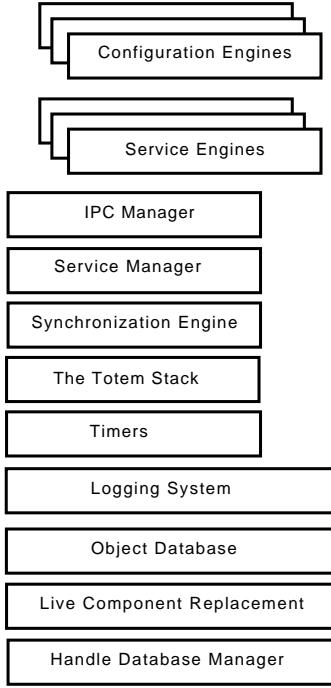


Figure 1: Corosync Cluster Engine Architecture

The Corosync Cluster Engine supports a fully componentized plug-in architecture. Every component of the Corosync Cluster Engine can be replaced by a different component providing the same functionality at processor start time.

Figure 1 depicts the architecture of the Corosync Cluster Engine process.

The subsections in this paper are organized by dependency, not importance. Every component used in the Corosync Cluster Engine is critical to creating a cluster software engine.

3.2 Handle Database Manager

The handle database manager provides a reference counting database that maps in O1 order a unique 64-bit handle identifier to a memory address. This mapping can then be used by libraries or other components of the Corosync Cluster Engine to map addresses to 64-bit values.

The handle database supports the creation and destruction of new entries in the database. Finally, mechanisms exist to obtain a reference to the object database entry and release the reference.

```

struct iface {
    void (*func1) (void);
    void (*func2) (void);
    void (*func3) (void);
};

/*
 * Reference version 0 of A and B interfaces
 */
res = lcr_ifact_reference (
    &a_ifact_handle_ver0,
    "A_iface1",
    0, /* version 0 */
    &a_iface_ver0_p,
    (void *)0xaaaa0000);

a_iface_ver0 = (struct iface *)a_iface_ver0_p;

res = lcr_ifact_reference (
    &b_ifact_handle_ver0,
    "B_iface1",
    0, /* version 0 */
    &b_iface_ver0_p,
    (void *)0xbbbb0000);

b_iface_ver0 = (struct iface *)b_iface_ver0_p;

a_iface_ver0->func1();
a_iface_ver0->func2();
a_iface_ver0->func3();

lcr_ifact_release (a_ifact_handle_ver0);

b_iface_ver0->func1();
b_iface_ver0->func2();
b_iface_ver0->func3();

lcr_ifact_release (b_ifact_handle_ver0);

```

Figure 2: Example of using multiple interfaces in one application

Garbage collection occurs automatically and a user-supplied callback may be called when the reference count for a handle reaches zero to execute destruction of the handle information.

3.3 Live Component Replacement

Live Component Replacement is the plug-in system used by the Corosync Cluster Engine. Every component in the engine is an LCR object which is loaded dynamically. LCR objects are designed to be replaceable at runtime, although this feature is not yet fully implemented.

The LCR plug-in system is different from all other plug-in systems in that a complete C interface is plugged into the process address space, instead of simply one function call. Figure 2 demonstrates the use of the LCR system.

LCR objects are linked statically or dynamically. When an interface is referenced, an internal storage area is checked to see if the object has been linked statically. If it has been linked statically, a reference will be given to the user. If it isn't found in the internal storage area, the `lcrcso` directory on the storage medium will be scanned for a matching interface. If it is found it will be loaded and referenced to the user; otherwise, an error is returned.

The live component replacement plug-in system is used extensively throughout the Corosync Cluster Engine to provide dynamic run-time loading of interfaces.

3.4 Object Database

The object database provides an in-memory non-persistent storage mechanism for the configuration engines and service engines.

The object database is a collection of objects. Every object has a name and is stored in a tree-like structure. Every object has a parent. Within objects are key and value pairs which are unique to the object. Figure 3 depicts a partial object database layout.

The object database provides an API for the creation, deletion, and searching for objects. The database also provides mechanisms to read and write key and value pairs. Finally, the database provides a mechanism to find objects, iterate objects within a tree, and iterate keys within an object.

Objects have specific requirements. The object database allows multiple objects with the same name to be stored in the database with the same parent. Every object may contain key and value pairs. An object's key is unique and its value is a binary blob of data.

Because the object database is often used in parsing by the configuration engine, a special API is provided to automatically detect failures in the storing of keys and associated values within an object. On object creation, a list of valid keys for that object can be registered as well as a validation callback for each key. If the user

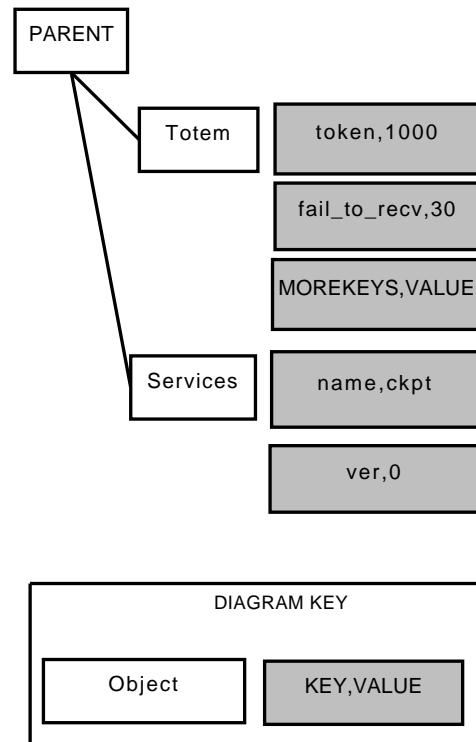


Figure 3: Typical Object Database Layout

of the API specifies an invalid key when modifying an object within the object database, the modification request will be rejected with an error. When the key is valid, before the key is modified, the validation callback is called. This validation callback verifies the contents of the value using the user-registered callback. If the callback returns an invalid value, the modification request is rejected.

3.5 Logging System

A common logging system is available to service engines as well as the rest of the Corosync Cluster Engine software stack. The logging system is completely non-blocking and uses a separate thread in the process address space to filter and output logging information. The logging system is a generically reusable library available to third-party processes as well as service engines. In the case that multiple service engines use the logging system, only one thread is created by the Corosync Cluster Engine.

The logging system supports logging with complete `printf()` style argument processing. Information may be printed to `stderr`, a file, and/or `syslog`.

A logging system may contain any number of components, called tags, which allow runtime filtering of debug messages and 8 levels of tracing messages to the logging output medium. Each tracing type may be separately filtered so specific trace numbers may be used for specific functionality.

A unique feature of the logging system is that a logging system and logging components are initialized through a constructor definition at the beginning of the C code for the file. The configuration options may also be changed at runtime. Additionally, the logging system supports the `fork()` system call.

3.6 Timers

Nearly every service engine requires the use of timers, so a timer system is provided. Time is represented in nanoseconds since the epoch, or January 1, 1970.

Timers may be set to expire at an absolute time. Another type of timer allows expiration in a certain number of nanoseconds into the future.

When a timer expires, it executes a callback registered at timer creation time to execute software code desired by the service engine designer.

3.7 The Totem Stack

The Totem Single Ring Ordering and Membership Protocol [Amir95] implements a totally ordered extended virtual synchrony communication model [Moser94]. Unlike many typical communication systems, the extended virtual synchrony model requires that every processor agrees upon the order of messages and membership changes, and that those messages are completely recovered.

A property of virtual synchrony, called agreed ordering, allows for simple state synchronization of cluster services. Because every node receives messages in the same order, processing of messages occur once the Totem protocol has ordered the message. This allows every node in the cluster to remain in synchronization when processor failure occurs or new processors are included in the membership.

One key feature of the Totem stack is that it supports the ability to communicate redundantly over multiple

network interfaces. All data including the membership protocol is replicated over multiple network interfaces using the Totem Redundant Ring Protocol [Koch02].

Totem is implemented completely in userspace using user datagram protocol [Postel80] multicast. The protocol implementation can be configured to run within Internet Protocol version 4 [USC81] networks or Internet Protocol version 6 [Deering98] networks.

All communication may be, at user configuration, authenticated and encrypted using a private secret key stored securely on all nodes.

3.8 Configuration Engine

The Corosync Cluster Engine solves the issue of configuration file independence by providing the ability to load an application specific configuration engine. The configuration engine provides a method to read and write configuration files in an application specific way. These plug-ins configure the Corosync Cluster Engine as well as other components specific to an application plug-in.

In the event that the Corosync Cluster Engine executive is not running, the configuration engine can still be used by applications transparently to read and store configuration information.

3.9 Interprocess Communication Manager

The interprocess communication manager is responsible for receipt and transmission of IPC requests. The incoming IPC requests are routed via the service manager to the appropriate service engine plug-in. The service engine may send responses to a third-party process.

Every IPC connection is an abstraction of two file descriptors. One file descriptor is used for third-party process blocking request and response packets. The remaining file descriptor is used exclusively for non-blocking callback operations that should be executed by the third-party process. These two file descriptors are connected to each other during initialization of the IPC connection by the Interprocess Communication Manager.

3.10 Service Engine

A service engine is created by third parties to provide some form of cluster wide services. Some examples of these are the Service Availability Forum's Application Interface Specification checkpoint service, Pacemaker, or CMAN.

The service engine has a well defined live component replacement interface for run-time linking into the service manager. The service engine is responsible for providing a specific class of cluster service to a user via API or external control via the interprocess communication manager.

3.11 Service Manager

The service manager is responsible for loading and unloading plug-in service engines. It is also responsible for routing all requests to the service engines loaded in the Corosync Cluster Engine.

During Corosync Cluster Engine initialization, the configuration engine is loaded. The configuration engine then stores the list of service engines to load. Finally, the service manager loads every service engine.

Once the service manager loads a service, it is responsible for initializing the service engine. When the user requests an operation via the interprocess communication manager, that request is routed to the appropriate service engine by the service manager. The service manager is also responsible for sending membership changes to the service manager. A service engine replicates information via the low-level Totem Single Ring Protocol by transmitting messages. These transmitted messages are delivered via the service manager to a service engine. Finally, the service manager is responsible for routing synchronization activities with the synchronization engine.

3.12 Synchronization Engine

The synchronization engine is responsible for directing the recovery of all service engines after a failure or addition of a processor. A service engine may optionally use the synchronization engine, or set the synchronization engine functions to NULL, in which case they won't be used.

```

typedef uint64_t cpg_handle_t;

typedef enum {
    CPG_DISPATCH_ONE,
    CPG_DISPATCH_ALL,
    CPG_DISPATCH_BLOCKING
} cpg_dispatch_t;

typedef enum {
    CPG_TYPE_UNORDERED,
    CPG_TYPE_FIFO,
    CPG_TYPE_AGREED,
    CPG_TYPE_SAFE
} cpg_guarantee_t;

typedef enum {
    CPG_FLOW_CONTROL_DISABLED,
    CPG_FLOW_CONTROL_ENABLED
} cpg_flow_control_state_t;

typedef enum {
    CPG_OK = 1,
    CPG_ERR_LIBRARY = 2,
    CPG_ERR_TIMEOUT = 5,
    CPG_ERR_TRY AGAIN = 6,
    CPG_ERR_INVALID_PARAM = 7,
    CPG_ERR_NO_MEMORY = 8,
    CPG_ERR_BAD_HANDLE = 9,
    CPG_ERR_ACCESS = 11,
    CPG_ERR_NOT_EXIST = 12,
    CPG_ERR_EXIST = 14,
    CPG_ERR_NOT_SUPPORTED = 20,
    CPG_ERR_SECURITY = 29,
    CPG_ERR_TOO_MANY_GROUPS=30
} cpg_error_t;

typedef enum {
    CPG_REASON_JOIN = 1,
    CPG_REASON_LEAVE = 2,
    CPG_REASON_NODEDOWN = 3,
    CPG_REASON_NODEUP = 4,
    CPG_REASON_PROCDOWN = 5
} cpg_reason_t;

struct cpg_address {
    uint32_t nodeid;
    uint32_t pid;
    uint32_t reason;
};

#define CPG_MAX_NAME_LENGTH 128

struct cpg_name {
    uint32_t length;
    char value[CPG_MAX_NAME_LENGTH];
};

#define CPG_MEMBERS_MAX 128

```

Figure 4. The Closed Process Group Interface Definitions

The synchronization engine has four states

- sync_init
- sync_process
- sync_activate
- sync_abort

The first step in the synchronization process for a service engine is initialization. The `sync_init` call in a service engine stores information for executing the recovery algorithm created by the service engine designer.

The `sync_process` is executed to process the recovery operation. Because the Totem protocol transmission queue may become full on the processor executing recovery, `sync_process` may have to return without completing by returning a negative value. If synchronization was completed, a value of zero should be returned.

If at any time during synchronization, a new processor joins the membership or a processor leaves the membership, the `sync_abort` call will be executed to reset any state created by `sync_init`.

After synchronization has completed on all nodes, `sync_activate` is called to activate the new data set for the service engine.

3.13 Default Service Engines

The Corosync Cluster Engine provides a few default service engines which are generically useful. Other default service engines will be provided in the future.

3.13.1 Closed Process Group Service Engine

The closed process group API and the associated service engine are responsible for providing closed process group messaging semantics. Closed process groups are a specialization of the process groups semantics [Birman93].

Any process may join a process group. A process is a system task with a process identifier, often called a PID. Once joined, a join message is sent to every process in the membership. The contents of the join message are

the process ID of the process and the processor identifier that the joining process on which the process is running. When the process leaves the process group, either voluntarily, or as a result of failure, a leave message is sent to every remaining processor.

The closed process group service engine allows the transmission and delivery of messages among a collection of processors that have joined the process group.

The definitions in Figure 4 and API in Figure 5 are used to implement the closed process group system. At all times, the extended virtual synchrony messaging model is maintained by this service.

To join a process group, `cpg_join()` is used in a C program. The user passes the process group to join. To leave a process group, `cpg_leave()` is used. Failures automatically behave as if the process had executed a `cpg_leave()` function call. Messages are sent to every node in the process group using the C function `cpg_mcast()`.

Changes in the process membership and delivery of messages are executed using the `cpg_dispatch()` C function call. This function calls the `cpg_deliver_fn_t()` function to deliver messages and `cpg_confchg_fn_t()` to deliver membership changes. These functions are registered during initialization with the `cpg_initialize()` function call.

3.13.2 Configuration Database Service Engine

The configuration database service engine provides a C programming API to third-party processes to read and write configuration information in the object database. The API is essentially the same as that used in the object database.

The configuration database service C API may operate when the Corosync Cluster Engine is not running for configuration purposes. In this operational mode, a configuration engine is loaded and automatically used to read or write the object database after the user of the C API has made changes to the object database.

4 Library Programming Interface

4.1 Overview

The library programming interface is useful for third-party processes that wish to access a Corosync service

```

typedef void (*cpg_deliver_fn_t) (
    cpg_handle_t handle,
    struct cpg_name *group_name,
    uint32_t nodeid,
    uint32_t pid,
    void *msg,
    int msg_len);

typedef void (*cpg_confchg_fn_t) (
    cpg_handle_t handle,
    struct cpg_name *group_name,
    struct cpg_address *member_list,
        int member_list_entries,
    struct cpg_address *left_list, int
        left_list_entries,
    struct cpg_address *joined_list, int
        joined_list_entries);

typedef struct {
    cpg_deliver_fn_t cpg_deliver_fn;
    cpg_confchg_fn_t cpg_confchg_fn;
} cpg_callbacks_t;

cpg_error_t cpg_initialize (
    cpg_handle_t *handle,
    cpg_callbacks_t *callbacks);

cpg_error_t cpg_finalize (
    cpg_handle_t handle);

cpg_error_t cpg_fd_get (
    cpg_handle_t handle, int *fd);

cpg_error_t cpg_context_get (
    cpg_handle_t handle, void **context);

cpg_error_t cpg_context_set (
    cpg_handle_t handle, void *context);

cpg_error_t cpg_dispatch (
    cpg_handle_t handle, cpg_dispatch_t
        dispatch_types);

cpg_error_t cpg_join (
    cpg_handle_t handle,
    struct cpg_name *group);

cpg_error_t cpg_leave (
    cpg_handle_t handle,
    struct cpg_name *group);

cpg_error_t cpg_mcast_joined (
    cpg_handle_t handle,
    cpg_guarantee_t guarantee,
    struct iovec *iovec, int iov_len);

```

Figure 5: The Closed Process Group Interface API

engine. The library programming interface provides handle management and connection management with the `hdb` inline library and the `cslib` library.

4.2 Handle Database API

The handle database API, shown in Figure 6, is responsible for managing handles that map to memory blocks. Handle memory blocks are reference counted and the handle memory area is automatically freed when no user references the handle. The API is fully thread safe and may be used in multithreaded libraries.

When creating a handle database, the function `hdb_create()` should be used. When destroying a handle database, the function `hdb_destroy()` should be used.

To create a new entry in the handle database, use the function `hdb_handle_create()`. Once the handle is created, it will start with a reference count of 1. To reduce the reference count and free the handle, the function `hdb_handle_destroy()` should be executed.

Once a handle is created with `hdb_handle_create()`, it can be referenced with `hdb_handle_get()`. This function will retrieve the memory storage area relating to the handle specified by the user. When the library is done using the handle, `hdb_handle_put()` should be executed.

4.3 Corosync Library API

The Corosync Library API, defined in Figure 7, provides a mechanism for communicating with Corosync service engines. A library may connect with the Corosync Cluster Engine by using `cslib_service_connect()`. This function returns two file descriptors. One file descriptor is used for request and response messages. The remaining file descriptor is used for callback data that shouldn't block normal requests.

Once an IPC connection is made, a request message can be sent with `cslib_send()`. A response may be received with `cslib_recv()`. These functions generally shouldn't be used unless the size of the message to be received is variable length.

When the size of the message to be received is known, `cslib_send_recv()` should be used. This will send a request, and receive a response of a known size.

```

struct hdb_handle {
    int state;
    void *instance;
    int ref_count;
};

struct hdb_handle_database {
    unsigned int handle_count;
    struct hdb_handle *handles;
    unsigned int iterator;
    pthread_mutex_t mutex;
};

void hdb_create (
    struct hdb_handle_database
        *handle_database);

void hdb_destroy (
    struct hdb_handle_database
        *handle_database);

int hdb_handle_create (
    struct hdb_handle_database *handle_database,
    int instance_size,
    unsigned int *handle_id_out);

int hdb_handle_get (
    struct hdb_handle_database *handle_database,
    unsigned long long handle,
    void **instance);

void hdb_handle_put (
    struct hdb_handle_database *handle_database,
    unsigned long long handle);

void hdb_handle_destroy (
    struct hdb_handle_database *handle_database,
    unsigned long long handle);

void hdb_iterator_reset (
    struct hdb_handle_database
        *handle_database);

void hdb_iterator_next (
    struct hdb_handle_database *handle_database,
    void **instance,
    unsigned long long *handle);

```

Figure 6: The Handle Database API Definition

All of these functions handle recovery of message transmission on short reads or writes, or in the event of signals or other system errors that may occur.

Finally, it is useful to poll a file descriptor, especially in a dispatch routine. This can be achieved by using `cslib_poll()` which is similar to the poll system call

```

cslib_service_connect (
    int *response_out,
    int *callback_out,
    unsigned int service);

cslib_send (int s,
    const void *msg,
    size_t len);

cslib_recv (int s,
    const void *sg,
    size_t len);

cslib_send_recv (
    int s,
    struct iovec *iov,
    int iov_len,
    void *response,
    int response_len);

cslib_poll (
    struct pollfd *ufds,
    unsigned int nfds,
    int timeout);

```

Figure 7: The Corosync Library API Definition

except it retries on signals and other errors which are recoverable.

5 Service Engine Programming Model and Interface

5.1 Overview

A service engine consists of a designer-supplied plug-in interface coupled with the implementation of functionality that uses Corosync Cluster Engine APIs.

A service engine designer implements the plug-in interface. This interface is a set of functions and data which are loaded dynamically. The service manager directs the service engine to execute functions. Some of the service engine functions then use four APIs which are registered with the service engine to execute the operations of the Corosync Cluster Engine.

5.2 Plug-In Interface

The full plug-in interface is a C structure depicted in Figure 8. The interface contains both data and function

calls which are used by the service manager to direct the service engine plug-in.

The `name` field contains a character string which uniquely identifies the service engine name. This field is printed by the Corosync Cluster Engine to give status information to the user.

The `id` field contains a 16-bit unique identifier registered with the Corosync Cluster Engine. This unique identifier is used to route library and Totem requests to the proper service engine by the service manager.

When private data is needed to store state information, the interprocess communication manager allocates a block of memory of the size of the parameter `private_data_size` during initialization of the connection.

The `exec_init_fn` field is a function executed to initialize the service engine. The `exec_exit_fn` field is a function executed to request the service engine to shut down. When the administrator sends a `SIGUSR2` signal to the Corosync Cluster Engine process, the state of the service engine is dumped to the logging system by the `exec_dump_fn` function.

The `lib_init_fn` field is a function executed when a new library connection is initiated to the service engine by the interprocess communication manager. The `lib_exit_fn` field is a function executed when the IPC connection is closed by the interprocess communication manager.

The main functionality of a service engine is managed by the service engine using the `lib_engine` and `exec_engine` parameters. These parameters contain arrays of functions which are executed by the service manager.

A service engine connection is routed to the proper `lib_engine` function by the service manager. When a library connection requests the service engine to execute functionality, the connection's `id` is used to identify the function in the array to execute. The `lib_engine_count` contains the number of entries in the `lib_engine` array.

The function then would generally use the various APIs available within the `corosync_api_v1` structure to create timers, send Totem messages, or respond with a message using the interprocess communication manager.

When Totem messages are originated, they are delivered to the proper `exec_engine` function by the service manager to every processor in the cluster. The proper `exec_engine` function is called based upon the service id in the header of the function. The `exec_engine_count` contains the number of entries in the `exec_engine` array.

The design of a service engine should take advantage of the Totem ordering guarantees by executing most of the logic of a service engine in the `exec_engine` functions. These functions generally respond to the library request that originated the Totem message using the interprocess communication manager API.

5.3 Service Engine APIs

5.3.1 Overview

There are four sets of functionality within Corosync service engine APIs shown in Figure 9.

5.3.2 Timer API

The timer api allows a user-specified callback to be executed when a timer expires. Timers may either be defined as absolute or at some duration into the future.

The `timer_add_duration()` function is used to add a callback function that expires into a certain number of nanoseconds into the future. The `timer_add_absolute()` function is used to execute a callback at an absolute time as specified through the number of nanoseconds since the epoch.

If a timer has been added to the system, and later needs to be deleted before it expires, the designer can execute `timer_delete()` function to remove the timer.

Finally, a service engine can obtain the system time in nanoseconds since the epoch with the `timer_get()` function call.

5.3.3 Interprocess Communication Manager API

The Interprocess Communication Manager API includes functions to set and determine the source of messages, to obtain the IPC connection's private data store,

```

struct corosync_lib_handler {
    void (*lib_handler_fn) (void *conn, void *msg);
    int response_size;
    int response_id;
    enum corosync_flow_control flow_control;
};

struct corosync_exec_handler {
    void (*exec_handler_fn) (void *msg, unsigned int nodeid);
    void (*exec_endian_convert_fn) (void *msg);
};

struct corosync_service_engine {
    char *name;
    unsigned short id;
    unsigned int private_data_size;
    enum corosync_flow_control flow_control;
    int (*exec_init_fn) (struct objdb_iface_ver0 *, struct corosync_api_v1 *);
    int (*exec_exit_fn) (struct objdb_iface_ver0 *);
    void (*exec_dump_fn) (void);
    int (*lib_init_fn) (void *conn);
    int (*lib_exit_fn) (void *conn);
    struct corosync_lib_handler *lib_engine;
    int lib_service_count;
    struct corosync_exec_handler *exec_engine;
    int (*config_init_fn) (struct objdb_iface_ver0 *);
    int exec_service_count;
    void (*confchg_fn) (
        enum totem_configuration_type configuration_type,
        unsigned int *member_list, int member_list_entries,
        unsigned int *left_list, int left_list_entries,
        unsigned int *joined_list, int joined_list_entries,
        struct memb_ring_id *ring_id);
    void (*sync_init) (void);
    int (*sync_process) (void);
    void (*sync_activate) (void);
    void (*sync_abort) (void);
};

struct corosync_service_handler_iface_ver0 {
    struct corosync_service_handler *(*corosync_get_service_handler_ver0) (void);
};

```

Figure 8: The Service Engine Plug-In Interface

```

typedef void *corosync_timer_handle;

struct corosync_api_v1 {
    int (*timer_add_duration) (
        unsigned long long nanoseconds_in_future,
        void *data, void (*timer_nf) (void *data),
        corosync_api_handle_t *handle);

    int (*timer_add_absolute) (
        unsigned long long nanoseconds_from_epoch,
        void *data, void (*timer_fn) (void *data),
        corosync_timer_handle_t *handle)

    void (*timer_delete) (corosync_timer_handle_t timer_handle);

    unsigned long long (*timer_time_get) (void);

    void (*ipc_source_set) (mar_message_source_t *source, void *conn);

    int (*ipc_source_is_local) (mar_message_source_t *source);

    void *(*ipc_private_data_get) (void *conn);

    int (*ipc_response_send) (void *conn, void *msg, int mlen);

    int (*ipc_dispatch_send) (void *conn, void *msg, int mlen);

    void (*ipc_refcnt_inc) (void *conn);

    void (*ipc_refcnt_dec) (void *conn);

    void (*ipc_fc_create) (
        void *conn, unsigned int service, char *id, int id_len,
        void (*flow_control_state_set_fn)
            (void *context, enum corosync_flow_control_state flow_control_state_set),
        void *context);

    void (*ipc_fc_destroy) (
        void *conn, unsigned int service, unsigned char *id, int id_len);

    void (*ipc_fc_inc) (void *conn);

    void (*ipc_fc_dec) (void *conn);

    unsigned int (*totem_nodeid_get) (void);

    unsigned int (*totem_ring_reenable) (void);

    unsigned int (*totem_mcast) (struct iovec *iovec, int iov_len,
        unsigned int gaurantee);

    unsigned void (*error_memory_failure) (void);
};


```

Figure 9: The Service Engine APIs

and to send responses to either the response or dispatch socket descriptor. Messages are automatically delivered to the correct service engine depending upon parameters in the message header.

The `ipc_source_set()` will set a `mar_message_source_t` message structure with the node id and a unique identifier for the IPC connection. A service engine uses this function to uniquely identify the source of an IPC request. Later this `mar_message_source_t` structure is sent in a multicast message via Totem. Once this message is delivered, the Totem message handler then can respond to the ipc request by determining if the message was locally sent via `ipc_source_is_local()`.

Each IPC connection contains a private data area private to the IPC connection. This memory area is allocated on IPC initialization and is determined from the `private_data` field in the service engine definition. To obtain the private data, the function `ipc_private_data_get()` function is executed by the service engine designer.

Every IPC connection is actually two socket descriptors. One descriptor, called the response descriptor, is used for requests and responses to the library user. These requests are meant to block the third-party process using the Corosync Cluster Engine until a response is delivered. If the third-party process doesn't desire blocking behavior, but may want to execute a callback within a dispatch function, the service engine designer can use `ipc_dispatch_send()` instead.

There are other APIs which are useful to manage flow control, but they are complex to explain in a short paper. If a designer wants to use these APIs, they should consider viewing the Corosync Cluster Engine wiki or mailing list.

5.3.4 Totem API

The Totem API is extremely simple for service engines to use with only three API functions. These functions obtain the current node ID, allow a failed ring to be reenabled, and allow the multicast of a message. Conversely, most of the complexity of Totem is connected to the Corosync service engine interface and hidden from the user.

To obtain the current 32-bit node identifier, the function `totem_nodeid_get()` function can be called. This is

useful when making comparisons of which node originated a message for service engines.

When Totem is configured for redundant ring operational mode, it is possible that an active ring may fail. When this happens, a service engine can execute `totem_ring_reenable()` via administrative operation to repair a failed redundant ring.

Service engines do a majority of their work by sending a multicast message and then executing some functionality based upon the multicasted message parameters. To multicast a message, an io vector is send via the `totem_mcast()` API. This message is then delivered to all nodes according to the extended virtual synchrony model.

5.3.5 Miscellaneous APIs

Currently many of the subsystems in the Corosync Cluster Engine are tolerant of failures to allocate memory. The exception to this rule may be the service engine implementations themselves. When a non-recoverable memory allocation failure occurs in a service engine, the `api_error_memory_failure()` is called to notify the Corosync Cluster Engine that the service engine calling the function has had a memory malfunction.

In the future, the Corosync Cluster Engine designers intend to manage memory pools for service engines to avoid any out of memory conditions or memory process starvation.

6 Security Model

The Corosync Cluster Engine mitigates the following threats:

- Forged Totem messages intended to fault the Corosync Cluster Engine
- Monitoring of network data to capture sensitive cluster information
- Malformed IPC messages from unprivileged users intended to fault the Corosync Cluster Engine

The Corosync Cluster Engine mitigates those threats via two mechanisms:

- Authentication of Totem messages and IPC users
- Secrecy of Totem messages with the usage of encryption

7 Integration with Third Party Projects

7.1 OpenAIS

OpenAIS [OpenAIS] is an implementation of the Service Availability Forum's Application Interface Specification. The specification is a C API designed to improve availability by reducing the mean time to repair through redundancy.

Integration with OpenAIS was a simple task since a majority of the Corosync functionality was reduced from the OpenAIS code base. When OpenAIS was split into two projects, some of the internal interfaces used by plug-ins changed. The usage of these internal APIs were modified to the definitions described in this paper.

7.2 OpenClovis

OpenClovis [OpenClovis] is an implementation of the Service Availability Forum's Application Interface Specification. OpenClovis uses some portions of the Corosync services. Specifically, it uses the Totem protocol APIs to provide membership for its Cluster Membership API.

7.3 OCFS2

The OCFS2 [OCFS2] filesystem can use the closed process group api to communicate various pieces of state information about the mounted cluster. Further the CPG service is used for supporting Posix Locking because of the virtual synchrony feature of the closed process group service.

7.4 Pacemaker

Pacemaker [Pacemaker] is a scalable High-Availability cluster resource manager formerly part of Heartbeat [LinuxHA]. Pacemaker was first released as part of Heartbeat-2.0.0 in July 2005 and overcame the deficiencies of Heartbeat's previous cluster resource manager:

- Maximum of 2-nodes
- Highly coupled design and implementation
- Overly simplistic group-based resource model
- Inability to detect and recover from resource-level failures
- Pacemaker is now maintained independently of Heartbeat in order to support both the OpenAIS and Heartbeat cluster stacks equally.

Pacemaker functionality is broken into logically distinct pieces, each one being a separate process and able to be rewritten/replaced independently of the others:

- cib—Short for Cluster Information Base. Contains definitions of all cluster options, nodes, resources, their relationships to one another and current status. Synchronizes updates to all cluster nodes.
- lrmd—Short for Local Resource Management Daemon. Non-cluster aware daemon that presents a common interface to the supported resource types. Interacts directly with resource agents (scripts).
- pengine—Short for Policy Engine. Computes the next state of the cluster based on the current state and the configuration. Produces a transition graph contained a list of actions and dependencies.
- tengine—Short for Transition Engine. Co-ordinates the execution of the transition graph produced by the Policy Engine.
- crmd—Short for Cluster Resource Management Daemon. Largely a message broker for the PE, TE, and LRM. Also elects a leader to co-ordinate the activities of the cluster.

The Pacemaker design of one process per feature presented an interesting challenge when integrating with Corosync which uses plug-ins/service engines to expand its functionality. To simplify the task of porting to the Corosync Cluster Engine, a small plug-in was created to provide the services traditionally delivered by Heartbeat.

At startup, the Pacemaker service engine spawns Pacemaker processes and respawns them in the event of failure. Cluster-aware components connect to the plug-in

using the interprocess communication manager. Those applications can then send and receive cluster messages, query the current membership information, and receive updates.

The Pacemaker components use the Pacemaker service engine features indirectly via an informal API which is used to hide details of the chosen cluster stack. The abstraction layer can automatically determine the operational stack and chose the correct implementation at runtime by checking the runtime environment. Once the Pacemaker service engine and abstraction layer were functional, Pacemaker was made stack independent, as shown in Figure 10, with little effort.

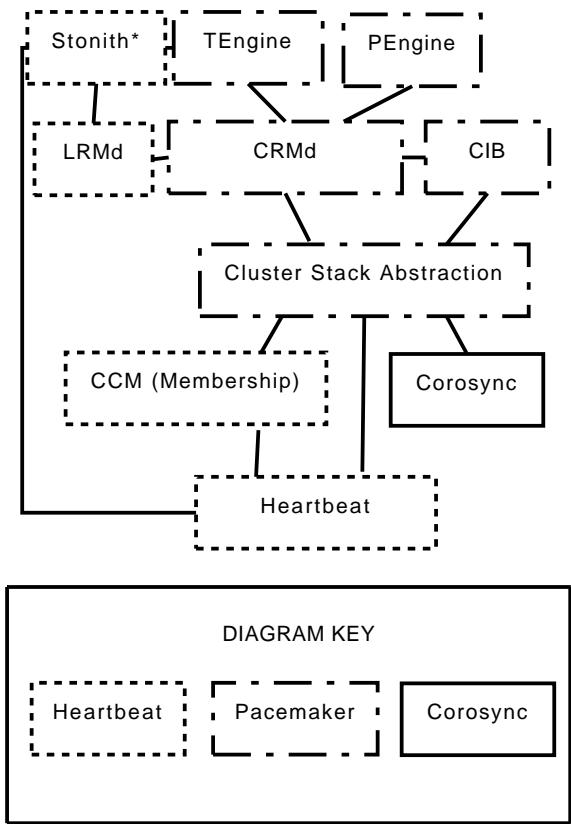


Figure 10. Pacemaker Dual Stack Architecture

Pacemaker components exchange messages consisting mostly of compressed XML-formatted strings. Representing the payload as XML is not efficient, but the format's verboseness means it compresses well, and complex objects are easily unpackable by numerous custom and standard libraries.

In order to accommodate Pacemaker, the Corosync Cluster Engine designers added ordered service engine shutdown. When an administrator or another service engine triggers a shutdown of the Corosync Cluster En-

gine, the service engines clean up and exit gracefully. This allows the Pacemaker service engine to organize for resources on a node to be migrated away gracefully and eventually stop its child processes before the Corosync Service Engine process exits.

7.5 Red Hat Cluster Suite

The Red Hat Cluster Suite [RHCS] version 3 uses the Corosync Cluster Engine. The Red Hat Cluster Suite uses a service engine called CMAN to provide services to other Red Hat Cluster Suite services.

Quorum is the main function of the CMAN service and is a strong dependency in all of the Red Hat Cluster Suite software stack. Quorum ensures that the cluster is operating consistently with more than half of the nodes operational. Without quorum, filesystems such as the Global File System can lead to data corruption.

The Quorum disk software communicates with the CMAN service via an API. The quorum disk software provides extra voting information to help the infrastructure identify when quorum has been met for special criteria.

Red Hat Cluster Suite uses a distributed XML-based configuration system called CCS. CMAN provides a configuration engine which reads Red Hat Cluster Suite specific configuration format files and stores them within the object database. This configuration plug-in overrides the default parsing of the `/etc/corosync/corosync.conf` configuration file format.

The libcman library provides backwards compatibility with the cman-kernel in Red Hat Enterprise Linux 4. This backwards compatibility is used by a few applications such as CCS, CLVMD, and rgmanager.

Red Hat Cluster Suite, and more specifically the Global File System component, makes use of the Closed Process Groups interface that is standardized within the CPG interface included in the Corosync Cluster Engine.

8 Future Work

The Corosync Cluster Engine Team intends to improve the scalability of the engine. Currently, the engine has been used in a physical 60 node cluster. The engine has been tested in a 128 node virtualized environment.

While these environments demonstrated the Corosync Cluster Engine works properly at large processor counts, the team wants to improve scalability to even larger processor counts and reduce latency while improving throughput.

The Corosync Cluster Engine designers desire to add a generically useful quorum plug-in engine so that any project may define its own quorum system.

Finally, the team wishes to add a generic fencing engine and mechanism for multiple plug-in services to determine how to fence cooperatively.

9 Conclusion

This paper has presented a strong rationale for using the Corosync Cluster Engine and demonstrated the design is generically useful for a variety of third-party cluster projects. This paper has also presented the current architecture and plug-in developer application programming interfaces. Finally, this paper has presented a brief overview of some of our future work.

References

- [Corosync] The Corosync Cluster Engine Community. *The Corosync Cluster Engine*, <http://www.corosync.org>
- [Amir95] Y. Amir, L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, and P. Ciarfella. *The Totem Single-Ring Ordering and Membership Protocol*, ACM Transactions On Computer Systems, 13(4), pp. 311–342, November, 1995. <http://www.cs.jhu.edu/~yairamir/archive.html>
- [Moser94] L.E. Moser, Y. Amir, P.M. Melliar-Smith, and D.A. Agarwal. *Extended Virtual Synchrony*, ACM Transactions on Computer Systems 13(4):311-342, November, 1995. Proceedings of DCS, pp. 56–65, 1994.
- [Dake05] S. Dake and M. Huth. *Implementing High Availability Using the SA Forum AIS Specification*, Embedded Systems Conference, 2005.
- [SaForumAIS] Service Availability Forum. *The Service Availability Forum Application Interface Specification*, <http://www.saforum.org/specification/download>
- [Birman93] K.P. Birman. *The Process Group Approach to Reliable Distributed Computing*, Communications of the ACM 36(12): 36-56, 103, 1993.
- [Koch02] R.R. Koth, L.E. Moser, and P. M. Melliar-Smith. *The Totem Redundant Ring Protocol*, ICDCS 2002:598-607.
- [Postel80] J. Postel. *User Datagram protocol*, Darpa Internet Program RFC 768, August 1980.
- [USC81] University of Southern California. *Internet Protocol*, Darpa Internet Program RFC 791, September 1981.
- [Deering98] S. Deering and R. Hinden. *Internet Protocol, Version 6 (IPv6) Specification*, IETF Network Working Group, December 1998.
- [OpenAIS] The OpenAIS Community. *The OpenAIS Standards Based Cluster Framework*, <http://www.openais.org>
- [OpenClovis] The OpenClovis Company. *OpenClovis*, <http://www.openclovis.org>
- [OCFS2] The Oracle Cluster File System Community. *The Oracle Cluster Filesystem*, <http://oss.oracle.com/projects/ocfs2>
- [Pacemaker] The Pacemaker Community. *Pacemaker*, <http://www.clusterlabs.org>
- [LinuxHA] The Linux-HA Community. *Linux-HA*, <http://www.linux-ha.org>
- [RHCS] Red Hat Cluster Suite. *The Linux Cluster Community Project*, <http://sources.redhat.com/cluster/wiki>

LTTng: Tracing across execution layers, from the Hypervisor to user-space

Mathieu Desnoyers

École Polytechnique de Montréal

mathieu.desnoyers@polymtl.ca

Michel Dagenais

École Polytechnique de Montréal

michel.dagenais@polymtl.ca

Abstract

This presentation discusses the upcoming changes to be proposed to the kernel tracing field by the LTTng community. It will start by explaining what has been mainlined (per-cpu atomic operations, Linux Kernel Markers.) Then, the focus will turn to the patch set currently developed and for which the mainlining process is in progress. An important part of this presentation will talk about the efficient system-wide user-space tracing infrastructure being designed. Work done for tracing across execution layers, including the Hypervisors, will also be shown.

The mainlining status of kernel tracing will be a key element of this talk. Considering the increasing amount of news articles written on this subject, many attendees, from the kernel hacker to the system administrator, should find interest in this presentation.

1 Introduction

Since last year’s symposium, where the need the industry has for a tracer in the Linux kernel has been demonstrated [1], the expectations from the Linux community for tracing tools matching DTrace [2] seem to have grown [4]. A lot has happened since then in the various tracing projects, with results still waiting to find their way into the kernel mainline.

This paper presents the current state of the work performed in the LTTng project which have been integrated or is planned to be integrated in the Linux kernel. It details the “Immediate Values,” improvements for the “Linux Kernel Markers” and discusses the kernel instrumentation patch set, based on the markers, submitted to the Linux community.

2 Related Work

Other projects with similar goals have already tackled areas of the tracing problem. Credit must be given to the K42 team [9] at IBM Research for developing a highly scalable operating system implements a lock-free, mostly atomic trace buffering mechanism (except for subbuffer switch.) The Kprobes developers at IBM, Intel, and Hitachi and the Djprobes [7] team at Hitachi have also pioneered the area of dynamic kernel code modification on the x86 architecture, providing the ability to insert custom instrumentation based on breakpoints or jumps based on dynamic code modification. The shortcoming of these two methods seems to be the performance impact of the breakpoint and the fact that none of these can guarantee access to the local variables in the middle of a function, since they can be optimized away by the compiler.

The SystemTAP [6] project is built on top of Kprobes and the Linux Kernel Markers to provide a scriptable language to create probes, which can be connected on any of those two information sources to extract information from the running kernel. LTTng learned from the lessons brought by the first generation of tracer, LTT [10]. It also reused the instrumentation found in LTT.

More recent work includes the “Driver Tracing Infrastructure” (DTI) [8] and the “Generic Trace Setup and Control” (GTSC), which aim at providing a standard driver tracing infrastructure for drivers.

3 Mainlining Status

In the past years, the LTTng project has proved its usefulness and yet, the ground work required in the Linux kernel before a kernel tracer can really become usable is not over. The next section will present the pieces of

infrastructure required by LTTng which have been integrated in the mainline kernel.

3.1 Linux Kernel Markers

LTTng depends on the Linux Kernel Markers [3] to provide the instrumentation of the core kernel. It uses the Linux Kernel Markers as primary information source, but could connect to other sources of information if needed. The markers provide an interface to source code instrumentation that simplifies adapting to code source changes, separating the concept of “high level trace event” from the actual code source. The markers can be dynamically activated, and can provide information to probes registering on specific markers from either the code kernel or GPL modules. Other projects, such as SystemTAP [6], also support hooking on markers.

3.2 Per-CPU Atomic Operations

The LTTng kernel tracer does not only need to be fast, but it also needs to be reentrant with respect to other execution contexts, when it reserves space in its memory buffer. Using per-CPU data structures and buffers helps eliminating false sharing and eliminates concurrency coming from other processors. However, local interrupts, both maskable and non-maskable (NMI), will try to write events to the same trace buffers concurrently.

The algorithm for lock-less NMI-safe buffer management [5] is based on extensive use of the compare-and-swap atomic operation. It is known, however, to be slower than interrupt disable on SMP systems. The Per-CPU atomic operations, also known as “local ops” do best of both: they offer reentrancy with respect to NMI contexts and are faster than interrupt disable on many architectures. The reason for such performance is that these operations don’t need neither LOCK prefix nor memory barriers, since they update memory local to a given CPU.

In addition to the LTTng tracer, the Per-CPU Atomic Operations are currently being used in an experimental patch for the SLUB allocator, which uses the local compare-and-swap primitive in the allocate and free fast paths. Initial performance improvements range from two to threefold compared to the version using interrupt disable.

4 Forthcoming Kernel Changes

4.1 Immediate Values Optimization

The immediate values are a derivative work of the Linux Kernel Markers. They provide an infrastructure to encode, in the instruction stream of the Linux kernel, static and global variables which are read-often, but updated rarely. The read-side does not have to read any information from data cache, since it’s already encoded in the form of an immediate value at each variable reference site; therefore, all the information needed is present within the instruction stream.

Updates are done dynamically by updating the immediate values in the load immediate instructions on a live running kernel, upon each variable modification.

The original goal of immediate values was to provide a very efficient activation mechanism for the Linux Kernel Markers. With their current version, in kernel 2.6.25, each encountered marker adds a memory read to check if the marker is enabled. The impact on data cache therefore grows as more markers are added to kernel cache-hot instruction paths.

Using the immediate values, to encode the branch condition in the instruction stream, helps solving this problem. Instead of polluting the data cache, markers based on immediate values encode the branch condition directly in the instruction stream. Assembly example of immediate values use by markers on x86_32 and x86_64 goes as follow. The first example focuses on the added code to `schedule()` cache-hot code. It adds 2 bytes for the immediate value load, a 2-byte test and a 6-byte conditional near jump, for a total of 10 bytes.

```
356: b0 00          mov    $0x0,%al
358: 84 c0          test   %al,%al
35a: 0f 85 1e 03 00 00 jne   67e <schedule+0x449>
```

For smaller functions such as `wake_up_new_task()`, the conditional jump only takes 2 bytes since the offset can be expressed as a short jump, for a total of 6 bytes.

```
848e: b0 00          mov    $0x0,%al
8490: 84 c0          test   %al,%al
8492: 75 7f          jne   8513 <wake_up_new_task+0x97>
```

This infrastructure can be used simply by replacing every reference to a static or global variable “var” by a `imv_read(var)` and by changing each update to the variable by an `imv_set(var)`, the latter being a pre-emptable function. Variables with size of 1, 2, 4, or 8 bytes can be referred to. If the architecture does not support updating one of these type size on a live system, a normal variable read is used. This is the case for a 8 bytes variable on a 32 bits x86, which cannot be encoded as an immediate value of a single instruction, and for variables larger than 2 bytes on PowerPC, because instructions are limited to 4 bytes in size and take only 2 bytes operands. If no immediate value optimization is implemented for a given architecture, the generic fallback is used: a standard memory read.

Some work is currently being done to improve even further immediate values used as boolean condition for a branch. The goal is to minimize the impact of disabled markers on a running system, replacing the `mov`, `test` and `branch` instructions by a sequence of 2-byte nops and either a 2-byte short jump or a 1-byte `nop` and 5-byte jump. Since the compiler might reorder instructions between the `mov`, `test`, and `jne/je` instructions, this optimization is only done when the pattern is detected as unmodified by the compiler. Initial results show that the 97% of the 120 trace points added to the Linux kernel in the LTTng instrumentation do not suffer from such compiler modifications on x86_32 and that the success rate stays at 90% on x86_64. Knowing for sure where the `test` and `branch` instructions are would require some work on the compiler.

The performance impact of a loop instrumented with different techniques is shown in tables 1 and 2. This loop executes some ALU work in the baseline. It is then compared with the performance impact of the same loop with an added inactive marker using a sequence of `mov`, `test`, and `branch` instructions, and with a normal marker reading a memory variable.

It is then compared with the “`ftrace`” approach, using a function call replaced by nops. The latter method is also used in DTrace. The second column shows the same results with a baseline which flushes the data cache containing the information accessed to show how each method behaves when the data is cache cold. We can see that the cache cold impact is much higher for the disabled function call when it references information not present in the cache or in the registers. This is required to perform the stack setup, even if the function call is

disabled with nops. The non-optimized markers have a similar data cache impact.

The difference of impact between the cached runs could be considered as non-significant and amortized by the pipeline, but the real difference comes from the uncached memory accesses, where the runtime cost ranges from 41.8 to 154.7 cycles.

It must be noted that, on the code size aspect, the markers also add about 50 bytes in an unlikely branch. With `gcc -O2` or `-freorder-blocks`, this branch is placed away from cache-hot instructions and therefore does not stress the instruction cache. The data added by each marker is placed in a special section, only needed when the markers are activated.

4.2 Instrumentation

Once the marker infrastructure is in place to support instrumentation, the following step to have a useful tracer is to start integrating a core instrumentation set in the kernel. The instrumentation proposed in the LTTng project is divided into architecture independent and dependant patch sets.

Architecture independent instrumentation is by far the largest, yet the simplest, instrumentation with 86 markers inserted in the filesystem, inter-process communication, kernel, memory management, networking, and library code. Its simplicity comes from the fact that it only instruments C code in a straightforward way. It’s therefore easy to benefit from the small performance overhead of the markers.

The architecture dependant instrumentation currently supports the following architectures, from the most complete to the less: x86_32, x86_64, PowerPC, ARM, MIPS, SuperH, Sparc, and S/390. Instrumentation at the assembly level requires some extra mechanisms to efficiently extract information from system calls. Those are implemented in the form of a new `TIF_KERNEL_TRACE` thread flag added to every architecture. It can be enabled or disabled at runtime to control system call tracing activation for all the system threads. This new thread flag is tested in assembly to check if the `do_syscall_trace()` functions, which contains the system call entry and exit markers, must be called.

In addition to the instrumentation of the kernel code, dumping the kernel structures requires the addition of

x86 Pentium 4, 3.0GHz, Linux 2.6.25-rc7	Added cycles (cached)	Added cycles (uncached)
Optimized marker	0.002	0.07
Normal marker	0.004	154.7
Stack setup + (1+4 bytes) NOPs (6 local var.)	0.04	0.6
Stack setup + (1+4 bytes) NOPs (1 pointer read, 5 local var.)	0.03	222.8

Table 1: Comparison of markers and disabled function impact on x86_32

AMD64, 2.0GHz, Linux 2.6.25-rc7	Added cycles (cached)	Added cycles (uncached)
Optimized marker	-1.2	0.2
Normal marker	-0.3	41.8
Stack setup + (1+4 bytes) NOPs (6 local var.)	-0.5	0.01
Stack setup + (1+4 bytes) NOPs (1 pointer read, 5 local var.)	2.7	51.8

Table 2: Comparison of markers and disabled function impact on x86_64

new in-kernel accessors. This information extraction is typically done at trace start to have a complete picture of the operating system state. When a trace is examined in a viewer, this recorded initial state can be updated using the information in the trace, and the system state is thus available for viewing and analysis purposes for the whole trace duration. Functionality must be added to dump the important kernel structures in the trace buffers, in a way that permits to identify when the data structures are changing concurrently. Typically, the /proc file system expects the kernel structures to stay unchanged between two consecutive reads. If they change, it will result in the loss of information that can't be linked with the element being added or removed from the structures. The output text will be truncated at the offset of the currently requested read operation.

The LTTng state dump module dumps the kernel structures to the trace in multiple iterations, releasing the locks after a fixed number of elements, to make sure operations such as dumping all the memory maps of all the processes in the system won't generate high latencies. Detection of concurrent data structure modification is done by the rest of the kernel instrumentation; since every manipulation to these data structures is traced, the trace analyzer can re-create the data structure at any given point of the trace after the end of state dump.

5 User Space Tracing

Work performed in the user-space tracing area involved porting the Linux Kernel Markers to user-space so that they can be used in libraries. The linker scripts are modified to add a new section which contains the markers placed in each object. A library init function is linked with each object to allow registration of the markers to the kernel through an additional system call.

Activation of markers can then be done system-wide. It would allow to easily turn on instrumentation of the NPTL pthread mutexes at the user-space level, or to instrument glibc memory allocation primitives and link this information with the kernel memory requests.

As a first step, the extraction of information could be done through a string, passed as an argument to a trace system call. The reason for using system calls rather than other mechanisms is that this technique does not depend on other libraries to open files and help instrumenting user-space programs executed at boot time.

Eventually, extracting the information without going through a system call would help to minimize tracing performance impact. It would, however, imply that shared buffers should be made accessible for writing to each traced process. Because of security concerns, these buffers cannot be shared between the various processes, as done in the K42 research operating system. There is therefore still work to do in this area.

6 Hypervisor Tracing

The Xen hypervisor has already its own tracer, xentrace. It exports fixed-size data to userspace through a buffer shared with a process running on domain0. The process communicates with the hypervisor to activate tracing through hypercalls.

An experimental port of LTTng to the Xen hypervisor has been realized. The lttng-xen daemon has been created by modifying the lttng daemon to use new hypercalls rather than debugfs. The same has been done to lttctl and liblttctl: they have been ported to use hypercalls rather than a netlink socket. Because we use variable-sized events, which represent the data in its most compact form, we were able to generate traces twice as small as xentrace.

The main interest in having a tracer extracting information in the same format as the operating system and userland is to help analyze concurrency, race and other timing problems across execution domains.

7 Conclusion

With Kprobes and Linux Kernel Markers already in the mainline kernel, the road seems to be opening for integration of more parts required to have a solid tracing infrastructure in the kernel, namely the immediate values, a kernel instrumentation, and eventually, support for userspace tracing.

Once the kernel goals are reached, the focus will be easier to turn on the other aspects of tracing, which includes the choice of userland markers and standardization of hypervisor tracing.

References

- [1] Martin Bligh, Rebecca Schultz, and Mathieu Desnoyers. Linux kernel debugging on google-sized clusters. In *Proceedings of the Ottawa Linux Symposium 2007*, 2007.
- [2] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *USENIX '04*, 2004.
- [3] Jonathan Corbet. Kernel markers. August 2007.
- [4] Jonathan Corbet. On dtrace envy. August 2007.
- [5] Mathieu Desnoyers and Michel Dagenais. The lttng tracer : A low impact performance and behavior monitor for gnu/linux. In *Proceedings of the Ottawa Linux Symposium 2006*, 2006.
- [6] Frank Ch. Eigler. Problem solving with systemtap. In *Proceedings of the Ottawa Linux Symposium 2006*, 2006.
- [7] Masami Hiramatsu and Satoshi Oshima. Djprobes - kernel probing with the smallest overhead. In *Proceedings of the Ottawa Linux Symposium 2006*, 2006.
- [8] David Wilder. Unified driver tracing infrastructure. In *Proceedings of the Ottawa Linux Symposium 2007*, 2007.
- [9] Robert W. Wisniewski and Bryan Rosenburg. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In *Supercomputing, 2003 ACM/IEEE Conference*, 2003.
- [10] Karim Yaghmour and Michel R. Dagenais. The linux trace toolkit. *Linux Journal*, May 2000.

Getting the Bits Out: Fedora MirrorManager

Matt Domsch

Dell

Matt_Domsch@Dell.com

Abstract

Fedora is fortunate to have several hundred volunteer mirror organizations globally. MirrorManager tracks all of these mirror servers and automatically directs users to a local, fast, current mirror. It has several unique features, including registration of private mirrors and designation of preferred mirrors by IP address—a great benefit to corporations, ISPs, and their users; and automatic direction of Internet2 clients to Internet2 mirrors. This paper presents the web application architecture that feeds updates to over 200,000 users each day. It provides instructions for setting up local private Fedora or EPEL mirrors for schools, companies, and organizations, and explains how you can volunteer to help distribute Fedora worldwide.

1 Introduction

The Fedora Project (hereafter ‘Fedora’) is a leading-edge Linux distribution that provides the newest and best Free and Open Source Software to millions of users worldwide. MirrorManager (MM) [9] is the tool developed to get that software out to those users accurately, quickly, and inexpensively.

To assist with this distribution, Fedora is fortunate to have several hundred volunteer mirror organizations globally. These organizations provide manpower (responsive system administrators), servers, storage, and copious bandwidth. Each mirror server carries a subset of the content available on the Fedora master servers. It is often fastest and least expensive for these mirrors to serve users whom are “local” network-wise. MM tracks all of these mirror servers and automatically directs users to a local, fast, current mirror.

We present MM from three aspects. Section 3 shows how end users download software transparently using MM. Section 4 shows how mirror system administrators interact with MM. Section 5 goes behind the scenes into the design of the MM software itself.

2 Background

There are three factors to consider when scoping the size of the distribution channel you need: number of users, size of the software, and available network bandwidth.

By conservative estimates [7], Fedora has nearly 2 million users worldwide. Neglecting the number of users who buy or receive free CDs, at a minimum each user downloads one CD worth of material (about 700MB). This equates to at least 1.4 Exabytes of data to serve for each release. With a single 45 Mbit/second T3 network connection, it would take over 8 years to serve all this content. Security and bugfix updates could easily double this number. At this rate, Fedora releases occur every 6 months, we’d fall behind very quickly (not to mention lose our entire user base!).

As for total disk space, Fedora keeps at least the current release (at time of press, Fedora 9), the previous release (Fedora 8), and the next previous release (Fedora 7) online and available for download. Each Fedora version release, including packages, CD and DVD images, and daily security and bugfix updates, can consume up to 200GB of disk space. In addition, alpha and beta test releases, and the “rawhide” tree (the development tree for what will be the next major release), are posted regularly. These consume a bit less space than a full release. Overall, about 1TB of space is constantly needed on the master servers and for each full mirror.

While our mirror organizations are altruistic, they’re also not overly wasteful. Each mirror may choose to carry only a subset of the available content, such as omitting lesser-used architectures and debug data. This means it’s not sufficient to know which mirrors exist, but we must also know which content each carries. This precludes using a simple DNS round-robin redirector.

Further complicating matters, due to historical ways in which the content was offered via rsync modules, each

mirror server may publish their tree of the Fedora content at paths of their choosing—often not matching that of the master servers. This makes it even more important that tools can discover the content a mirror carries, and at which URLs that content is served—a naïve redirect would fail miserably.

Organizations have several reasons why they choose to become a Fedora mirror. Generally, they have many Fedora users locally, and for those users, it's faster (and for the organization, less expensive) if they can pull that content from a local mirror rather than across the Internet multiple times. For large Internet Service Providers or organizations, the savings can be quite dramatic.

Organizations that are part of Internet2, or one of the high speed research and educational networks that peer with it, often have significantly lower costs and higher bandwidth when passing traffic over Internet2 than over their commercial links. Fedora itself does not have any public download servers that are accessible via Internet2, but more than half of the Fedora public mirror servers are accessible via Internet2. By directing users to local or Internet2-connected mirrors, they can get the benefit of high speed downloads at a reduced cost.

2.1 Sidebar: Preventing Meltdown

One of the driving forces behind MM is to get the bits to end users as fast as possible. A related goal is to keep Fedora's primary sponsor, Red Hat, online during release week.

In October 2006, Fedora had around 100 active mirrors. During the days leading up to a release, individual mirror admins would report by email that they were synced. However, the list of mirrors was managed manually, included in release announcements manually, and generally quite error-prone (dozens of text files had to be updated correctly, once for each mirror reporting ready).

When Fedora Core 6 was released that month, demand was immense—over 300,000 installs in the first three weeks—larger than ever seen for a Red Hat Linux or Fedora release. A few dozen mirrors were synced in time for the release, but nowhere near sufficient capacity to handle the demand. It didn't help that the web page most users were being directed to in order to begin their download pointed them to use Red Hat's own servers, not mirror servers.

On top of this, an apparent Distributed Denial of Service attack was mounted against Red Hat's own servers on release day. Talk about kicking you when you're down.

The result: for the week following the Fedora Core 6 release, significant portions of Red Hat's network became unusable for anything other than responding to the DDoS attack and serving Fedora content. You can imagine the joy this brought to Red Hat executives. The mirrors were annoyed that they would finally get synced, only to not be listed on the mirror list web pages (the Fedora sysadmins were busy trying to handle the traffic and keep everything running, and were slow getting those manual lists updated). Chaos and confusion.

Thus MM was born, to address the shortcomings of manually updating dozens of text files, and to ensure all known mirrors were accounted for and being put to good use.

Six months later, MM made its debut with the Fedora 7 release. Fortunately, there was no DDoS attack this time, and while there were some growing pains getting all the mirrors listed in the database, it went quite smoothly.

In November 2007, Fedora 8 was released. With every confidence in MM and the mirrors themselves, the Red Hat servers were removed from public rotation—Red Hat served bits to the mirrors, but served very few end users directly. From Red Hat's perspective, the release went so smoothly they didn't even know it happened. Users were able to get their downloads quickly. Life was good.

3 Getting the Bits: End Users

End users have several options for downloading Fedora CDs, DVDs, and packages. Outside the scope of MM, Fedora serves the content via BitTorrent. However, tools such as yum do not use BitTorrent, and network restrictions by a user's organization may prevent BitTorrent or other peer-to-peer download methods.

Critical to the goal of delivering mirrored content to users quickly is the *redirector* which automatically redirects user download requests to an up-to-date, close mirror, using several criteria:

- The user's IP address is compared against a list of network blocks as provided by each mirror server.

If a user is on a network served by a listed mirror server, the user is directed to that network-local mirror. This should be the fastest and least expensive way to serve this user.

- If the user is on a network served by Internet2 or its peers, they are redirected to another Internet2-connected mirror in their same country, if available. MaxMind's open source and zero-cost GeoIP database provides country information.
- Users are directed to mirrors in their same country, if any.
- Users are directed to mirrors on their same continent, if any.
- Users are directed to one of the mirrors globally.

This search algorithm, while not always perfect, provides a pretty good approximation of the Internet topology, and in practice has shown to provide acceptable performance for users. In the event a user wants to manually choose a mirror, he or she can look at the list of available up-to-date mirrors [6].

To override this search algorithm in some way (e.g. because GeoIP guesses the country incorrectly, or because the actual network you're on is near a border with another country where there is a faster mirror), users may append flags to the URLs used ([3] or [5]). Table 1 describes the available flags.

4 Hosting the Bits: Mirrors

MM offers several features aimed specifically to assist mirror server administrators most efficiently serve their local users, as well as global users, such as:

- The ability to have “private” mirrors—those which serve only local users and which are not open to the general public.
- The ability to specify the network blocks of their organization. Local users from that organization will be automatically directed to their local mirror.
- The ability to specify the specific countries a mirror should serve.

- The ability to preferentially serve users on Internet2 and related research and educational networks.

These features help keep down bandwidth costs for serving Fedora users.

4.1 Signing Up

These are the steps involved with registering as a Fedora mirror, either to serve the public, or to serve your own organization.

1. Create yourself a Fedora Account System account [2]. You should have one account per person in your organization who will maintain your mirror. You will be able to list these people as administrators for your mirror site.
2. Log into the MM web administration interface [10].
3. Create a new Site. Sites are the administrative container, and where your organization can get sponsorship credit for running a public mirror. Public mirrors are listed on a fedoraproject.org web page with a link to each sponsoring organization.
4. Create a new Host. Hosts are the individual machines, managed under the same Site, which serve content. Sites may have unlimited numbers of Hosts.
5. Add Categories of content for each Host. Most mirrors carry the “Fedora Linux” category (current releases and updates), while some also carry the “Fedora EPEL” (Extra Packages for Enterprise Linux) [1], “Fedora Web” (web site), and “Fedora Secondary Arches” (secondary architectures such as ia64 and sparc) categories.
6. Add your URLs for each Category. Most mirrors serve content via HTTP and FTP; some also serve via rsync.

In addition, you can set various bits about your Site and Host, including its country, whether it's connected via Internet2 or its peers, whether it's private or public, your local network blocks, etc.

Table 1: mirrorlist flags

Flag	Description
country=us, ca, jp	Return the list of mirrors for the specified countries.
country=global	Return the global list of mirrors instead of a country-specific list.
ip=18.0.0.1	Specify an IP address rather than the one the server believes you are connecting from.

Private Sites or Hosts are those which expect to only serve content to their local organization. As such, they will not appear on the public-list web pages. Hosts default to being “public” unless marked “private” on either the Site (which affects all Hosts), or individually on the Hosts’s configuration page. Private Hosts are ideal for universities who have one mirror for internal users, and another they share with the world. Private hosts are returned to download requests based on matching client IP to a Host’s netblock.

Netblocks are a feature unique to MM. You may specify all of the IPv4 and IPv6 network blocks, in CIDR format, that your mirror should preferentially serve. Users whose IP addresses fall within one of your netblocks will be directed to your mirror first. There is one security concern, as this could allow a malicious mirror to direct specific users to them. However, as all content served by the mirror system as a whole is GPG-signed by the Fedora signing keys, to be successful the attacker would have to convince the target user to accept their GPG keys as well, which, one hopes, would be unlikely. Mirrors may not set overly large netblocks without MM administrator assistance, further limiting the scope of such possible attack.

Internet2 detection is done by regularly downloading and examining BGP RIB files from the Internet2 log archive server. This data includes all the CIDR blocks visible on Internet2 and its peer research and educational networks worldwide. Clients determined to be on Internet2 will be preferentially directed to a mirror on Internet2 in their same country, if possible. By setting the Internet2 checkbox for the Host, your Host will be included in that preferential list. In addition, private Hosts on Internet2 may be happy to serve clients on Internet2, even if they don’t fall within the Host’s list of netblocks. MM provides this option as well.

Each Host should list the IP addresses from which they download content from the master servers. These addresses are entered into the rsync Access Control List

on the master servers, as well as on the Tier 1 mirrors. This is used to limit the users who may download content from the master mirror servers, so as to not overload them.

4.2 Syncing

Fedora employs a multi-tier system [4] to speed deployments, similar to other Linux distributions. Tier 1 mirrors pull from the Fedora master servers directly, Tier 2 mirrors pull from the Tier 1 servers. Private mirrors pull from one of the Tier 1 or 2 mirrors.

Unique to MM, the tool `report_mirror` is run on each mirror server immediately after each `rsync` run completes. This tool informs the MM database about the full directory listing of content carried by that mirror. The MM database for each Site contains a password field, used by `report_mirror` to authenticate this upload, so as to not expose an individual user’s Fedora Account System username and password.

5 Architecture

The MM software follows a traditional 3-tier architecture of database back-end, application server, and front-end web services. It is written in python, and leverages the TurboGears rapid application development environment. However, some specific design decisions were made to address the memory consumption and multi-threaded locking challenges that python imposes. We split the most often hit web services out from the application server, exactly to address the memory demands.

5.1 Application Server and Database

MM uses TurboGears [13], with the SQLAlchemy [12] object-relational mapper layer for most data, and the SQLAlchemy [11] mapper for integration with the Fedora Account System. The application server provides several entry points:

- The administrative web interface [10], where mirror administrators register their mirrors and can see the perceived status.
- A limited XMLRPC interface used by the `report_mirror` script, run on the mirror servers, to “check in” with the database.
- A web crawler, which detects which mirrors are up-to-date. In conjunction with the `report_mirror` script, this follows the “trust, but verify” philosophy. Mirrors which are unreachable, even temporarily, are removed from the redirector lists.

The database itself can be anything that SQLObject can speak to, including PostgreSQL and MySQL. SQLObject takes care of creating the proper tables and mapping rows into objects. For speed and memory efficiency, some queries are implemented in SQL directly.

5.2 Crawler

The second half of the “trust, but verify” philosophy is the web crawler. This application first updates its record of content found on the master servers. For each public Host, it then scans, using lightweight HTTP HEAD or FTP DIR requests (depending on protocols served by that Host), each file that Host is expected to contain. For large directories full of RPM files, only the most recent 10 files are scanned to cut down on extra unnecessary lookups. Directories where all the files match the master servers are marked up-to-date in the database; unreachable servers or those whose content does not match are marked as not up-to-date, effectively preventing clients from being directed to those Hosts’ directories. The crawler can run against several Hosts at once, limited only by available memory on the crawling system.

The crawler extends python’s `httplib` to use HTTP keep-alives. This lets it scan about 100 files per server per TCP connection using HTTP HEAD calls which do not download the actual file data, and thus are very fast.

5.3 Web Services

5.3.1 Mirrorlist Redirector

To the end user, the most critical service MM provides is the mirrorlist redirector [5], which directs users to a

mirror for the content they request. This service receives all the requests generated by yum looking for package updates, and individuals downloading CD and DVD images from the front page of fedoraproject.org. These services operate on a cache of the database, containing pre-computed answers to most queries, for maximum speed.

As this application gets hundreds of hits each second, a pure `mod_python` solution was infeasible—it simply wasn’t fast enough, and the memory consumption (upwards of 30MB per `httpd` process waiting to service a client) overwhelmed the servers. So, we split the application into two parts: a `mod_python mirrorlist_client` app, which marshalls the request and performs basic error checking and HTTP redirection, and a `mirrorlist_server` app, which holds the cache and computes the results for each client request. `mirrorlist_server` fork()s itself on each client connection, keeping the cache read-only (so `copy-on-write` is never invoked), which eliminates the memory consumption problems and python interpreter startup times. The two communicate over a standard Unix socket. Client requests are answered in about 0.3 seconds on average.

This pair of applications is then replicated on several web servers, distributed globally. This reduces the likelihood of a single server or even data center failure bringing down the service as a whole. In the event of application server or database layer failure, the web services can operate on the cached data indefinitely, until the back ends can be made available again.

5.3.2 Publiclist pages

Aside from the redirector, the second user-visible aspect of MM are the “publiclist pages,” web pages that list each up-to-date available public mirror and its properties, including country, sponsoring organization, bandwidth, and URLs to content. These pages are rendered once per hour into static HTML pages and served via HTTP reverse proxy servers, again to make use of caching. This keeps the traffic load manageable, even on very active major release days.

6 Future Work

There are several features MM does not currently provide which would be useful additions.

- MM lists each mirror's available bandwidth, but does not use this information when choosing which mirrors to return in what order. This causes both relatively fast and slow mirrors in the same country to be returned with equal probability. MM should take into account a given Host's available bandwidth, and return a list of mirrors probabilistically favoring the faster mirrors.
- `report_mirror` does not work from behind a HTTP proxy server. Private mirrors need to run this tool, but are often stuck behind such a proxy. This is actually a shortcoming of python's `urllib`.
- Metalink [8] downloads, which would let users pull data from several mirrors in parallel. This is somewhat controversial, as it increases the load on the mirrors (they wind up serving more random read requests, which are much slower than streaming reads). But it might let metalink-aware download tools do a better job of choosing a “close” mirror than MM does.

7 Conclusion

MM has been very effective in getting Fedora content to users quickly and easily. Furthermore, it has decreased the bandwidth burden of Fedora's primary sponsor, Red Hat, by making good use of the contributions from hundreds of volunteer mirror organizations worldwide. Its architecture allows it to serve millions of users, and to scale as demand grows. It's simple and fast for users, and saves money for mirror organizations—a win all around.

8 Acknowledgments

MM is primarily developed for the Fedora Project on behalf of the author and his employer, Dell, Inc. It is licensed under the MIT/X11 license.

MM includes GeoLite data created by MaxMind, available from <http://www.maxmind.com/>.

The Fedora Project is grateful to the hundreds of mirror server administrators and their organizations who help distribute Free and Open Source software globally.

9 About the Author

Matt Domsch is a Technology Strategist in Dell's Office of the CTO. He has served on the Fedora Project Board and as the Fedora Mirror Wrangler since 2006.

References

- [1] Extra Packages for Enterprise Linux. <http://fedoraproject.org/wiki/EPEL>.
- [2] Fedora Account System. <https://admin.fedoraproject.org/accounts>.
- [3] Fedora download site. <http://download.fedoraproject.org>.
- [4] Fedora mirror tiering. <http://fedoraproject.org/wiki/Infrastructure/Mirroring/Tiering>.
- [5] Fedora mirrorlist used by yum. <http://mirrors.fedoraproject.org/mirrorlist>.
- [6] Fedora Project public mirror servers. <http://mirrors.fedoraproject.org>.
- [7] Fedora Project statistics. <http://fedoraproject.org/wiki/Statistics>.
- [8] Metalink. <http://www.metalinker.org>.
- [9] MirrorManager. <http://fedorahosted.org/mirrormanager>.
- [10] MirrorManager administrative interface. <https://admin.fedoraproject.org/mirrormanager/>.
- [11] SQLAlchemy. <http://sqlalchemy.org>.
- [12] SQLObject. <http://sqlobject.org>.
- [13] TurboGears. <http://turbogears.org>.

Applying Green Computing to clusters and the data center

Andre Kerstens
SGI
kerstens@sgi.com

Steven A. DuChene
SGI
sduchene@sgi.com

Abstract

Rising electricity costs and environmental concerns are starting to make both the corporate IT and scientific HPC worlds focus more on green computing. Because of this, people are not only thinking about ways to decrease the initial acquisition costs of their equipment, but they are also putting constraints on the operational budgets of that same equipment. To address this challenge, we use both commercial and open-source Linux tools to monitor system utilization and closely track the power usage of those systems. The results of our monitoring are then used to make real-time decisions on whether systems can be put to sleep or shutdown altogether. In this paper we show how to use the Ganglia monitoring system and Moab scheduling engine to develop a methodology that guarantee the most efficient power usage of your systems by helping Moab make intelligent decisions based on real-time power data and incoming workload.

1 Introduction

In the last five years, corporate and research data centers have grown significantly due to the increasing demand for computer resources. Not only has the power used by these computer systems roughly doubled over this period, but also the energy consumed by the cooling infrastructure to support these computer systems has increased significantly. In addition to the resulting increase in data center capital and operational costs, this expanding energy use has an impact on the environment in the form of carbon-dioxide emissions that are created as an unwanted by-product of the electricity generation. In their Report to Congress on Server and Data Center Energy Efficiency [1], the Environmental Protection Agency (EPA) estimates that servers and data centers consumed about 61 billion kilowatt-hours in 2006 (1.5 percent of the total US electricity consumption) and that this will double by the year 2011 (an annual growth rate

of 9 percent). Recent findings by the Uptime Institute [2] show that the EPA numbers are probably too conservative and that the annual growth rate from 2006 to 2011 is more likely to be 20 to 30 percent. No matter who is right in the end, it is obvious that measures have to be taken to limit the increase in power consumption. On a global level, organizations like the Green GridSM have started defining metrics for energy efficiency. They are developing standards and measurements methods to determine data center efficiency against these metrics. On a local level, we can increase the efficiency of our data centers and reduce operating costs by decreasing the power and cooling loads of our computing infrastructure.

To achieve this objective, we propose to use intelligent monitoring and control of data center resources. With a combination of open-source and commercial Linux software like Ganglia [3] and Moab [4], we will be able to monitor system utilization as well as closely track the power usage of those systems. The information collected are used to make real-time decisions on whether systems can be put to sleep, run in a lower power mode, or shutdown altogether. The rest of this paper is organized as follows: After some history on green computing efforts in Section 2, we discuss the details of our methodology in Section 3, and show a case study in Section 4. We conclude the paper in Section 5.

2 Green Computing

Wikipedia defines Green Computing as the study and practice of using computing resources efficiently [5]. This comes down to designing computer systems that optimize the performance of the system compared to the cost of running (i.e., electricity) and operating it (i.e., power distribution and cooling).

In the past decade there have been some disconnected efforts by government, academic, and corporate facilities as well as data center managers and system vendors

to increase the energy efficiency of servers and other computing resources. On a national and global level, organizations such as Green GridSM [6], ASHRAE [18], and Climate Savers Computing InitiativeSM[19] are defining metrics for energy efficiency and are developing methods to measure energy efficiency against these metrics. For example, Green GridSM has defined the Power Usage Effectiveness (PUE) metric [7] that enables data center operators to estimate the energy efficiency of their data centers and determines if energy efficiency improvements need to be made.

There have been a number of other attempts to manage the consumption of server systems and clusters. Rajamani and Lefurgy worked on identifying system and workload factors in power-aware cluster request distribution policies [13]. Elnozahy *et al.* have studied dynamic voltage scaling and request batching policies to reduce the energy consumption of web server farms [14]. Fan *et al.* studied the aggregate power usage characteristics of up to 15 thousand servers for different classes of applications over a period of approximately six months to evaluate opportunities for maximizing the use of the deployed power capacity of data centers and assess over-subscribing risks [15]. Chase *et al.* looked at how to improve the energy efficiency of server clusters by dynamically resizing the active server set and how to respond to power supply disruptions or thermal events by degrading service in accordance with negotiated Service Level Agreements (SLAs) [12]. Pinheiro *et al.* developed a system that dynamically turns cluster nodes on or off by considering the total load on the system and the performance implications of changing the current configuration [16].

Our goal is to expand these efforts to full data centers or clusters of data centers. For example, by moving workloads around based on the cost of power and cooling in various geographical locations or by delaying workloads to run during off-peak electricity rates, additional cost savings can be realized.

3 Methodology

Our method of green computing consists of three primary components:

1. Collect data to monitor resource state (power/temperature);

2. Interfacing to power management facilities; and
3. Enabling intelligent policies to control power consumption and remove hot spots.

The following sections discuss these components in detail; they will be combined in a case study that is described in Section 4.

3.1 Data Collection

Most modern computer systems, if not all, contain a service processor that provides out-of-band remote management capabilities. In most open or commodity-based systems, this service processor is a Baseboard Management Controller (BMC) which provides access to Intelligent Platform Management Interface (IPMI) capabilities. Various tools to access the BMC can be used to monitor sensor information like temperature, voltages, fan speeds, and power status. The BMC also provides remote network access to power on and power off systems. The BMC operates independent of the processor and the operating system, thus providing the ability to monitor, manage, diagnose, and recover systems, even if the operating system has crashed or the server is powered down, and as long as the system is connected to a power source. Other service-processor-based, out-of-band management systems such as RSA cards, iLO, ALOM, ILOM, or DRAC implement similar feature sets using vendor-specific tool sets.

Most currently available BMCs support either IPMI 1.5 or IPMI 2.0 with common sensors of fan speeds, cpu temperatures, board temperature, cpu voltages, power supply voltages, etc. On a system with a BMC that supports IPMI 2.0, a power sensor is more likely to be present that reports watts being used by the system. The sensor data returned from `ipmitool` run against a BMC that supports IPMI 1.5 looks like this:

CPU Temp 1	29 degrees C	ok
CPU Temp 2	28 degrees C	ok
CPU Temp 3	no reading	ns
CPU Temp 4	no reading	ns
Sys Temp	27 degrees C	ok
CPU1 Vcore	1.31 Volts	ok
CPU2 Vcore	1.31 Volts	ok
3.3V	3.26 Volts	ok

5V	4.90 Volts	ok
12V	11.81 Volts	ok
1.5V	1.49 Volts	ok
5VSB	4.85 Volts	ok
VBAT	3.28 Volts	ok
Fan1	13200 RPM	ok
Fan2	11200 RPM	ok
Fan3	13200 RPM	ok
Fan4	11200 RPM	ok
Fan5	13200 RPM	ok
Fan6	11100 RPM	ok

Once the access to the sensor data from the BMC is confirmed across the cluster, the various sensor values can be pulled into a cluster monitoring system like Ganglia, Nagios, or MRTG. Since the authors of this paper are most familiar with it, Ganglia will be used to monitor and record historical data about the systems, clusters, and other data center power and cooling equipment. Ganglia has web-based data-display mechanisms as well as command-line tools to peek at the data stream. The web-based display is based around `rrdtool` just like MRTG.

By getting this sensor data into a monitoring tool like Ganglia, historical data becomes available for performance trend analysis and post-problem root cause analysis.

Modern data center infrastructure equipment such as Power Distribution Units (PDU), Uninterruptible Power Supplies (UPS), chillers, and Computer Room Air Conditioning (CRAC) units are IP-enabled and understand network protocols (e.g., SNMP or HTTP) for communication with other systems on the network. Using this capability, the electric current through a UPS or a PDU branch circuit can be measured and the temperature of the water in a chiller or the return line of a CRAC unit can be requested. Some models of rack PDUs can measure the power draw per PDU outlet. This provides an opportunity for measuring power usage of servers that do not have IPMI capabilities. Often these infrastructure devices can also be controlled over the network, but that discussion falls outside of the scope of this paper.

Adding these data sources from CRAC units or PDUs into the monitoring system provides a more complete picture of data center conditions over long periods of time or for spot analysis of daily, weekly, or monthly trends. For example, it could show that a particular set

of systems or areas of a data center get unusually hot on weekends. The cause of this could be that some error in data center facilities setup is not taking into account system loads on weekends.

3.2 Power Management Interface

Various power states are available in a Linux system, i.e., everything from a 100% power utilization to powered-off. By being able to intelligently control the current power level based on current and future system load expectations, we can take maximum advantage of the power savings available. Within an HPC environment, where a job control system is used to process incoming workloads, the system load expectations can be fairly predictable. By placing the power state of HPC cluster client systems under control of the job control scheduler, the incoming workload can drive the power demands of the cluster. In this case, nodes can be switched off when the workload is light and switched on again when the workload is expected to go up. However, even when a system is running a job, there are power savings benefits possible by controlling the power usage of non-critical system components. Modern processors, disks, and other components can have varying states of power usage. By taking advantage of the ability to dynamically control the power state, the system is able to make adjustments when a device is idle for a significant period of time. The degree of significance here is different for a processor versus a disk or other components: processor idle times can be of milliseconds, while disk idle times are in the range of seconds.

There are currently four different processor power-saving states described in the Advanced Configuration and Power Interface (ACPI) specification [10]: C0, C1, C2, C3, and C4. Table 1 shows the power usage of the Intel Core 2 Duo processor in each of these states. Note that the lower the power-saving state, the longer it will take to wake up from that state.

Many of these numbers come from the LessWatts organization [8], which does research into power saving for Linux systems [9]. To see the current power state as well as power states supported by a system use:

```
cat /proc/acpi/processor/CPUx/power
```

where x is a number ranging from 0 to the number of CPUs in the system. For example, the output on a Core2Duo system looks as follows:

C-State	Max Power Consumption (Watt)
C0	35
C1	13.5
C2	12.9
C3	7.7
C4	1.2

Table 1: Intel Core 2 Duo maximum power consumption in the different C-states

```
active state: C3
max_cstate: C8
bus master activity: 00000000
maximum allowed latency: 2000 usec
states:
C1: type[C1] promotion[C2] demotion[--]
latency[001] usage[00000010]
duration[00000000000000000000]
C2: type[C2] promotion[C3] demotion[C1]
latency[001] usage[181597540]
duration[0000000631489359035]
*C3: type[C3] promotion[--] demotion[C2]
latency[057] usage[1438931278]
duration[0000006636332340366]
```

With the use of a script, the active power state of the system CPUs can be monitored over time with Ganglia.

When a low-power state is entered, it is best to stay in that state as long as possible for the greatest energy savings. Unfortunately, older Linux kernels have a regularly occurring interrupt, called a timer tick, that is used to alert the kernel when some housekeeping tasks have to be performed. This feature limited the usefulness of the lower power states (for example, C3 or C4), because the system could only stay in that state in between timer ticks (1, 4, or 10 ms). In newer Linux kernels, starting with 2.6.21 for x86 and 2.6.24 for x86_64, a tick-less timer was introduced which made it possible to keep the processor in a lower power state for a longer amount of time, at least until the next timer event occurred. Unfortunately, there is still much code around (e.g., applications, device drivers) which does not take energy efficiency into account and which triggers the kernel hundreds of times per second to wake it up and do some work. Until this situation changes (and it is indeed slowly changing due to projects like Lesswatts.org [8]), saving power on Linux using power-saving states continues to be a struggling task.

The ACPI specification also defines four system sleep states:

- S1 – Stopgrant – Power to CPU is maintained, but no instructions are executed. The CPU halts itself and may shut down many of its internal components. In Microsoft Windows, the “Standby” command is associated with this state by default.

- S3 – Suspend to RAM – All power to the CPU is shut off, and the contents of its registers are flushed to RAM, which remains on. In Microsoft Windows, the “Standby” command can be associated with this state if enabled in the BIOS. Because it requires a high degree of coordination between the cpu, chipset, devices, OS, BIOS, and OS device drivers, this system state is the most prone to errors and instability.

- S4 – Suspend to Disk – CPU power shut off as in S3, but RAM is written to disk and shut off as well. In Microsoft Windows, the “Hibernate” command is associated with this state. A variation called S4BIOS is most prevalent, where the system image is stored and loaded by the BIOS instead of the OS. Because the contents of RAM are written out to disk, system context is maintained. For example, unsaved files would not be lost following an S4 transition.

S4 is currently not supported by the 2.4.x kernel series in Linux, but you might have good luck with SWSUSP. Some machines offer S4_BIOS whose support is considered to be experimental within Linux/ACPI.

- S5 – Soft Off – System is shut down, however some power may be supplied to certain devices to generate a wake event—for example, to support automatic startup from a LAN or USB device. In Microsoft Windows, the “Shut down” command is associated with this state. Mechanical power can usually be removed or restored with no ill effects.

These sleep states are activated by writing values to the file /sys/power/state. The current state can be queried by reading this file.

Processor voltage and frequency scaling are other techniques for managing power usage that have been available in consumer platforms for some years and only recently have been made available to server-class processors [11]. Processor voltage scaling is used

to run a processor (or processors) at a lower voltage than the maximum possible while frequency scaling is used to run a processor at a lower frequency than the maximum possible to conserve power. In Linux 2.6.x systems, frequency scaling can be controlled through the directory `/sys/devices/system/cpu/cpu0/cpufreq/`. There are several governors available to control the frequency scaling behavior of the system, e.g., conservative, on-demand, power-save, user-space, performance. The list on a specific system can be shown by viewing the file `/sys/devices/system/cpu/cpu0/cpufreq/scaling_available_governors`. The governor can be changed by writing a new value into the file `/sys/devices/system/cpu/cpu0/cpufreq/scaling_governor`. Both processor voltage and frequency scaling techniques are heavily used on notebooks and other systems containing mobile processors. Unfortunately mobile processors are not used for data center workloads normally and thus many of the power-saving features discussed above are not applicable to server platforms. On the other hand, vendors like Intel are planning to bring power management features usually found in their mobile line of processors like the Core 2 Duo to server platforms like the Xeon family of processors [9]. By passing on the power-savings features from the mobile market to the server market, a whole new range of possibilities for energy conservation is entering the data center.

In HPC clusters, the concept of shutting down nodes is a relatively new concept since most HPC clusters leave nodes running 24/7 except for scheduled maintenance outages. This is not necessary with current workload management systems. One issue is that cluster administrators often perceive shutting down nodes as a probable cause of power supply failures or hard drive spin-up failures. However, our view is that if components are going to break, and this is inevitable, they should do so in a known, controlled manner at a time that jobs are not scheduled on these resources. Deliberately taking a node down is the clearest indicator of the reliability of your cluster. Doing these tests, everything from UPS loading, to switch fabric, to control and mediation, as well as the hardware of the node itself is affected.

3.3 Scheduling and Control

The power management functionality discussed in Section 3.2 is not necessarily passed through to the con-

trol of the workload scheduler. We will start by doing coarse-grained control through Moab, the workload scheduler we chose in this paper.

Moab uses IPMI or similar capabilities to monitor temperature information, check power status, power-up, power-down, and reboot compute nodes. This information can be utilized to make intelligent scheduling decisions that can save energy, limit or remove hot spots in the data center, and open up the possibility of implementing chargeback structures for users, groups, or departments based on energy usage.

In this context, the first action is to specify a pool of idle nodes that are accessed by the scheduler when the workload of a cluster changes. To achieve this, the scheduler utilizes workload prediction algorithms to determine when idle nodes are required to run a queued workload and switches them on accordingly by taking into consideration the time the node needs to boot up. Initially all nodes in the pool will be idle and switched on for instant workload response, but when nodes have not been requested after a specified time, the nodes go to the off state and power down. The status of the idle pool is transparent to the end users and workload. If service level agreements (SLAs) are in place, the idle pool can be dynamically adjusted based on the requested quality of service (QoS) level. To maximize the number of nodes that are powered off, and if memory bandwidth requirements allow it, jobs can be densely packed together on nodes instead of running all on separate nodes. Checkpoint/restart job migration strategies can be part of this scheme to make sure that power consumption is minimized.

A second method of energy conservation is to utilize cost-aware scheduling. The scheduler needs to be made aware of expensive peak and cheaper off-peak electricity rates and the times these rates are in effect. Only time-critical workloads are scheduled during the more expensive time periods where the peak electricity rate is in effect, while other, less time-critical workloads are deferred to off-peak time periods. This type of scheduling can be extended to incorporate summer and winter rates which are used by many utility companies to provide seasonal discounts to their customers. If an organization operates in multiple geographically dispersed data centers or co-locations, one could go a step further and migrate workloads to the least expensive data center based on the time of day at those locations and the electricity rates that are in effect in these locations. Tak-

ing advantage of any significant rate differences from these distinct locations, countries, and even continents, additional cost savings can be realized. From a user perspective, as long as the input and output data sets are readily available, it would not make a difference if their workload is run in New York or in Hong Kong, but from a energy cost perspective it makes a significant difference.

Gartner's research, performed in 2007, shows that 47% of data centers in the US will run out of power by 2008 [17]. For such data centers, Moab can be instructed to utilize daily limits based on watts per user or group. Again, these can vary for different times of a day and different seasons.

Moab's ability to learn application behavior can be utilized to find out on which systems certain applications use minimum power with acceptable performance. The data that is gathered during the learning process can then be used to create the system mapping which defines the optimal performance/power ratio for each application. This mapping can subsequently be used during workload scheduling.

Not only can Moab make scheduling decisions to optimize power usage in a data center, but it also can be configured to make sure that any localized hot spots are minimized in a computing facility. Hot spots occur when certain systems or nodes are highly utilized, but do not receive sufficient cooling to keep temperature at an acceptable level. It is important that any server or data center issues leading to recurring hot spots are investigated as soon as possible, because over-temperature events in a server can be directly correlated to failure rates of systems. A node that runs hot all the time has a larger probability of component failures than a node that is kept within its specified temperature range. Most modern server systems have service processors that allow real-time remote monitoring of temperatures and fan speeds. This data can be used to set limits which can then be used to make intelligent scheduling decisions to resolve or minimize the effects of hot spots. For example, several parts of a workload can be distributed over certain nodes to best balance the heat dissipation in the data center. If a raised floor environment is used to supply server racks with cool air, often the systems that are mounted higher in a rack receive less cool air and thus run hotter. A scheduling policy can be implemented, based on data gathered during a learning process, that will always schedule workloads with lower processor

utilization (and thus generate less amounts of heat) on nodes that are located in these higher temperature areas of a rack or data center.

In this section we have discussed a number of ways to schedule workload with the goal of lowering power consumption of and removing hot spots in a data center. The next section shows a detailed case study that uses a number of these methods in a data center facility.

4 Case Study

Let's consider this hypothetical scenario: Doodle Computing Inc. owns a co-location data center with four large cluster systems which are used by many manufacturing companies to run Computer Fluid Dynamics (CFD) and Finite Element Analysis (FEA) studies to design and optimize their products. Figure 1 shows the layout of the data center in detail. As shown in the figure, the cluster racks are laid out in a hot/cold aisle configuration to create an effective separation of hot and cold air. The Computer Room Air Conditioning (CRAC) units are placed at the end of the hot aisles to provide a short distance for the hot waste air to return to the unit. The cold aisles contain perforated floor grates to allow the cold air from underneath the raised floor to come up to the air intake of the servers in the racks.

Doodle Computing Inc. currently owns four different clusters: A, B, C, and D, all with approximately the same processor speeds, but with different power efficiencies. Table 2 shows the number of nodes/blades each cluster contains and how much power each of these nodes/blades consumes.

Cluster	Number of Nodes	Power Usage / Node (Watts)
A	1040	700
B	560	500
C	280	500
D	2688	1000

Table 2: Cluster Properties

Clusters B and C are some years older and contain a mix of 1U and 2U nodes with dual single-core processors per node. Cluster A is only one year old and consists of high-density blades with two dual-core processors each. As shown in the figure, one of the racks of cluster A runs a little hot (depicted by the red dot). Cluster D

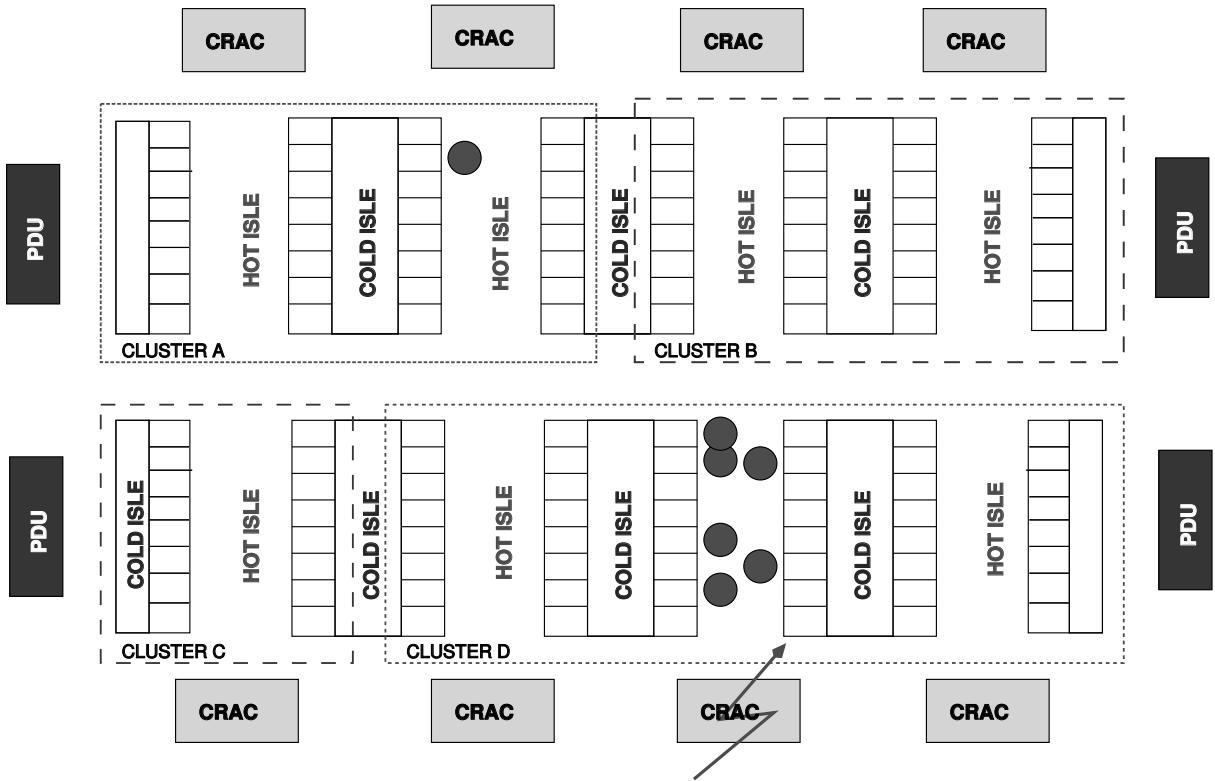


Figure 1: Doodle Computing Inc.’s co-location data center layout

is the newest cluster in the data center and consists of very high density blades with four quad-core processors each. Due to the high power usage of the nodes and the fact that the CRAC unit serving that aisle is broken and does not cool the air sufficiently any more, there are a number of hot spots in the second hot aisle of that cluster (see Figure 1 for details). The monthly energy bill for Doodle Computing Inc. of approximately \$160,000 has led the management team to believe that savings should be possible if the data center can more intelligently manage the resources. Doodle Computing Inc. has standardized on Moab and TORQUE as their scheduler and resource managers. The servers and cluster nodes of clusters A, B, and D have a BMC interface and provide temperature and power information to the outside world, while cluster C only reports temperature. To make it possible to monitor the power usage of this cluster, the facilities department has installed IP-enabled rack PDUs in the racks of cluster C. This type of PDU can report on the power usage of every outlet and submit that information as a reply to an HTTP request.

There are two different electricity rates in effect in the location where Doodle Computing Inc. operates their

co-location data center: \$0.10/kWh from 7 AM to 7 PM and \$0.05/kWh from 7 PM to 7 AM. By implementing several of the green policies that were discussed in Section 3.3 of this paper, significant power savings can be achieved and the observed problem with hot spots in the Doodle Computing Inc. co-location data center is resolved. We will show in the remainder of this section how this is done.

Moab has knowledge of the power usage of compute nodes in all of the clusters and thus it can make intelligent decisions where the incoming workload has to be scheduled to optimize power usage. For our scenario, let’s consider a job that runs optimally on 64 processors or processor cores, has an approximate running time of 4 hours, and whose result has to be available within 24 hours of submission. For such a job, Moab can calculate the respective cost of running the job on each cluster as follows:

- Cluster A 16 nodes of four cores, each is needed to run the job. This represents a power usage of $16 \times 700 = 11.2$ kW. For a four-hour run, 4×11.2

= 44.8 kWh is needed. The cost of this run is $44.8 * \$0.10 = \4.80 .

- Cluster B and C 32 nodes of two processors, each is needed to run the job. This represents a power usage of $32 * 500 = 16$ kW. For a four-hour run, $4 * 16 = 64$ kWh is needed. The cost of this run is $64 * \$0.10 = \6.40 .
- Cluster D for nodes of 16 cores, each is needed to run the job. This represents a power usage of $4 * 1000 = 4$ kW. For a four-hour run, $4 * 4 = 16$ kWh is needed. The cost of this run is $16 * \$0.10 = \1.60 .

It is obvious that the job most efficiently runs on cluster D. If a node on cluster D is not available, cluster A is the next choice, followed by clusters B or C. Because the result of the job only has to be available in 24 hours, even more cost savings can be made by scheduling this job at a different time of the day, e.g., after 7 PM when the resource cost is lower. By running the job at night, the cost on cluster D decreases to a mere 80 cents (\$2.40 on cluster A and \$3.20 on cluster B and C).

For all clusters, idle pools are created for nodes that have been switched off. Moab can instruct clusters to switch off any compute nodes that have not been utilized in the past hour and for which no reservations or jobs in the queue exist. For example, assume that at 9 PM cluster A is 50 percent utilized, cluster B is 40 percent utilized, cluster C is not utilized at all, and cluster D is 80 percent utilized. Also assume that reservations exist that utilize all of the clusters at 100% starting from 7 o'clock the next morning. In such a situation, half of the nodes of cluster A (520 nodes) can be switched off and move to the idle pool until 7 AM. This represents a cost saving of: $520 * 700W * 10h * 0.05 = \182 . In addition, 60 percent of the nodes in cluster B (336 nodes) can be moved to the idle pool which represents a cost saving of $336 * 500W * 10h * 0.05 = \84 . All the nodes in cluster C (280 nodes) can be moved to the idle pool for a saving of $280 * 500W * 10h * 0.05 = \70 . Last but not least, 20% of the nodes in cluster D (538 nodes) can be moved to the idle pool for a cost saving of $538 * 1000W * 10h * 0.05 = \269 . This means that a total of \$605 is saved by switching off compute nodes and move them into the idle pool for 10 hours.

To solve the hot-spot problems, Moab is provided with node temperature limits and instructed to assess the temperature output of the node BMCs on a regular basis.

This way, if it finds that the temperature of the nodes in the second hot aisle of cluster D is too high, the workload can be migrated to cooler nodes in the same cluster. If D nodes are not be available, the workload can be migrated to cluster A, B, or C. Moab can react even more proactively if it has access to data from the CRAC units so it can react to any problems with those units. In this case, the workload can be migrated more promptly making sure that hot spots do not get a chance to occur and nodes in that hot aisle are switched off or turned to sleep if the CRAC unit has an extended problem. The hot spot found in cluster A can be taken care of in a similar way.

5 Conclusion

In this paper we present several techniques for monitoring and controlling power consumption and temperature. Through a case study we show how these tools can be practically deployed in a data center facility. In particular we show how real-time monitoring and intelligent scheduling of workload can be efficiently utilized to lower the energy cost of data centers. In the scenario we have used, we also show ways to limit or remove temperature hot spots.

References

- [1] EPA Report to Congress on Server and Data Center Energy Efficiency, 2007, http://www.energystar.gov/ia/partners/prod_development/downloads/EPA_Datacenter_Report_Congress_Final.pdf
- [2] Uptime Institute, <http://uptimeinstitute.org>
- [3] Ganglia, <http://ganglia.sourceforge.net>
- [4] MOAB, <http://www.clusterresources.com/pages/products/moab-cluster-suite.php>
- [5] Wikipedia Green Computing, http://en.wikipedia.org/wiki/Green_computing
- [6] The Green Grid, <http://www.thegreengrid.org>

- [7] Green Grid Metrics: Describing datacenter power efficiency, 2-20-2007,
http://www.thegreengrid.org/gg_content/Green_Grid_Metrics_WP.pdf
- [8] Less Watts: Saving Power with Linux on Intel Platforms, <http://www.lesswatts.org/>
- [9] Less Watts Whitepaper, http://oss.intel.com/pdf/lesswatts_whitepaper.pdf
- [10] ACPI specification Version 3.0b, 10/6/2006, ACPI Working Group,
<http://www.acpi.info/spec.htm>
- [11] Intel Corporation. Dual-Core Intel Xeon Processor LV and ULV Datasheet.
<http://download.intel.com/design/intarch/datashts/31139101.pdf>, September, 2006.
- [12] J.S. Chase, D.C. Anderson, P.N. Thakar, and A.M. Vahdat, Managing Energy and Server Resources in Hosting Centers, Proc. 18th Symp. Operating Systems Principles, ACM Press, 2001, pp. 103–116.
- [13] K. Rajamani and C. Lefurgy, On Evaluating Request-Distribution Schemes for Saving Energy in Server Clusters, Proc. IEEE Intl Symp. Performance Analysis of Systems and Software, IEEE CS Press, 2003, pp. 111–122.
- [14] M. Elnozahy, M. Kistler, and R. Rajamony, Energy Conservation Policies for Web Servers, Proc. 4th Usenix Symp. Internet Technologies and Systems, Usenix Assoc., 2003, pp. 99–112.
- [15] X. Fan, W. Weber, and L.A. Barroso, Power Provisioning for a Warehouse-sized Computer, Proc. of the ACM International Symposium on Computer Architecture, San Diego, CA, June 2007.
- [16] E. Pinheiro, R. Bianchini, E.V. Carrera, and T. Heath, Dynamic Cluster Reconfiguration for Power and Performance, Compilers and Operating Systems for Low Power, Kluwer, 2003, pp. 75–94.
- [17] Gartner: 2007 Data Center Conference Poll Results for Power and Cooling Issues.
- [18] ASHRAE, <http://www.ashrae.org/>

Introduction to Web Application Security Flaws

Jake Edge

LWN.net

jake@lwn.net

Abstract

You hear the names of the most common web security problems frequently: cross-site scripting, SQL injection, cross-site request forgery—but what do those terms mean? This paper will provide an introduction to those vulnerabilities along with examples and ways to avoid them. This introduction is language-independent, as the problems can occur in any language used to develop web applications.

Developers of web applications sometimes get caught up in the excitement of developing the application and forget to consider the security implications. This paper will help them get a handle on what to avoid so that the excitement doesn't get squashed by an attacker. Others who are curious about the kinds of attacks made against web applications will also find much of interest.

1 Introduction

Web application vulnerabilities make up a fairly large slice of security vulnerabilities reported on Bugtraq and other security mailing lists. In addition, they are probably the type of vulnerability that Linux users are most likely to come across.

The consequences of a web vulnerability vary greatly, from full compromise of a vulnerable server application—potentially the server machine itself as well—to stealing authentication information so that a user's account, often on an unrelated site, can be accessed by an attacker. This highlights the broad reach of web application vulnerabilities as they can affect particular sites *or* the users who visit them.

2 Hypertext Transfer Protocol (HTTP)

Hypertext Transfer Protocol (HTTP) is the language spoken by web applications. It is a fairly simple, text-oriented protocol that is easy to read and understand. A

web browser sends an HTTP request and awaits a response from the server. That response is generally text in Hypertext Markup Language (HTML), but can also be other types of data: images, audio, video, etc. The browser then displays the response from the server and awaits another user action (e.g. clicking a link, submitting a FORM, using a bookmark to go elsewhere, etc.).

The most common HTTP requests are of the following three types:

- **HEAD** – this retrieves the headers and dates associated with a page so the browser can determine whether it can use its cached version of the object (HTML, image, etc.).
- **GET** – This is the workhorse of HTTP. Retrieve content based on a URL, with parameters passed as part of the URL (e.g. `http://foo.com/bar?baz=42`).
- **POST** – This is used by FORMs. The parameters are encoded into the request and POSTed to a specific URL, which is specified in the FORM tag.

Figure 1 shows a short example of using `telnet` to talk to a web server. The “GET” and “Host:” lines are typed by the user (followed by two carriage returns) with the response from the server following. The headers are sent first followed by a blank line and then the content, in this case the HTML of the document (abbreviated in the figure).

There is an important distinction between GET and POST that web application programmers should be aware of. GET requests should be *idempotent*, that is they should not change the state of the application. Multiple GET requests should return the same content, unless the underlying state of the application has been changed via a POST request. It is common for web

HTTP EXAMPLE

```
$ telnet lwn.net 80
Connected to lwn.net.

GET /talks/ols2008/ HTTP/1.1
Host: lwn.net

HTTP/1.1 200 OK
Date: Mon, 28 Apr 2008 03:54:40 GMT
Server: Apache
Last-Modified: Mon, 28 Apr 2008 03:49:48 GMT
ETag: "428105-30c-4815495c"
Accept-Ranges: bytes
Content-Length: 780
Content-Type: text/html

<html>
<head>
<title>OLS 2008</title>
</head>
...
```

Figure 1: Example of HTTP using telnet

applications to have state-changing links (which correspond to GETs), though there are two good reasons not to do that.

One classic example is a web page with links to delete content that looked something like: `http://somehost.com/delete?id=4`. A web spider then came along to index the site and found all of these links to follow—promptly deleting all the content on the site. The other reason to avoid state-changing GET requests is to prevent trivial cross-site request forgery as will be described below.

Another important thing to note about HTTP is that it is a stateless protocol. There is no inherent state information kept by the server and client. Each request is an entity unto itself. Various mechanisms have been used to achieve stateful web applications, the most common is the idea of a *session*. Sessions are typically set up by the server, given some kind of identification number (i.e., session ID), and then set as a *cookie* in the user’s browser in the response from the server. Cookies are persistent values, associated with a particular website, that are sent by a browser whenever a request is made of that website.

3 User input cannot be trusted

Many web application vulnerabilities share a fairly simple characteristic: insufficient or incorrect filtering of user-controlled input. This input can come as part of the URL, from FORM data, or from cookie values. These inputs make up most, if not all, of the attack surface of an application and *must* be appropriately filtered before use. The filtering should use a whitelisting, rather than blacklisting, approach—only allowing known-good inputs is far safer than trying to construct a list of all “bad” inputs.

One common mistake that web application programmers make is to assume that all traffic generated to their program will come from a web browser. They assume that certain things are “impossible” because a web browser does not do it that way. This is a grievous error, as it is trivial to generate HTTP traffic from any programming language—many provide full-featured libraries to do just that. An attacker can use a browser and easily manipulate parameters passed as part of the URL in a GET request, but using FORMs is no protection. Generating a POST request with the appropriate parameters is a simple task that is often done in Javascript as part of an exploit. Cookie values can also be created or stolen from another user.

Perhaps the most common manifestation of this mistake is in using Javascript validation. Web application programmers will often write some Javascript to run on the browser to validate values typed into a form. For example, a form might have a place to type in an IP address, with Javascript that ensured the values were legitimate—integers in the right range—popping up an alert box if the values were not valid. This may help some users and is a reasonable thing for a web application to do. The mistake is in not doing the same validation on the server. Any validation done by Javascript needs to be repeated on the server side. There is no guarantee that the user has Javascript enabled—even if the application tries to force it—or even that it is a browser at the other end. A program can easily be written to submit any value of any kind for that parameter. Browser-based limitations on length or type of a field in a form are *not* enforced if the browser is not present.

4 Cross-site scripting (XSS)

One of the more common vulnerabilities seen for web applications is *cross-site scripting* (XSS¹). XSS results from taking user input and echoing it back to the browser without properly filtering it for HTML elements. Many web applications allow users to store some content, a comment on a story for example, in a database on the server. This content is then sent back to the browser for that user or others as appropriate for the application. Consider the following “content” `<script>alert ("XSS")</script>`. If that is sent to the browser unchanged, it will cause Javascript to pop up an alert.

Any user input that gets sent back to the browser is a potential XSS vector. One common mistake is for an error message to contain the unrecognized input, which is helpful for a legitimate user who made a mistake, but can also be used as part of an XSS attack. Typically, XSS vulnerabilities are described with a proof-of-concept that just uses a Javascript alert which tends to make some underestimate the power of XSS. It is important to note that XSS attacks are capable of anything Javascript can do, which is a *lot*.² One of the more common uses of XSS is to steal cookie information from the browser which can then be used for session hijacking or other attacks.

There are two major flavors of XSS, the non-persistent XSS, where the attack comes as part of the link—like the error message example above—and persistent XSS, where the attacker stores something persistently on the server that can attack each time that content is accessed. Both flavors can have serious effects, but the amount of malicious code that can be contained in a link is somewhat limited, whereas the database will happily store a great deal more. Vulnerable applications may be storing page contents for MySpace or Facebook-like uses, comments on a blog posting, or some other lengthy content, any of which may be effectively unlimited in size.

The only defense against XSS is to filter all user input before echoing it back to the browser. Table 1 shows the

Input Character	Output HTML Entity
<	<
>	>
((
))
&	&
#	#

Table 1: Character mapping for HTML entities

recommended filtering rules. Mapping each listed character to its HTML entity equivalent will prevent XSS.

Depending on the implementation language, there may be functions (like `htmlentities()` or `cgi.escape()`) that do some or all of the filtering job. Note that some implementations may not do all of the recommended transformations, which could possibly lead to an XSS hole.

5 Cross-site request forgery (XSRF or CSRF)

Another type of web vulnerability—in some ways related to XSS—is *cross-site request forgery* (CSRF or XSRF³). XSRF abuses the trust that a web site has in the user, typically in the form of cookies, to cause an action on that site from an unrelated site.

To see how this works, consider a state-changing GET on a particular web site, perhaps one for a broadband router. If a particular URL on the site, `http://router/config?setDNS=1.2.3.4` for example, will change the router’s DNS setting, a completely unrelated attack site could use that URL in an image tag: ``. This would cause a request to be made of the router *with* any cookie the browser has stored for the router. If the cookie was used for authentication and had not yet expired, the action would be performed.

It is not just GETs that can be attacked via XSRF, POSTs are vulnerable as well. Using Javascript—and often hidden away in an IFRAME—FORMs can be constructed and submitted with values under the control of an attacker. A user could be lured to the attack site via a link in an email or other web page. Once they arrive, the attack site could generate a FORM submission to a

¹For web abbreviations, CSS was already taken by Cascading Style Sheets.

²For some examples, including network scanning behind firewalls, stealing web browser history, and more, see <http://jeremiahgrossman.blogspot.com/2006/07/my-black-hat-usa-2006-presentation.html>.

³For consistency with XSS, this paper will use XSRF.

```
Username: x
Password: ' OR 1=1; --
Query: SELECT id FROM users WHERE name=' $name' AND password=' $pword'
Results in: SELECT id FROM users WHERE name=' x' AND password=' ' OR 1=1; --'
```

Figure 2: SQL injection example

```
$stmt = prepare("SELECT id FROM users WHERE name=? AND password=?")
execute($stmt, $name, $pword)
```

Figure 3: SQL prepared statement and placeholder example

completely unrelated site—a banking, stock trading, or shopping site for example—which would be transmitted along with any cookies the user has for the site. If the user has recently logged in, the form action will be taken, just as if the user visited the site and filled in the form that way.

For high-profile sites, the URL is easy to know, but even for local devices like the broadband router mentioned above, the URL can often be guessed. It is very common for a particular model of router to be handed out en masse to subscribers of a particular service—often by default their IP address is fixed. So an attacker that wanted to do a phishing scam might choose a vulnerable router type and try to do XSRF attacks to 192.168.1.1, for example.

Getting rid of XSRF holes is somewhat complicated. First, as described above, state-changing GETs must be eliminated from the site. These are clearly the easiest XSRF hole to exploit.

For FORMs, there needs to be something that cannot be reliably predicted—or brute forced—in the FORM data. The best way to do that is with a TYPE=HIDDEN FORM element that has unpredictable NAME and VALUE attributes. Each should be generated separately, be long enough to resist brute force, and be tracked on the server side associated with the session ID. Whenever a FORM is submitted, the NAME and VALUE of the element should be validated, with any action dependent on that validation.

It should be noted that XSS can provide a means for Javascript to read the FORM and gather the required information, so a site must be free of XSS issues or the avoidance mechanism above can be circumvented.

It should also be noted that for very sensitive operations, re-authenticating the user can provide absolute protection against XSRF—assuming a good password has been chosen. It is common for web applications to require the current password before changing to a new password, which is an example of this technique.

6 SQL injection

SQL injection is, after XSS, the most commonly reported web site application flaw. Because many web sites are backed by some kind of SQL database, there are a large number of applications that are potentially vulnerable. SQL injections can lead to data loss, data disclosure, authentication bypass, or other unpleasant effects.

SQL injection abuses the queries that the web site does as part of its normal operation by injecting additional SQL code—under the control of the attacker—into the query. It is usually done through parameters to GET or POST requests by taking advantage of naïve—or nonexistent—attempts to protect the query from the parameter values.

In Figure 2, there is an example of how a SQL injection can happen. The SQL query is generated by interpolating the FORM variables for username and password into the statement. Under normal circumstances, when a user is trying to log in, the SQL statement works fine to select an ID if the username/password matches someone in the database. If an attacker types in the '**OR 1=1; --**'⁴ string for the password, he modifies the query as shown. This has the effect of returning every row in the *users* table. Typically application code is written to just take the

⁴The “--” tells the SQL engine to ignore the rest of the query, similar to a comment.

first returned result in that case, which should be a valid user—and may in fact be the first user added, which is often the administrative user.

Depending on how the web application is structured and what database system is used, other abuses are possible. Some databases allow multiple statements separated by “;” so a password of ‘; **DROP TABLE users;** --’ would end up removing all users from the database. There are ways to use SQL injection to discover all of the tables in the database and their contents, again depending on the database system.⁵

When trying to work out how to create a SQL injection for a site, an attacker may need to try multiple different techniques. The error messages returned by the web application often make it easier to determine what needs to be added to the injection to make it work because they disclose what the problem is (i.e., “Missing parentheses,” “Unterminated string,” and the like). Even unhelpful error messages can give clues to an attacker if the application responds differently to well-formed SQL that uses correct table and column names versus illegal SQL. That difference can be exploited by a technique known as *blind SQL injection*.

Thankfully, there are straightforward ways to avoid all SQL injection attacks. Converting an existing codebase may be somewhat tedious and time-consuming, but the method is easy. Essentially all of the techniques boil down to having the database treat the user input as a single entity that is used in the proper place in the query—as opposed to textually substituting that text into a query string.

The overall best technique is to use *prepared statements* with *placeholders* for the values that are to be used in the query. Different database systems use different placeholder syntax—and various languages’ database libraries obscure it more—but a common choice is the “?” character.

Queries are then created using the placeholder and passed to the database `prepare()` function. Figure 3 gives an example in a kind of pseudocode. Instead of textually substituting the `$name` and `$pword` variables into the query, the database system uses them internally to match. Doing it that way, the only way the query will

⁵Microsoft’s SQL Server is said to be particularly susceptible to this.

return any results is if there is a user named x with the password ‘ OR 1=1; --’.

If the database (or language library) does not have placeholder support, strong consideration should be given to changing to one that does. If that is impossible, any database library should have some kind of support for a database-specific `quote()` function. This will take the user input and do whatever necessary to escape special characters in the input so that they can be used directly in the query string.

Stored procedures offer similar protections to prepared statements, but are set up in the database itself ahead of time. It is somewhat less flexible than just tossing a query in where needed, but will also handle parameters in a safe method.

7 Authentication bypass

Authentication bypass comes in various flavors, but at the core it is a way to circumvent logging in while still being able to perform privileged actions. Unlike cracking a password—which still uses the standard authentication mechanism—authentication bypass, as the name implies, circumvents the authentication method completely. It abuses some aspect of the application to “reach around” the requirement to be logged in.

While not truly an authentication bypass, default passwords that remain unchanged have essentially the same effect. If default passwords are for some reason required—and finding a way to not require them would be a better choice—it should be difficult to get very far with the application without changing the default.

Applications should be structured in such a way that it is impossible to view or submit a page that is privileged without also supplying the proper credentials. One common mistake is for an application to check the URL against a list of privileged URLs, requiring authenticated users for any that are on the list. This kind of testing can fall prey to *aliasing*.

Most web servers will allow multiple URLs to reach the same page, but those URLs can look quite different. A trivial example is `http://vulnsite/foo//bar` which is equivalent—in web server terms—to `http://vulnsite/foo/bar`, but is very different when

matching the URL. By adding the extra slash, an attacker gets around the authentication requirement. Similar things can be done using HTML URL encoding (using %2F instead of /, for example).

Another common mistake is to assume that any links or forms that are only presented to the user after they have logged in are somehow protected. While in the normal course of events, the user has no access to those elements *through the application*, there is nothing stopping an attacker from using them. Some applications will use a separate program to process FORM submissions—without checking authentication—believing that because those FORM URLs are only presented post-login, they cannot be accessed otherwise. Both of these are a kind of “security through obscurity” that provides no protection at all.

It is imperative that the code for each page that requires authentication check for it before displaying or taking any action. If a separate program is used to process FORMs, it must also check authentication. No matter what kind of aliasing might be happening, the page code must be invoked, so that is the proper place for checking credentials.

8 Session hijacking

As described above, *sessions* are a standard way for adding state to HTTP. A session is assigned a particular ID that is stored in a cookie. Each HTTP request from the client is accompanied by the session ID, allowing the application to track a related series of requests. For applications that require authentication, the session stores the status of that authentication. This means that a valid session ID can be presented to the application by an attacker to hijack it.

This hijacking works only as long as the session is valid. Short-lived sessions—on the order of minutes—reduce the window of vulnerability, but can be annoying to users because they have to reauthenticate whenever the session times out. For sensitive web applications, it is worth the user annoyance.

An attacker can gain access to the values of a victim’s cookies in a number of ways. If the application runs on an unencrypted connection, cookie values can be sniffed on the wire. XSS provides another means to get access to cookie values. Some web applications do not even

use cookies—instead sending the session ID in the URL or a hidden FORM element—making it even easier to access them.

Some applications store IP address information in the session which is verified on each subsequent request. This technique is not particularly useful, as there are often lots of computers sharing a single IP address (e.g. NAT); also, some ISPs effectively assign a new IP on each request which would require the user to re-authenticate for each page accessed.

Re-authentication is an important safeguard for extremely sensitive operations. It can be annoying, but does protect against leaked session IDs.

9 Conclusion

The vulnerabilities presented are the most common flaws that are found in web applications. There are others, of course, but these should be at the top of any web application designer’s list. Properly handling user input while ensuring that authentication is correctly implemented will go a long way towards securing these applications.

Modern web frameworks—like Ruby on Rails, Django, and others—often provide mechanisms to eliminate or seriously lessen these vulnerabilities. Quite a bit of thought has gone into the security model of these frameworks and there is typically an active security group maintaining them. For new web applications, it is worth looking into them so as to benefit from those protections.

Around the Linux File System World in 45 minutes

Steve French

IBM

Samba Team

sfrench@us.ibm.com

Abstract

What makes the Linux file system interface unique? What has improved over the past year in this important part of the kernel? Why are there more than 50 Linux File Systems? Why might you choose ext4 or XFS, NFS or CIFS, or OCFS2 or GFS2? The differences are not always obvious. This paper will describe the new features in the Linux VFS, how various Linux file systems differ in their use, and compare some of the key Linux file systems.

File systems are one of the largest and most active parts of the Linux kernel, but some key sections of the file system interface are little understood, and with more than 50 Linux file systems the differences between them can be confusing even to developers.

1 Introduction: What is a File System?

Linux has a rich file system interface, and a surprising number of file system implementations. The Linux file system layer is one of the most interesting parts of the kernel and one of the most actively analyzed. So what is a file system? A file system “is a set of abstract data types that are implemented for the storage, hierarchical organization, manipulation, navigation, access, and retrieval of data.” [4]

But a “file system” can also mean a piece of code, i.e., a Linux kernel module used to access files and directories. A file system provides access to this data for applications and system programs through consistent, standard interfaces exported by the VFS. It enables access to data that may be stored persistently on local media or on remote network servers/devices, or that may be transient (such as debug data or kernel status) stored temporarily in RAM or special devices.

The virtual file system interface and file systems together represent one of the larger (over 500 thousand

lines of code), most active (averaging 10 changesets a day!), and most important kernel subsystems.

2 The Linux Virtual File System Layer

The heart of the Linux file system, and what makes Linux file systems unique, is the virtual file system layer, or VFS, which they connect to.

2.1 Virtual File System Structures and Relationships

The relationships between Linux file system components is described in various papers [3] and is important to understand when carefully comparing file system implementations.

2.2 Comparison with other Operating Systems

The Linux VFS is not the only common file system interface. Solaris, BSD, AIX, and other Unixes have similar interfaces, but Linux’s has matured rapidly over the 2.6 development cycle. Windows, originally with an IFS model similar to that of OS2, has evolved its file system interface differently than Linux, and has a very rich set of file system features, but at the cost of additional complexity. A reasonably functional file system can be much smaller in Linux than in most other operating systems (look at `shmemfs` for example), including Windows.

2.3 What has changed in the VFS

During the past year, no new file systems were added, although one (`smbfs`) was deprecated. In the previous year, three new file systems were added: `ecryptfs` (allowing per-file encryption), `gfs2` (a new clustered

file system), and `ext4` (an improved, more scalable version of `ext3`). The file system area did improve dramatically though during the past year. From 2.6.21.1 to 2.6.25, the size of the VFS code grew almost 7% (from 38 KLOC to 41 KLOC). The total size of the file systems and VFS together (the `fs` directory and all subdirectories) grew about 6% (from 487 KLOC to 518 KLOC). 3612 changesets from almost 400 developers were added during the year (about 8.4% of the total kernel changesets), and resulting in adding or changing over 200,000 lines of kernel file system code over the year. There has been huge progress.

Interestingly, despite thousands of code changes, the VFS interface, the API needed to implement a file system, changed only in minor ways, although file system drivers from the 2.6.21 source tree would require minor changes to compile on 2.6.25. The `exportfs` operations (needed to export a file system over the network via NFS) reworked its required methods (2.6.24). The `debugfs` and `sysfs` file systems also changed their external interfaces (2.6.24). The `debugfs` changes make it easier to export files containing hexadecimal numbers. Write-begin and Write-end methods were added to the VFS to remove some deadlock scenarios (2.6.24). New security operations were added to control mount and umount operations (2.6.25). SMBFS was deprecated (CIFS replaces it), but this did not affect the VFS interface. The kernel (2.6.22) now supports setting (not just getting) nanosecond inode timestamps via the new `utimensat` (2) system call. This call is an extension to `futimesat` (2) which provides the following:

- nanosecond resolution for the timestamps.
- selectively ignoring the `atime` or `mtime` values.
- selectively using the current time for either `atime` or `mtime`.
- changing the `atime` or `mtime` of a symlink itself along the lines of the BSD `lutimes` (3) functions.

A similar API call is being added to POSIX.

In the previous year, splice support was added. Splice is a mechanism to receive file data directly from a socket, and can dramatically improve performance of network

applications like Samba server when they are reading file data directly from a socket. The name of a common structure element in the dentry changed as it moved into to the new `f_path` structure (2.6.20). The `readv` and `writev` methods were modified slightly and renamed to `aio_readv` and `aio_writev` (2.6.19). The `inode` structure shrunk (2.6.19), which should help memory utilization in some scenarios. There were changes to the `vfsmount` structure and `get_sb` (2.6.18). A new `inotify` kernel API (2.6.18) was added to fix some functional holes with the `DNOTIFY` ioctl. The `statfs` prototype was changed (2.6.18). Support for `MNT_SHRINKABLE` was added (2.6.17) to make implementation of global namespaces (such as NFS version 4 and CIFS DFS) possible. Shrinkable mounts are implicit mounts, and are cleaned up automatically when the parent is unmounted.

3 File Systems from a to xfs

There are many Linux file systems—each for a different purpose. Within the `fs` directory of the Linux kernel are 60 subdirectories, all but five contain distinct file system drivers: from `adfs` (which supports the Acorn Disk Filing System used on certain ARM devices) to `XFS` (a well regarded high performance journaling file system). Linux also can support out of kernel file systems through `FUSE` (the `Filesystems in User Space` driver).

4 File Systems

4.1 Types of File Systems

Conventionally we divide file systems into four types: local file systems, cluster file systems, network file systems, and special-purpose file systems. Local file systems are used to store data on a local desktop or server system, typically using the disk subsystem rather than network subsystem to write data to the local disk. Local file systems can implement POSIX file-API semantics more easily than network file systems, which have more exotic failure scenarios to deal with, and are limited by network file system protocol standards. Cluster file systems aim to achieve near-POSIX file-API semantics, while writing data to one or more storage nodes that are typically nearby, often in the same server room. Cluster file systems are sometimes needed for higher performance and capacity. In such file systems, which

often use SANs or network attached block storage, more disks can be connected to the system, and more actively used at one time, than could be achieved with a local file system running on a single system.

5 Local File Systems

5.1 EXT4

In 2.6.23 kernel, `ext4` added various scalability improvements including `fallocate()` support, increasing the number of uninitialized extents, and removing the limit on number of subdirectories. In addition, support for nanosecond inode timestamps was added (needed by Samba and some network services). The development of advanced snapshot and reliability features in ZFS have led to consideration of longer-term file system alternatives to `ext4`. One promising candidate is `btrfs` which was announced last year and is under development by Chris Mason at Oracle (although still experimental).

5.2 EXT2 and EXT3

With the obvious need to improve the scalability of the default local file system (`ext3` or `ext2` on many distributions), attention has focused on the follow-on to `ext3`, `ext4`. Despite this, there were 88 changesets added which affected `ext3` over the past year, from over 50 developers, changing over 1000 lines. This is a surprisingly high number of changes for a file system in “maintenance” mode.

5.3 JFS

IBM’s JFS, which is in maintenance mode, had 45 changesets throughout the year (mostly global changes to structures, and minor code cleanup) but few new features.

5.4 XFS

In 2.6.23, XFS added a “concurrent multi-file data streams” feature to improve video performance and support for “lazy superblock counters” to improve performance of simultaneous transactions.

5.5 UDF

The “Universal Disk Format” is very important due to the explosion in numbers of writable optical drives. UDF added support for large files (larger than 1GB) in 2.6.22.

6 Network File Systems

6.1 NFS version 3

NFS version 3 defines 21 network file system operations (four more than NFS version 2) roughly corresponding to common VFS (Virtual File System) entry points that Unix-like operating systems require. NFS versions 2 and 3 were intended to be idempotent (stateless), and thus had difficulty preserving POSIX semantics. With the addition of a stateful lock daemon, an NFS version 3 client could achieve better application compatibility, but still can behave differently than a local file system.

6.2 NFS version 4

NFS version 4, borrowing ideas from other protocols including CIFS, added support for an open and close operation, became stateful, added support for a rich ACL model similar to NTFS/CIFS ACLs, and added support for safe caching and a wide variety of extended attributes (additional file metadata). It is possible for an NFS version 4 implementation to achieve better application compatibility than before without necessarily sacrificing performance. This has been an exciting year for NFS development with a draft of a new NFS RFC point release (NFS version 4.1) under active development with strong participation of the Linux community (including CITI and various commercial NFS vendors). The NFS version 4.1 specification is already 593 pages long. NFS version 4.1 includes various new features that will be challenging for the Linux file system to support fully, including directory oplocks (directory entry information caching), NFS over RDMA, and pNFS. pNFS allows improved performance by letting a server dispatch read and write requests for a file across a set of servers using either block- or object-based mechanisms (or even using the NFSv4 read and write mechanism). Some NFS version 4.1 features likely will be merged into the kernel by early next year, but their complexity has been challenging to the community.

6.3 NFS improvements

Over the past year, NFS has improved significantly, and has had more changes than any other file system. The SunRPC layer (which NFS uses for sending data over the network) now supports IPv6, although some smaller supporting patches are still being evaluated. Due to scarcity of IPv4 addresses in some countries, IPv6 support is becoming more important, especially as some government agencies are beginning to require IPv6 support in their infrastructure. NFS over RDMA, which provides performance advantages for workloads with large writes, is partially integrated into mainline (some of the server portions are not upstream yet). Server-side security negotiation is upstream. A new string-based mount options interface to the kernel has been added, which allows new options to be implemented in kernel without necessarily requiring `nfs-utils` (user-space tools) update, and eases long-term NFS packaging. Forced unmount support, which had been removed from NFS a few years earlier, was readded (2.6.23). This allows “`umount -f`” to better handle unresponsive servers. Also added to NFS in kernel version 2.6.23 was support for “`nosharecache`” which allows two mounts, with different mount options, from the same client to the same server export. When this mount option is not specified, the second mount gets the same superblock, and hence the same mount options as the first. With this new mount option, the user may specify different mount options on a second mount to the same export.

6.4 CIFS

IPv6 support was added (2.6.22). Additional POSIX extensions were added (2.6.22) to improve POSIX application compatibility on mounts to Samba servers. The most exciting changes to CIFS over the past year, though, have been the addition of Kerberos support and the addition of DFS support. MS-DFS is a mechanism for traversing and managing a global name space for files and is commonly used in larger networks. The Samba server already supported DFS, but the Linux kernel did not, until this year. Released earlier this spring, large amounts of interoperability documentation by Microsoft may allow us to improve our support more quickly, not just for newer servers, but also for older servers. Older dialects of SMB, sometimes more than 15 years old, are still in use in some places. This documentation is also making development of an in-kernel SMB2

client implementation easier. Currently `smb2` support is being prototyped as a distinct module from `cifs`, to make it easier to make rapid changes, and because SMB2 is turning out to be much different than CIFS. Although sharing some information levels with SMB and CIFS, SMB2 has a much simplified set of commands that are largely handle- (rather than path-) based, and are even more efficient to parse, which should allow improved performance in the long run. SMB2 also allows improved asynchronous operation and request dispatching, while also adding better reconnection support via a new durable handle feature of the protocol. The prototype SMB2 client should be available (in experimental form) before the end of the year. Although the server for CIFS, Samba, is not in-kernel, it should be mentioned that with the recent release of the enormous amount of network interoperability documentation by Microsoft, the Samba team already has made great progress with the Samba 4 SMB2 server, already passing most functional tests.

6.5 AFS

There are multiple versions of AFS, the OpenAFS implementation which is more complete in function, but not in mainline kernel, and what a year ago was only a minimal implementation in-kernel. The AFS version in the mainline kernel has improved dramatically over the past year. In 2.6.22, support for basic file write was added, and support for directory updates (`mkdir`, `rename`, `unlink`, etc.) and `krb4` ticket support was added via RPC over AF_RXRPC sockets.

7 Cluster File Systems

There are two cluster file systems in the mainline kernel, GFS2 from Red Hat/Sistina and OCFS2 from Oracle. There are also two popular cluster file systems that are not in the mainline, but that are mentioned because they are often used in high-end compute clusters: Lustre (now owned by Sun) and IBM’s GPFS. GFS2 supports more file system entry points than OCFS2 (which has worse locking and ACL support), but OCFS2 does support sufficient features to address the needs of some clustered databases.

7.1 OCFS2

In 2.6.22 OCFS2 added support for sparse files. Over the year the OCFS2 team and other kernel developers

added 247 changesets to OCFS2, more than 16,000 lines of code.

7.2 GFS2

This area had a busy year, with 243 changesets added: many small stability patches and bug fixes, but multiple performance enhancements were added as well.

8 Future Work

Going forward there are many challenges to address in the Linux File System area, but among the most interesting are the following: clustering improvements, new hierarchical storage management features, improved error handling and detection in the local file system, support for new network file systems (SMB2), and improved network file systems (NFS version 4.1).

8.1 Clustering

With the need for reliable server failover, and the need for dynamic reconfiguration of complex networks, clustering is becoming more important, but there is no clear winner among the cluster file systems, and two of the most popular choices still remain out-of-kernel. In addition, the NFS version 4.1 protocol adds the ability to support parallel NFS, to better distribute load across a set of NFS servers without requiring cluster file system software to be installed on all clients. Similar features are being investigated for CIFS. Samba server support for clustering is greatly improved through the work of Tridge, and others on the SOFS team, on `ctdb`. `Ctdb` has proven to be a very useful library for high performance cluster state management and recovery. NFS server support for running over a clustered file system also has improved in the past year through work by CITI and by IBM and others.

8.2 New Local File Systems

Among the biggest long term challenges in the file system area remains error handling and recovery. As disks get ever larger and yet error rates stay constant, error detection and recovery is reaching a critical point. One of the design goals of `btrfs` is to address this problem, as well as to improve HSM features. Many new systems

now contain a mix of storage which includes disk and solid state. Since these devices perform very differently, local file systems must add features to optimize performance on hybrid systems which contain both. Whether in the long term we will need two local file systems, `ext4` and `btrfs`, due to performance differences on particular popular workloads, or whether one Linux file system will win and be used for most workloads remains to be seen. `EXT4` scalability is improved but would require substantial changes to handle ever increasing disk errors on ever larger disks as well as `btrfs` already does. In some operating system, the file system and volume manager layers of the operating system are more tightly coupled than in Linux. This eases the addition of better support for dynamic reconfiguration of disk subsystems when failing disks are added or removed on the fly. The development of `btrfs` may open up discussion of changes to the volume management layer as well as changes to Linux to support DMAPI, the standard for disk management used by many storage management applications. Adding support for part or all of DAPI to the VFS and `btrfs` or `ext4` would allow for improved backup and disk management. Currently a subset of DAPI is supported but on XFS only.

8.3 New Network File Systems

An SMB2 file system prototype, written by the author, is being coded and tested currently. Since SMB2 is significantly different than SMB and CIFS protocols in the way it handles path names, and in the way it handles file handles, less code could be shared between it and the existing `cifs` module, so it is being written as a distinct module. This also allows it to be updated quickly without impacting the existing `cifs` module. SMB2 will become more important in the coming year since it is the default network file system for current Microsoft servers and clients, and matches reasonably well to Linux. NFS version 4.1 also includes new features which will need to be explored in the Linux VFS, including how to support directory oplocks (directory delegations).

9 Legal Statement

This work represents the view of the author and does not necessarily represent the view of IBM. IBM and GPFS are registered trademarks of International Business Machines Corporation in the United States and/or other countries. Microsoft,

Windows, and Windows Vista are either a registered trademark or trademark of Microsoft Corporation in the United States and/or other countries. UNIX is a registered trademark of The Open Group in the United States and other countries. POSIX is a registered trademark of The IEEE in the United States and other countries. Linux is a registered trademark of Linus Torvalds. Other company, product, and service names may be trademarks or service marks of others.

References

- [1] O. Kirch. Why NFS Sucks. *Proceedings of the 2006 Ottawa Linux Symposium*, Ottawa, Canada, July, 2006. <http://ols.108.redhat.com/reprints/kirch-reprint.pdf>
- [2] Linux CIFS Client and Documentation. <http://linux-cifs.samba.org>
- [3] S. French. Linux File Systems in 45 minutes: A Step by Step Introduction to Writing or Understanding a Linux File System. <http://pserver.samba.org/samba/ftp/cifs-cvs/ols2007-fs-tutorial-smf.pdf>
- [4] File System. Wikipedia, The Free Encyclopedia, Retrieved 28 April 2008. <http://en.wikipedia.org/wiki/Filesystem>
- [5] Linux Weekly News, API changes in the 2.6 kernel series <http://lwn.net/Articles/2.6-kernel-api/>
- [6] Kernel Newbies. Linux Changes report. <http://kernelnewbies.org/LinuxChanges>

Peace, Love, and Rockets!

Bdale Garbee
HP Open Source & Linux Chief Technologist
bdale@gag.com

Abstract

My son and I enjoy building and flying model rockets. But when we went looking for an electronic altimeter to measure how high our flights were going, the products we found provided limited features, and required the use of proprietary software for configuration and to extract the data recorded... and that's no fun!

This paper gives a brief overview of the resulting work to develop open hardware and associated open source software to satisfy our altitude curiosity, and provides pointers to sources of more information. The live conference presentation will include a more detailed report on our progress and plans for more sophisticated payloads for our higher-powered rocket projects, punctuated with photos and video clips.

This material should be interesting to anyone curious about open small embedded systems. The hardware is ARM-based, licensed under the TAPR Open Hardware License, and implemented entirely using open source design tools. The software is built on FreeRTOS using GNU tools and a variety of open source libraries.

1 The Basic Idea

Model rocketry is a popular hobby in which rocket-shaped models are built, launched, and recovered by a variety of means to be flown again. One of the first questions people ask about model rockets is: "How high did it go?" This question can be answered by visual observations and some fairly simple math, but modern electronics and miniature sensors also make it possible (and more fun!) to measure altitude and other flight parameters directly.

A number of commercial model rocket avionics systems exist, are reasonably priced, and work well. The problem is that they really aren't "hackable" to add new features, or try out different approaches. In some cases,

even the serial protocol used to speak to the units from a PC is explicitly proprietary and only useful with provided software for Windows or Mac systems. And that's just no fun!

2 Role of Avionics

Beyond the simple "how high did it go?" question, there are a number of other dynamic parameters that can be interesting to measure in flight. This is particularly true at the more advanced end of the hobby, where flights may exceed the speed of sound, or where experimental propellants, motor casings, and nozzle designs may be under evaluation.

Another significant role of avionics in many model rockets is to control the recovery system, firing ejection charges to deploy parachutes or streamers. If the rocket is moving too fast when the recovery system deploys, it can cause damage to the vehicle or recovery system due to the sudden changes of velocity and resulting energy transfers at ejection. The objective is therefore usually to cause ejection to happen as close to the flight apogee as possible, because that's where the rocket is moving at minimum speed. Simple models accomplish this by flying with a motor that includes a delay element that burns through in a predictable time before firing an ejection charge. But with active electronics, apogee can be directly sensed, eliminating variations due to weather conditions, exact takeoff weights due to payload changes, etc.

For very high flights, particularly on windy days, an additional feature that active on-board electronics can enable is "dual deployment" in which a small drogue chute or streamer is ejected at apogee, followed by deployment of the main recovery parachute at a pre-determined altitude much closer to the ground. This allows the rocket to return towards earth in a controlled but rapid descent to minimize how far down-range it drifts, yet

touch down at a safe speed once the main parachute deploys.

Even more sophisticated systems include a radio down-link to the ground for live updates, video from on-board cameras, or even live position information if the rocket is equipped with a GPS receiver. A radio uplink could even be used to command events on board from the ground.

3 What We Built

The Altus Metrum project intends to deliver a completely open (hardware and software) recording altimeter for model rockets. Sized to fit airframes as small as 24mm in diameter, with flexible battery choices, the design bases most operations on a barometric pressure sensor, but also includes a three-axis accelerometer and temperature sensor. Enough non-volatile memory is included to support data logging through the entire flight, and a USB interface allows easy programming, data recovery, and operational power when not in flight. Other features include two serial ports to support an on-board GPS receiver and RF downlink, and support for firing two ejection charges using “electric match” low-current igniters to support dual-deploy or staging activities.

The hardware design is based around the single-chip LPC-2148 microcontroller from NXP, which is an ARM7TDMI-S core with 512k of flash memory, 32k of RAM, USB, and lots of analog, digital, and serial I/O on-board. Non-volatile storage of flight data is provided in a Microchip 24FC1025 CMOS serial EEPROM, which is 128k by 8 bits with an I2C interface. The sensor complement in the initial prototype includes the Freescale MP3H6115A pressure sensor, Freescale MMA7260QT 3-axis accelerometer, and Microchip MCP9700A linear temperature sensor. A Honeywell 2-axis magnetic sensor was evaluated but not included because of the large circuit board area required for supporting circuitry.

The hardware was designed entirely using open source tools, including `gschem` and `pcb` from the gEDA suite, the features of `digikey.com` for parts selection and data sheet access, `gerbv` and the service of `freedfm.com` for circuit board verification, and the services of `barebonespcb.com` for quick and cheap circuit board fabrication. An Olimex ARM-USB-OCD JTAG interface is used with `gdb` via `openocd` for hardware testing and firmware development and debugging.

The firmware is written mostly in C with some ARM assembler, runs from the on-chip flash using the on-chip RAM, and stores flight data to the serial EEPROM. USB serial emulation provides a console interface for interaction with the software during ground testing and to retrieve data after flight. Software development uses GCC, newlib, FreeRTOS, and the LPCUSB packages, and is derived from a FreeRTOS demo package written for the Olimex LPC-P2148 evaluation board by J.C. Wren.

The hardware design carries the TAPR Open Hardware License (OHL), which was created to be “GPL-like” for hardware designs. The software is licensed GPL “v2 or later.”

4 Current Status

First article prototypes are completely assembled and mostly tested. Enough problems were found and fixed that more v0.1 boards are unlikely to be assembled, a revision of the circuit board design is called for instead. The weather in Colorado has been mostly unsuitable for flight testing since the hardware was developed, and the flight software is not quite finished, so there have been no flight tests yet as of the time of talk submission.

By the time of the conference, we hope to have completed testing and evaluation of the initial hardware including some amount of flight testing. The design will then be updated and a new circuit board revision released, with work to integrate a GPS receiver core and RF downlink continuing in parallel.

5 For Further Information

This project can be found at <http://altusmetrum.org>, with more information on our hobby rocketry activities appearing at <http://gag.com/rockets>.

Information about the TAPR Open Hardware License may be found at <http://tapr.org/ohl>.

Secondary Arches, enabling Fedora to run everywhere

Dennis Gilmore
Fedora Project OLPC
dgilmore@fedoraproject.org

Abstract

There are many different types of CPU out there. From SPARC to Alpha, Arm to s390. Most people run i386 or x86_64; how can you get a distro for your favorite exotic preference? Fedora has been doing a lot of work to allow interested people to build and support the architecture of their choice. Work has been going on for arm, alpha, ia64, s390, and SPARC.

This paper will cover what has been done to enable motivated people to bring up and support new architectures. The same tools used to make Fedora can be used to layer products on top of Fedora also. If you're interested in MIPS, PA-RISC, or just want to know what is involved in putting together a distro, this paper is also for you.

1 What is a secondary architecture?

Any architecture that is not mainstream? Fringe architectures? Anything not x86 or x86_64?

Fedora is starting by saying that it is anything that is not x86 or x86_64 based, however ppc is currently a special case. We are building ppc and ppc64 with the primary architectures, however the release engineering for it is being done by the Fedora community. This is also not to say that the way things are done now is the way they will always be. For instance if one architecture started building up market share then it would be evaluated for primary architecture inclusion.

The secondary architecture team is responsible for providing all of the building and hosting resources. The team also gets full access to Fedora cvs. The same source cvs checkout will be used to build across all architectures.

Fedora package maintainers are encouraged to look at and fix build failures on secondary architectures. However, they are not required to do so. We understand

that there are people who have no interest in supporting secondary architectures. The architecture teams are there to ensure things get fixed, and to provide architecture specific knowledge to interested maintainers.

We are actively looking for sponsorship of a master mirror for secondary architectures to reduce the barrier of entry for all. While the barrier of entry is high for a new Secondary Architecture it will ensure that the people proposing the architecture are committed to ensuring that it is a success.

The initial proposal was written by Tom Callaway with the following purpose. As an open community, Fedora encourages motivated individuals who care about architectures which are currently unsupported [1]. With the ultimate goal of supporting as many architectures as people want to support.

2 Why support secondary architectures?

The more architectures that you build the same code on, the more portable the code base. Lots of people make a claim that their code is portable because they write to standards, such as POSIX. Many times they really have no idea if their code is truly portable.

What is portable code? The same code running in different places. We are testing portability by building on big endian CPUs, little endian CPUs, CPUs with different instruction sets. We could also test portability by building on different OS's such as FreeBSD, Darwin, or Hurd on the same CPU.

We also scratch people's itches. If you want to know more about Linux and how a distribution is put together, then working on a secondary architecture is a great place to start.

3 Who is doing secondary architectures?

- HP, Intel, SGI, and Red Hat for Itanium

- Red Hat and IBM for S390
- Marvell for ARM
- Fedora Community for SPARC
- Fedora Community for ALPHA

Lots of different parts of the Fedora community are working on actively ensuring the success of Secondary Architectures. Secondary architectures are not a new idea. Aurora SPARC Linux, Alpha Core, and RHEL have been doing it for years. This is a new process that simplifies and speeds up getting other architectures built.

4 How do Secondary Architectures work?

All packages will be built on the primary hub first. Once a package has been successfully built it will be queued on the secondary hub. There is a many-to-one relationship between primary and secondary hubs.

Initially all automated communication is one way. A long term goal is to push builds back up to the primary architecture hub. `kojisd` is a new tool that has been written to enable automatic builds on secondary hubs. `kojisd` ensures that the secondary hubs have the same or newer build of everything used in the buildroot as what was used on the primary hub. Architecture teams are responsible for ensuring that bugs get fixed, and doing the release engineering.

5 Architecture resources needed (provided by architecture team)

- 1TB disk space per architecture

This will be an ever growing amount of disk. `koji` has a garbage collection mechanism to clean up old builds, however Fedora builds a huge number of packages, the number of builds will continue to grow as new packages are added. For Fedora-9 there were ~20000 builds.

- web server

Used for the `koji` frontend and hub, running apache with mod_python, the hub uses xmlrpc for all communication. Fedora provides a SSL cert to be used for user authentication. This allows users to use the same credentials and work on any architecture.

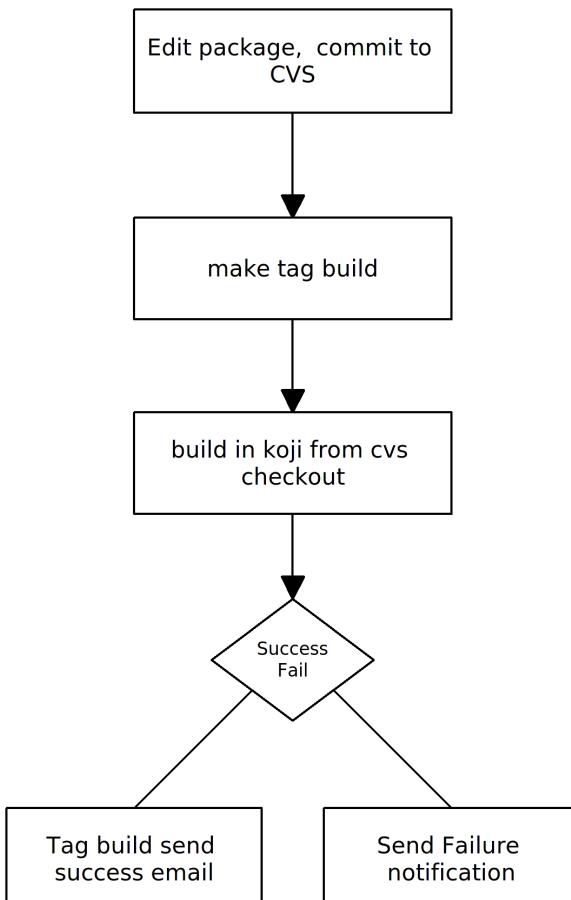


Figure 1: previous workflow

- Database server

Postgresql server, depending on the hardware, this can run on the web server. Ideally this is a dedicated server. Some tables like the rpmfiles table contain over 90,000,000 rows in the Fedora installation. Fedora is able to provide scripts to enable backups of the database. Postgres 8.3 is recommended due to it having working autovacuum support.

- builders

At least one but preferably more, they need to have read only access to the file system that koji stores its packages on, this can be via http, nfs, iscsi, etc. In order to run createrepo tasks, at least one builder needs the local access to the koji file system.

- bandwidth

Pushing up images, rawhide, and updates takes up bandwidth, uploading 30 to 60 gigabytes for a release at 1 mbit/s takes a long time.

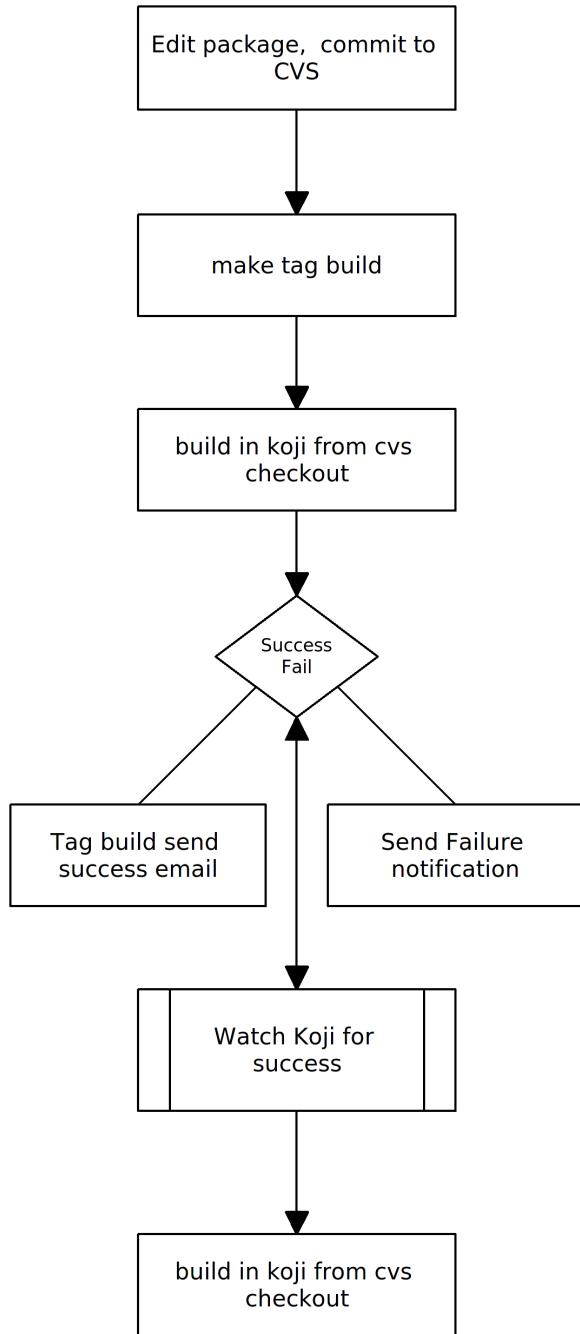


Figure 2: new workflow workflow

- mirrors

Right now we are unable to provide mirrors. We are working on enabling hosting and mirroring of release trees. Secondary architectures will always be responsible for hosting their own infrastructure.

- Sand Box Machines (optional but recommended)
- Gives a workspace for Fedora maintainers to do test builds and debug failures using mock. This

will be an environment that will help people who want to help but don't have access to hardware.

6 Fedora Provided Resources

- Package CVS

Every package built for all architectures must come from Fedora cvs. This will ensure that we can be in compliance with the GPL, everyone will also know where to go.

- User authentication

All users on secondary architecture hubs are identified by using the same certificates, there is seamless SSL authentication across all secondary architecture hubs.

- Packagers

Fedora has 596 packagers maintaining over 6000 source packages. Many, but not all of these people, will help fix bugs on secondary arches for the packages they maintain.

- Framework

There is an existing framework and mechanism to support new contributors and contributions. Package management, workflow management, and user support are all taken care of, greatly lowering the barrier to entry of starting your own distribution project. Fedora has resources for hosting upstream projects called fedorahosted. As well as VoIP and mailing lists for communications. Fedora uses freenode for irc communications.

- Best practice guidelines

Fedora has a set of packaging guidelines <http://fedoraproject.org/wiki/Packaging/Guidelines> which are constantly evolving and growing as new types of software are added to package repositories. Fedora is at the forefront of Linux development. Many things like selinux, xen, gcj, OpenJDK and NetworkManager shipped first in Fedora. Fedora is also heavily involved in upstream development of many technologies. New gnome releases regularly ship first in Fedora.

- Release engineering tools and support

Fedora's insistence that everything be open means that not a single proprietary tool is used to produce the distribution. In the past the hardest part

in doing your own release was getting tools to do composes. With the merge of Core and Extras Red Hat's previous tools were dropped and new open source tools were created. Fedora Release Engineering is entirely done in the open, and release engineering team members are available to help with issues. As things move forward we will have new problems to tackle, being open we will be able to resolve them all.

7 Existing Fedora Tools

- **koji**

<http://koji.fedorahosted.org>

Terminology

In Koji, it is sometimes necessary to distinguish between the package in general, a specific build of a package, and the various rpm files created by a build. When precision is needed, these terms should be interpreted as follows:

Package:

- The name of a source rpm. This refers to the package in general and not any particular build or subpackage. For example: kernel, glibc, etc.

Build:

- A particular build of a package. This refers to the entire build: all arches and subpackages. For example: kernel-2.6.9-34.EL, glibc-2.3.4-2.19.

RPM:

- A particular rpm. A specific arch and subpackage of a build. For example: kernel-2.6.9-34.EL.x86_64, kernel-devel-2.6.9-34.EL.s390, glibc-2.3.4-2.19.i686, glibc-common-2.3.4-2.19.i64.

Koji Components

Koji is comprised of several components:

- koji-hub is the center of all Koji operations. It is an XML-RPC server running under mod_python in Apache. koji-hub is passive in that it only receives XML-RPC calls and relies upon the build daemons and other components to initiate communication. koji-hub

is the only component that has direct access to the database and is one of the two components that have write access to the file system.

- kojid is the build daemon that runs on each of the build machines. Its primary responsibility is polling for incoming build requests and handling them accordingly. Koji also has support for tasks other than building. Creating install images is one example. kojid is responsible for handling these tasks as well. kojid uses mock for building. It also creates a fresh buildroot for every build. kojid is written in Python and communicates with koji-hub via XML-RPC.
- koji-web is a set of scripts that run in mod_python and use the Cheetah templating engine to provide a web interface to Koji. koji-web exposes a lot of information and also provides a means for certain operations, such as cancelling builds.
- koji is a CLI written in Python that provides many hooks into Koji. It allows the user to query much of the data as well as perform actions such as build initiation.
- kojira is a daemon that keeps the build root repodata updated.

Koji organizes packages using tags. In Koji a tag is roughly analogous to a beehive collection instance, but differ in a number of ways:

- Tags are tracked in the database but not on disk.
- Tags support multiple inheritance.
- Each tag has its own list of valid packages (inheritable.)
- Package ownership can be set per-tag (inheritable.)
- Tag inheritance is more configurable.
- When you build you specify a target rather than a tag.

A build target specifies where a package should be built and how it should be tagged afterwards. This allows target names to remain fixed as tags change through releases [2].

- **mash**

<http://mash.fedorahosted.org>

mash is a tool that creates repositories from koji tags, and solves them for multilib dependencies.

- **pungi**

<http://pungi.fedorahosted.org>

The pungi project is two things. First and foremost it is a free opensource tool to spin Fedora installation trees and isos. It will be used to produce Fedora releases from Fedora 7 on, until it is replaced by something better. Secondly, pungi is a set of python libraries to build various compose-like tools on top of. Pungi provides a library with various functions to find, depsolve, and gather packages into a given location. It provides a second library with various functions to run various Anaconda tools on the gathered packages and create isos from the results.

- **livecd-tools**

livecd-tools lives in a git repo at <http://fedorahosted.org/>. In a nutshell, the livecd-creator program

- Sets up a file for the ext3 file system that will contain all the data comprising the live CD.
- Loopback mounts that file into the file system so there is an installation root.
- Bind mounts certain kernel file systems (/dev, /dev/pts, /proc, /sys, /selinux) inside the installation root.
- Uses a configuration file to define the requested packages and default configuration options. The format of this file is the same as is used for installing a system via kickstart.
- Installs, using yum, the requested packages into the installation using the given repositories.
- Optionally runs scripts as specified by the live CD configuration file.
- Relabels the entire installation root (for SELinux.)
- Creates a live CD specific initramfs that matches the installed kernel.
- Unmounts the kernel file systems mounted inside the installation root.
- Unmounts the installation root.
- Creates a squashfs file system containing only the ext3 file (compression.)

- Configures the boot loader.
- Creates an iso9660 bootable CD.

8 New Tools Needed

- **kojisd**

kojisd is a new tool written to monitor one hub and on success of tasks to repeat the task on a second hub. There are two things that we want to do. Maintain tags/targets and build packages using a buildroot which is as identical as possible. Due to the high possibility during rampup that packages will fail we are saying that if we have a newer build on the secondary hub then that's ok. It also simplifies buildroot management. We can follow the same principles that the primary hub does. The one thing we add is to ensure that for instance things depending on a specific version of xulrunner are built against that version since build timings will vary on different architectures.

Many options in kojisd are configurable. For instance you can choose to import packages based on architecture, and whitelist/blacklist targets to build for. For Secondary Architecture purposes we will be importing noarch packages. For someone layering products on top of Fedora they can import all builds.

9 Secondary architecture Teams

We currently have teams working on ARM, Alpha, Itanium, S390, and SPARC.

9.1 ARM

- Hub: Not yet available.
- Team Page: <http://fedoraproject.org/wiki/Architectures/Arm>

Arm is being lead by Lennert Buytenhek. The baseline ARM CPU architecture that we have chosen to support is ARMv5, Little Endian, Soft-Float, EABI. We believe that this provides a nice baseline and that the pre-built packages and root file system images will be usable on many of the modern ARM CPUs, including XScale, ARM926, and ARM-11, etc.

9.2 Alpha

- Hub: <http://alpha.koji.fedoraproject.org>
- Team Page: <http://fedoraproject.org/wiki/Architectures/Alpha>

The Alpha team is being lead by Oliver Falk and has Jay Estabrook as a member. Fedora Alpha started out as the AlphaCore Linux Project, when Red Hat stopped support for Alpha with Red Hat Linux 7.1. Fedora Alpha is the continuation of the AlphaCore efforts, in an official capacity as part of the Fedora Project.

9.3 Itanium

- Hub: <http://ia64.koji.fedoraproject.org>
- Team Page: <http://fedoraproject.org/wiki/Architectures/IA64>

The IA64 team is being lead by Doug Chapman, and has Tim Yamin, Prarit Bhargava, Yi Zhan, Jes Sorensen, and George Besher as members. The IA64 work is based on unofficial releases composed from builds as a side effect of Fedora's buildsystem before the merge. It was maintained in sorts after and is now becoming official.

9.4 S390

- Hub: <http://s390.koji.fedoraproject.org>
- Team Page: <http://fedoraproject.org/wiki/Architectures/s390x>

The S390 team is being lead by Brad Hinson and has Brock Organ as a member. The work on s390 is being based on RHEL5.

9.5 SPARC

- Hub: <http://sparc.koji.fedoraproject.org>
- Team Page: <http://fedoraproject.org/wiki/Architectures/SPARC>

The SPARC team is lead by Tom Callaway, and has Peter Jones, Patrick Laughton, and I as members. We have a wide range of SPARC hardware. Fedora, unlike Aurora SPARC Linux, will support only UltraSPARC and higher hardware due to lack of an upstream kernel maintainer for older SPARC. All user land is being built for sparcv9 and sparc64, with sparcv9 recommended for use.

10 How can the Secondary Architecture work help layering products on top of Fedora?

Using the tools written for secondary architectures, you are able to pull into your own koji setup builds from Fedora as they happen. Using a trigger mechanism you are able to build your own packages when new packages are built in Fedora.

References

- [1] Fedora Secondary Architecure proposal, <http://fedoraproject.org/wiki/TomCallaway/SecondaryArchitectures>.
- [2] Fedora wiki Describing koji. <http://fedoraproject.org/wiki/Koji>.

Application Testing under Realtime Linux

Luis Claudio R. Gonçalves
Red Hat Inc.
lgoncalv@redhat.com

Arnaldo Carvalho de Melo
Red Hat Inc.
acme@redhat.com

Abstract

In the process of validating the realtime kernel and validating third-party applications under this kernel, it was necessary to build a set of small tools to understand different behavior presented by applications and the kernel itself.

We have identified several practices and common mistakes that could be harmful to performance and determinism in a Linux RT environment. We used systemtap and other tools to identify these problems and in some cases, to fix them or test alternatives without touching the application code.

1 Introduction

This paper talks about experiments conducted on systems with PREEMPT_RT Realtime enabled kernels. The main features of the PREEMPT_RT patch were already described in papers [1] and the Project wiki page [2].

The main goal of PREEMPT_RT is to offer determinism, predictability to the highest priority tasks in the system. The project is moving in a fast pace, and most of its mature features have already been merged upstream.

In the search for determinism, several well established entities had been touched and changed. Examples are:

- sleeping spin_locks – in the PREEMPT_RT patch most spin_locks were converted to rt_mutexes and can sleep. This leads to better kernel preemption capabilities.
- threaded interrupts – Interrupt Service Routines are now threaded. This prevents, for instance, lower priority tasks doing heavy I/O activity from creating latencies in higher priority tasks.

- threaded softirqs – each softirq has its thread and so the system administrator has the ability to change their priorities in order to favor the ones more important for his application.
- priority inheritance – avoid priority inversion by boosting the priority of a lower priority thread to the priority of the thread waiting for the resource, if higher. This is possible, in part, because rt_mutexes have the concept of ownership.

Sleeping spin_locks and threaded IRQs are bound together, because it is necessary to have a process context in order to sleep. This way, interrupt processing had to be delegated to specialized threads. Sleeping spin_locks are also building blocks for priority inheritance. These features are the foundations on which PREEMPT_RT is built.

Realtime is all about determinism, predictable behavior. There are certain practices in application development that may hurt these premises and lead a system to present a Byzantine behavior.

Along with determinism comes extra flexibility, as entities like interrupt handlers can now be prioritized, allowing the user to tune the environment to better serve particular workloads. User applications can have priorities higher than any IRQ or kernel thread. Of course, application errors in such a high priority can lead to problems. A busy loop could starve disk IRQ or any other subsystem.

2 About the Tests

Most of the examples presented here are based in experience acquired while testing customer applications. Most of the tests happened under the “It runs slower on RT!” pressure. That alone was enough motivation to find the root cause of the observed behavior.

On one hand, the objective of Realtime is determinism and sometimes the cost of determinism is a negative impact on performance—other areas such as High Availability suffer from the same problem. On the other hand, the hit on performance should be as light as possible and Realtime, in fact, can improve performance in several scenarios.

That said, there are four main possibilities for the performance penalties observed:

- Problem in the kernel – a simple example is the case where a user application was aborting due to system load reaching the defined limit in the test, that was caused by a bug in the kernel system load calculation routine;
- Problem in the application – user application was multithreaded and not all the threads were running in the same priorities, or at least in reasonable priorities, causing the lowest priority thread to starve;
- Problem in both – race condition in user application that was more likely to be reached in Realtime, due to system architecture, triggered a kernel bug in a code path not usually exercised;
- Incorrect comparison – comparing results from running the test application in tuned environment versus running the test application in a system with out of the box settings.

2.1 Avoid using `sched_yield`

A good description on the problem of using `sched_yield` on Realtime was written [3]. After discussing the problem of “trying to help the scheduler,” the author notes:

One example can be found in recent changes to the iperf networking benchmark tool, that when the Linux kernel switched to the CFS scheduler, exhibited a 30% drop in performance. Source code inspection showed that it was using `sched_yield` in a loop to check for a condition to be met. After a fix was made, the performance drop disappeared and CPU consumption went down from 100% to saturate a gigabit network to just 9%. [4]

The use of `sched_yield` is not recommended in Realtime and that serves as a good example how systemtap can be used to identify usage of a certain system call or kernel function and account the usage:

```
#! stap

# pid, process_name and number of calls
# to sched_yield()
global process_list

probe syscall.sched_yield {
    p = pid()
    e = execname()

    if (process_list[p,e])
        process_list[p,e] += 1
    else
        process_list[p,e] = 1
}

probe end {
    foreach ([pid+, name] in process_list)
        printf ("%s[%d] called sched_yield %d times\n",
               name, pid, process_list[pid, name])
}
```

This systemtap script will run until `Ctrl+C` is pressed. Once interrupted, this systemtap script will inform you of the processes that have called `sched_yield()` and the number of times it was called by each process. For example:

```
[root@lab tmp]# stap sched_yield.stp
ping[2848] called sched_yield 14 times
```

2.2 Bad Priority Assignment

Due to PREEMPT_RT nature, interrupts are threaded and so are several other kernel subsystems. A user application running at the highest available priority, or a priority high enough to be above certain kernel subsystems, if not carefully crafted, can freeze the system. A good piece of advice is to *avoid running applications at the highest available priority*.

A simple example of this case would be the highest priority application running a busy loop, creating statistics that will be sent to disk before the loop ends. That was what probably happened with the `gtod_latency` test in this email thread [5], where disk IRQ thread was unable to run until `gtod_latency` finished its busy loop.

The opposite case, where the application under test runs at the lowest priority, or on a priority so low that any

other process could preempt it, is also a disaster if the application is expected to have a high throughput. Observing `/proc/<pid>/status` can provide valuable information, especially in the `nonvoluntary_ctxt_switches` field. This field keeps track of the number of times this process has been preempted by other tasks, so a high number in this field would result in high latencies in process because everytime a process gets preempted, its work is interrupted until it gets rescheduled to run again.

This is the easiest case to observe and fix. However, sometimes this problem can be hid when several threads have reasonable priorities except for one with an unacceptable priority. The common way of verifying this is just a matter of using `ps`:

```
[root ~]# ps -emo pid,tid,policy,pri,rtprio,cmd
...
3826    --   -  /usr/lib64/.../firefox-bin
- 3826 TS  19  -
- 3848 TS  19  -
- 3849 TS  19  -
- 3855 TS  19  -
- 3856 TS  19  -
- 3857 TS  19  -
- 3869 TS  19  -
- 8854 TS  19  -
...
...
```

What is the best priority to use? That depends on various factors such as what the application does, how it works, what the most important resource for application is , and the like. It should be noted here that the system can be fine-tuned to favor the IRQ or softirq that is most important for the application.

Another important thought to bear in mind is: what will be the scheduler policy of use? Will that be `SCHED_FIFO` or `SCHED_OTHER`? Changing scheduler policy and priorities can enhance performance or turn your application into a Denial-of-Service tool.

The tool used to run a given application with the desired priority and under the desired scheduler policy is `chrt`. This tool can also be used to modify priority and scheduler policy of running processes. It is also possible for the application to set these parameters by itself.

For some of the tests, running the test application at a higher priority and with a better scheduler policy solved the performance issue.

2.3 Resource Allocation

There is a saying in realtime stating that every needed resource should be allocated in advance. Dynamic allocations are prone to resource contention and other latencies which are the worst offenders in realtime.

Whenever an application requests memory allocation, the kernel tries to carry this task as fast as possible. Eventually, these memory pages may not be available and a certain level system reorganization will be required, delaying the requested memory delivery. Sometimes not all requested memory pages are ready to use and when accessed, if not ready, they may trigger a *minor page fault*. In this case, the kernel will need to perform a few steps to solve the situation. This scenario can get more complicated, and the requested memory allocation could trigger a *major page fault*, where the kernel may need to swap memory pages of another application in order to free memory pages for the requesting application. These latencies can be big enough to hurt the realtime constraints of the application.

This case can be even worse because it involves I/O accesses and depending on the system configuration, it may become a nightmare. Imagine a system using a network file system where in order to get the requested memory pages, the system has to flush some buffers through the network. In the context of realtime applications that may lead to unbounded latency spikes and the end of determinism.

Another interesting point is that when a process runs into a page fault it will be frozen, with all its threads, until the kernel handles the page fault.

There are several ways to address this case [6], with the use of `mlock()` and related functions being the common solution. Unfortunately, the real solutions here require changes in the source code.

Information about the amount of minor and major page faults that already happened to a given application can be obtained using:

- `/proc/<pid>/stat` – this file has a huge amount of data about the process indicated by `pid`. There are four fields presenting the number of minor and major page faults faced by this process and the number of page faults faced by its child

```
# if you have problems hooking on exit_mm, use profile_task_exit instead.

probe kernel.function("exit_mm") {
    printf("%s(%d:%d) stats:\n", execname(), pid(), tid())
    printf("\tPeakRSS: %dKB \tPeakVM: %dKB\n",
        $task->mm->hiwater_rss*4, $task->mm->hiwater_vm*4)
    printf("\tSystem Time: %dms \tUser time: %dms\n",
        $task->stime, $task->utime)
    printf("\tVoluntary Context Switches: %d \tInvoluntary: %d\n",
        $task->nvcsw, $task->nivcsw)
    printf("\tMinor Page Faults: %d \t Major Page Faults: %d\n",
        $task->min_flt, $task->maj_flt)
}
```

Figure 1: Simple script for observing resource usage by processes

processes. The position of this information may change from kernel version to kernel version and it is recommended looking for “contents of the stat file” in the file `filesystems/proc.txt` in your kernel documentation.

- `getrusage` – this function returns resource usage information for a given process. Although not all information fields are available in Linux, page fault information is present.
- `task_struct` – the two methods above gather information from processes’ `task_struct`. It is possible to write a simple systemtap script to get this information directly from the source. Later in this session, we will present a systemtap script that gets page faulting information when a given process exits.

Some system calls are known to generate page faults; e.g., `fopen` calls `mmap` to allocate memory which can lead to a page fault. Some are known to present latencies or to have a time resolution that may not be good enough for some applications such as `select` timeout mechanism which uses a `jiffies` based timer. Careful thought and engineering can overcome these issues. A simple approach is to have a separate thread to deal with file operations and other latency prone tasks.

Creating threads on-the-fly can also be a problem if there is a constraint for the time between the event that should trigger thread creation and this new thread handling the event. The same thought is valid for creating new processes through `fork()`.

As already said, there are several ways to get resource usage information, such as: gathering information from `/proc/<pid>/status` in regular time basis; using the `getrusage` infrastructure; or writing a systemtap script that gathers information. One important point to keep in mind when choosing the method is *when* would be the best moment to get the information. Maximum memory usage would be best acquired when the process exits. Time spent in thread creation should be measured during thread creation. The first two methods are harder to synchronize with specific events. Systemtap scripts are more flexible and powerful.

The systemtap script below was used to observe resource usage by processes. When a process finishes, the script prints the maximum amount of real and virtual memory used by the application, CPU time used, voluntary and involuntary context switches, and how many page faults occurred during process execution. The script is simple, and shown in Figure 1.

Two interesting notes about the systemtap script in Figure 1:

- this script gathers information from processes’ `task_struct`, and it would be easy to show more information that may be of interest for the user if needed.
- when a process finishes, it calls `exit()` that calls `do_exit()`. The script hooks on `exit_mm()` that is halfway in `do_exit()`—meaning that *after* the script collects data, there still are a few extra steps until the process finishes. The only data that may be different is the system time as the process will spend more time until it really vanishes.

This script was key to observe that processes were spending a long time inside of `do_exit`—in some cases, up to 10 seconds. That helped understanding why performance comparisons carried using the `time` command were so horrible on PREEMPT_RT. The urge to fix this issue was placated by the perception that the process is exiting, it is already out of the run queue and its resources will be freed when needed and when the kernel can do that without disturbing the system.

Resource allocation also comprehends CPU allocation and CPU isolation. For an application with strict real-time constraints, that makes perfect sense reserving one or more CPU in a SMP machine. In some cases it is enough to be sure that thread A and thread B will run in different CPU—that can be easily achieved using the `taskset` command. Sometimes it is necessary to be sure that nothing else will run in a given CPU set, only the application—in this case the best solution is to use the `isolcpus` parameter in the kernel boot command line [7].

When taking in account the fact that PREEMPT-RT has more kernel threads running simultaneously due to threaded IRQ and softirq, it is clear that there are more processes competing for CPU time and that the scheduler works harder to accommodate all tasks. CPU isolation is a valuable tool especially when the application under test does not follow the guidelines discussed in this paper.

2.4 Using libautocork

What to do when a customer application shows 60% performance degradation when running on PREEMPT_RT? And what if after careful system tuning the performance degradation is about 40%?

As the application was network based, the first idea was asking the customer all the information about packet sizes, interval between packets, socket flags in use, and everything else that could give clues to solve the issue. Of course, source code was requested for inspection. Most of the specific questions were not answered and access to source code was denied due to internal security policies.

Several attempts to reproduce the problem in the laboratory, based on the information available, failed miserably. At this point, tests were carried by the customer

and our main concern was to not abuse or bother a customer that was as cooperative and helpful as possible.

Several small systemtap scripts were written to get socket information, packet size, protocols in use and everything else. At a certain point Nettaps [8] was created, a set of tsets and systemtap scripts collecting ideas discussed during the tests and a mix of the most relevant scripts already written. These scripts were sent to the customer and he was able to provide us valuable information on the internals of his application and runtime information.

The `drill.stp` [8] script collects information on the number of times each thread calls `writev`, `poll`, `select`, and `sched_yield`. It also collects statistics on lock contention, per thread and per lock. The last bit of information provided by this script is detailed information on network connections, including buffer size, packet size, average sizes for both buffers and packets, the status on Nagle usage, delayed ACK and similar information. An excerpt from a `drill.stp`'s test run:

```
[root@lab nettaps]# stap -I tapset drill.stp
thread: tid name nrwritev nrpoll nrselect
nrsched_yield
thread: 1934 auditd 6 0 6 0
thread: 1935 auditd 0 0 0 0
thread: 2652 sshd 0 0 4 0
thread: 3300 sshd 0 5 0 0
thread: 3301 sshd 0 0 8 0
thread: 3302 sshd 0 1 37 0
lock: tid futex nrcontentions avg min max
lock: 1934 0x000055555576d508 1 43672 43672 43672
lock: 1935 0x000055555576d534 1 1960 1960 1960
connection: tid saddr sport daddr dport nrbuf
avgbfsz minbfsz maxbfsz nrpkts avgpktsz minpktsz
maxpktsz nagle da ka wr
connection: 2652 192.168.0.200 22 192.168.0.100
38915 2 392 176 608 2 392 176 608 0 0 0 2
connection: 3300 192.168.0.200 22 192.168.0.100
36042 1 20 20 20 2 10 0 20 0 7 0 5
connection: 3301 192.168.0.200 22 192.168.0.100
36042 7 297 32 744 8 260 0 744 0 7 0 5
connection: 3302 192.168.0.200 22 192.168.0.100
36042 17 58 48 96 15 57 48 96 0 7 0 5
```

The second script sent to the customer, also part of Nettaps, was `lnlat` [8]. This script gathers information about packet flow, such as time spent by a packet when traveling from the network interface up to the user space application waiting for it, time between a user space application sending a packet and that packet reaching the network, network stack latency, and buffer size statistics. Running `lnlat` produces data like Figure 2.

```
[root@lab nettaps]# stap -I tapset lnlat.stp
```

latency(ns)		buffer size									
entry	local address	port	remote address	port	avg	min	max	avg	min	max	
user_in	10.0.0.152	43667	10.0.0.152	20975	38802	38802	38802	0	0	0	
user_in	10.0.0.164	20975	10.0.0.164	52965	37717	37717	37717	0	0	0	
user_in	10.0.0.164	51375	10.0.1.234	2222	35590	35590	35590	0	0	0	
user_in	10.0.0.164	20975	10.0.0.164	52964	51630	51630	51630	0	0	0	
user_in	10.0.0.164	20975	10.0.0.164	52995	32706	32706	32706	0	0	0	
user_in	10.0.0.152	20975	10.0.0.152	43687	32676	32676	32676	0	0	0	
user_in	10.0.0.152	20975	10.0.0.152	43666	50457	50457	50457	0	0	0	
user_in	10.0.0.164	37451	10.0.1.234	2222	40151	36293	44010	0	0	0	
tcp_out	10.0.0.164	20975	10.0.0.164	52966	1750	818	8150	48	0	232	
tcp_out	10.0.0.164	52966	10.0.0.164	20975	12360	6260	26967	7604	624	16384	
tcp_out	10.0.0.152	20975	10.0.0.152	43666	1708	928	2488	20	0	40	
tcp_out	10.0.0.152	43666	10.0.0.152	20975	5404	5404	5404	40	40	40	
tcp_out	10.0.0.152	20975	10.0.0.152	43668	1710	837	7104	48	0	232	
tcp_out	10.0.0.152	43668	10.0.0.152	20975	13161	6734	33385	7604	624	16384	
tcp_out	10.0.0.152	20975	10.0.0.162	41931	197615	8602	1154511	2585	496	4344	
tcp_out	10.0.0.164	52995	10.0.0.164	20975	1078	853	1589	3	0	40	
tcp_out	10.0.0.164	20975	10.0.0.164	52995	3493	1042	6633	713	0	840	
tcp_out	10.0.0.164	51371	10.0.1.234	2222	1876	1236	3082	20	0	34	
tcp_out	10.0.0.152	43687	10.0.0.152	20975	1144	864	1707	3	0	40	
tcp_out	10.0.0.152	20975	10.0.0.152	43687	3461	850	6736	713	0	840	
tcp_out	10.0.0.164	20975	10.0.0.164	52965	1478	931	2025	20	0	40	

Figure 2: Output from `stap -I tapset lnlat.stp`

Analyzing information received from the customer, that was not the one in the examples just shown, we were able to understand that:

- customer application was TCP/IP based;
- customer application was using the default Nagle algorithm [9];
- sending millions of small-sized packets;
- sending packets in a burst, without inter-packet interval.

Being the most used transport protocol poses a fantastic challenge for TCP to meet many different needs. Several heuristics were introduced over time as new application use cases and new hardware features appeared and as well kernel architecture optimizations were implemented. In an impressive way, default TCP/IP settings are able to satisfy most users.

As one example of the heuristics in use, TCP delays sending small buffers, trying to coalesce several before generating a network packet. This normally is very effective, but in some cases this heuristic is not the better fit.

Applications that want lower latency for the packets to be sent, especially when working with small packets, will be harmed by this TCP heuristic. There is a knob for applications that want to avoid this algorithm, a socket option called `TCP_NODELAY`. Applications can use it through `setsockopt` sockets API:

```
int one = 1;
setsockopt(descriptor, SOL_TCP, TCP_NODELAY,
&one, sizeof(one));
```

But for this to be used effectively, applications must avoid doing small, logically related buffer writes as this will make TCP send these multiple buffers as individual packets, and `TCP_NODELAY` can interact with receiver optimization heuristics, such as ACK piggybacking, and result in poor overall performance. In other words, both sender and receiver have to be in tune.

There are several sources of latency for TCP connections [10] and sometimes the solutions applied to a given operating system network stack may not be the best one for another. This becomes clear when working with applications ported from one operating system to another.

It was suggested to the customer using `TCP_NODELAY` to solve, at least partially, the performance drop. But the customer argued that even though the change seemed to

be simple, touching the code was not an option. If we could prove him that a considerable performance gain would be perceived he could try convincing his managers.

Using ideas discussed when trying to correlate data obtained from the Futex Contention systemtap example script [11] and the actual locks in the application, a Glibc stub was written to set TCP_NODELAY on the sockets used by a given application. The results were satisfactory but the behavior of TCP_NODELAY was not the optimal way of enhancing these connections. Then libautocork [12] was written.

The TCP socket option called TCP_CORK is a less known option, present in a similar fashion in several OS kernels, sometimes under a different name. The purpose of this option is to put a cork in the socket, preventing packets from being sent through this socket. When the application decides it is time to send the packet or packets, it just removes the cork.

Applying the cork to a socket is just a matter of setting TCP_CORK with a value of 1, as in this excerpt of code:

```
int one = 1;
setsockopt(descriptor, SOL_TCP, TCP_CORK,
    &one, sizeof(one));
```

That tells TCP to wait for the application to remove the cork before sending any packets, just appending the buffers it receives to the socket in-kernel buffers. That allows applications to build a packet in kernel space, something that can be required when using different libraries that provide abstractions for layers.

One example is on the SMB networking protocol code, where headers are sent together with a data payload, and better performance is obtained if the header and payload are bundled in as few packets as possible.

When the logical packet was built in the kernel by the various components of the application, it is time to remove the cork and send the packet. It is important to note that the kernel does not have a way to identify, on behalf of the application, that the packet is ready and must be sent. To remove the cork the application uses:

```
int zero = 0;
setsockopt(descriptor, SOL_TCP, TCP_CORK,
    &zero, sizeof(zero));
```

That makes TCP send the accumulated logical packet right away, without waiting for any further packets from the application, something that it would do in other cases, to fully use the network maximum packet size available.

In order to verify the impact of TCP_CORK on the customer application, without access to source code, we decided writing a library to be preloaded and interfere in a few socket related operations. There were other options, such as a systemtap script to touch the system calls related to the functions intercepted by libautocork but the preloaded library offered lower overhead and was simpler to write.

Using libautocork is just a matter of:

```
LD_PRELOAD=libautocork.so ./customer_app
```

Libautocork applies the cork to the sockets and automatically removes it whenever the application waits for an answer to what has been sent. The removal of the cork happens when the application calls `recv`, `recvmsg`, `select` and similar functions.

2.5 Comparing Apples to Apples

System configuration plays an important role in performance test results. In addition, it is important to compare results in the same environment, changing the minimal set of elements possible.

When a little knob is changed, the system may behave in a completely different way during the test. As these knobs vary from `/proc/sys` writable files to device drivers parameters in modules, it may be a hard job to record the exact environment where a test has been conducted. Even the default values for knobs that were not touched may vary from kernel version to kernel version.

Depending on the nature of the application under test, some knobs have no effect on the test. Some knobs may have a visible effect on the results. Determining all the available knobs and their values and eventually recording these values to replay a test or perform another test in the same environment is surely a difficult task.

Some effort has been done to address this need and projects like Tuna [13] and AIT are good examples of such tools. The authors recommend using similar tools to ease the task of comparing results, defining best configurations and even identifying regressions.

2.6 Conclusion

Testing applications under PREEMPT_RT was, most of time, a refreshing and enlightening task. Understanding why a given application presented such a different behavior or performance just by changing the kernel or why some race conditions and bugs were more likely to be triggered under PREEMPT_RT was really insightful. In fact, even kernel subsystems and device drivers had bugs and race conditions uncovered by PREEMPT_RT. The bugs were there all the time, but PREEMPT_RT was more likely to trigger them—this way, several bugs were found and fixed.

On the other hand, seeing all the theories about a given problem falling apart, being unable to reproduce a bug, having to solve a negative performance hit presented by an application without access to information or the source code was sometimes hard to manage. It was even harder to reproduce the exact test environment or the best configuration found without a tool. These moments required creativity and revived the joy of coding and hacking.

The authors hope the reader will benefit from the system tuning and development techniques briefly described in this paper. Hopefully, the readers will even feel inspired by the testing ideas explained through the text and endure the process of creating their own test devices and tools or even automating “bad coding techniques” checking. We hope that ultimately, the reader will find the tools presented here helpful and use them to solve his/her problems.

References

- [1] Steven Rostedt and Darren V. Hart, “Internals of the RT Patch,” *Proceedings of the Linux Symposium*, Ottawa, Ontario, Canada, 2007, pp. 161–172.
- [2] Real-Time Linux Wiki,
<http://rt.wiki.kernel.org>
- [3] Arnaldo Carvalho de Melo, *Techniques that can have its behavior changed when the kernel is replaced*,
<http://oops.ghostprotocols.net:81/acme/unbehaved.txt>
- [4] Ingo Molnar, *Re: Network slowdown due to CFS*,
<http://lkml.org/lkml/2007/9/26/132>
- [5] <http://www.mail-archive.com/linux-rt-users@vger.kernel.org/msg02364.html>
- [6] *HOWTO: Build an RT-application*, http://rt.wiki.kernel.org/index.php/HOWTO:_Build_an_RT-application
- [7] Kernel boot parameters, Documentation/kernel-parameters.txt
- [8] Nettaps,
<http://oops.ghostprotocols.net:81/acme/nettaps.tar.bz2>
- [9] Wikipedia, *Nagle's algorithm*,
http://en.wikipedia.org/wiki/Nagle%27s_algorithm
- [10] Robert A. Van Valois, Todd L. Montgomery, Steven R. Wright, and Eric Bowden, *Topics in High-Performance Messaging*,
<http://www.29west.com/docs/THPM/thpm.html#TCP-LATENCY>
- [11] Systemtap War Stories: Futex Contention,
<http://sourceware.org/systemtap/wiki/WSFutexContention>
- [12] Libautocork,
http://git.kernel.org/?p=linux/kernel/git/acme/libautocork.git;a=blob_plain;f=tcp_nodelay.txt
- [13] Arnaldo Carvalho de Melo, “If I turn this knob...what happens?,” *Proceedings of the Linux Symposium*, Ottawa, Ontario, Canada, 2008.

IO Containment

Naveen Gupta
Google Inc.
ngupta@google.com

Abstract

In existing Linux IO schedulers there is no way to differentiate latency-sensitive IO from the background IO. This means that jobs which have strong latency requirements cannot coexist with batch jobs. In the past various ad-hoc solutions have been used to limit latencies such as throttling background workload, periodic syncing to handle delayed writes, and limiting the queue length.

With increasing CPU and memory speeds we generally end up having multiple applications on the same machine sharing the disk. In the absence of differential service, it is not possible to run a latency-sensitive application along with a job doing batch updates. We have added *priorities* in the *anticipatory IO scheduler* so that an IO submitted by an application can be treated differently depending on the priority of IO. We are able to show considerable improvement in the latency of *high priority* applications using this mechanism.

1 Introduction

There is an effort within the Linux community to provide isolation for memory and network resources to run multiple jobs independently on a single machine. IO containment has been a difficult problem for various reasons. The non-deterministic nature of storage devices makes any kind of QoS or isolation a harder problem for IO as compared to other computing resources. Variations in an IO stream—like size, placement of data on platter, or seek distance—have played a major role in determining throughput and latency. This inability to quantify the device characteristics leads to inefficiency when performing any kind of isolation. Even the applications sharing these resources have vastly different performance and service quality requirements.

Disk isolation can be done over one or multiple IO characteristics. For workloads needing different ser-

vice rates—bandwidth scheduling—which may be implemented as a strict bandwidth guarantee or as proportional scheduling, may suffice. Latency bounds are needed for real-time applications sharing their storage resources, while allocation of time slices is needed for shared usage in applications wanting to schedule for non-deterministic things like rotational delay and seek time rather than raw throughput. Resource accounting frameworks may need to limit the number of seeks in a given container. Relative *priority* of a request as compared to other outstanding requests is helpful in separating *latency-sensitive* traffic from *best-effort* traffic. In context of web searches, we need to serve low-latency user traffic independent of background index updates.

The Linux mainline has support for four different IO schedulers [5]. They have mainly aimed towards improving bandwidth by reordering requests, submitting them in batches, and merging outstanding requests. The *deadline* scheduler tries to process all read requests within a specified time period by selecting expired read requests from a *FIFO* list. The *anticipatory* scheduler [14] is a non-work-conserving scheduler which reduces seeks for a sequential workloads. The *CFQ* scheduler allocates requests fairly amongst various processes, while *NO-OP* provides basic merging. Apart from *CFQ*, which allocates larger time slices to higher priority processes, there is no differential service in the current mainline IO schedulers.

Our current work uses *non-work-conserving* scheduling to provide *priority-based* scheduling by building upon the *anticipatory* scheduler. The current implementation of the *anticipatory* scheduler solves the problem of *deceptive idleness* [14] present in applications which issue synchronous reads. It identifies *deceptive idleness* as a condition where due to a short period of computation, the scheduler incorrectly assumes that the last request-issuing process has no further requests and dispatches the requests from another process. These workloads cause seek-optimizing schedulers to multiplex between

requests from different processes. Letting the disk remain idle for a short period lets the scheduler select the next nearby request from the same process, thereby increasing disk performance at the cost of a small loss in utilization. In our work we extend this concept to block pending requests from a *low priority* process for a short duration after submitting a request from a *high priority* process. This prevents an incoming *high priority* request from being delayed due to a dispatched *low priority* request. In addition to this a running batch of *low priority* processes can be interrupted on seeing a request of *high priority*.

2 Related work

Jassura et al. [1] have used both *non-work-conserving* and *work-conserving* approaches at the user and kernel levels to provide differentiated levels of service in web content hosting based on priority. Their main metric for quality of service for an HTTP request was latency. In their implementation, a request may be postponed if the load on the system is already high or if the request is of lower priority. Their solution was not restricted to a particular subsystem and was a preliminary step in investigating QoS mechanisms in web servers. In their study, they observed that they had to limit the number of processes to have differentiated performance. An important finding was that a *non-work-conserving* approach is better since in their implementation a *work-conserving* approach could not do much differentiation with a large number of processes. Our approach isolates latency even with multiple antagonists.

Seetarami Seelam et al. [23] proposed that throttling IO streams in high performance computing systems improved I/O performance. In Linux, similar approaches are sometimes used to limit the latency of a *high priority* job while running a throttled background job with decreased utilization. By using a *non-working-conserving* scheduler we have been able to improve isolation without throttling the IO streams.

Isolation based on time slicing has been used in *CFQ* [3], the *eclipse operating system* [6], and *YFQ* [22]. *Argon* [27] defined insulation as reduced interference between workloads, allowing sharing with a bounded loss of efficiency. But its focus was on throughput efficiency and though usually it reduced average response times, it could increase the variation in

latency and worst-case response times. Our implementation of *anticipatory priorities* insulates both the average and maximum latency.

Dynamic selection of IO schedulers [21] investigated the possibility of using runtime switching of IO schedulers to *deadline* for providing latency bounds while using *CFQ* for best throughput traffic. This selection was based on feedback from the IO system and the workload. Such a technique would be difficult to optimize for competing workloads while incurring no overhead for scheduler switches.

In order to compensate for the non-deterministic nature of disks, QoS schedulers rely on conservative estimates of bandwidth and latency. Some implementations use weighted fair queuing coupled with admission control. [7, 11, 15, 16, 19] all use a fair queuing/tag-based approach. Even though request stream patterns and the disk characteristics impact the performance of other applications, these studies either ignore or take a conservative estimate. Another approach taken in IO schedulers [19] is to take into account a disk model, but such techniques are not feasible with numerous disk drives, since most often it is not specified by the manufacturer and, models are an approximate representation for today's complex drives. Latency-bound IO in QoS scheduling is orthogonal to our problem, since we want *prioritized* traffic to be isolated. We don't want it to be delayed and scheduled later to satisfy pre-calculated bounds.

The badger Project [13] solved a similar problem of improving the latency of transaction synchronous requests while improving the throughput of asynchronous transaction database requests. They realized that the only latency control available for synchronous requests with current schedulers was to limit the number of pending background requests. Since they were focusing on a database, they chose to implement *priorities* in the *deadline* scheduler since it is the most-used IO scheduler for databases [18]. They implemented one *FIFO* per *priority* level where *priority* was determined when the deadline was set and that *low priority IO* would wait for a *high priority* request. This effectively improved the latency of *high priority* requests in their experiments with better throughput than the no-priority, rate-throttling approach. But since *priorities* determined when the deadline was set, the latency was still bound by this deadline. Our approach has no disk-parameter-agnostic bounds and the non-work conserving approach achieves tighter latency bounds [1]. Also, throughput [4] with the *antic-*

ipatory scheduler is inherently more than with *deadline* schedulers.

ABISS [10] is designed to provide guaranteed reading and writing bit-rates to applications, with minimal overhead and low latency. They use a custom *priority* scheduler and have support for *CFQ*. Their solution could use *anticipatory priorities* since it has lower latency for higher priority jobs than *CFQ*.

3 Design

The *anticipatory IO scheduler* [18] is a one-way elevator algorithm with limited backward movement. It has *FIFO* expiration times for both reads and writes. These queues are maintained separately and expiration times are tunable. This is similar to the *deadline* IO scheduler implementation for interrupting the elevator sweep. Another feature of the *anticipatory* scheduler is batching, where bunches of read requests or write requests are submitted together. The *anticipatory* scheduler alternates dispatching read and writes batches to the driver. Also, the scheduler anticipates when reads are dispatched to the driver one at a time. At completion, if the next request is close to the previous request or from same process, it is dispatched immediately.

Priority scheduling changes each of these policies to improve response time for a *high priority* process and prevents starvation of a *low priority* one. To change these policies, two data structures are also changed, `sort_list` and `fifo_list`. Changes to these structures are described below as part of policy changes.

Sort queue: For the one-way elevator, we add two queues per *priority* level so that the requests in a given level can be served independently of requests at any other level according to their layout on the disk. Instead of two `sort_lists`, we now have two `sort_list` structs for each level. These lists contain requests in sorted order according to their layout on disk. A backward seek can still occur when choosing between two IO requests where one is behind the elevator's current position, and the other is in front. As before, if the seek distance to the request in the back of the elevator is less than half the seek distance to the request in front of the elevator, then the request in the back can be chosen.

FIFO queue: Instead of one *FIFO* queue for each request direction, we now have two *FIFO* queues for each

priority level. Only when the requests from a given *priority* level are being served, the read or write from that level expires to interrupt the IO scheduler in its current one-way sweep or read anticipation. This prevents the starvation of requests in a given *priority* level. The expiration times for requests on these lists are tunable using the existing parameters `read_expire` and `write_expire`.

Changes to Anticipatory Core: In the vanilla *anticipatory* scheduler, when a read request completes, the next request is not dispatched to the driver unless it is from same process or a nearby request [18]. *In the priority scheduler, the next request will wait if it is from a process of lower priority, in anticipation of a request from a process of higher priority. When a request with higher priority is submitted by the application, it will break any anticipation happening in the scheduler.* This helps us prioritize a request over a batch of low-priority requests. It still maintains the earlier statistics on *think time* and *mean seek distance* for the process to decide if it is worthwhile to wait for a request. The read anticipation can be disabled using `antic_expire` set to zero.

Changes to Batch submission: In most IO schedulers, requests are served in batches where a batch is a set of requests of one kind (read or write). For a read/write request, the scheduler submits read requests until there are more read/write requests to submit and batch time has not been exceeded. Before scheduling requests for the alternate direction, the scheduler lets all requests from the previous batch complete.

In addition to the time limit imposed on batches by `read_batch_expire` and `write_batch_expire`, the *priority* scheduler limits the number of requests for a given *priority* level using tokens associated with that *priority* level. Instead of alternately serving read and write batches, the *priority scheduler* selects a particular *priority* level and submits requests in one direction as long as there are requests in the chosen direction or there are tokens left for that level. The batching of requests for a given *priority* level maintains the localization per *priority* level and gets fairly good results due to inherent batching in processes in a given *priority* level. As before, the read and write *FIFO* expiration times are checked only when scheduling IO of a batch for the corresponding (read/write) type.

Handling merges: Merging is another property of vari-

ous IO schedulers. If a request entering the IO scheduler is contiguous with a request that is already on the queue, either to the front or to the back of a request, it is called a front-merge candidate or a back-merge candidate. If the size of new request is less than what can be handled by the hardware, this new request is merged with the one already in queue. With *priorities*, a request is merged with an already existing request if it satisfies the above criteria and also if it is from the same *priority* level as the existing request. Merging of the request with a request from another level should be possible, but we need to see if this has any performance gain. Also, there is an issue of whether we need to change the *priority* band of the merged request, especially when dealing with logs and journaled file systems due to a large number of merges.

Handling writes

Attaching Priority: The *priority* of a request is derived from the current context and uses the infrastructure in the Linux kernel for *CFQ*. Since writes are submitted asynchronously, they get submitted in the context of background threads or the process doing direct reclaim. A prototype implementation was created to attach *priority* to the page struct in submission context. This *priority* is transferred to the request during submission.

Writes and Page cache: While serving cached read traffic, an application doing writes could effectively wipe out the page cache unless it is submitted to the elevator frequently. One of the possible solutions would be to throttle the dirty path with feedback from the elevator regarding available tokens for that *priority* level. A second approach would be to do memory isolation of an antagonist job. Finally, one can periodically sync the dirty page cache. For simplicity we adopted this approach to eliminate interference from background high-throughput jobs. We experimented using both periodic dropping of cache and direct IO for background write traffic.

Workload Description: We have attempted to isolate latency, in particular we want to improve the latency of seek-bound (low throughput) search traffic. When this search traffic is served, in the background, the data that is being used to serve this search traffic (index) is refreshed for more up-to-date results. This interferes with foreground search traffic. Most of the foreground traffic is random synchronous reads, some of which is sensitive to the amount of cache in the system. We have synthesized these two workloads to quantify the above

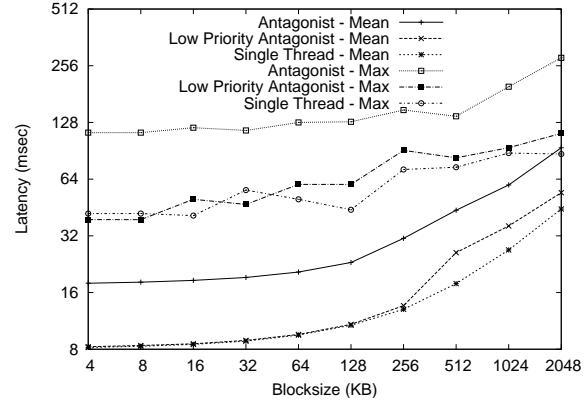


Figure 1: Random read latency – varying block size

problem.

1. Sequential read of 80% of the file followed by random reads to achieve good amounts of hits in undisturbed cache. This is run along with antagonist sequential read/write traffic.
2. Random reads with sequential background read/write.

4 Experimental Results

We evaluated the performance on two different configurations. The first one is a dual-core 2Ghz system with 8GB of RAM. The test disk is a SATA drive with a capacity of 400G and 8MB onboard cache. The second system is a dual-core 2.4Ghz system with 16GB of RAM and a SATA drive with a capacity of 500G and 8MB onboard cache. Both systems are running a customized 2.6.18 kernel with ext2. The experiments were conducted using synthetic workloads generated using fio [2]. The tests were run on separate drives not running any other workload.

1. Random and Sequential Read Latency

The first experiment is designed to measure the latency impact of random reads in the presence of competing random reads (Figure 1). It measures the mean and maximum latency of reads for different block sizes. When more than one thread is running, both use the same block size. The three cases in this experiment are

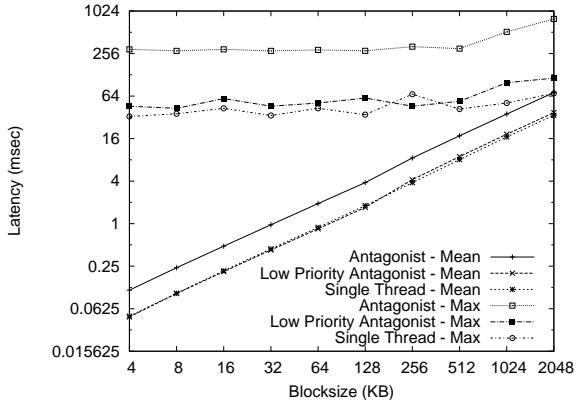


Figure 2: Sequential read latency – varying block size

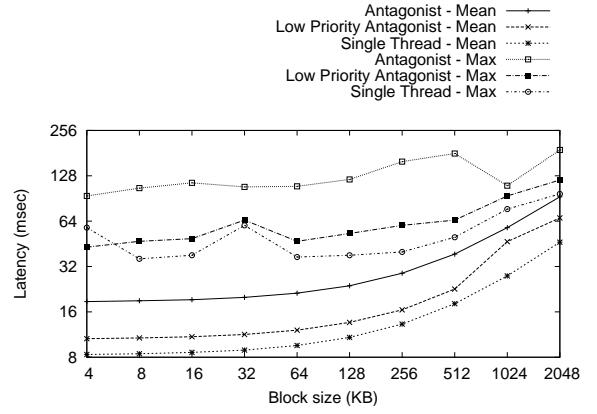


Figure 3: Direct IO latency – varying block size

- (a) *Single Thread*: One thread doing random reads.
- (b) *Antagonist*: Two threads doing random reads.
- (c) *Low Priority Antagonist*: Two threads doing random reads at two different *priority* levels.

When a competing thread is introduced without any *priority* (b), the average latency is almost two times as compared to when a single thread (a) is running and the max latency is more than double. On lowering the IO priority of the antagonist (c), the mean and the maximum latency for the main job is almost similar to when only a single thread is running. Changing the *priority* of the antagonist has isolated latency of foreground job.

For sequential reads, the results are similar. The mean latency and the maximum latency are isolated from competing workload on increasing the IO priority of main thread (Figure 2). In case (b), the maximum latency is more than 4 times, though the average latency is almost twice—like in the case of random reads. Here also, by increasing the *priority* of the main thread (c), the effects of an antagonist are minimal.

2. Direct IO

In the second comparison threads are doing random direct IO (Figure 3). Here also, we have three cases where:

- (a) *Single Thread*: One thread running doing direct IO.
- (b) *Antagonist*: Two threads doing direct IO at the same *priority* level.

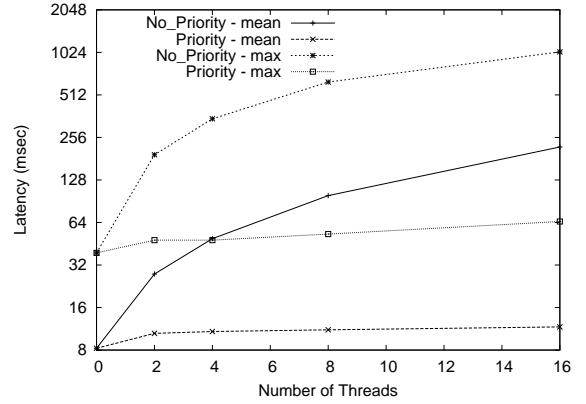


Figure 4: Multiple threads in background – block size 4k

- (c) *Low Priority Antagonist*: Two threads doing direct IO at different *priority* levels.

In case (c) the latency is reduced, since the *priority scheduler* anticipates a *high priority* IO even if there is a pending *low priority* IO in queue. This means that a *high priority* IO is not blocked behind a *low priority* even if the submissions happened one after the another. In the case of a *work-conserving* scheduler, a direct IO submission will be handed off to the driver and hence the latency of a *high priority* job is not isolated from *low priority* submissions. For all measured block sizes, the equal *priority* antagonist had more than twice the latency as compared to when a single thread is running, but when *higher priority* was assigned to the main thread, both the average and maximum latency were reduced to a large extent.

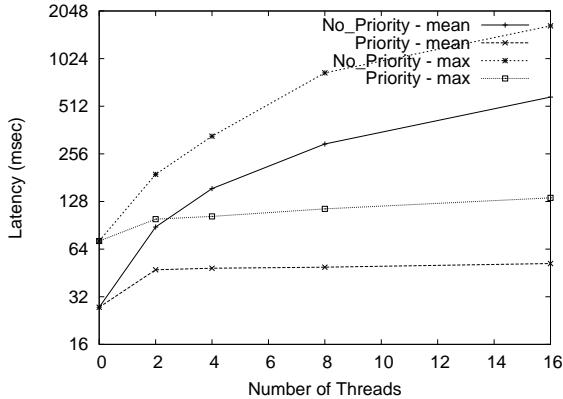


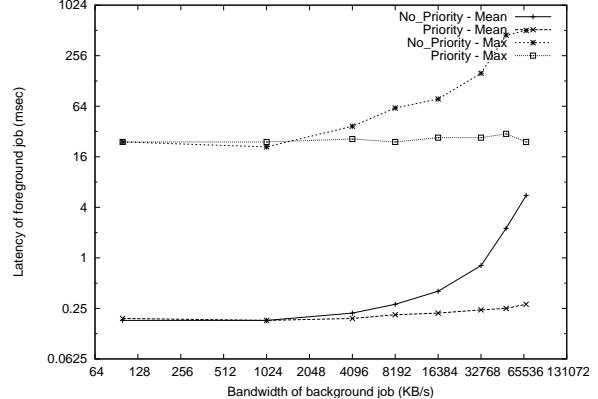
Figure 5: Multiple threads in background – block size 1m

3. Multiple Background Threads

Figure 4 shows the isolation provided to a high priority job in the presence of multiple low priority jobs. It has a thread doing 4k direct IO with a varying number of antagonist jobs doing direct IO of the same block size. When the antagonist threads run at the same *priority* as that of the main thread (*No Priority*), the latency (both average and maximum) roughly doubles with each doubling of the number of antagonists. On assigning higher priority to the main thread (*Priority*), the latency is fairly constant even when the number of background threads is increased. Figure 5 plots the results of changing the number of background threads when the block size is 1MB. Irrespective of block size, varying the number of background threads has little or no effect on the latency of the foreground job. Earlier *user-level* and *kernel-level* approaches to using *priorities* in both *work-conserving* and *non-work-conserving* schedulers [1] required restricting the number of jobs to obtain differentiated performance. The current scheme has no such limitation.

4. Rate-limited Background Load

Sometimes in order to limit the latency of a typical latency-sensitive job which is not throughput-bound, the background thread needs to be rate limited. This reduces the probability of interference of IO from an antagonist on the main workload. Due to this, the utilization of the disk drive is reduced by a large extent. Figure 6 plots latency of the foreground job while the throughput of the



background.

- (c) buf: 64k-sized sequential buffered IO in background.
- (d) buf-fadv: 64k-sized sequential buffered IO where the cache is dropped using `FADVISE_DONT_NEED` [17] after every 16MB.
- (e) buf-prio: 64k-sized sequential buffered logging traffic running at lower priority.
- (f) buf-prio-fadv: In addition to lower priority, sequential buffered antagonist has cache for the antagonist being dropped after every 16MB.
- (g) direct-64k-prio: Direct IO of block size 64k at lower priority.
- (h) buf-sync-16m: 64k sequential buffered background IO with sync every 16MB.

Type -	mean (msec)	max (msec)	std dev -	b/w MB/s
no-load	5.97	149	7.81	-
direct-64k	33.97	373	83.34	1.9
buf	111.78	528	160.05	0.6
buf-fadv	40.67	784	97.00	1.6
buf-prio	9.34	182	12.42	6.7
buf-prio-fadv	7.74	112	11.05	8.1
direct-64k-prio	7.48	116	10.32	8.3

Table 1: Random cached reads with background sequential reads

5. Random Cached Reads With Background Reads

In this setup, 8GB of the 10GB file is read sequentially into memory and will form cache for the foreground thread doing 64k random reads. Table 1 shows the mean, max, and standard deviation of latency as well as the average bandwidth of the foreground job for various kinds of reads in the background. A random read in the presence of cached data performs worse when a sequential buffered read traffic (c) is running in background, since the logging traffic quickly destroys the cache and, in the absence of *priority*, competes with the foreground random traffic. Lower mean latency when using `fadvise` (d) confirms that the major portion of increase in latency for the buffered case comes from the cache being wiped out. Using lower priority either with buffered (e) or direct

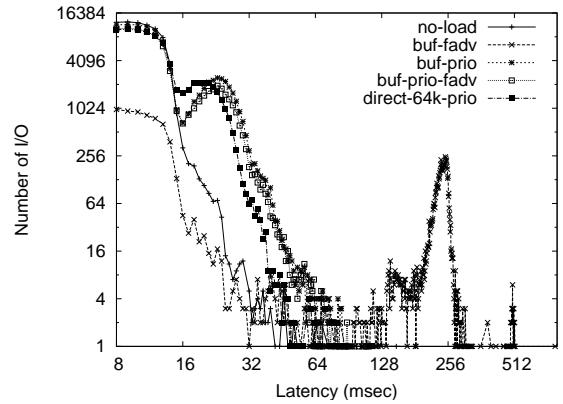


Figure 7: Foreground Cached Random reads – Background sequential reads

(g) reads in the background reduces both the average and tails of the foreground job. Reads which constantly drop cache and run at lower priority (f) further reduce the average and tails. Direct IO at lower priority (g) and buffered with cache drop (f) perform the best as they reduce the latency and at the same time increase bandwidth of the foreground job. Direct (g) is marginally better than (f).

Figure 7 shows the distribution of per-IO latency of the main thread. Even though Table 1 shows that buffered read running with lower priority alone (e) has a slightly longer tail as compared to when *priority* is used along with `fadvise` (f) or direct IO (g) for background threads, the graph clearly shows that running a lower priority read antagonist in all three cases (running alone (e), with `fadvise` (f), and with direct IO (g)) has almost similar distribution. Running `fadvise` alone (d) has a very long tail due to contention in the scheduler.

6. Random Cached Reads With Background Writes

Similar to the last experiment, here also 8GB from a 10GB file are pre-cached into memory sequentially, followed by the main thread performing 64k random reads. In this case, the background load is sequential writes of various kinds, unlike in last experiment, where we were reading in the background. Here, direct IO (b) performs the worst due to contention in scheduler. Sequential (c) is a lot better, but dropping cache periodically; (d) improves the average latency further. Since the background writes can quickly wipe the cache, using *priority* alone (e) does not reduce the average as

Type -	mean (msec)	max (msec)	std dev -	b/w MB/s
no-load	6.0	102	7.86	-
direct-64k	36.60	508	88.29	1.7
buf	12.89	891	30.83i	4.8
buf-fadv	7.49	236	15.68	8.3
buf-prio	12.15	258	15.10	5.1
buf-prio-fadv	8.17	133	12.06	7.6
buf-sync-16m	9.36	349	17.45	6.7
direct-64k-prio	6.27	99	8.60	9.9

Table 2: Random cached reads with background sequential writes

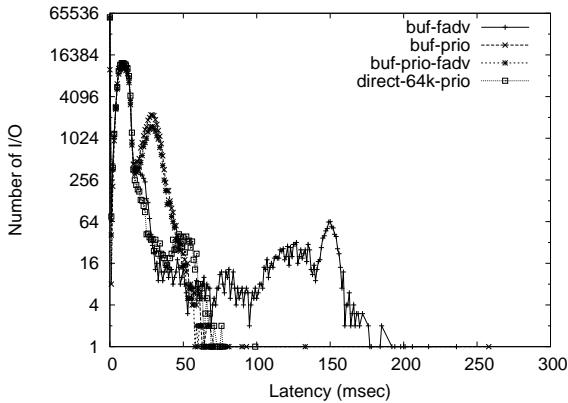


Figure 8: Foreground Cached Random reads – Background sequential writes

compared to buffered writes without *priority* (c). Similar to the results of the last experiment, lower priority direct writes (g) or buffered writes along with dropping cache (f) give the best performance both for latency (mean or max) and bandwidth. Using *priority* moves direct IO (g) from being the worst antagonist to least intrusive antagonist. Also, it reduces the tail for both direct (g) and buffered (f) logging traffic. Figure 8 has per-IO latency distribution for the main thread and shows the long tail of *buf-fadv* approach. Though Table 2 shows a higher value of max for *buf-prio*, Figure 8 clearly shows that for all cases using *priority* (e, f, g), the tails end around 100msec. Doing sync alone (h) does not perform as well as *fadvise* (d) due to cache effects.

7. Random Reads With Background Reads

Unlike the previous two experiments, there is no

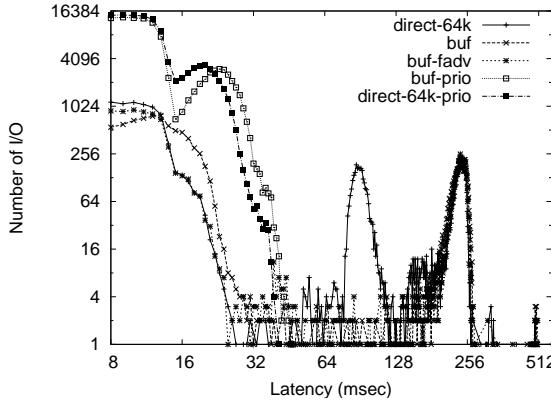


Figure 9: Foreground Random reads – Background sequential reads

sequential caching read in the next two experiments. In Table 3 we measure the latency and throughput of a thread doing 64k random reads with different kinds of antagonists reading sequentially. Unlike in the case of caching reads, *fadvise* does not help here, both mean and maximum latency when using *fadvise* (d) is almost similar to buffered reads (c). Using *priority* (e) reduces the average and maximum latency by eliminating contention in the scheduler queues. It helps in increasing bandwidth and reducing latency for both direct (g) as well as buffered IO (e). Direct IO seems to be slightly better, but the difference between it and buffered IO is within experimental noise. Figure 9 shows the per-IO latency distribution and in all cases where *priority* is used (e, f, g), tails are shortened by a large amount.

8. Random reads with background writes

Type -	mean (msec)	max (msec)	std dev -	b/w KB/s
no-load	8.91	32	9.34	6959
direct-64k	93.46	579	138.14	714
buf	115.29	506	161.29	577
buf-fadv	113.87	507	161.08	583
buf-prio	11.88	47	13.59	5294
buf-prio-fadv	11.47	63	13.05	5475
direct-64k-prio	11.13	46	12.34	5633

Table 3: Random reads with background sequential reads

Type -	mean (msec)	max (msec)	std dev -	b/w KB/s
no-load	9.16	105	9.66	6782
direct-64k	100.94	512	145.47	654
buf	12.93	698	30.01	4939
buf-fadv	11.39	408	19.36	5516
buf-prio	13.09	155	15.66	4825
buf-prio-fadv	12.53	325	15.03	5033
buf-sync-16m	11.47	218	19.14	5482
direct-64k-prio	9.66	96	10.67	6450

Table 4: Random reads with background sequential writes

Table 4 has the results of running a 64k random read thread along with various kinds of sequential write threads running in the background doing 64k-sized IO. This is similar to experiment 6, but with no caching reads. Unlike experiment 6, buffered writes (c) in the background are better than direct IO (b), since in this case wiping the cache does not have an impact on the foreground random read thread. Using direct IO (b) without priority causes heavy contention in the scheduler, unlike the normal write requests. In fact, using priority (e) or `fadvise` (d) has negligible impact on the mean latency, though the latency tail is reduced to a large extent by using `priority`. The large maximum latency in Table 4 when using both `fadvise` and priority (f) is due to an outlier. Looking at Figure 10 it is clear that using `priority` eliminates long tails for all kinds of logging workloads (e, f, g). And like experiment 6, direct IO has changed from being the worst antagonist to the least intrusive one.

5 Conclusion and Future Work

Using *priorities* in a *non-work-conserving* scheduler, we have been able to isolate latency of both buffered and direct high-priority synchronous requests. Increasing the number of background jobs has small/negligible impact on latency. Even though we tagged asynchronous requests, multiple writers still perform non-deterministically. For our example workload’s (1) cache-sensitive reads, interference from background reads can be effectively isolated using `priority`. In the case of background writes, using `priority` along with cache drop hints or direct IO reduces the average. For

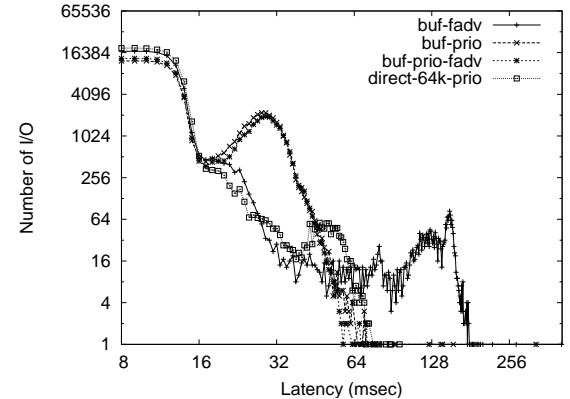


Figure 10: Foreground Random reads – Background sequential writes

random reads (2), *priority* is able to handle reads/writes as antagonists. Using *priority* we can effectively eliminate tails in all cases.

In our solution we either bypassed cache (direct IO) or used explicit hints to drop cache (`fadvise`) for background writes, but in a generic solution apart from feedback controlled IO throttling, we need prioritized submission in background writes like `pdflush`. Though we used `page struct` as a prototype to tag asynchronous *priorities*, in context of `cgroups` we could use `ioprio` of a control group associated with the page as proposed by Fernando [8]. Another non-container solution would be to use `io_context` to hold this information.

There is a push to create an IO subsystem controller for dividing available bandwidth among various `cgroups`. The Linux kernel mailing list had a proposal to limit the bandwidth [20] based on values configured in a control group filesystem. [28] [12] [25][9] are proportional bandwidth-scheduling solutions, while [12] is an IO scheduler solution, [25] is the solution in the device-mapper driver. [26][24] are attempts in the `cfq` scheduler to add fairness control per group rather than per process. While we agree that dividing bandwidth proportionally is needed for I/O controller, we would also like to be able to attach `priority` to a process or control group. This is a more feasible solution than adding latency and bandwidth requirements to a control group. In this solution, a group having higher priority would be given preference within its quota of bandwidth.

Proposed interface for such a scheduling scheme

1. *Using cgroup interface:* Add `blockio.bandwidth` and `blockio.priority` files per cgroup.

```
/dev/cgroup/group1/blockio.bandwidth 20
```

```
/dev/cgroup/group1/blockio.priority 0
```

```
/dev/cgroup/group1/blockio.prios_allowed BE(2-4)
```

In this scheme, all processes submitting IO within `group1` get 20% of available bandwidth, while being the highest priority (0) they are served before processes from any other group. Moreover, processes in `group1` are allowed to use only *best-effort* levels 2–4.

2. *Using ionice approach:* When not using cgroups, for per-process bandwidth/latency control, `ioprio_set()` / `ioprio_get()` can be used to specify bandwidth as well. This is overloading of the current definition used by *CFQ*, but we could add another class not conflicting with ones defined now.

6 Acknowledgements

Thanks to Mike Waychison, Grant Grundler, Shishir Verma, Paul Menagae, and Al Borchers for reviewing this document or providing technical guidance. Special thanks to Mike Waychison for code reviews and discussions.

References

- [1] J. Almeida, M. Dabu, A. Manikutty, and P. Cao. Providing differentiated levels of service in web content hosting. In *Workshop on Internet Server Performance*, pages 91–102, 1998.
- [2] Jens Axboe. Fio - flexible io tester. <http://freshmeat.net/projects/fio/>.
- [3] Jens Axboe. [patch][cft] time sliced cfq ver18. <http://lkml.org/lkml/2004/12/21/67>.
- [4] Jens Axboe. Time sliced cfq io scheduler. <http://lwn.net/Articles/113869/>.
- [5] Jens Axboe. Linux block io – present and future. In *Ottawa Linux Symposium*, pages 51–61, 2004.
- [6] J. Bruno, E. Gabber, B. Ozden, and A. Silberchatz. The eclipse operating system: Providing quality of service via reservation domain. In *USENIX Annual Technical Conference*, pages 235–246. USENIX, 1998.
- [7] John Bruno, Jose Brustoloni, Eran Gabber, Banu Ozden, and Abraham Silberschatz. Disk scheduling with quality of service guarantees. In *IEEE International Conference on Multimedia Computing and Systems*, volume 2, page 400, 1999.
- [8] Fernando Luis Vazquez Cao and Hiroaki Nakano. Cfq vs containers. http://iou.parisc-linux.org/lsf2008/IO-CFQ_vs_Containers-Fernando_Luis_V%e1zquez_Cao.pdf.
- [9] Fabio Checconi. Bfq i/o scheduler. <http://lkml.org/lkml/2008/4/1/234>.
- [10] Giel de Nijs, Benno van den Brink, and Werner Almesberger. Active block i/o scheduling system. In *Ottawa Linux Symposium*, pages 109–126, 2004.
- [11] Ajay Gulati, Arif Merchant, and Peter J. Varman. pclock: an arrival curve based approach for qos guarantees in shared storage systems. In *SIGMETRICS*, pages 3–24, 2007.
- [12] Naveen Gupta. [rfc] proportional bandwidth scheduling using anticipatory i/o scheduler. <http://lkml.org/lkml/2008/1/30/10>.
- [13] Christoffer Hall-Frederiksen and Philippe Bonnet. Using prioritized i/o to improve storage bandwidth in mysql. VLDB 2005.
- [14] Sitaram Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous i/o. In *SOSP*, 2001.
- [15] W. Jin, J. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. In *ACM Sigmetrics -Performance*, 2004.
- [16] C. Lumb, A. Merchant, and G. Alvarez. Facade: Virtual storage devices with performance guarantees. In *Conference on File and Storage Technology*, pages 131–144, 2003.

-
- [17] Andrew Morton. Usermode pagecache control: fadvise().
<http://lkml.org/lkml/2007/3/3/110>.
 - [18] Nick Piggin. Anticipatory i/o scheduler.
<http://lxr.linux.no/linux/Documentation/block/as-iosched.txt>.
 - [19] Lars Reuther and Martin Pohlack. Rotational-position-aware real-time disk scheduling using a dynamic active subset. In *24th IEEE International Real-Time Systems Symposium*, page 374, 2003.
 - [20] Andrea Righi. cgroup: limit block i/o bandwidth.
<http://lkml.org/lkml/2008/1/18/166>.
 - [21] S. Seelam, J. Babu, and P. Teller. Automatic i/o scheduler selection for latency and bandwidth optimization. In *Workshop on Operating System Interference in High Performance Applications*, 2005.
 - [22] S. Seelam and P. Teller. Virtual i/o scheduler: a scheduler of schedulers for performance virtualization. In *3rd international conference on Virtual execution environments*, pages 105–115, 2007.
 - [23] Seetharami R. Seelam, Andre Kerstens, and Patricia J. Teller. Throttling i/o streams to accelerate file-io performance. In *HPCC*, pages 718–731, 2007.
 - [24] Vasily Tarasov. cgroups: block: cfq: I/o bandwidth controlling subsystem for cgroups based on cfq. <http://lkml.org/lkml/2008/3/21/519>.
 - [25] Ryo Tsuruta. dm-band: The i/o bandwidth controller. <http://lkml.org/lkml/2008/1/23/106>.
 - [26] Satoshi UCHIDA. Yet another i/o bandwidth controlling subsystem for cgroups based on cfq. <http://lkml.org/lkml/2008/4/3/45>.
 - [27] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. Argon: Performance insulation for shared storage servers. In *5th USENIX Conference on File and Storage Technologies*, pages 61–76, 2007.
 - [28] C. Waldspurger and W. Weihl. Stride scheduling: Deterministic proportional-share resource management, 1995.

Linux Capabilities: making them work

Serge E. Hallyn

IBM LTC

serue@us.ibm.com

Andrew G. Morgan

Google Inc.

agm@google.com

Abstract

Linux capabilities have been partially implemented for many years, and in their incomplete state have been nearly unusable. In light of recent kernel developments, including VFS support and per-process support for bounding-set and secure-bits, capabilities have finally come of age. In this paper we demonstrate, with examples, how capabilities enhance the security of the modern Linux system.

1 Introduction

Linux helps users manage their data, and a single Linux system simultaneously manages the data of multiple users. Within the system, a particular user's property is generally contained in *files* which are annotated with a numerical ownership user-identifier (UID). Linux also manages and abstracts the computer hardware, offering programs an environment in which to execute. Part of this abstraction enforces data ownership. In order to honor the ownership of such data, Linux adheres to context-specific rules limiting how programs can manipulate, via *system-calls*, a specific user's data (the context in this case being the value of attributes like the user's UID).

To start running programs on a Linux system, an applicant-user generally leverages a program (such as *login*, *sshd*, or *gdm*) to authenticate their identity to the system and create a working context for them to invoke other programs that access their data. Such *login* programs are exceptionally special, insofar as they have the ability to change the user context (set the *current* UID). Changing user context is clearly a special operation, since if it were not then programs run from the context of any user could trivially be leveraged to create a different user's context and manipulate data belonging to that other user. The special property of these applications is commonly known as *privilege*, and this paper

concerns a newly completed mechanism for managing privilege within the Linux operating system.

Programs, in the context of authenticated users, can create data with access controls associated with them: create a file that anyone can read; create a file that only the creator can read or modify; etc. These forms of protection are known as Discretionary Access Control (DAC), and with more recent Linux extensions such as Access Control Lists (ACLs) can be quite elaborate [1]. The protection of such data is at the discretion of the owner of these files. Other mechanisms, such as Mandatory Access Control (MAC), enforce a system policy that restricts the ways in which users can share their data. Linux, via the Linux Security Module (LSM) [2] programming abstraction, natively supports simple MAC [3] and the more modern *type-enforcement* model [4, 5]. All of these mechanisms follow a tradition [6] of attempting to add real security to Linux and UNIX [7].

Managing a Linux system in the real world requires levels of reliability (continuity of service for multiple simultaneous users and uses) that must anticipate future problems: the need for data backups; configuration changes and upgrades to failing/obsolete hardware etc. There is also a recurrent need to work around and correct urgent issues: users accidentally instructing programs to delete files, users being removed from the system (and their data being archived or redistributed to other users); etc. These requirements, in addition to the need to *login* users (discussed above), lead to the fact that any system must provide the ability to override DAC and MAC security protections to "get things done." Viable systems need a *privilege* model.

The classic UNIX model for wielding privilege is to assign a special UID the right to do anything. Programs running in the context of this *super-user* are not bound by the normal DAC/MAC rules. They can read, modify, and change any user's data. In UNIX, UID=0 is the special context assigned to this administrative identity. To give this entity a more human touch, this user is also

known as `root`. What this means for programs is that, when they run in the context of `root`'s UID, system-calls can do special (privileged) things. The converse of this is also significant: when they run in this context, programs can't help breaking the normal DAC/MAC rules, potentially causing unintended damage to the system. For example, executing: `rm -rf /`, as `root` can have spectacularly bad consequences to a system. However, running this command as a *normal user* (in this paper, we'll call such a user: `luser`) results in a prompt error and no negative effects. This `luser` doesn't have a right to delete anything in the base directory of the filesystem.

Unprivileged users, however, must always perform some tasks which do require privilege. For instance, our `luser` must be able to change his password. That being said, the system must prevent `luser` from being able to read or change passwords for other users. Users execute programs that act for them, and a program exists to change passwords: `passwd`. This program must be invoked from the context of the `luser` but operate with sufficient privilege to manipulate the shared-system's password file (`/etc/shadow`). To this end, Linux executable files can have a special attribute *setuid*-bit set, meaning that the program can execute with the *effective* UID (EUID¹) of the user owning the program file. If the *setuid* `passwd` program file's owner is `root`,² then independent of the user context in which the program is launched, it will execute with the effective context of the super-user. That is, a program such as `passwd` will not be bound by any of the DAC/MAC rules that constrain other *regular* programs.

2 The Linux capability model

While the simple UNIX privilege mechanism has more or less sufficed for decades, it has long been observed that it has a significant shortcoming: that programs that require only some privilege must in fact run with full privilege. The dangers of such a lack of flexibility are well known, as they ensure that programming errors in privileged programs can be leveraged by hostile users

¹Details about effective, saved, and filesystem UIDs, groups, and group membership have been omitted from this discussion. That being said, through complexity, they have greatly added to the usability of the system.

²In practice, a shared need to edit a protected file like this can be achieved with ACLs—requiring a shadow-UID or group for example.

to lead to full system compromise [8]. Such compromises can be mitigated through the use of MAC, but at some fundamental level any privileged access to the hardware underpinning an operating system can violate even MAC rules, and bogging MAC implementations down with details about `root` privilege separation only increases policy complexity. In the real world, administrative access to override normal access control mechanisms is a necessary feature.

Over the years, the proponents of a more secure UNIX [7] explored various alternatives to the concept of an all powerful `root` user. An aborted attempt was even made to unify these enhancements into a single standard [9]. The downward trajectory in the mid to late 1990's of the closed-source vendor-constrained rival commercial UNIX implementations mired, and eventually halted, the ratification of this *standard*. However, not entirely disconnected from this slowdown was the rapid and perhaps inevitable rise of Linux—a truly open (free) system in the original spirit of the UNIX tradition. These modern ideas of incremental enhancements to the UNIX security model have now found a home in Linux [1, 10, 3, 11].

The proposed privilege model [9] introduced a separation of root privilege into a set of *capabilities*. These capabilities break the super-user's privilege into a set of meaningfully separable privileges [7]. In Linux, for instance, the ability to switch UIDs is enabled by `CAP_SETUID` while the ability to change the ownership of an object is enabled by `CAP_CHOWN`.

A key insight is the observation that programs, not people, exercise privilege. That is, everything done in a computer is via agents—programs—and only if these programs know what to do with privilege can they be trusted to wield it. The `UID=0` model of privilege makes privilege a feature of the super-user context, and means that it is arbitrary which programs can do privileged things. Capabilities, however, limit which programs can wield any privilege to only those programs marked with filesystem-capabilities. This feature is especially important in the aftermath of a hostile user exploiting a flaw in a *setuid-root* program to gain super-user context in the system.

This paper describes how to use the Linux implementation of these capabilities. As we will show, while support of legacy software requires that we sometimes maintain a privileged root user, the full implementation

of Linux capabilities enables one to box-in certain subsystems in such a way that the UID=0 context becomes that of an unprivileged user. As legacy software is updated to be capability-aware, a fully root-less system becomes a very real possibility [12].

2.1 Capability rules

Processes and files each carry three capability sets. The process *effective* set contains those capabilities which will be applied to any privilege checks. The *permitted* set contains those capabilities which the task may move, via the `capset()` system call, into its *effective* set. The *effective* set is never a superset of the *permitted* set. The *inheritable* set is used in the calculation of capability sets at file execution time.

Capabilities are first established, at program execution time, according to the following formulas:

$$pI' = pI \quad (1)$$

$$pP' = (X \& fP) | (pI \& fI) \quad (2)$$

$$pE' = fE ? pP' : \emptyset. \quad (3)$$

Here pI , pE , and pP are the process' inheritable, effective, and permitted capability sets (respectively) before `exec()`. Post-`exec()`, the process capabilities sets become pI' , pE' , and pP' . The capability sets for the file being executed are fI , fE , and fP . Equation 1 shows the task retains its pre-`exec()` inheritable set. Equation 2 shows the file inheritable and process inheritable sets are and'ed together to form a context-dependent component of the new process permitted set. The file inheritable set, fI , is sometimes referred to as the file's *optional* set because the program will only acquire capabilities from it if the invoking user context includes them in pI . By optional, we mean the program can gracefully adjust to the corresponding privileges being available or not. The file permitted set, fP , is also called the *forced* set because capabilities in that set will be in the process' new permitted set whether it previously had them in any capability sets or not (subject to masking with X). In Equation 3, the file effective capability set is interpreted as a boolean. If fE (also called the *legacy* bit) is set, then the process' new effective set is equal to the new permitted set. If unset, then pE' is empty when the `exec()` d program starts executing.

The remaining object in these rules, X , has, until recently, been an unwieldy *system-wide* capability bounding set. However, it has now become the per-process

capability bounding set. X is inherited without modification at `fork()` from the parent task. A process can remove capabilities from its own X so long as its effective set has `CAP_SETPCAP`. A task can never add capabilities to its X . However, note that a task can gain capabilities in pP' which are not in X , so long as they are both in pI and fI . The bounding set will be further discussed in Section 3.3.

When a new process is created, via `fork()`, its capability sets are the same as its parent's. A system call, `capset()`, can be used by a process to modify its three capability sets: pI , pP and pE . As can be seen in Equation 1, the inheritable set pI remains unchanged across file execution. Indeed it is only changed when the running process uses the system call to modify its contents. Unless pE contains `CAP_SETPCAP`, Linux will only allow a process to *add* a capabilities to pI that are present in pP . No special privilege is required to *remove* capabilities from pI . The only change to the permitted set, pP , that a process can make is to drop raised capabilities. The effective set is calculated at file execution, and immediately after `exec()` will be either equal to the permitted set or will be empty. Via `capset()` the process can modify its effective set, pE , but Linux requires that it is never a superset of the contents of the process' permitted set, pP .

Most software and distributions available currently depend on the notion of a fully privileged `root` user. Linux still supports this behavior in what we call *legacy-fixup* mode, which is actually the default. Legacy-fixup mode acts outwardly in a manner consistent with there being a `root` user, but implements super-user privilege with capabilities, and tracks UID-changes to *fixup* the prevailing capability sets. This behavior allows a `root` user to execute any file with privilege, and an ordinary user to execute a setuid-root file with privilege. When active, legacy-fixup mode force-fills the file capability sets for every setuid-root file and every file executed by the `root` user. By faking a full fP and full fI we turn a setuid-root file or a file executed by the `root` user into a file carrying privilege. This may appear distasteful, but the desire to support legacy software while only implementing one privilege model within the kernel requires it. As we will show in Section 4 legacy-fixup mode can be turned off when user-space needs no privilege or supports pure privilege through capabilities.

In the absence of VFS support for capabilities, a number of extensions to the basic capability model [9] were

introduced into the kernel: an unwieldy (global, asynchronous,³ and system crippling) bounding set [13]; the unwieldy (asynchronous and questionable) remote bestowal of capabilities by one process on another;⁴ the unwieldy (global, asynchronous, and system crippling) secure-bits;⁵ and the more moderately scoped `prctl(PR_SET_KEEPcaps)` extension.

All but the last of these have recently been made viable through limiting their scope to the current process, becoming synchronous features in the Linux capability model. The `prctl(PR_SET_KEEPcaps)` extension of legacy-fixup mode, which can be used as a VFS-free method for giving capabilities to otherwise unprivileged processes, remains so. When switching from the privileged `root` user to a non-`root` user, the task's permitted and effective capability sets are cleared.⁶ But, using `prctl(PR_SET_KEEPcaps)`, a task can request keeping its capabilities across the next `setuid()` system call. This makes it possible for a capability-aware program started with `root` privilege to reach a state where it runs locked in a non-`root` user context with partial privilege. As we discuss in Section 4, while legacy-fixup remains the default operating mode of the kernel, each of these *legacy* features can be disabled on a per-process basis to create process-trees in which legacy-fixup is neither available nor, indeed, needed.

3 Worked Examples

In this section we provide some explicit examples for how to use capabilities. The examples show how traditional setuid-root solutions can be emulated, and also what is newly possible with capabilities.

³Asynchronicity with respect to security context means that a task's security context can be changed by another task without the victim's awareness.

⁴The ability for one process to asynchronously change, without notification, the capabilities of another process, via the *hijacked* `CAP_SETPCAP` capability, was so dangerous to system integrity that it has been disabled by default since its inception in the kernel. The addition of VFS support disables this feature and restores `CAP_SETPCAP` to its intended use as documented in this paper (see Section 3.1).

⁵Securebits have been implemented in the kernel for many years, but have also been cut off from being available—without any API/ABI for manipulating them for almost as long.

⁶The actual semantics of legacy-fixup are more complicated.

3.1 Minimum privilege

In this example we consider an application, `ping`, that one might not even realize requires privilege to work. If you examine the regular file attributes of a non-capability attributed `ping` binary, you will see something like this:

```
$ ls -l /bin/ping
-rwsr-xr-x 1 root root 36568 May 2 2007 /bin/ping
$ ./bin/ping -q -c1 localhost
PING localhost.localdomain (127.0.0.1) 56(84)
bytes of data.
--- localhost.localdomain ping statistics ---
1 packets transmitted, 1 received, 0% packet loss,
time 0ms
rtt min/avg/max/mdev = 0.027/0.027/0.027/0.000 ms,
pipe 2
$
```

The `s` bit of the file's mode is the familiar setuid-executable bit. If we copy the file as an unprivileged user (`luser`) it loses its privilege and ceases to work:

```
$ cp /bin/ping .
$ ls -l ping
-rwxr-xr-x 1 luser luser 36568 Mar 26 17:54 ping
$ ./ping localhost
ping: icmp open socket: Operation not permitted
$
```

Running this same program as `root` will make it work again:

```
# ./ping -q -c1 localhost
PING localhost.localdomain (127.0.0.1) 56(84)
bytes of data.
--- localhost.localdomain ping statistics ---
1 packets transmitted, 1 received, 0% packet loss,
time 0ms
rtt min/avg/max/mdev = 0.027/0.027/0.027/0.000 ms,
pipe 2
#
```

In short, `ping` requires privilege to write the specially crafted network packets that are used to probe the network.

Within the Linux kernel there is a check to see whether this application is capable (`CAP_NET_RAW`), which means `cap_effective(pE)` for the current process includes `CAP_NET_RAW`. By default, `root` gets all effective capabilities, so it defaults to having more-than-enough privilege to successfully use `ping`. Similarly, when setuid-root, the `/bin/ping` version is also overly privileged. If some attacker were to discover

a new buffer-overflow [14] or more subtle bug in the ping application, then they might be able to exploit it to invoke a shell with root privilege.

Filesystem capability support adds the ability to bestow *just-enough* privilege to the ping application. To emulate just enough of its legacy privilege, one can use the utilities from libcap [10] to do as follows:

```
# /sbin/setcap cap_net_raw=ep ./ping
# /sbin/getcap ./ping
./ping = cap_net_raw+ep
```

What this does is add a permitted capability for CAP_NET_RAW and also sets the *legacy* effective bit, *fE*, to automatically raise this *effective* bit in the ping process (*pE*) at the time it is invoked:

```
$ ./ping -q -c1 localhost
PING localhost.localdomain (127.0.0.1) 56(84) bytes of data.
--- localhost.localdomain ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.093/0.093/0.093/0.000
ms, pipe 2
$
```

Unlike the setuid-root version, the binary ping is not bestowed with any privilege to modify a file that is not owned by the calling user, or to insert a kernel module, etc. That is, there is no direct way for some malicious user to subvert this *privileged* version of ping to do anything privileged other than craft a malicious network packet.⁷

So far, we have explained how to replace the setuid-root privilege of ping with file capabilities. This is for an unmodified version of ping. It is also possible to lock ping down further by modifying the ping source code to use capabilities explicitly. The key change from the administrator's perspective is to set ping's capabilities as follows:

```
# /sbin/setcap cap_net_raw=p ./ping
```

That is, no *legacy* effective bit, and no enabled privilege (just the potential for it) at exec() time. Within the ping application one can, using the API provided by libcap [10], prepare to manipulate the application's privilege by crafting three capability sets as follows:

⁷Of course, it may prove possible to leverage a rogue network packet to cause system damage, but only indirectly—by subverting some other privileged program.

```
/* the one cap ping needs */
const cap_value_t cap_vector[1] =
{ CAP_NET_RAW };
cap_t privilege_dropped = cap_init();
cap_t privilege_off = cap_dup(privilege_dropped);
cap_set_flag(privilege_off, CAP_PERMITTED, 1,
    cap_vector, CAP_SET);
cap_t privilege_on = cap_dup(privilege_off);
cap_set_flag(privilege_on, CAP_EFFECTIVE, 1,
    cap_vector, CAP_SET);
```

Then, as needed, the capability sets can be used with the following three commands:

```
/* activate: cap_net_raw=ep */
if (cap_set_proc(privilege_on) != 0)
    abort("unable to enable privilege");
/* ...do privileged operation... */
/* suspend: cap_net_raw=p */
if (cap_set_proc(privilege_off) != 0)
    abort("unable to suspend privilege");
/* ...when app has no further need of privilege */
if (cap_set_proc(privilege_dropped) != 0)
    abort("failed to irrevocably drop privilege");
```

Also, remember to clean up allocated memory, using cap_free(privilege_on) etc., once the capability sets are no longer needed by the application. These code snippets can be adapted for other applications, as appropriate.

In these code snippets, the inheritable capability set is forced to become empty. This is appropriate and suffices for applications that do not expect to execute any files requiring privilege, or which expect any privilege in subsequently executed programs to come from the file's forced set (*fP*). For an application like a user shell, the above snippets might be changed so as to preserve *pI*. This can be achieved by replacing the use of cap_init(), above, with the following sequence:

```
cap_t privilege_dropped = cap_get_proc();
cap_clear_flag(privilege_dropped, CAP_EFFECTIVE);
cap_clear_flag(privilege_dropped, CAP_PERMITTED);
```

A login process, in turn, would likely be authorized with CAP_SETPCAP, allowing it to actually fill *pI* further with specific capabilities assigned to the user being logged-in. Section 3.2 will begin to show how to use inherited privilege.

3.2 Inherited privilege

There are some programs that don't have privilege, per se, but wield it in certain circumstances: for example,

when they are invoked by `root`. One such application is `/bin/rm`. When invoked by `root`, `/bin/rm` can delete a file owned by *anyone*. Clearly, forcing privilege with the file permitted bits, as we did in the previous section, would give any invoker of `/bin/rm` such abilities and not represent an increase in security at all! To emulate `root`-is-special semantics for certain users, we employ the *inheritable* capability set (pI).

The basic setup for leveraging inheritable capabilities is to add file capabilities to `/bin/rm` as follows (in this case, we'll add the capability to the official `rm` binary):

```
# /sbin/setcap cap_dac_override=ei /bin/rm
```

Reviewing the capability formula, Equation 1, one can see that a process inherits its *inheritable* capabilities, pI , directly from its parent. In order to use inheritable capabilities, therefore, a process has to first acquire them. The `libcap` package provides a utility for reading the capabilities of a process:

```
$ /sbin/getpcaps 1
Capabilities for '1': =ep cap_setpcap-e
$
```

This says that `init`, the top of the process tree, and ancestor to all processes in a system, does not have any *inheritable* capabilities. That is, by default, no process will passively obtain any inheritable capabilities. However, `init` and its many privileged descendants, such as `login` and `su`, do have access to capabilities through their *permitted* sets, pP . To add a capability to its inheritable set, a process must either have that capability present in its *permitted* set, or be capable (`CAP_SETPCAP`)—have the single capability, `CAP_SETPCAP` in its *effective* set, pE . Leveraging this feature, the `libcap` package [10] contains two convenient methods to introduce inheritable capabilities to a process-tree: a simple wrapper program, `capsh`, and a PAM [15] module, `pam_cap.so`.

The `capsh` command is intended to provide a convenient command-line wrapper for testing and exploring capability use. It is able to alter and display capabilities of the current process and can be used to explore the nuances of the present example. We shall use `capsh` in Section 3.3. Here we will describe how to make use of the `pam_cap.so` PAM module.

The PAM module `pam_cap.so`, as directed by a local configuration file, sets inheritable capabilities based on the user being authenticated. In our example, we give a student administrator (`studadmin`) the ability to remove files owned by others. We set up a test file and a configuration file (as `root`) with the following commands:

```
# cat > /etc/security/su-caps.conf <<EOT
cap_dac_override    studadmin
none                *
EOT
# touch /etc/empty.file

# ls -l /etc/{empty.file,security/su-caps.conf}
-rw-r--r-- 1 root root 0 Mar 30 14:00 /etc/empty.file
-rw-r--r-- 1 root root 52 Mar 30 13:59 /etc/security/su-caps.conf
```

We then put the following line at the very beginning of the `/etc/pam.d/su` file:

```
auth optional pam_cap.so \
        config=/etc/security/su-caps.conf
```

Now anyone able to authenticate via `su` `studadmin` will become the regular user `studadmin` with the enhancement that they have an inheritable capability, `CAP_DAC_OVERRIDE`:

```
$ whoami
luser
$ su studadmin
Password:
$ whoami
studadmin
$ /sbin/getpcaps $$
Capabilities for '11180': = cap_dac_override+i
$
```

Having obtained this inheritable capability, `studadmin` can try it out by deleting a `root`-owned file:

```
$ rm /etc/empty.file
rm: remove write-protected regular
file '/etc/empty.file'? y
$ ls -l /etc/empty.file
ls: /etc/empty.file: No such file or directory
```

In passing, we note that when the `rm` command was prompting for the `y` response, it was possible to find the PID for this process and, from a separate terminal:

```
$ /sbin/getpcaps 15310
Capabilities for '15310': = cap_dac_override+eip
$
```

That is, observe that the formula Equation 2 did its work to raise the permitted, pP , capability for `rm`, and the legacy fE bit caused it to become effective for the process at `exec()` time.

It is instructive to try to remove something else using another program. For example, using `unlink`:

```
$ unlink /etc/security/su-caps.conf
unlink: cannot unlink
'/etc/security/su-caps.conf': Permission denied
$
```

Because this `unlink` application has no filesystem capabilities, $fI = fP = fE = 0$, despite the prevailing inheritable capability in pI , `unlink` cannot wield any privilege. A key feature of the capability support is that only applications bearing filesystem capabilities can wield any system privilege.

In this example, we have demonstrated how legacy applications can be used to exercise privilege through inheritable capabilities. As was the case in the previous example, legacy applications can be modified at the source code level, to manipulate capabilities natively via the API provided by `libcap`. Such a modified application would not have its legacy capability raised ($fE = 0$). The code samples from the previous section are equally applicable to situations in which an application obtains its capabilities from its inheritable set, we do not repeat them here.

3.3 Bounding privilege

The capability bounding set is a per-process mask limiting the capabilities that a process can receive through the file permitted set. In Equation 2, the bounding set is X . The bounding set also limits the capabilities which a process can add to its pI , though it does not automatically cause the limited capabilities to be removed from a task which already has them in pI . When originally introduced [13], the capability bounding set was a system-wide setting applying to all processes. An example intended usage would have been to prevent any further kernel modules from being loaded by removing `CAP_SYS_MODULE` from the bounding set.

Recently, the bounding set became a per-process attribute. At `fork()`, a child receives a copy of its parent's bounding set. A process can remove capabilities from its bounding set so long as it has the `CAP_`

`SETPCAP` capability [16]. Neither a process itself, nor any of its `fork()`d children, can ever add capabilities back into its bounding set. The specific use case motivating making the bounding set per-process was to permanently remove privilege from containers[17] or jails[18]. For instance, it might be desirable to create a container unable to access certain devices. With per-process capability bounding sets, this becomes possible by providing it with a `/dev` that does not contain these devices and removing `CAP_MKNOD` from its capabilities.⁸

The reader will note, in Equation 2, that X masks only fP . In other words, a process' permitted set can receive capabilities which are not in its bounding set, so long as the capabilities are present in both fI and pI . Ordinarily this means that a process creating a “secure container” by removing some capabilities should take care to remove the unwanted capabilities from both its bounding and inheritable sets. Thereafter they cannot be added back to pI . However, there may be cases where keeping the bits in the inheritable and not the bounding set is in fact desirable. Perhaps it is known and trusted that the capability will only be in fI for trusted programs, so any process in the container executing those programs can be trusted with the privilege. Or, the initial container task may take care to spawn only one task with the capability in its pI , then drop the capability from its own pI before continuing. In this way the initial task in a container without `CAP_MKNOD`, rather than mounting a static `/dev`, could keep `CAP_MKNOD` in pI while running a trusted copy of `udev`, from outside the container, which has `CAP_MKNOD` in its fI . The `udev` process becomes the only task capable of creating devices, allowing it to fill the container's `/dev`.

Here is an example of dropping `CAP_NET_RAW` from the bounding set, using `capsh` [10]. So doing, we can cause `ping` to fail to work as follows:

```
# id -nu
root
# /sbin/getcap ./ping
./ping = cap_net_raw+ep
# /sbin/capsh --drop=cap_net_raw \
--uid=$(id -u luser) --
$ id -nu
luser
$ ./ping -q -c1 localhost
ping: icmp open socket: Operation not permitted
$ /bin/ping -q -c1 localhost
ping: icmp open socket: Operation not permitted
```

⁸This requires a (hopefully) upcoming patch causing mounts by a process which is not capable (`CAP_MKNOD`) to be `MNT_NODEV`.

The `--drop=cap_net_raw` argument to `/sbin/capsh` causes the wrapper program to drop `CAP_NET_RAW` from the bounding set of the subsequently invoked bash shell. In this process tree, we are unable to gain enough privilege to successfully run `ping`. That is, both our capability-attributed version, and the setuid-root version attempt to force the needed privilege, but the prevailing bounding set, X , suppresses it at execution time.

In an environment in which the bounding set suppresses one or more capabilities, it is still possible for a process to run with these privileges. This is achieved via use of the inheritable set:

```
# id -nu
root
# /sbin/setcap cap_net_raw=eip ./ping
# /sbin/capsh --{inh,drop}=cap_net_raw \
--uid=$(id -u luser) --
$ ./ping -q -c1 localhost
PING localhost.localdomain (127.0.0.1) 56(84)
bytes of data.
--- localhost.localdomain ping statistics ---
1 packets transmitted, 1 received, 0% packet loss,
time 0ms
rtt min/avg/max/mdev = 0.037/0.037/0.037/0.000 ms,
pipe 2
```

That is, as per Equation 2, the bounding set, X , does not interfere with the $pI \& fI$ component to pP' .

There are some subtleties associated with bounding set manipulation that are worth pointing out here.

The first is that the bounding set does limit what capabilities can be *added* to a process' inheritable set, pI . For example, as `root`:

```
# /sbin/capsh --drop=cap_net_raw --inh=cap_net_raw
Unable to set inheritable capabilities:
Operation not permitted
#
```

This fails because, by the time we attempt to add an inheritable capability in the working process, we have already removed it from the bounding set. The kernel is just enforcing the rule that once pI and X are *both* without a particular capability, it is irrevocably suppressed.

The second subtlety is a warning, and relates to a bug first highlighted in association with the `sendmail` program [16]. Namely, for *legacy* programs that require forced capabilities to work correctly, you can cause them to fail in an unsafe way by selectively denying them privilege.

When a legacy program makes the (common) assumption that an operation must work because the program is known to be operating with privilege (a previous privileged operation has succeeded), with capabilities, it can be fooled into thinking it is operating in one privilege level when it actually isn't. Since privilege is now represented by independent capabilities, one can leverage the bounding set to deny a single capability that is only needed later at a more vulnerable time in the program's execution.

The `sendmail` issue was in a context where the dropping of an inheritable capability by an unprivileged parent of the setuid-root sendmail caused sendmail to launch a program as `root` when it thought it was running in the context of the `luser`. The significance of the bug was that an unprivileged `luser` could exploit it.

The kernel was fixed to make this particular situation not occur. However, the bounding set actually recreates a similar situation, and while sendmail has since been fixed to protect it from this problem, many other legacy setuid-root applications are expected to suffer from this same issue. Non-legacy applications are not susceptible to this subtlety because they can leverage the `libcap` API to look-before-they-leap and check if they have the needed privilege explicitly at runtime.

The significant difference between the old problematic situation and this present case, is that to exploit this issue you need to be able to alter the bounding set and that, itself, requires privilege. That being said, this subtlety remains. Be careful, when using the bounding set, to avoid leveraging it suppress privilege in general when it is more appropriate to supply *optional* capabilities as needed via the inheritable set. Caveat emptor!

3.4 No privilege at all

In general, unprivileged users need to run privileged applications. However, sometimes it may be desirable to confine a process, and any of its children, ensuring that it can never obtain privilege. In a traditional UNIX system this would not be possible, as executing a setuid-root program would always raise its privilege.

To completely remove privilege from a process in a capability-enabled Linux system, we must make sure that both sides of Equation 2 are, and always will be, empty. We can suppress fP by emptying the bounding

set, X . Since a capability can never be added back into X , this is irrevocable. Next, we can suppress the second half of the equation by emptying pI using `capset`. Thereafter the process cannot add any bits not in X (which is empty), back into pI . The legacy compatibility mode refills fI whenever a setuid root binary is executed, but we can see in Equation 2 that capabilities must be in both fI and pI to appear in pP' . Now, regardless of what the process may execute, neither the process nor any of its children will ever be able to regain privilege.

4 Future changes

At the conclusion of Section 2.1 we observed that the capability rules are perverted for files run by the super-user. When the super-user executes a file, or when any user executes a setuid-root file, the file's capability sets are filled. Since historically Linux had no support for file capabilities, and since without file capabilities a process can never wield privilege in a pure capability system, this *hack* was unfortunate but necessary. Now that the kernel supports file capabilities, it is only userspace which must catch up. As applications become capability-aware, it will be desirable to remove the legacy `root`-as-super-user support for those applications. While infrastructure to support disabling it system-wide has been present for as long as the `root`-as-super-user hack has existed, support to do this for application sets has only recently been accepted into the experimental `-mm` tree [19]. It is expected to be adopted in the main Linux tree [20], and may have done so by the time of publication.

With the per-process *securebits*, the root user exception can be “turned off” for capability-aware applications by setting the `SECURE_NOROOT` and `SECURE_NO_SETUID_FIXUP` flags using `prctl()`. These are per-process flags, so that a system can simultaneously support legacy software and capability-aware software. In order to lock capability-aware software into the more secure state in a such a way that an attacker cannot revert it, both bits can be locked by also setting `SECURE_NOROOT_LOCKED` and `SECURE_NO_SETUID_FIXUP_LOCKED`.

To nail the residue of problematic partial privilege for legacy applications, discussed in Section 3.3, we are considering adding a requirement that any legacy application which is made privileged with $fE \neq 0$ must

execute with $pP' \geq fP$. That is, if the bounding set, X , suppresses a *forced* capability ($fP < fP \& X$), and the inheritable sets ($pI \& fI$) do not make up for its suppression (see Equation 2), `exec()` will fail with `errno = EPRIV`. This change will enforce what is presently only a convention that legacy applications should run with all of their *forced* (fP) capabilities raised, or are not safe to run at all.

5 Conclusion

The intent of this paper has been to demonstrate that the Linux *capability* implementation, with VFS support, is a viable privilege mechanism for the Linux kernel. With examples, we have shown how these capabilities can and should be used. What remains is for user-space applications to start using them.

That being said, privilege is not the only use of the `root` identity. There are many files, such as are to be found in `/proc/` and `/etc/`, that are owned by `root`. Even without super-user privilege, a process running in the context of an impotent `root` user, can still do a large amount of damage to a system by altering these files. Here, DAC and MAC based security will continue to be important in securing your Linux system.

Acknowledgments

It is our pleasure to thank Chris Friedhoff and KaiGai Kohei for collaboration; Chris Wright, James Morris, and Stephen Smalley for advice and careful scrutiny; Olaf Dietsche and Nicholas Simmonds for posting alternative implementations of file capabilities; and Andrew Morton for patiently sponsoring the recent capability enhancements to the kernel. AGM would additionally like to thank Alexander Kjeldaaas, Aleph1, Roland Buresund, Andrew Main and Ted Ts'o for early `libcap` and kernel work; the anonymous chap that let AGM read a copy of a POSIX draft sometime around 1998; and Casey Schaufler for persuading the POSIX committee to release an electronic copy of the last POSIX.1e draft [9]. Finally, we'd like to thank Heather A Crogale for her insightful comments on an early draft of this paper.

References

- [1] Andreas Grünbacher. POSIX Access Control Lists on Linux. USENIX Annual Technical Conference, San Antonio, Texas, June 2003.

- [2] Chris Wright et al. Linux Security Modules: General Security Support for the Linux Kernel. 11th USENIX Security Symposium, 2002.
- [3] Casey Schaufler. The Simplified Mandatory Access Control Kernel. linux.conf.au, 2008.
- [4] W.E. Boebert and R.Y. Kain. A practical alternative to hierarchical integrity policies. In Proceedings of the Eighth National Computer Security Conference, 1985.
- [5] Peter Loscocco, Stephen Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference, June 2001.
- [6] The Trusted Computer System Evaluation Criteria (*the Orange Book*). <http://www.fas.org/irp/nsa/rainbow/std001.htm>.
- [7] Samuel Samalin, Secure UNIX, McGraw-Hill, 1996.
- [8] Whole system compromises are regularly documented on websites such as *Bugtraq*: <http://www.securityfocus.com/archive/1>.
- [9] The last POSIX.1e draft describing capabilities: <http://wt.xpilot.org/publications/posix.1e/download.html>.
- [10] The capability user-space tools and library: <http://www.kernel.org/pub/linux/libs/security/linux-privs/libcap2/>.
- [11] See for example: <http://people.redhat.com/sgrubbs/audit/>.
- [12] Advocates for transforming systems to be capability based such as: <http://www.friedhoff.org/fscaps.html>.
- [13] Introduction of the *global* bounding set to Linux, <http://lwn.net/1999/1202/kernel.php3>.
- [14] An example of a buffer-overflow in the ping binary (not exploitable in this case): <http://www.securityfocus.com/bid/1813>.
- [15] Linux-PAM, <http://www.kernel.org/pub/linux/libs/pam>; Kenneth Geisshirt, Pluggable Authentication Modules: The Definitive Guide to PAM for Linux SysAdmins and C Developers. Packt Publishing. 2006.
- [16] Discussion of the unfortunately named *sendmail-capabilities-bug*: <http://userweb.kernel.org/~morgan/sendmail-capabilities-war-story.html>.
- [17] Linux Containers, <http://lxc.sourceforge.net/>.
- [18] Poul-Henning Kamp, Robert N.M. Watson: Jails: Confining the omnipotent root. Proceedings of second international SANE conference, May 2000.
- [19] Andrew Morton's kernel patch series: <http://www.kernel.org/patchtypes/mm.html>
- [20] Linus' kernel tree: <http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=summary>

Issues in Linux Mirroring: Or, BitTorrent Considered Harmful

John Hawley

3Leaf Systems

warthog19@eaglescrag.net

Abstract

The Linux community has risen to the challenge of sharing distributions by developing an ad-hoc worldwide collaborative mirroring infrastructure capable of withstanding some of the heaviest network traffic imaginable. It is already capable of moving tens of terabytes a day and it is continuing to grow and expand to meet the needs of a demanding user base. However this ad-hoc infrastructure is not without its faults. Distribution maintainers have complicated, non-intuitive websites to direct users to downloads. Things are made worse by a lack of communication amongst the major releases throughout the year, and a user base that is not always correct in its requests and demands that it puts upon the system.

As the infrastructure grows, the administrators seek out new ways to help manage the stress on everyone involved. BitTorrent has been heralded as one such technology; however, its claims of being better, faster, and more manageable seem to fall short. BitTorrent itself seems to have an upper limit to its capacity that does not match the existing infrastructure. In fact, it has significant downsides to the maintainers, the mirrors, and the users, making it unsuitable as a large-scale primary distribution mechanism.

1 Distributions

Distribution maintainers are at the very core of the Linux mirroring infrastructure. They are creators of the data that the mirrors will be providing to users, and wield a significant amount of control over the experience that both mirrors and users have when downloading. Mirror maintainers and end-users have different issues that distribution maintainers must do their best to respect and work with.

Users, however, are by far the biggest challenge facing both distribution maintainers and their mirrors. Users

are an un-relenting mob, capable of bringing some of the largest, and fastest, machines to a crawl. The effect users can have on mirrors during a major release is not dissimilar to a distributed denial of service attack, with the added affect that each user is fighting to use as much bandwidth as can be obtained.

1.1 Keeping it simple

Linux users already face many problems with Linux—getting it shouldn't be one of them. Currently, users trying to download a distribution are asked a multitude of questions, many of which can be unclear and not well understood by the user. This complicates the download process, making it difficult for users to make good choices for their needs. Options should be kept to a minimum by default, as the more options exposed to a user, the more potential for confusion. Where possible, choices should be guessed at for the user, such as choosing a download-mirror based on the geographic location of the user's IP address. A clear and simple mechanism to override the default should be available, so that users can correct or alter the assumptions if necessary.

For example, when users wish to download a distribution, they should be directed by the distribution's website to a download page. The page should default to downloading the latest version, indicate the mirror currently selected, and display "download" icons for each processor architecture. The mirror should be shown in a drop-down list so it can be changed easily. If multiple formats exist (CDs and DVDs, for example), clear icons listing the processor architecture and the format should be present. In the case of a single-file download (like DVD ISOs), upon clicking the icon, the download should just commence without further user action. In the case of multiple file downloads (like CD ISOs), users should then be directed directly to the directory on the mirror server that has the CD ISO images in it so that they may select and download each file on their own.

Archives should be linked to on the page, and a similar strategy to the current version should be used. A full listing of mirrors and their contents should be linked on a separate page, should a user wish to manually browse.

This particular strategy encompasses a number of simplifications to users, and gives distributions much more flexible control over the distribution process. For starters, having a centrally controlled download page gives the distribution a common and simple way to direct users to resources—in this case, their ISO images. It also gives the distribution the ability to attempt to spread the load amongst mirrors, by having a mirror declare its download speed, the country it is located in and what countries it serves a distribution to; the distribution site may intelligently choose, using something like geographic IP lookup, where a user is attempting to download from and provide a mirror that serves that country with sufficient capacity. It also gives the users a very clear and obvious path to get the data, by use of clear icons defining their available choices and providing a simple means of getting the data.

1.2 Mirrors are your friends, treat them with care

There is a growing trend to place more and more requirements on mirrors. They need to mirror more data, requiring more disk space. They may be asked to verify this data, both by internal scripts and by allowing external crawlers to browse their filesystems. And, of course, they need to be able to handle an ever-growing user base. Each new requirement slowly adds a straw to their backs. At some point, even the most powerful mirrors must ask the question—is this too much?

Thankfully, there are ways that this load can be managed so that mirrors don't become overburdened—for instance, limiting the amount of data that needs to be mirrored, having a controlled schedule for crawlers, and spreading out the dates when distributions are released.

1.2.1 Diet Time: Mirrors choice in Legacy data

With new releases coming out regularly, space on a mirror is becoming a greater concern. An average release, ranging from 5GB on the low end to 20GB on the high end, is quite a bit of data that not only has to be stored, but served. Archiving older releases is essential, and distributions already doing this should be applauded. However, distribution maintainers should give

mirrors the choice to help by mirroring those archives. This can either be done as an additional target to sync from, or by making the archives available in some other mirror-friendly format. This will not only alleviate loads on slower archive machines, but it also provides legacy users with guaranteed and stable means of downloading packages and ISO images into the future.

1.2.2 Blowing disk cache: Filesystem traversal pain

Distributions have a need to know when a mirror has been updated and to verify that it is up to date and should be included as a valid mirror. The easiest way to do this is to either externally crawl the mirror or to have the site admins add a local process that runs to crawl the repositories and report the results. Both methods have advantages and disadvantages; however, distributions and mirror administrators should be very aware of what these processes do to the servers, as each method causes a linear traversal of the filesystem. This traversal can and does push active data out of the disk cache, causing more data to be sought from disk instead of from the memory cache. This results in severe performance penalties for busy and active mirrors. These kinds of checks should be done sparingly at best as to prevent thrashing of the mirror's disks. A recommendation would be that these kinds of checks be performed at most twice per day, per distribution. This should give distribution maintainers reasonable verification of a mirror's status without causing undue additional stress on the mirrors.

1.2.3 Talk to your neighbor: Scheduling Releases

Lastly, when it comes to distributions, there is one thing that would help mirror administrators immensely—communication amongst the distribution maintainers themselves.

It is becoming common for distributions to follow a set release schedule. While this is a boon to mirror administrators, as they can now easily plan downtimes, upgrades, etc., there is a problem in its current state. A number of these release schedules have become very close together, to the point where in 2007 three major distributions had releases all within the same week of each other. This causes what is best described as chaos

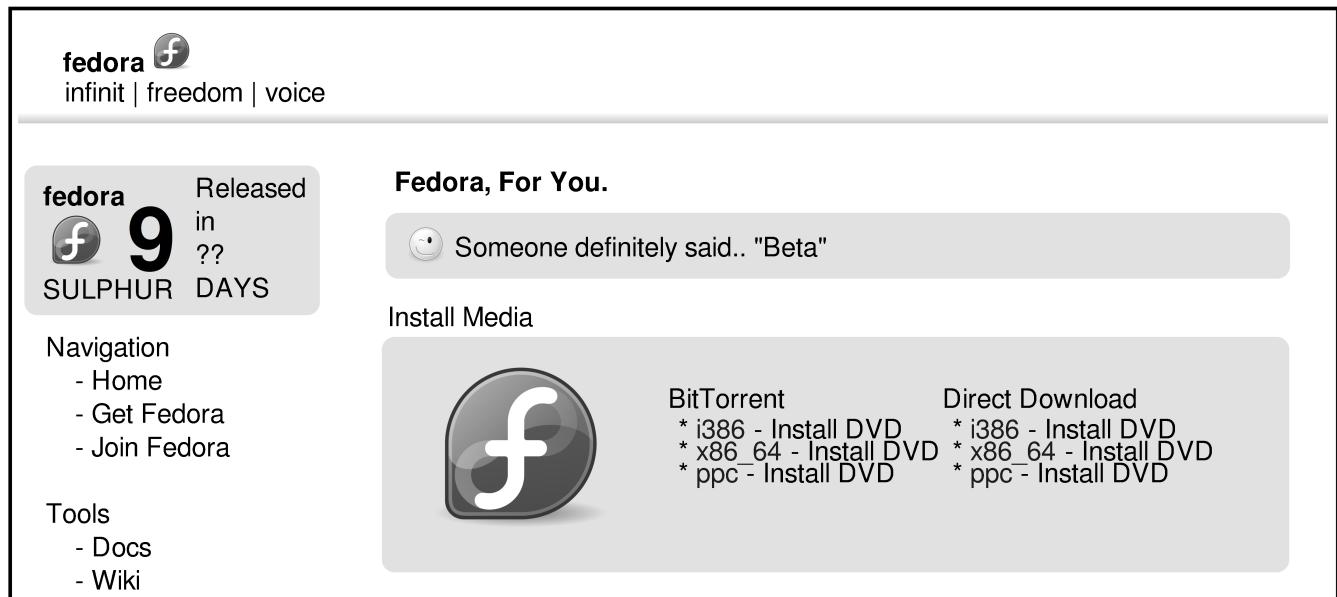


Figure 1: Current Fedora download page

on the mirrors. Where a single distribution could be considered a mad rush, having three distributions release simultaneously is akin to a swarm of locusts.

Handling a release means keeping the working dataset (CD and DVD ISOs, packages, etc.) in memory, having fast enough disk to fetch what's not in memory, and having sufficient network bandwidth. The main problem in this scenario is that while many mirrors are capable of the strain of a single release, tripling the working set's size will greatly exceed the memory available on most systems. Using the Fedora Project as an example, and assuming that only the ISO images are served, this is a baseline of about 18GB of data—a dataset that many mirrors are easily able to hold entirely in RAM. But triple that to 55GB or so of data, and even the largest mirrors must now constantly read everything from disk in order to serve data. This is compounded by the corresponding increase in download requests; more people downloading data means less bandwidth for everyone, thus downloads take longer and load is substantially increased on the mirror servers. For this reason, better communication amongst the distribution maintainers is essential to mitigate these overlaps in releases, and provide the best possible experience for everyone.

2 A better understanding for the user

User:

noun

1. a person who makes use of a thing; someone who uses or employs something
 2. a person who uses something or someone selfishly or unethically [syn: exploiter]
- ... ¹

Users are the reason the mirrors exist in the first place: they are the client and the customer, and as a whole are a very demanding and diverse group. Each individual brings a very different set of expectations, needs, and goals when he or she goes to download the data that is being served. However, there are some things users should be aware of, and keep in mind, to gain a better understanding of what is going on under the surface. This knowledge will help them make better choices in their downloading, from mirror selection, to package selection. This will have set the expectations they bring to the entire downloading process.

¹<http://dictionary.reference.com/browse/user>

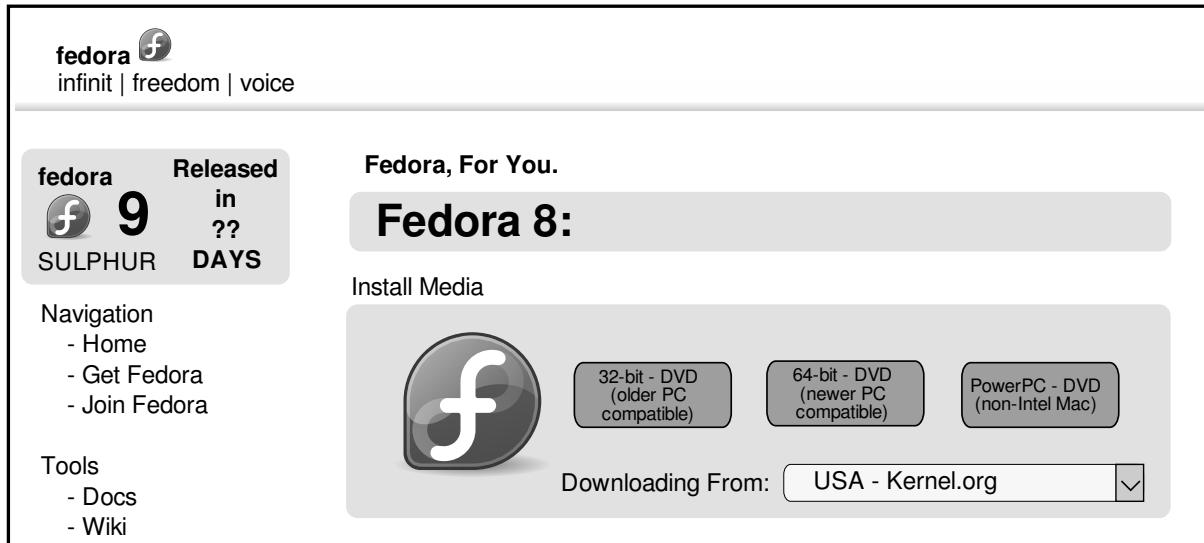


Figure 2: Proposed Fedora download page

2.1 The user isn't always right

Seeing the users as the clients or the customers is quite natural; they are seeking to download data from the distributions. However, going against accepted wisdom when it comes to customers, users are not always right. It is not that their opinion is explicitly invalid, it is that the vast majority of users are ignorant of the complexities in the entire mirroring process, and as a result lack sufficient insight and details to make the correct choices or significant suggestions. This can be alleviated primarily by the users recognizing their own limitations, and that their primary goal is to download data.

2.1.1 Users see things from their perspective

Consider a group of people sitting around a table with an irregular object in the middle of it. Ask each person in this group to describe what they are looking at, and you will get a slightly different response from each of them as they attempt to describe what, to their perspective, is a completely different object. Users are in a similar situation, as each user is staring at the entire mirror process from a different perspective; some go directly to a mirror, some to a distribution, and some follow links on third-party sites to get at the data they are seeking. They are also seeing just one side, the user-facing side, of a much greater system that is working behind the scenes. Users should be aware that what they are seeing is the

culmination of a huge amount of work, theory, and practice, so that they may click a link of some sort and download their data.

Users may not be aware of the ramifications in suggesting or demanding changes in the entire mirroring structure. These consequences may be non-obvious and have significant impact, of which a case in point: many users consider BitTorrent to be something that can and does help alleviate the loads of mirrors; however, this is not strictly the case, and in fact may be detrimental to the mirrors and distributions. Push from the user community at large can have consequences that distribution maintainers and mirrors must carefully balance. To ensure the best user experience for everyone, it is sometimes necessary to discard even popular suggestions.

2.1.2 Diversity of Thought

There are by far more users than there are distribution and mirror administrators. Therefore the push from the user community for enhancements and changes can be quite strong and varied. This does not mean, unfortunately, that the outcry for change in the mirroring infrastructure is necessarily valid or useful. Distribution and mirror administrators should be cautious and careful in implementing the demands of their user base, as many changes are ultimately detrimental. This is not to say that user suggestions are all invalid, but rather to note

that they may have consequences that adversely affect the entire mirroring process. It is also possible that a suggestion that is brought forth is attempting to solve a problem that no one is actually facing. Users should be careful of getting swept up in hype or marketing about specific solutions or technologies; they may be great for certain applications, but they are not always guaranteed to give a better user experience, or may provide benefit to a small minority, but be detrimental to the majority.

It should be kept in mind that every user may be capable of putting forth suggestions, but with there being more users than administrators to experiment, analyze, and verify things, there will always be technologies, ideas, or problems that cannot be addressed or investigated. Distribution and mirror administrators' ranks are filled with incredibly smart and dedicated people who are working on a multitude of problems that many of these individuals relish working on; however, they are a small bunch of people and their time is not unlimited. If there is a problem, it should be voiced, but it should not be taken for granted that it will be solved immediately just because the issue was raised, or that any solution chosen to solve the problem will necessarily match anything suggested.

2.1.3 Beware Arm-chair Administrators

Many users do have valid points, concerns, and issues. There are, however, some that do not have a willingness to accept that they may be wrong. There are countless individuals, the world over, who are working within the systems they have at their disposal in universities, corporations, non-profits, and their own personal equipment to provide mirrors. Each mirror administrator knows the limitations of what they can and cannot do with what they have, and the distributions have a nigh-impossible job of herding these volunteers into a coordinated force capable of providing, with amazing efficiency, the huge amount of data that is downloaded 24 hours a day, 7 days a week, 365 days a year. These people are not perfect, but users who proclaim that they themselves “can do it better,” that the administrators are incompetent, or who lambaste the complicated and often hard decisions either that these administrators are forced to make, are usually wrong. If these individuals that claim they can “do it better” are convinced they are, they should make constructive criticism and suggestions to the distribution and mirror maintainers. Every-

one is open to criticism, constructive suggestions, and help. Users who assume their solution, when put forth, is the absolute correct solution should look back on the last two sections. Each user when putting forth a suggestion or comment should consider that he isn't seeing the entire picture, and it may be that the opinion causes more harm than good.

3 BitTorrent

Since its inception in 2001, BitTorrent has been proclaimed as the means of eliminating the mirror infrastructure, and that it will provide a faster, better means of content distribution. It is based not on downloading from a central repository, or repositories (the typical mirror infrastructure), but on a central point that coordinates the masses of users who wish to download the data and harnesses their collective bandwidth to alleviate the load from the mirror servers and increase total available bandwidth. This is accomplished through every user participating in both downloading and uploading content to the cloud. Distributions and mirrors have recently been exploring or adopting BitTorrent as an alternative means to download their content. This in part due to a perceived user demand, and to explore the possibilities of this technology as a means to more effectively use the resources at the disposal of the distributions and mirrors. However, the motivation should be questioned, beginning with, what problem is BitTorrent really solving? If relatively few nodes perform the vast majority of the uploading, how is this any different or better than providing the same files via more traditional mechanisms like HTTP and ftp? Is BitTorrent straightforward enough for the average user to understand the complex implications of using it, as opposed to traditional download mechanisms? With the rising resentment against BitTorrent from Internet service providers, is this going to adversely affect BitTorrent as a download mechanism? These are but a few of the questions that must be asked about BitTorrent as the answers to these questions affect every layer in the mirroring infrastructure: distributions, mirrors, and users alike.

3.1 What it's good for / Where it's useful

BitTorrent's original intent was to provide a simple mechanism to alleviate the problem of downloading large amounts of data when there is no established mirroring infrastructure in place, or the mirroring infrastructure is incapable of handling the demand put upon

it. In 2001, this was a serious concern, as it was quite possible for large and popular datasets to cause the meltdown of both servers and network infrastructure. In some cases, this caused noticeable slowdowns and bottlenecks on the entire Internet. BitTorrent's intention was to come to the rescue by distributing the combined load to every user who was participating. By taking advantage of the aggregate resources available, users were then able to download faster, and in downloading, helped make downloads for others faster by also uploading the content that a user has.

As BitTorrent has matured and become more accepted, it has been found to be exceedingly useful for moving large datasets of any type, be it multimedia, software, or anything, really. BitTorrent performs best in scenarios where there is more than a single server and client in the cloud. This has become particularly popular where there is not, or cannot be, large and established mirroring infrastructures. This is seen in small open-source projects with large datasets and small followings, but more commonly in illegal downloading. While BitTorrent has been popular for these smaller, more targeted, distribution channels, there are a few commercial exceptions² that are providing torrents.

3.2 Where BitTorrent falls flat on its face

While BitTorrent has the ability to create a respectable distribution mechanism where none exists, by its very nature it has an Achilles heel when large number of users are in the cloud. The tracker, or the controlling unit of the cloud, must pass messages to each of the clients being used. This puts a load on the tracker, and sets a finite limit to how fast it can respond to and process the data in the cloud. As the cloud increases in size, it does not keep the same level of efficiency or productivity when pitted against a mirror structure or a very large user base, such as the one used to distribute Linux. There are facets of BitTorrent that make it particularly painful to an established mirroring structure, especially if the mirrors themselves participate in the BitTorrent cloud.

²Warner Brothers, Paramount, and BitTorrent Inc.'s own entertainment network.

3.2.1 What the numbers show

With BitTorrent's rise in popularity, kernel.org has been running experiments exploring its use as more distributions attempt to push it as a download mechanism. During these tests, data has been recorded and analyzed for many distributions. This paper discusses the Fedora 7 and 8 releases, as they are most consistent and established of these numbers. Kernel.org on both of these occasions joined and participated in the BitTorrent cloud from machines that were dedicated to this purpose. These machines were not providing the same data over traditional download mechanisms like HTTP and ftp. The numbers reflecting the Fedora 7 release used a stock configuration of rTorrent, which would be the normal and expected setup for a typical user. The only exception to this was that the two machines running in this experiment each had three instances of rTorrent running simultaneously. The numbers reflecting the Fedora 8 release, however, add two additional machines and the original two machines maintained three instances of rTorrent, while the two added machines each ran five instances. For the Fedora 8, release rTorrent's configuration was also modified to allow for the maximum possible peers, simultaneous uploads, upload and download rates. Figures 3 and 4 show the amount of data moved by the cloud as a whole versus the data moved by Kernel.org acting as a part of the cloud.

It should be noted, in the case of Fedora 8, that Kernel.org's Pub 1 and Pub 2 servers were explicitly throttled. This was done to prevent bandwidth issues to the machines serving HTTP, ftp, and rsync, which reside on the same network.

The numbers for BitTorrent reveal quite a bit, not the least of which is that a small change in configuration can cause a dramatic change in the behavior of the BitTorrent client. The data also brings into question BitTorrent's ability to keep up with the mirroring needs of a major distribution such as the Fedora Project. In the Fedora 8 release, it can be shown that it is quite possible (in fact, quite probable), that a very few number of nodes are performing the vast majority of the work in the BitTorrent cloud. This is likely due to people leaving the cloud once they have completed their downloads: there is no continuing advantage for the user to continue uploading into the BitTorrent cloud after acquiring the full download. This leaves the cloud increasingly dependent on the few seeders who have a full copy of the

Fedora 7

	size	downloaded	data transferred by BitTorrent cloud	percent of total	transferred by Kernel.org ^a
Fedora-7-KDE-Live-i686	686MiB	4,900	3,200,000	18.85%	603,269.4
Fedora-7-KDE-Live-x86_64	831MiB	1,615	1,280,000	46.06%	589,629.2
Fedora-7-Live-i686	699MiB	8,044	5,360,000	12.95%	693,861.8
Fedora-7-Live-x86_64	779MiB	3,084	2,290,000	15.03%	344,127.6
Fedora-7-i386	2.79GiB	33,909	92,590,000	3.35%	3,097,718.9
Fedora-7-ppc	3.49GiB	957	3,260,000	28.03%	913,751.0
Fedora-7-x86_64	3.3GiB	10,448	33,730,000	6.45%	2,175,682.0
Totals:			141,710,000	5.94%	8,418,039.9

	Pub 1 ^b	Pub 2 ^c	Total
Fedora-7-KDE-Live-i686	301,655.9	301,613.5	603,269.4
Fedora-7-KDE-Live-x86_64	145,080.6	444,548.6	589,629.2
Fedora-7-Live-i686	346,413.4	347,448.4	693,861.8
Fedora-7-Live-x86_64	175,252.0	168,875.6	344,127.6
Fedora-7-i386	1,503,176.6	1,594,542.3	3,097,718.9
Fedora-7-ppc	460,143.1	453,607.9	913,751.0
Fedora-7-x86_64	1,111,975.6	1,063,706.4	2,175,682.0
Totals:	4,043,697.2	4,374,342.7	8,418,039.9

^athis is the total amount of data transferred through BitTorrent by Kernel.org's Pub1 and Pub2 servers

^bMachine was un-throttled, and has 1gbps of upstream bandwidth

^cMachine was un-throttled, and has 1gbps of upstream bandwidth

Figure 3: Fedora 7 BitTorrent downloads of the cloud as a whole and of `kernel.org`

data and who are dedicated enough to stay in the cloud despite having a complete download, or mirrors such as `kernel.org` acting explicitly as a seeder.

BitTorrent's performance also falters when you directly compare it to more traditional download mechanisms such as HTTP, FTP, or rsync. For our testing purposes BitTorrent was given 15% more usable bandwidth, and four machines while the traditional download mechanisms used only two machines. Despite these advantages, BitTorrent did not outshine the traditional download methods. For the Live CD images BitTorrent only moved more data in four of the five torrents. In the more popular DVD install images BitTorrent was unable to keep up lagging by 72% and 204.12% for the x86_64 and i386 downloads respectively, and beating out the PPC downloads by a small margin. Looking beyond pure number of bytes moved, BitTorrent moved 33,111 images as a whole. This pales in comparison to the mirroring infrastructure which has a hundred or so mirrors in it, and with a single mirror, `Kernel.org`, moved

21,901 images. These numbers, however, reiterate BitTorrent's primary purpose: a distribution mechanism for downloads that do not have more structured mirroring and distribution mechanisms.

3.2.2 Immensely manual process for admins

The classic distribution mechanisms (HTTP, ftp, and rsync) are very simple for both distribution and mirror administrators. Simply put the files in a downloadable location, and the mirrors download the data to their servers. When the time is correct, the distribution and the mirrors perform what is commonly known as a "bit flip" (or a changing of the file permissions) to allow normal users to acquire the data. This is quite simple for both parties; in fact, if a mirror admin wished, after initial setup was done in such a way as to download from the distribution on a regular basis, the distribution is the only entity that needs to manually change the permissions on the data and those changes will propagate to

Fedora 8

	size	downloaded	data transferred by BitTorrent cloud	percent of total	transferred by Kernel.org ^a
Fedora-8-Live-KDE-i686	698MiB	6,710	4,460,000	28.24%	1,259,591.4
Fedora-8-Live-KDE-x86_64	805MiB	1,663	1,270,000	29.55%	375,280.9
Fedora-8-Live-i686	697MiB	10,642	7,070,000	22.08%	1,561,068.2
Fedora-8-Live-ppc	698MiB	641	437,550	36.18%	158,286.5
Fedora-8-Live-x86_64	766MiB	2,649	1,930,000	25.2%	486,375.4
Fedora-8-dvd-i386	3.28GiB	33,111	106,380,000	22.81%	24,261,040.5
Fedora-8-dvd-ppc	3.96GiB	1,071	4,140,000	36.48%	1,510,322.9
Fedora-8-dvd-x86_64	3.71GiB	12,017	43,550,000	28.86%	12,569,610.7
Totals:		169,237,550	24.92%	42,181,576.5	
	Pub 1 ^b	Pub 2 ^c	Pub 3 ^d	Pub 4 ^e	Total
Fedora-8-Live-KDE-i686	232,696.2	257,395.4	221,166.9	548,333.9	1,259,591.4
Fedora-8-Live-KDE-x86_64	79,563.7	78,471.2	68,880.2	148,365.8	375,280.9
Fedora-8-Live-i686	286,141.2	322,965.5	242,441.7	709,519.8	1,561,068.2
Fedora-8-Live-ppc	35,926.2	36,520.0	29,412.0	56,428.3	158,286.5
Fedora-8-Live-x86_64	97,050.5	109,541.5	82,232.9	197,550.5	486,375.4
Fedora-8-dvd-i386	4,956,911.9	5,492,479.7	3,586,870.5	10,224,778.4	24,261,040.5
Fedora-8-dvd-ppc	381,919.8	300,517.5	299,703.8	528,181.8	1,510,322.9
Fedora-8-dvd-x86_64	2,479,605.9	2,760,286.6	1,751,454.1	5,578,264.1	12,569,610.7
Totals:		8,548,815.4	9,087,677.4	6,282,162.1	17,991,422.6
42,181,576.5					

^athis is the total amount of data transferred through BitTorrent by Kernel.org's Pub1 and Pub2 servers

^bBandwidth throttled to a max of 240.8mbps for the machine

^cBandwidth throttled to a max of 240.8mbps for the machine

^dMachine has a maximum of 100mbps of bandwidth due to upstream provider

^eMachine was un-throttled, and has 1gbps of upstream bandwidth

Figure 4: Fedora 8 BitTorrent downloads of the cloud as a whole and of kernel.org

the mirrors automatically. This is very straightforward, simple, easy to verify, and robust for all parties involved in the mirroring process. BitTorrent is not, at least in its current implementation, quite as simple for the distribution or mirror administrators to set up.

The process for BitTorrent is more cumbersome from the distribution administrator's point of view. The administrator must put together package sets and create the torrents, which takes some additional effort. Typically these torrents are unavailable until the point at which "bit flip," or release, occurs; so there is no way for the mirrors themselves to join the cloud early. There is an added difficulty that per distribution, per release, the torrent files are either inconsistent in where they are, or not present at all. It is also made more difficult if the files defining the torrent are not present, or they are not in a location where the torrent file is expecting the ISO images to be. This makes it immensely time-consuming for the mirror admins to participate, should they choose,

in the BitTorrent cloud, as they must hand-craft the entire structure, or face re-downloading the data once the torrents are available. There is a means of setting up a more automatic searching of the file system to find and automatically join torrents; however, this would cause additional load on the system, as it will have to walk the entire file space regularly in search of those torrents, and again the structure in many cases is not set up to have the files pre-configured in the correct structure.

3.2.3 Loading of a machine

BitTorrent is designed to manage the cloud and all of the portions of the images that are available. While this works well when a small number of machines are asking a host for data, it does not scale to thousands. This causes the hosting machine to get, effectively, random requests for sections of the data, meaning that it can not sequentially read the file out, and take advantage of

Fedora 7

	Mirrors1			Mirrors2			Totals	BitTorrent Ratio (From Above)
	HTTP	Ftp	Rsync	HTTP	Ftp	Rsync		
Fedora-7-Live-x86_64.iso	144	24		146	31		345	441.75
Fedora-7-Live-i686.iso	1,062	184		1,031	142		2,419	992.64
Fedora-7-Live-KDE-x86_64.iso	120	21		104	24		269	709.54
Fedora-7-Live-KDE-i686.iso	664	129		633	94		1,520	879.40
Fedora-7-ppc-DVD.iso	145	20		107	17		289	441.75
Fedora-7-x86_64-DVD.iso	2,337	281	17	2,145	226	47	5,053	320.46
Fedora-7-i386-DVD.iso	10,579	1,193	55	9,146	865	63	21,901	365.43

Fedora 8

	Mirrors1			Mirrors2			Totals	BitTorrent Ratio (From Above)
	HTTP	Ftp	Rsync	HTTP	Ftp	Rsync		
Fedora-8-Live-ppc.iso	42	5	18	37	2	14	118	226.40
Fedora-8-Live-x86_64.iso	178	31	34	162	16	30	451	634.36
Fedora-8-Live-i686.iso	1,339	111	50	1,167	80	38	2,785	2,238.90
Fedora-8-Live-KDE-x86_64.iso	67	29	33	67	17	30	243	456.19
Fedora-8-Live-KDE-i686.iso	588	82	50	547	71	37	1,375	1,803.80
Fedora-8-ppc-DVD.iso	171	27	16	141	14	13	382	398.71
Fedora-8-x86_64-DVD.iso	2,703	364	38	2,322	238	47	5,712	3,307.36
Fedora-8-i386-DVD.iso	10,716	951	55	9,416	712	51	21,901	7,201.36

read-ahead when sending data to clients. This random access across the disk, which will only get more frequent with the number of clients in the cloud, will very quickly begin to adversely impact the system, causing a rise in load and added stress to the disk. On a system that may already be serving traditional download methods, this constant seeking on the disk can cause loads to rise dramatically, impacting performance for both BitTorrent and the traditional download methods. This makes its use on a normal mirror machine questionable due to the adverse impact it would have on the system.

3.3 Increasing problem to users

There are several technical and logistical reasons that BitTorrent is unsuitable for mirror usage, but there is also a number of hurdles and complexities to a user, both external and internal to their control, that can adversely affect a user's experience in using BitTorrent. BitTorrent, to make it usable, needs to have a routable port for other clients to connect to and request data from. This, however, poses an issue for users, as many of them are behind a NATed firewall that they may or may

not control. On top of that, many users are unaware of this particular issue and don't know that a port needs to be forwarded to their computer. This causes confusion and a lack of understanding about why it "doesn't just work." Things like rsync and HTTP do this without firewall changes. Users may also not be able to control the network they are on, for example a corporate network, where a user is unable to alter the firewall to make use of BitTorrent, thus making the experience painful and unusable.

BitTorrent has also come under fire from Internet Service Providers (ISPs) who feel that BitTorrent is primarily being used for illegitimate purposes. This has lead many large, and small, ISPs³ to begin performing various things to either slow down BitTorrent traffic or to outright block it. This can be problematic for users, and may be undetectable by them, causing frustration at the inability to find what the cause is.

³Comcast is probably the most famous currently; however, Azureus, a Java-based BitTorrent client, keeps a list of known problem ISPs at http://www.azureuswiki.com/index.php/Bad_ISPs

3.4 BitTorrent—too late to the party

While evaluating the feature list and promises of BitTorrent, it seems like it could be the silver bullet that solves many problems for distributions and mirror administrators. However, during the course of real testing and looking at BitTorrent from the perspective of the distributions, the mirror administrators, and the users, there are a number of rather serious concerns and issues that come up that should give all three groups concern. Requests are coming from users to provide BitTorrent, and maintainers are seeking ways of making their distribution process faster and better. Users are seeking a magic bullet, and they have been lead to believe that BitTorrent is it. They want a faster and easier means of downloading the data they want. The reality of the matter, however, is that their calls and howls for BitTorrent to be provided are not made with a full understanding of the impact it has on the system. Much of the infrastructure to provide the user a better experience is in place today. There are hundreds of mirrors around the world ready to mirror the data and distributions have created the infrastructure to manage and pre-distribute the data to the mirrors before release. All that is left to be done is for the distributions to provide a simple and straightforward user interface to interact with so that users can simply download the data they are seeking.

Distributions should endeavor to make and *keep* things simple and straightforward for the end user. Users should be given few (but clear) options, and choices should be limited to the bare few needed. Distributions should endeavor to provide only the most popular formats as a default, leaving less-common formats to be generated by the end users themselves, with a mechanism provided by the Distribution. The intent is not to take all options away from the users, but rather to make things as straightforward and simple for the majority, giving the minority tools to meet their more specific needs. Along with this “simpler is better” approach, basic coordination amongst the distributions is critical, mainly to prevent overlapping release schedules, but to provide better discussion and feedback on what mirroring practices are working and which aren’t. There are issues in the mirroring infrastructure currently, but these are solvable problems. With a better understanding of the issues and problems faced by everyone, solutions and practices can be put in place to better serve.

Linux, Open Source, and System Bring-up Tools

How to make bring-up hurt less

Tim Hockin

Google, Inc.

thockin@google.com

Abstract

System bring-up is a complex and difficult task. Modern hardware is incredibly complex and it's not getting any simpler. Bringup engineers spend far too much time hunting through specs and register dumps, trying to find the source of their problems. There are very few tools available to make this easier.

This paper introduces three open source tools that can help ease the pain, and hopefully shorten bring-up cycles for new platforms. SGABIOS is a legacy option ROM which implements a simple serial console interface for BIOS. Iootools is a suite of low-level utilities which enables rapid prototyping of register settings. Prettyprint is a powerful library and toolset which allows users to easily examine and manipulate device settings.

Introduction

Sometimes you get lucky on a bring-up, and things just work the way they are supposed to. More often than not, though, something goes wrong. Unfortunately, it's usually many "somethings" that go wrong. When things do go wrong, someone has to figure out what happened and how to fix it.

Platforms today are vastly more complicated than they were just a few years back. Almost nothing works when the system powers on. It all needs to be configured. When the inevitable "something" goes wrong, determining the cause can be an overwhelming task.

Of course, there are no magic bullets, but there are tools that can help to make solving some of these problems easier.

1 Terminology

Before diving in, it's important that we are all speaking the same language:

Platform: Sometimes used as a synonym for motherboard, a platform is really the combination of components that make up a computer system. This includes the CPU or CPU family, the memory controller, the IO controller, the DRAM, the IO devices, and usually the system firmware.

Bring-up: The process of evolving a platform from an expensive *objet d'art* into a fully operational computer system. This process usually involves debugging and/or working around the hardware, configuring the system in the BIOS, and hacking the drivers and kernel into shape. It often includes superstitious rituals, cynical prayers, and lots of cussing.

BIOS: Basic Input Output System. The BIOS is the software that executes when a PC powers on, and is primarily responsible for configuring the hardware.

Device: A piece of hardware that is logically self-contained. While a typical southbridge is a single chip, it is usually viewed as a collection of devices, such as disk controllers, network interfaces, and bridges.

Chipset: A hardware chip or chips that provide the bulk of the IO on a platform. Chipsets are typically tested and sold as a single unit. These generally include a *north bridge* which contains one or more memory controllers as well as high-speed IO bridges, and a *south bridge* which contains lower-speed devices such as storage and legacy bus interfaces.

Register: An addressable set of bits exposed by a device. Most devices contain many registers. Registers generally hold control and status bits for the device, and can be mapped into a multitude of address spaces such as PCI config space, memory, or IO space.

2 Serial Console for the Unwashed Masses

An obvious place to start is to get the BIOS output as it boots. Just about anyone who has ever booted up a PC has seen the BIOS output on the screen. This is, however, not very useful. Most servers do not have a monitor plugged in to them at all times. VGA-capable chips, while not particularly high-tech, are not free to buy or run. Why require one on every server?

It is a sad fact that many platforms available today *still* do not have serial console support. Those that do offer it usually offer it as an up-sell on the BIOS, and the implementation quality is often questionable.

Some implementations provide side-band interfaces, which only get used to print certain information. This is not particularly useful to anyone, and is fortunately not seen much any more. Some implementations do what is called *screen scraping* which depends on a real VGA device with real VGA memory to store the screen contents. They periodically scan the VGA memory and send updates on the serial port. Some implementations support text output but completely break down in the face of “advanced” features like cursor movement or color.

2.1 Solving It Once and for All

In order to provide a consistent feature set, one Google engineer chose to solve this once and (hopefully) for all. Thus was born SGABIOS—the *Serial Graphics Adapter*. SGABIOS is a stand-alone option ROM which can be loaded on a platform to provide serial console support. It provides a robust set of VGA-compatible features which allow most BIOS output to be converted to serial-safe output. It supports basic cursor movement, color, text input, and large serial consoles.

The easiest way to use SGABIOS is to make your BIOS load it as an option ROM. You can try to convince your board vendor to include it as an option ROM in the BIOS build, or you can use tools (usually provided by the BIOS vendor) to load an option ROM into a BIOS image. If this is not an option for you, all is not lost. There are commercially available add-in debug cards which have option-ROM sockets. In a pinch, many network and other cards have programmable ROMs which can be made to load an arbitrary option ROM.

When started, SGABIOS attempts to detect if there is a terminal attached. If detected, SGABIOS will

adapt its internal structures to the detected terminal size. SGABIOS then traps INT 10h, the legacy “print something” BIOS function, and INT 16h, the legacy “read keyboard” function. The final result is that any well-behaved BIOS, option ROM, or legacy OS will now be using the serial port transparently. However, there are some badly behaved programs which attempt to write to VGA memory directly. SGABIOS can not fix those applications. Fortunately, this does not seem to be a very big problem.

We have successfully run SGABIOS with LILO and GRUB, as well as DOS. It works wonderfully for the uses we have found, though it does have its limitations. Some applications, such as LILO, query INT 10h for previously displayed data. Because there is no VGA memory backing it, SGABIOS only stores a small amount of the most recently printed output. This has been good enough to handle the applications we have found to do this, but it does have the potential to fail. As with so many things, it is a tradeoff of memory size vs. functionality.

You can find SGABIOS at <http://sgabios.googlecode.com>.

3 Simple Access to Registers

A recurring situation in my office is that you can boot, but something is not right. You might have some ideas on what it could be, but you need to run some additional tests. You need to modify some registers.

You could have the board vendor build some test BIOSes with the various settings. That’s not going to be an effective, scalable, or timely solution.

You could build a custom kernel which programs the desired changes; at least you control that part. It’s still a pretty heavyweight answer, and the hardware test team folks are not really kernel hackers. This approach is better than the last one, but not good.

One might ask “Hold on, doesn’t the kernel expose some APIs that let me fiddle with registers?” Why yes, it does. Now you only have to write some simple programs to do these tests. But again, the test team is not really C programmers. There must be something simpler.

3.1 Introducing Iotools

A simple, scriptable interface to device registers allows anyone who can do basic programming to deal with this problem. Almost anyone is now able to trivially read and write registers, thereby enabling a whole new debugging army.

This is the goal of `iotools`. The `iotools` package provides a suite of simple command-line tools which enable various forms of register accesses. They are mostly thin wrappers around Linux kernel interfaces such as `sysfs` and device nodes. `Iotools` also includes a number of simple logical operation tools, which make manipulating register data easier.

The `iotools` “suite” is actually a single binary, a la `busybox`. This allows for simple distribution and installation on target systems. The `iotools` binary is less than 30 kilobytes in size when built with shared libraries. Building it as a static binary obviously increases the size, depending on the `libc` it is linked against. This should make `iotools` suitable for use in most size-sensitive environments, such as flash or initramfs.

A note of caution is warranted. Writing to registers on a running system can crash the system. You should always understand exactly what you are changing, and whether there might be a kernel driver managing those same registers. Sometimes it is enough to simply unload a driver before making your changes. Other times you just have to go for it.

3.2 What's in Iotools?

At the time of writing, the `iotools` suite includes tools to access the following register spaces:

- **PCI:** Read and write registers in PCI config space. This includes both traditional config space (256 bytes per device) and extended config space (4 Kbytes per device) for those devices which support it. Access is provided by `sysfs` or `procfs` and is supported as 8-bit, 16-bit, and 32-bit operations.
- **IO:** Read and write registers in x86 IO ports. This covers the 64-Kbyte space only. Access is provided by `IN` and `OUT` instructions and is supported as 8-bit, 16-bit, and 32-bit operations.

- **MMIO:** Read and write memory-mapped registers or physical memory. This provides access to the entire 64-bit physical memory space via `/dev/mem`. It supports 8-bit, 16-bit, and 32-bit operations.
- **MSR:** Read and write x86 model-specific registers on any CPU. This provides access to the full 32-bit MSR space via `/dev/cpu/*/msr`. It supports only 64-bit operations (all MSRs are 64 bits).
- **TSC:** Read the CPU timestamp counter on the current CPU. This is provided by the `RDTSC` instruction and is always a 64-bit operation.
- **CPUID:** Read data from the `CPUID` instruction on any CPU. This provides access to the full 32-bit `CPUID` space via `/dev/cpu/*/cpuid`.
- **SMBus:** Read and write registers on SMBus devices. This is provided by the `/dev/i2c-*` drivers and supports 8-bit, 16-bit, and block operations.
- **CMOS:** Read and write legacy CMOS memory. Most PCs have around 100 bytes of non-volatile memory that is accessed via the real-time clock. Access is provided by the `/dev/nvram` driver, and only supports 8-bit operations. This should be used with caution. CMOS memory is often used by the system BIOS, and changing it can have unintended side effects.

In addition to the register access tools, `iotools` also includes several tools to perform logical operations on numbers. These tools are important because they support 64-bit operations and treat all numbers as unsigned, which can be a problem in some shell environments.

- **AND:** Produce the logical AND of all arguments.
- **OR:** Produce the logical inclusive OR of all arguments.
- **XOR:** Produce the logical exclusive OR of all arguments.
- **NOT:** Produce the bitwise NOT of the argument.
- **SHL:** Shift the argument left by a specified number of bits, zero-filling at the right.

- **SHR:** Shift the argument right by a specified number of bits, zero-filling at the left (no sign extension).

3.3 A Simple Example

Suppose you need to test the behavior of enabling SERR reporting on your platform. This is controlled by bit 8 of the 16-bit register at offset 4 of each PCI device. You could whip up a quick script:

```
#!/bin/bash

function set_serr {
    # SERR is bit8 (0x100) of
    #     16-bit register 0x4
    OLD=$(pci_read16 $1 $2 $3 0x4)
    NEW=$(or $OLD 0x100)
    pci_write32 $1 $2 $3 4 $NEW
}

# hardcoded list of PCI addresses
set_serr 0 0 0
set_serr 0 0 1
set_serr 0 0 2
```

You can do better than this, though. You can trivially make this script loop for each PCI device:

```
#!/bin/bash

function set_serr {
    # SERR is bit8 (0x100) of
    #     16-bit register 0x4
    OLD=$(pci_read16 $1 $2 $3 0x4)
    NEW=$(or $OLD 0x100)
    pci_write32 $1 $2 $3 4 $NEW
}

# for each bus, dev, func
for B in $(seq 0 255); do
    for D in $(seq 0 31); do
        for F in $(seq 0 7); do
            pci_read32 $B $D $F 0 \
                >/dev/null 2>&1
            if [ $? != 0 ]; then
                # does not exist
                continue;
            fi
            set_serr $B $D $F
        done
    done
done
```

This version takes a bit longer to run, but works regardless of the devices in the system. You can shorten the run time significantly by putting a sane upper bound on the number of buses. Few systems have more than 20 or 30 buses, even in this era of point-to-point PCI Express buses.

This is the sort of tool that someone with very basic shell scripting skills can produce in just a few minutes with `iotools`.

You can find `iotools` at <http://iotools.googlecode.com>.

4 Making it Simpler

The previous section shows just one example of the sorts of problems that arise during bring-up. Frankly, it wasn't a particularly complicated problem, and the solution is bordering on real programming. Worse than that, it requires that the person doing the work remember several "magic" numbers. Which register is this bit in? How wide is that register? Which bit is it? Taken further, the problem quickly becomes very difficult.

Suppose you want to examine or configure something more complicated, like PCI Express advanced error reporting (AER). AER is a *capability* in PCI terminology. That means that some devices will support it and some will not. The only way to find out is to ask each device. Further, each device might put the AER registers at a different offset in their PCI register set. As if that is not enough, some devices have different AER register layouts, depending on what kind of device they are and which version of the specification they support.

Doing this in an `iotools` script is certainly possible; it just isn't so simple anymore. Google needed something that internalizes and hides even more of the details. This gave rise to `prettyprint`.

4.1 An Unfortunate Name

The original goal of `prettyprint` was this: to dump the state of all the registers in the system in a diff-friendly format. This would allow us to use one of our favorite debug tools, which we call "*Did you try rolling back the BIOS?*" Boot with BIOS A, `prettyprint` the system. Boot with BIOS B, `prettyprint` the system. Then `diff` the results.

Like the previous examples, there are other ways of doing this. They all resulted in a screenful of numbers, followed by a few hours of digging through datasheets to find what each bit of each differing register means. The only thing worse than going through this process and finding that the difference is undocumented is going through this process multiple times.

Instead, `prettyprint` attaches a datatype to field values, allowing it to produce output which is not only `diff`-friendly, but which is also human-friendly.

4.2 Fundamentals of Prettyprint

`Prettyprint` has two fundamental constructs: *registers* and *fields*. In keeping with the common vernacular, a register is a single addressable set of bits. Registers have a defined, fixed width, but they have no intrinsic meaning.

Fields, on the other hand, are of arbitrary width and are the only entity with meaning. Fields can be defined as a set of register bits (*regbits*), constant bits, or even as procedures. Every field has a datatype, and the result of reading a field is a value that can be evaluated against that datatype to produce a human-readable string.

4.3 The Power of Fields

Let's look at a the simple example from Section 3.3. For each PCI device there is a 16-bit register at offset 4 called `%command` (the `%` is a convention to indicate a name is register). For each PCI device there is also a field called `serr`. This field is exactly 1 bit wide, and is composed of bit 8 of `%command`. When accessing this field, one can interpret its value as a boolean, where a value of 1 = "yes" and a value of 0 = "no".

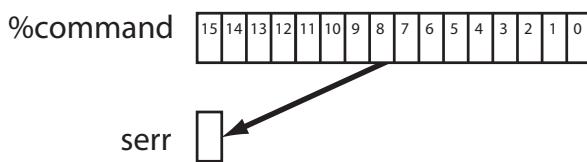


Figure 1: A simple field

Now, when you dump the state of a device, you can see a line item that says `serr: yes`.

This is vastly more useful than a hexadecimal number about which I have to remember that bit 8 being set means `SERR` is enabled. Even better, since I now have a system that understands `serr` directly, I can write to it just as easily as I can read from it.

4.4 Binding Fields to Devices

The previous example glossed over the details of "for each PCI device." This is a key aspect of `prettyprint`'s power. Registers are defined in an abstract way, divorced of exactly which device or access method they employ. They simply have *addresses*. When it comes time to use these registers, `prettyprint` passes control to the drivers which enable each class of device. A *binding* is used to map which abstract registers belong to which driver.

When starting up, `prettyprint` can find hardware devices in one of two ways. Firstly, you can tell it where a device is found. This is the only option for some devices, especially legacy devices. For example, to tell `prettyprint` about the serial port, you would have to tell it something to the effect of, "*There exists a serial port in IO space, at address 0x3f8.*" In so doing, you have given `prettyprint` enough information to bind the serial port registers and fields to a driver and address.

Better still, you can let `prettyprint` discover some devices. Many modern devices can be discovered either through the hardware itself, such as PCI, or through simple interfaces, such as ACPI. In this case, the driver has a discovery routine which will find devices and bind them as it finds them. This is how we are able to define things like `serr` as something that exists "for each PCI device."

4.5 About the Implementation

`Prettyprint` is written in C++. I can hear the cries of frustration already. Why C++? Because I thought that the problem decomposed nicely into an object-oriented model, and because I wanted to improve my C++.

`Prettyprint` has been designed from the start as a library to be used as a backend by various applications. From state dumping utilities to interactive shells to FUSE filesystems, anything is possible.

4.6 Defining Registers And Fields

So how does one go about defining a device? One of the things that the choice of C++ brought to the project was a way to manipulate the language syntax. The end goal is to have an actual interpreted language which is used to define devices. Until then, we have a set of C++ classes, functions, and templates which define a pseudo-language.

This pseudo language is intended to make the definition of registers and fields as simple as possible. Let's look at the SERR example:

```
REG16 ("%command", 0x04);
FIELD("serr", "yesno_t",
      BITS("%command", 8));
```

That's pretty straightforward. We define `%command` as a 16-bit register at address 0x04. We define `serr` as a field with datatype `yesno_t`, composed of bit 8 from `%command`.

Frequently, a field maps directly to a register. To simplify this, `prettyprint` understands *regfields*. For example, the PCI “`intpin`” field is the only consumer of the `%intpin` register.

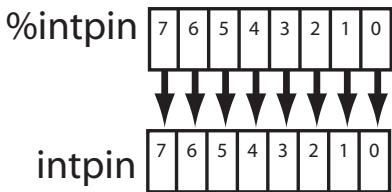


Figure 2: A regfield

We can express that as:

```
REG8 ("%intpin", 0x3d);
FIELD("intpin", "int_t",
      BITS("%intpin", 7, 0));
```

Or we can take the equivalent regfield shortcut:

```
REGFIELD8 ("intpin", 0x3d, "int_t");
```

Let's consider a more complicated example. In a PCI-PCI bridge, there are several registers which control the address ranges which are decoded by the bridge. They are implemented as two different registers, which combine to form a logical 64-bit address. The low 20 bits of both the `base` and `limit` register are fixed to 0 and 1, respectively.

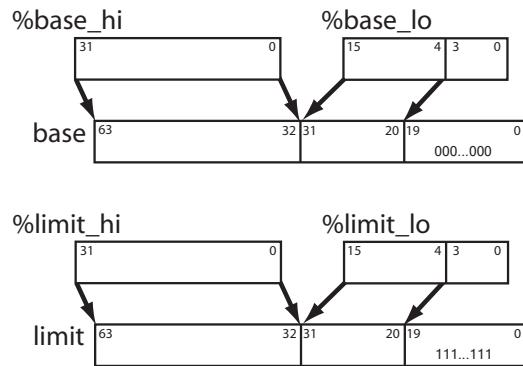


Figure 3: Complex fields

In `prettyprint`, this is expressed as:

```
REG16 ("%base_lo", 0x24);
REG32 ("%base_hi", 0x28);
REG16 ("%limit_lo", 0x26);
REG32 ("%limit_hi", 0x2c);

FIELD("base", "addr64_t",
      BITS("%base_hi", 31, 0) +
      BITS("%base_lo", 15, 4) +
      BITS("%0", 19, 0));
FIELD("limit", "addr64_t",
      BITS("%limit_hi", 31, 0) +
      BITS("%limit_lo", 15, 4) +
      BITS("%1", 19, 0));
```

Notice the use of `%0` and `%1` as registers. These are the *magic registers*. When read, `%0` always returns all logic 0 bits. Likewise, `%1` always returns all logic 1 bits. Also notice that the bits in a field are defined from most significant to least significant. A field can be arbitrarily long, and can be composed of any number of regbits.

4.7 Scopes and Paths

The examples so far have been relatively small. In reality the `%command` register has a number of fields that derive from it. All told, there are thousands of fields in

each PCI device. `prettyprint` provides *scopes* as a mechanism for grouping related things together.

Think of scopes like directories in a filesystem. Each scope has a name and a set of contents. A scope can contain registers, fields, or other scopes. Like the filesystem metaphor, `prettyprint` has paths. There is a conceptual *root* of the path tree, and each register, field, and scope can be named by a unique path. Also like a UNIX directory tree, path elements are separated by a forward slash (/), and two dots (..) means the parent scope.

The `%command` register from our previous examples actually looks something like this:

```
REG16 ("%command", 0x04);
OPEN_SCOPE ("command");
FIELD ("io", "yesno_t",
       BITS ("../%command", 0));
FIELD ("mem", "yesno_t",
       BITS ("../%command", 1));
FIELD ("bm", "yesno_t",
       BITS ("../%command", 2));
FIELD ("special", "yesno_t",
       BITS ("../%command", 3));
FIELD ("mwinv", "yesno_t",
       BITS ("../%command", 4));
FIELD ("vgasnoop", "yesno_t",
       BITS ("../%command", 5));
FIELD ("perr", "yesno_t",
       BITS ("../%command", 6));
FIELD ("step", "yesno_t",
       BITS ("../%command", 7));
FIELD ("serr", "yesno_t",
       BITS ("../%command", 8));
FIELD ("fbb", "yesno_t",
       BITS ("../%command", 9));
FIELD ("intr", "yesno_t",
       BITS ("../%command", 10));
CLOSE_SCOPE();
```

4.8 Datatypes

Each field can be evaluated against its datatype. `Prettyprint` defines a number of primitives:

- **int**: a decimal number
- **hex**: a hexadecimal number
- **enum**: an enumerated value
- **bool**: a binary enum

- **bitmask**: a set of name bits

These primitives are used to create several pre-defined datatypes:

- **int_t**: a number
- **hex_t**: a hexadecimal number
- **hex4_t**: a 4-bit hexadecimal number
- **hex8_t**: a 8-bit hexadecimal number
- **hex12_t**: a 12-bit hexadecimal number
- **hex16_t**: a 16-bit hexadecimal number
- **hex20_t**: a 20-bit hexadecimal number
- **hex32_t**: a 32-bit hexadecimal number
- **hex64_t**: a 64-bit hexadecimal number
- **hex128_t**: a 128-bit hexadecimal number
- **addr16_t**: a 16-bit address
- **addr32_t**: a 32-bit address
- **addr64_t**: a 64-bit address
- **yesno_t**: a boolean, 1 = "yes", 0 = "no"
- **truefalse_t**: a boolean, 1 = "true", 0 = "false"
- **onoff_t**: a boolean, 1 = "on", 0 = "off"
- **enabledisable_t**: a boolean, 1 = "enabled", 0 = "disabled"
- **bitmask_t**: a simple bitmask

Without doubt, any reasonably complex device will need to define its own datatypes. `Prettyprint` allows datatypes to be defined at any level of scope, and to be used in any scope below the definition—similar to C.

- **INT(name, units?)**: define a new int type with optional units.
- **HEX(name, width?, units?)**: define a new hex type with optional width and units.

- **ENUM(name, KV(name, value), ...)**: define a new enum type with the specified named values.
- **BOOL(name, true, false)**: define a new bool type with the specified true and false strings.
- **BITMASK(name, KV(name, value), ...)**: define a new bitmask type with the specified named bits.

Sometimes you want to define a new datatype for exactly one field. Rather than come up with a good name for it, each of the datatype definitions supports an ANON_ prefix, which removes the name argument and produces an anonymous datatype. For example, the previous PCI intpin example used `int_t` as the datatype. In reality, we want an enumerated type. This is the only field that will use this type, so we want to declare it anonymously:

```
REGFIELD8 ("intpin", 0x3d, ANON_ENUM (
    KV("none", 0),
    KV("inta", 1),
    KV("intb", 2),
    KV("intc", 3),
    KV("intd", 4)));
```

4.9 Advanced Techniques

So far, we've seen how `prettyprint` can be used to define simple registers and fields. Unfortunately, few hardware devices are so simple. Because the `prettyprint` "language" is actually a dialect of C++, there is a lot of power at your fingertips.

Hardware registers are at a premium. Often the hardware will overload the meaning of some bits depending on the state of other bits. `Prettyprint` supports the conditional definition of registers and fields.

Let's look at another example. In a PCI bridge's IO decode window, there is a `width` field. That field determines whether the high half of the `base` field is valid.

```
REG8 ("%base_lo", 0x1c);
REG16 ("%base_hi", 0x30);

FIELD ("width", ANON_ENUM (
    KV("bits16", 0),
    KV("bits32", 1)),
BITS ("%base_lo", 3, 0));

if (FIELD_EQ ("width", "bits16")) {
    FIELD ("base", "addr16_t",
        BITS ("%base_lo", 7, 4) +
        BITS ("%0", 11, 0));
} else { // bits32
    FIELD ("base", "addr32_t",
        BITS ("%base_hi", 15, 0) +
        BITS ("%base_lo", 7, 4) +
        BITS ("%0", 11, 0));
}
```

In this example you see the usage of `FIELD_EQ()`. This performs a read of the `width` field and compares the result against the value specified. Comparisons can be done by string or by number, thanks to function overloading in C++. The above example could have just as easily (though less maintainably) used:

```
FIELD_EQ ("width", 0)
```

The actual evaluation of the a comparison is done by the specific datatype, which is the only place that can actually determine what it means to compare values. `Prettyprint` supports the following comparison operations:

- **FIELD_EQ**: the field is equal to the specified comparator.
- **FIELD_NE**: the field is not equal to the comparator.
- **FIELD_LT**: the field is less than the comparator.
- **FIELD_LE**: the field is less than or-equal-to the comparator.
- **FIELD_GT**: the field is greater than the comparator.
- **FIELD_GE**: the field is greater than or-equal-to the comparator.
- **FIELD_BOOL**: the field is boolean TRUE, equivalent to NE 0.

- **FIELD_AND:** the field matches the comparator.

Again, because the prettyprint “language” is really just C++, almost any native construct will work. There are some limitations, though.

To start with, C++ will not allow a `switch` statement on a non-integer value, so you can not switch on enumerated strings. In the eventual prettyprint language implementation, this will be supported.

Secondly, control statements are evaluated just once, as the tree of registers and fields is being built. Later changes to control bits do not change the tree structure. This is something we want to enable in the prettyprint language, but we do not have support for it yet.

4.10 Discovering Specific Devices

Throughout these examples, we have looked at standard PCI fields and registers. The PCI standard covers only a fraction of the available PCI register space. Almost every PCI device defines its own non-standard register set. What about those extra registers and fields?

In the same way that prettyprint can discover generic devices, such as PCI, it can also discover specific devices. A device definition can register itself for discovery through a specific driver. When the driver’s discovery mechanism detects the registered device, as determined by a driver-specific signature, it invokes the specific device code, rather than the generic.

For example, a device definition for an AMD Opteron might register with the PCI driver for the vendor and device pair (0x1022, 0x1100). When the PCI driver finds that vendor and device pair, the Opteron-specific device code would be invoked, rather than the generic PCI device code.

Rather than re-encoding the entire PCI specification, the generic PCI code can be invoked from the Opteron code. This allows device code to extend standard devices with very little effort.

4.11 The Directory Tree

The prettyprint code is broken into four components. The top-level directory contains the core classes

and functions that make up the prettyprint library. This includes `pp_register`, `pp_field`, `pp_datatype`, etc.

The `drivers` subdirectory contains the driver modules. Currently prettyprint only supports Linux, though it would not be hard to add support for other operating systems. At the time of writing, the `drivers` directory contains drivers for:

- **PCI:** via `/sys` or `/proc`
- **MEM:** via `/dev/mem`
- **IO:** via IN and OUT instructions
- **MSR:** via `/dev/cpu/*/msr`
- **CPUID:** via `/dev/cpu/*/cpuid`

The `devices` subdirectory contains device code, written in the prettyprint “language.” When we have a real language parser, this is the code that will be rewritten in the new language. Prettyprint currently has support for:

- **PCI:** most of the fields for generic PCI and PCI Express devices, including many capabilities.
- **CPUID:** very basic CPUID fields

Lastly, the `examples` subdirectory contains example programs which use the prettyprint library.

4.12 FUSE and Prettyprint

One of the more exciting examples is the `pp_fs` application. This is a FUSE filesystem which allows direct access to registers and fields.

Using `pp_fs`, the example of setting SERR on all devices becomes trivial:

```
$ find /pp -wholename \*command\*/serr \
| while read X; do
    echo -n "$X: $(cat $X) -> "
    echo "yes" > $X
    cat $X
done
/pp/pci.0.0.0.2/command/serr: no -> yes
/pp/pci.0.0.1.0/command/serr: no -> yes
/pp/pci.0.0.0.1/command/serr: no -> yes
/pp/pci.0.0.0.0/command/serr: no -> yes
```

4.13 Current Status

The current examples demonstrate the capabilities of `prettyprint`. `pp_discover` has already proven to be a useful tool at Google. But there is still a lot of work to do in many areas.

`Prettyprint` is under active development. A great way to get involved is to encode new hardware devices into the `prettyprint` language. The larger our device repository gets, the more useful it becomes.

You can find `prettyprint` at
<http://prettyprint.googlecode.com>.

5 Acknowledgements

Thanks to Nathan Laredo for pushing to make SGABIOS a reality.

Thanks to Aaron Durbin for busybox-ifying `io-tools` on a whim, and to all the Google platforms folks who have added tools to it.

Thanks to Aaron Durbin, Mike Waychison, Jonathan Mayer, and Lesley Northam for all their help on `prettyprint`.

Thanks to Google for letting us hack on fun systems and release our work back to the world.

Audio streaming over Bluetooth

Marcel Holtmann
BlueZ Project
marcel@holtmann.org

Abstract

During the last year the Linux Bluetooth community worked hard to establish a solution for streaming audio using the Bluetooth wireless technology. This solution is based on multiple standards for headset (mono quality) and headphones (stereo/high quality) that have been published by the Bluetooth SIG. It also includes support for remote control and meta data information like song titles, etc.

This includes work on the open source implementation of the Subband-codec which has been highly improved and can now measure up against any commercial implementation of this codec.

Introduction

The initial Bluetooth specification [1] came with support for the Headset profile (HSP) which allowed connecting mono headsets to cellphones and enabled voice playback and capture support. Later specifications introduced the Handsfree profile (HFP) which added support for caller identification and additional status indication to allow better integration with cellular networks. In addition to the mono headset support in HSP, the Bluetooth SIG created the Advanced Audio Distribution Profile (A2DP) to support high quality audio streaming.

During the last three years various attempts have been made to add proper support for all three profiles to the Linux Bluetooth stack. The initial attempt was with the Bluetooth ALSA project [2] and produced a kernel module called *btsco*. It only supported the Headset profile. The second attempt was the PlugZ project which derived from the Bluetooth ALSA project. It collected various attempts for Headset profile (*headsetd*) and A2DP (*a2dpd*). Both daemons came with plugins for the Advanced Linux Sound Architecture (ALSA) [3]. Figure 1 shows the planned architecture

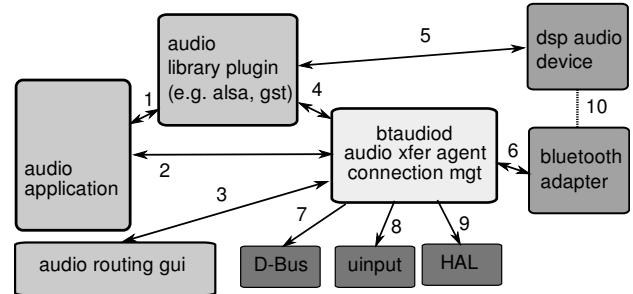


Figure 1: PlugZ architecture

of PlugZ. The support for GStreamer [4] and D-Bus [5] was never finished.

The Bluetooth ALSA project and PlugZ project had various deficiencies. The main problem was that both were too tightly coupled with ALSA. For PlugZ the support for alternate media frameworks was planned, but never integrated since the design was not flexible enough. The other issue was that neither of them were real zero-copy designs. The audio data was always copied two times. This killed the latency and increased the CPU load. The Bluetooth audio service was created to solve the issues of Bluetooth ALSA and PlugZ and fully replace them.

Technical background

The Headset profile and Handsfree profile use an RF-COMM channel as control channel. The protocol of this channel is based on standard AT commands with Bluetooth specific extensions. An example of these extensions is the volume control or the buttons to accept or reject calls. For the transport of the audio stream, the Bluetooth Synchronous Connection Oriented link (SCO) channel is used. The SCO channel is a designated channel within the Bluetooth piconet that allows the transport of 8 kHz Pulse Coded Modulation (PCM) audio over the air that will be encoded using Continuous Variable Slope Delta (CVSD) modulation.

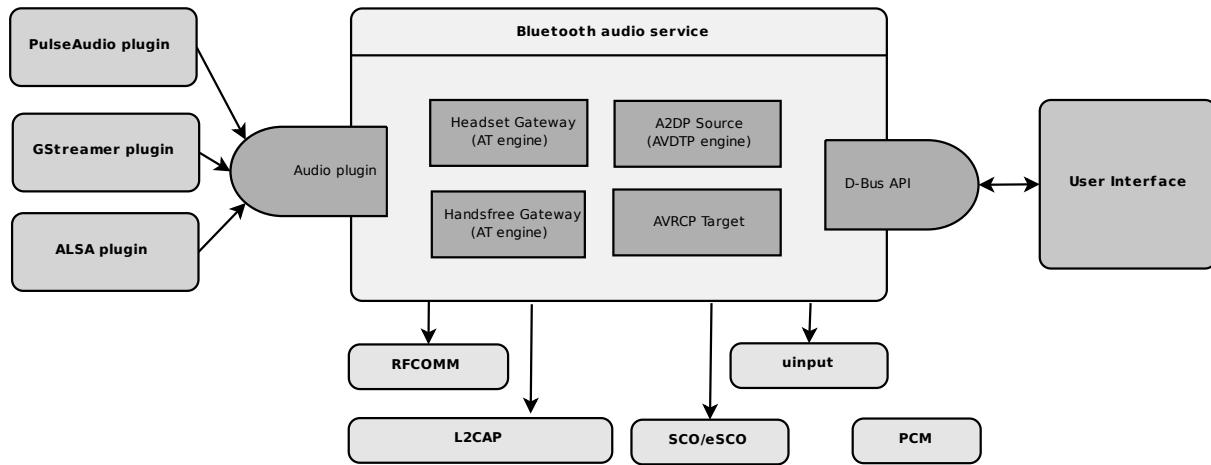


Figure 2: Bluetooth audio architecture

The audio data transferred over the SCO channel can be provided via the normal Host Controller Interface (HCI) hardware driver or via a PCM back-channel. In case of a desktop computer, the HCI will be used. In case of an embedded device (for example a mobile phone), the SCO channel will be directly connected via a PCM interface to the main audio codec.

The Advanced Audio Distribution Profile uses the Logical Link Controller and Adaptation Protocol (L2CAP) and the Audio/Video Distribution Transport Protocol (AVDTP) for control and streaming. This protocol defines a binary control protocol and a binary streaming protocol based on RTP [6]. To compress the audio stream a royalty free Subband-codec (SBC) has been introduced together with the A2DP specification. In addition to the mandatory SBC it is possible for Bluetooth devices to support MP3, ACC or other vendor codes.

Bluetooth audio service

The focus during the Bluetooth audio service development was to fix all the limitations of Bluetooth ALSA and PlugZ and present a flexible infrastructure that could be used for all Bluetooth audio related profiles. The following requirements were identified during the design:

- Treat mono and high quality profiles as equal

With the Service Discovery Protocol (SDP) it is possible to retrieve the supported profile list from

any remote Bluetooth device. Together with the information about the audio stream it is possible to select the correct profile automatically and do the needed conversion transparent for the user.

- Integrate with all multimedia frameworks

Choosing ALSA as the basic multimedia framework is not the best choice. Actually ALSA is pretty bad when it comes to virtual soundcards and that is what Bluetooth audio is. There is no audio hardware directly attached to the system. All headsets, headphones or speakers are connected via an invisible radio link.

The frameworks GStreamer and PulseAudio [7] are much better when it comes to handling virtual audio devices. So there is no need to treat them as second class citizens.

- Low-latency and high performance

In cases where the host has to handle all audio data processing, it should be done the most efficient way and data copying should be avoided at all costs. This increases the performance and at the same time results in good latency. In addition this will reduce the power consumption.

- Full integration with D-Bus

Provide a full D-Bus interface for control and notifications. It should allow creating, configuring and controlling audio connections.

Integrate with the Audio/Video Remote Control Profile (AVRCP) for handling keys and displays on remote devices.

The current release of BlueZ contains the audio service with support for HSP, HFP, A2DP and AVRCP. It comes with plugins for ALSA and GStreamer. Figure 2 shows the high-level architecture of the audio service.

Subband-codec implementation

During the work on PlugZ and later the Bluetooth audio service, the open source community produced an LGPL licensed implementation of SBC. This implementation has been highly optimized for performance and quality by the Instituto Nokia de Tecnologia (INdT) in Brazil [8].

The source code of the SBC implementation can be found at its Sourceforge project [9] or as part of the BlueZ source code [10].

ALSA support

The ALSA plugin has been created to allow legacy applications to use Bluetooth audio streaming. To make use of a remote headset, headphone or speaker the device has to be configured first. The file *.asoundrc* in the home directory needs to be extended with the lines from Figure 3.

```
pcm.bluetooth {
    type bluetooth
    device 00:11:22:33:44:55
}
```

Figure 3: ALSA configuration entry

This creates a virtual PCM device *bluetooth* which can now be used as if it were a normal soundcard, for example with *aplay -D bluetooth example.wav*.

GStreamer support

With the usage of GStreamer the possibilities become more flexible. The GStreamer framework allows a lot of configuration since everything can be abstracted into elements or containers and then a pipe can be constructed out of them.

The GStreamer plugin that provides access to the Bluetooth audio services consists of multiple elements that can be combined in various ways. Figure 4 shows the details of these elements.

```
# gst-inspect bluetooth
Plugin Details:
  Name:           bluetooth
  Description:   Bluetooth plugin library
  Filename:      libgstbluetooth.so
  Version:       3.30
  License:       LGPL
  Source module: bluez-utils
  Binary package: BlueZ
  Origin URL:   http://www.bluez.org/
rtpsbcpay: RTP packet payloader
a2dpsink:  Bluetooth A2DP sink
avdtsink:  Bluetooth AVDTP sink
sbcparse:  Bluetooth SBC parser
sbcdec:    Bluetooth SBC decoder
sbcenc:    Bluetooth SBC encoder
bluetooth: sbc: sbc
7 features:
+-- 6 elements
+-- 1 types
```

Figure 4: GStreamer plugin

Besides the elements the plugin also contains the definition for the SBC data type. This allows GStreamer enabled applications to load or store files with and SBC encoded audio stream.

The *sbcparse*, *sbcdec* and *sbcenc* elements contain the SBC implementation and besides using them for Bluetooth device, the architecture of GStreamer would allow them to be used within other multimedia applications.

The *a2dpsink* and *avdtsink* provide an easy abstraction of the inner workings of A2DP and its underlying protocol AVDTP. The *rtpsbcpay* element provides the RTP payload with SBC encoded data. An alternative would be to use an MP3 payloader from a different GStreamer plugin. This however only works if the headset also supports an MP3 encoded stream which is not a mandatory feature.

GNOME integration

The audio service provides an extensive D-Bus API to control the audio devices and to allow user applications

easy access. The current integration into the Bluetooth GNOME application has just started. Figure 5 shows the initial integration.



Figure 5: GNOME integration

The missing piece is integration with the Bluetooth wizard to provide an easy setup of Bluetooth audio devices.

Future work

Currently work is undergoing to develop the PulseAudio plugin to integrate directly with the third major multimedia framework.

The Bluetooth SIG has released updates of the HFP, HSP, A2DP and AVRCP specifications. These new specifications include updates for Simple Pairing support and meta-data transfer. The meta-data transfer allows to transfer ID3 information like song titles and artist information to the headset. In the future headsets or speakers with a display could use the meta-data transfer to display them. The support of meta-data transfer within the GStreamer plugin is work in progress.

Conclusion

The design and implementation of the current architecture has been widely accepted and used in products like the Nokia N810 [11].

The quality and performance of the Subband-codec can easily measure up against any commercial implementation. The architecture is flexible enough to support embedded hardware and also advanced systems with DSP or codec offload of the audio processing.

References

- [1] Bluetooth Special Interest Group:
Bluetooth Core Specification Version 2.0 + EDR, November 2004
- [2] Bluetooth ALSA Project:
<http://bluetooth-alsa.sf.net/>
- [3] Advanced Linux Sound Architecture (ALSA):
<http://www.alsa-project.org/>
- [4] GStreamer Multimedia Framework:
<http://www.gstreamer.net/>
- [5] freedesktop.org:
D-BUS Specification Version 0.12
<http://dbus.freedesktop.org/doc/dbus-specification.html>
- [6] Request for Comments:
RTP: A Transport Protocol for Real-Time Applications (RFC3550)
- [7] PulseAudio Sound Server:
<http://www.pulseaudio.org/>
- [8] Instituto Nokia de Tecnologia (INdT):
<http://www.indt.org.br/>
- [9] Subband-codec (SBC) implementation:
<http://sbc.sf.net/>
- [10] BlueZ Project:
<http://www.bluez.org/>
- [11] Nokia N810 Internet Tablet:
<http://www.nokiausa.com/A4626058>

Cloud Computing: Coming out of the fog

Gerrit Huizenga
IBM Corporation
gh@us.ibm.com

Abstract

Cloud computing is a term that has been around for a while but has been storming into the mainstream lexicon again, although with a lot of confusion about what Cloud Computing really means. Many vendors are jumping into Cloud like solutions and are labeling every new activity as somehow related to cloud computing. This paper explores some aspects of what makes a cloud, and distinguishes cloud computing from provisioning, utility computing, application service providers, grids, and many other buzzwords, primarily by focusing on the technology components that make up a cloud.

Further, this paper will briefly explore the factors that have made cloud computing a popular topic, including the current availability of Linux®, the advances in some of the virtualization technologies, and some interesting evolution in terms of provisioning and virtual appliances, and, of course, some of the current providers of technologies that are debatably clouds today. This paper also makes a projection as to the rise of a new era of Internet scale computing which will be enabled by the cloud and identify some of the technologies that will need to further evolve to make cloud computing as ubiquitous as the Internet.

1 Clouds everywhere! Or is it just fog?

You've seen the buzz by now. Cloud computing is *it*. The next big thing. The future of computing. There will be only five clouds in the future. Computing power is on the verge of being as pervasive as the network bandwidth on the Internet. But, wait a minute. What exactly *is* a cloud, you might ask? And, the answers are a bit more foggy.

Some of the first answers point to Google. There! That's a cloud! Look at Amazon's EC2—the Elastic Compute

Cloud—it has the word cloud in its name, it must be a Cloud, right? Microsoft must have a cloud, right? Or, another favorite response: We'll know it when we see it.

Probably the best way to start to understand what cloud computing might actually mean is to look at some of the existing technologies that are *not* cloud computing. There are quite a few of these, and all have some distinct properties of their own. As it turns out, cloud computing encompasses a number of the properties from these technologies which is probably why they are so commonly confused.

Some of the alternate technologies covered here include utility computing, grid computing, cluster computing, and distributed computing. We'll also look at Application Service Providers (ASPs), Software as a Service (SaaS), and Hardware as a Service (Haas). Later, this paper will also briefly show the relationship between cloud computing and Software Oriented Architecture (SOA).

After examining the technologies that exist distinctly from clouds, this paper examines the challenges in the industry are driving a push towards cloud computing, and how cloud computing evolves from a number of existing technologies. Cloud computing expands beyond these current properties, usually by drawing on the best techniques present in those pre-existing technologies. Clouds are often seen from a user perspective, but creating, managing, and ultimately implementing a number of the advanced capabilities of clouds is a key topic discussed later in this paper.

Finally, we'll provide some insights on the evolution of clouds, where we *really* are with respect to clouds in the industry, and when cloud computing will be as prevalent as the Internet.

2 Technologies that are *not* cloud computing

Everyone always wants to start with the question: So just what *is* cloud computing? And, I prefer to start that answer with an analysis of what common buzz words and technologies are *not* cloud computing. And, by means of that path to the answer, I will assert that cloud computing is actually something new—not just a new name for an old technology. There are plenty of old technologies in this space, and a few of the more common technologies that are confused with cloud computing are utility computing and grid computing. Starting with those, we will begin with a short description of what they are, and in each case, show how each relates to cloud computing. Once those related technologies have been put into context, we'll dive into what cloud computing actually is.

2.1 Utility Computing

So far, most of the industry buzz around cloud computing has actually been based on providing computational capability to end users, usually with the potential for charging for the use of that compute power. Basically, with utility computing “somebody else” owns the computers or compute resources, manages them, and sells you access to capacity. This form of computing draws its name directly from the public utilities which handle the creation/management of the resources that you use every day, such as water, electricity, natural gas, etc. As an end user, you don’t have to worry about managing those resources nor do you generally worry about access to those resources other than some standardized means of tapping into that resource. Then, based on the resources that you use, you get a bill.

Utility computing eliminates the need for you to acquire and manage your own compute resources, eliminates the start up costs in acquiring capital, configuring machines, performing the basic systems management and systems administration, and allows you to focus your efforts on simply running your application. Of course, in the utility computing model, *someone* still has to buy that hardware, manage that hardware, provide access, security, authentication and, in some cases, even the basic applications that allow you to run your application. However, the term *utility computing* has no particular implication as to what systems are provided, what interfaces are available, or how applications are developed, provisioned, or otherwise utilized within a utility computing

service. So, in essence, utility computing is really focused on a means of using hardware managed by someone else—be it another company, or perhaps a division of your own company, government, educational or research institution.

Utility computing is often viewed as a type of use case provided by several other types of computing, such as Grid Computing or Cloud Computing, although effectively anyone with a computer could provide access to their resources, potentially with cost recovery for the time and services used, and that would effectively be utility computing. Utility computing in general does not define the means of access to the compute resources, the existence or need for any APIs, or even what types of workloads would be supported by the underlying compute resources.

Utility computing could also refer to access to storage and the related cost recovery charged by those service providers.

2.2 Grid Computing

Grid computing is an idea which started with a vision of making compute resources sharable and broadly accessible, ultimately providing a form of utility computing as described above. Grid development, though, has historically evolved from a deep computing research focus which has been heavily backed by access to large compute clusters. Most of the interesting grid-related projects are those doing deep research with cluster aware programs, developed with a willingness to use a grid-aware library such as the Globus Toolkit™ which facilitates compute and data intensive applications which typically require high levels of inter-machine communication. These toolkits and applications provide the ability to locate services within the grid, provide for communication between processes, simplify the ability to partition a workload into a grid-aware application, and provide the application with secure access to data. The Globus Toolkit has recently evolved from a library approach to a more Web-centric approach based on the Web Services Definition Language (WSDL) and the Simple Object Access Protocol (SOAP) to encapsulate interprocess communication into Web-based protocols.

Most grids today provide a utility computing model, including charge-back capabilities. These grids provide

access to single machines or to entire clusters of machines in most cases. Further, many of these grids assume that each machine in the cluster is running similar or identical software packages. The packaging mechanisms that are typically used in Grid environment help to ensure that all machines are running the same software packages, have access to similar/identical versions of the base libraries, and provide the same sets of grid middleware whenever possible. They all typically allow the installation of additional packages as well, but the primary goal is on the build-up of a base stack which is as close to identical between machines as possible.

Grid computing has also recently evolved its use cases to the point of supporting commercial datacenter workloads, although it is much more difficult to find examples of commercial data centers running Grids for general purpose workloads today.

While Grids operate as providers of utility computing, they can also be used as a means of organizing a local, regional or corporate data center into a high-powered computing engine for research. It is a model which is highly targeted to provide simplification of management and processing for compute- or data-intensive workloads. However, its design strongly favors the scientific computing community and is only just evolving, at least conceptually, into a more general-purposed paradigm.

When we look at cloud computing, we'll see that it bears many similarities to grid computing, and, in fact, borrows many concepts from grid computing. In the future, it is also quite likely that cloud computing and grid computing will have a high degree of convergence because both models share many of the same core principles and have a very similar vision of the ultimate state of the computing world. Ian Foster, often viewed as the father of grid computing, also writes occasionally about clouds in his blog.¹

2.3 Cluster computing

Cluster computing is again somewhat similar to Grid computing, and, in many ways, can most easily be viewed as a subset of Grid computing. Cluster computing is typically based on a set of machines with shared access to storage, all operating as part of a single workload. Workloads are often designed to be cluster aware

and are often designed around a shared-nothing workload which limits the need for a coherent locking model, or a shared-everything model, usually based on some form of a distributed locking model. Clusters are typically used for many of the same workloads that are targeted towards Grids, although many clustered installations use a more limited form of inter-process communication, fewer libraries, and, quite often, more custom programming. Classic models for dividing a problem up into discrete components, such as the calculation of a Mandlebrot set or projects like the SETI (Search for Extraterrestrial Intelligence) project can easily divide a large working set into a large number of discrete problems.

Clusters differ from grids in a few visible ways. First, clusters are very homogenous with respect to the hardware and software installed. Most of the entries in the Supercomputer Top 500 list² tend to be large numbers of identical machines effectively operating as one massive multiprocessor computer. Where Grid computing tries to keep the software stack close to identical, large scale clusters or supercomputers tend to have an *identical* stack such that every machine is an identical cog in the great supercomputer. Supercomputers are typically viewed as a single aggregate machine with their throughput measured as the sum of the potential throughput in each machine.

Also, each cluster is typically designed to run one "job" at a time, typically a long running job whose goal is to address some problem so large that it would be nearly impossible to tackle in any other conventional sense. This usually means that the number of actual workloads which are available to be run on any given cluster is a very small list of applications over any particular period of time.

Cluster computing, however, brings some very interesting lessons of its own to Cloud Computing. One of the first of those stems from the fact that today's supercomputer is tomorrow's mid range computer. Clusters have started by using very high processor counts and system counts as the way to get to their very high compute limits. And with these large numbers of systems (as many as 128,000!), the tools to install and manage those systems become very important. Simply installing the operating systems and applications can become a bottleneck without excellent tools. And, managing those systems in

¹<http://ianfoster.typepad.com/blog/2008/01/theres-grid-in.html>

²<http://www.top500.org/>

the face of potential failures, firmware updates, network reconfigurations, and so on, is critical to keeping those expensive systems operational. One of the key properties for simplifying the management of those systems tends to rely on their relative homogeneity, which we'll come back to briefly when discussing cloud computing.

2.4 Distributed Computing

Again, a subset of Grid Computing, Distributed Computing is worth an independent mention in this space, primarily because it is the form of computing which assumes that multiple workloads operate across many machines, working together to solve a particular problem or implement a business workflow. In the most common cases, distributed computing can be a set of coordinating compute based workloads, distributed within an intranet or across the Internet, each contributing something to the end result. Distributed computing has less dependence on similarity of operating systems, platforms, or internal data representations than Cluster computing does. Instead, the real focus is on providing either a method for communication—either very tight coupling in the case of something like grid, or potentially very light coupling in the typical SOA deployments. However, the end result is that the compute activities for a single operation are distributed across a wide range of hardware and software, where, at the extreme, those couplings must be well-managed; creation of the various workloads must be coordinated; communication points and protocols must be well-defined and managed at some higher level. In some cases, distributed computing relies on a single “parent” process which coordinates all of the components of the workflow. In other cases, the components exist as services which reside at well-known locations. In many business situations, those services or other workload components reside at locations which must be identified in a configuration file or by a similar mechanism and then managed throughout the lifecycle of that portion of the business process.

Just like with Grid and Cluster Computing, Distributed computing has several lessons which are relevant to the evolution of Cloud computing. These include the realization that in most environments today, the workload is actually highly distributed, both within a corporate, research, education or government intranet but also more globally within the Internet. As such, many of these interoperational connections need to be configured and managed. The connections between applica-

tions need to be maintained, the network configurations need to be maintained, and, on within the networking space, security between applications must also be maintained. These problems are often addressed today by the systems administrator or by the author of the software. We'll look at how another approach in the context of cloud computing can also help with managing these connections and the relevant security issues.

2.5 Provisioning

The most common solutions that I see today being labelled as Cloud Computing are those with a set of applications and the ability to provision, or deploy, those applications within a utility computing framework. And, to be clear, I believe that provisioning and utility computing are aspects present around cloud computing, but a simple provisioning capability on top of utility computing is not cloud computing. Further, most of the provisioning solutions I see in use today are capable of deploying a single application or a single set of applications onto a single machine. While that may be a start towards cloud computing, it is honestly a very small start.

Provisioning capabilities come in many forms. Those used today in Cluster computing typically specialize in deploying the same operating system and applications to thousands of machine nearly simultaneously. In the recent past, applications like Tivoli's Provisioning Manager® (TPM) provides the ability to deploy and configure operating systems and complex applications to bare metal or, recently, to any of several hypervisors. Provisioning solutions within utility computing environments today, such as Amazon.com®'s Elastic Compute Cloud (EC2) tend to provision single images, usually wrapped as a virtual appliance, to one of a few pre-set virtual platform configurations.

Provisioning can include anything from deploying an application onto an already running operating system, up through deploying a set of complex virtual appliances onto a newly configured set of virtual machines, complete with virtual networks and configured access to storage. There are a number of software applications to aid in provisioning, including 3tera's AppLogic which allows graphical display of the connections between virtual appliances that are to be provisioned.

Provisioning, including complex, multi-tier virtual appliance provisioning is a key component of cloud computing. However, one of the larger gaps in provisioning

related to cloud computing is the definition and standardization of the virtual appliances as well as tools to help create those virtual appliances. Further for true adoption and deployment, the ability to manage a catalog of virtual appliances will be key.

2.6 Application Service Providers

A growing trend over the past 10–15 years has been the growth of the application service providers (ASPs). They range in nature from more conventional business applications, like hosted version of SAP, to the more current excursion of Google into Google Docs. Common questions on cloud computing still include references to ASPs, which typically just host one or more applications which allow users to use those applications without having to install or manage those applications themselves. However, there is very little flexibility on what applications a given provider makes available, and no true elasticity of the underlying resources that would be possible from true cloud computing. As we will see shortly, one of the major benefits of cloud computing is a level of elasticity in the use of compute resources and an underlying dynamic infrastructure.

2.7 Software as a Service

The new and updated term for ASP is Software as a Service (SaaS), although as the internet and the capabilities for software hosting have matured, there have been some subtle evolution of the term. Historically, ASPs provided physically isolated systems for different customers, sometimes even going as far as to provide physically isolated networks or VPNs for their larger customers. Today, many SaaS environments provide isolation based strictly on successful authentication. This in some ways is a reflection of the relative maturity of security isolation solutions more than perhaps anything else. Another shift has been a transition towards native Web interfaces on top of today's software applications. Historically, most provided their own UI (or GUI). Today, most front ends are very Web-friendly, with SOAP or REST based interfaces.

Ultimately, though, ASPs, SaaS, and Utility Computing are examples of services that can be provided by a Cloud computing environment. And, the biggest difference between these models as provided today and their deployment on a cloud computing-based environment

is that the underlying management of the platforms and services will be transparently managed and will allow these configurations to be deployed by more than just a few providers with highly sophisticated support staffs.

2.8 Hardware as a Service

Since we can offer software as a service, why not offer hardware (or platform) as a service (HaaS, PaaS)? Today a number of utility computing providers are effectively doing that. Amazon's Elastic Compute Cloud (EC2) or IBM's Research Compute Cloud (RCC) are effectively offering direct access to hardware to their respective customers. In both of these modern cases, the interesting difference is that the hardware being offered up is typically virtualized. This allows some additional functionality and flexibility on the back end because it is now easier to over-provision resources which helps improve overall utilization of the hardware resources. And, this improved utilization enables an improved thermal footprint in the data center, meaning that there is reduced power consumption for compute power and most likely reduced cooling needs as well.

Ideally, that flexibility provides yet another lesson which is applied in cloud computing. Specifically, that separation of the physical resources from their virtualized counterparts allows for some additional management benefits within the data center which implements a cloud computing. And, the proof points with current providers validate that the technologies have advances to the point where we can separate the physical resources from the virtual resources. VMware's VMotion product provides some hints as to what might be possible in that space if correctly harnessed under cloud computing.

Further, recent advances in Linux which allow the dynamic addition of memory or processors to a physical machine, and the ability to pass those capabilities on to an instance of Linux running in a virtual machine, enable a level of elasticity in the hardware which will ultimately enable the most efficient use of resources in cloud computing.

2.9 Software Oriented Architecture (SOA)

Rather than extol the virtues of SOA, we'll suffice to say that SOA is sufficiently prevalent within Enterprises today that any solution which claims to be Cloud Computing must clearly enable SOA as a model for application

deployment. Clouds should be effectively deployment platforms which enable advanced models such as SOA. And, based on our earlier look into provisioning and our quick looks at Software and Hardware as a Service, it should be clear that the ability to deploy components of software applications as virtual appliances would be an extremely powerful building block for creating a cloud of applications and services.

3 If that's not Cloud Computing, What is?

Thus far we have avoided any real discussion of what Cloud Computing is, although we've dropped a few hints along the way. But why has the buzz on cloud computing started now? What has made this topic jump so quickly to prominence? To understand that, let's look at a few of the pressures on data centers, compute power, and the challenge of creating and deploying new applications quickly.

3.1 First, how did we get here?

The rise to prominence of Cloud Computing stems from several sources. One is clearly related to the marketing hype engines that are always looking for something new. Directly related to that, Google's CEO, Eric Schmidt, Ph.D., has been widely quoted on his definition, specifically:

It starts with the premise that the data services and architecture should be on servers. We call it cloud computing—they should be in a *cloud* somewhere. And that if you have the right kind of browser or the right kind of access, it doesn't matter whether you have a PC or a Mac or a mobile phone or a BlackBerry or what have you—or new devices still to be developed—you can get access to the cloud...

This is a powerful vision and it is possible to see some of the same goals as utility Computing, grid computing, Application Service Provider(s) (ASPs), and Software as a Service (SaaS) embodied in this thinking, as well as much more. For instance, the ubiquitous access to software and applications from hand held devices is contained within this vision as well. The definition provides

a powerful vision as well as something of a marketing and hype alignment vehicle, without getting into any of the pesky little details that go with that vision.

However, even that short vision doesn't go into *why* cloud computing might be useful. And to do that, we have to look to the Enterprise Data Center as well as into the pressures that are inhibiting innovation from small companies and individuals.

3.2 The IT Crisis

We have been approaching a crisis in Information Technology for quite a while now, and that impending crisis has been forcing a lot of innovation into methods of avoiding that crisis. What crisis, you ask? Okay, it may not be a real *crisis*, per se. But, throughout the IT industry, the cost of managing hardware, operating systems, and applications has been on the rise for a long time. In fact, that cost has gone up to the point that some analyst figures suggest that IT management costs in the data center range from 25 to 45 percent of the total IT budget.³ That means that money which could otherwise be targeted towards specific new development in support of a company's value-add is instead going directly towards maintenance of just the server and software environment they already have. That directly limits a company's ability to invest in innovation or increased capacity in a way that impacts a company's bottom line. What kind of solution would substantially reduce those IT management costs?

Or, viewed from another direction, the cost of supplying existing servers with electricity and cooling data centers has escalated to the point that energy costs are now exceeding the costs of the actual hardware in many data centers. And, those power costs are often going to support capacity that is only needed in peak situations or sometimes to support failover capacity. As a result, data centers are spending a lot of money on equipment and power which is not directly contributing to the company's revenue most of the time. This waste capacity flies directly in the face of the *Green* movement as well—data centers that are only 10 to 20 percent utilized are still consuming precious raw materials and contributing to pollution and global warming without much value-add to organizations. What if there were

³I even saw a recent Microsoft presentation on Hyper-V indicating that number was as high as 70%!

a paradigm shift which substantially reduced the waste computing cycles, or enabled better sharing of existing computing cycles?

And, viewed from a third perspective, the time to develop and deploy new applications has been rising, especially as space, access to power, and complexity of the development environment has increased. Many innovators today have to request new hardware from their IT department; that hardware needs to be ordered, and, when it arrives, it has to have a place to be installed that has sufficient space, power, and cooling. Once installed, the OS must be installed, key applications must be identified and installed, and, oh, make sure that all the applications you've selected are inter-operable, and then, at last, you are ready to begin development. That cycle in many companies, both small and large, often approaches 3 or in some cases, even 6 months from "idea" to "ready to develop." What if that development cycle could start within hours, either based on a commercial provider's offering of compute cycles, or even your own enterprise's existing compute resources?

Cloud computing provides attractive answers to all of these scenarios. Now if only we could figure out the definition of cloud computing that provides all those answers!

4 Cloud Computing: A Vision

In many ways, the Internet provides an ideal model for Cloud Computing. The Internet provides bandwidth to everyone and happens to hide nearly all of the details of the underlying mechanism of the hardware providing that access to network bandwidth. From a user perspective, compute power should ideally be as ubiquitous for the end user as network bandwidth is today. In fact, some people have suggested that there is a Cloud with a capital *C*, just as there is an Internet with a capital *I*.

A simple way of stating this would be that "Cloud Computing provides ubiquitous access to compute resources for any user, anywhere."

Okay, so that's a pretty simple vision statement, but what does it say for people who want to do more than access a web service (we can already do that) or instantiate a pre-wrapped appliance on a utility computing service provider's platform? What is the development model? How do I get access to a machine to do proprietary development? How do I handle the set up and installation,

configuration, and management of the unique development environment that I need for my development? How do we handle licensing for those software applications that aren't part of the open source ecosystem—or even those that are part of the open source ecosystem but still have maintenance fees and licenses for operation?

As an example, suppose I wanted to run Red Hat Enterprise Linux with Oracle and Rational ClearCase® installed? What if I want to implement a 3-tier database, middleware, web server environment? Our vision is a little bit lacking on the finer details of how cloud computing is actually deployed and made available.

Further, suppose that I want to be a cloud computing provider or perhaps even share some of my existing compute capacity with other people—maybe even charging some minor fee for access to my unused cycles? I guess the vision leaves out some of those details as well. In fact, given the vision and the hype, it is really unclear as to whether or not I could even be a cloud computing provider. Clearly, we need some more definition about what clouds actually are, how one creates them, maintains, them, and what a cloud actually can provide today.

4.1 Vision: Meet Reality

It is time to separate our cloud into two distinct points of view. One point of view will focus on what services and capabilities a cloud *provides*. The other point of view will look at what technologies are present within a cloud. These two views will allow us to distinguish a user of a cloud from a maintainer of a cloud. And, buried in this analysis is the assumption that there will be, for the foreseeable future, more than just one cloud in the sky. Specifically, much like our earlier analogy of the Internet, the cloud will be the composite view of all of those individual clouds which will initially spring up in isolation. Following the existence of many clouds, there is the hope that someday, much like the view of Grid computing, all of the providers will be loosely connected, again like the Internet, to make a single, ubiquitous view of a single, well-connected cloud.

4.2 What services does the cloud provide?

This is the easy question, the question that is the most visible to end users. Specifically, the cloud provides a

set of services in the form of utility computing, or even grid computing to end users. Those services could, in theory, be physical hardware or virtual hardware. They could be operating system instances or virtual appliances. They could be operating systems instances with a catalog of software which can be easily installed on them, or which can receive custom software written by and provided by the end user. They could provide for simple Internet connectivity or perhaps they even provide some level of access to virtual private networks (VPNs) or virtual LANs (vLANs) so that the end user could deploy multiple servers cooperating with some level of security protection to help in isolating proprietary data.

These clouds could provide access via the Internet—but they could also be wired directly into private intranets, either physically or virtually, enabling the applications running on the cloud to have access to data or applications residing in an end-user's internal, private network. The Cloud provides an adjustable number of resources, be they physical machines or software appliances, where the user can adjust the number of machines running their workload based on demand.

Of course, as we expand the level of definition of clouds, the astute observer will question whether these clouds provide enough security or reliability to satisfy all users. For instance, would two Wall Street trading companies both put their private applications and data on the same cloud? Do we have strong enough security isolation in place today in our operating systems, hypervisors, virtual LAN technologies, virtualized storage access to enable true and safe isolation between competitors?

What about the latency of access? If this cloud is “just out there” somewhere, how long does it take to get data between any set of applications in the cloud or between the cloud and other internal machines? What is the bandwidth between your machine and the cloud's environment—will I get the bandwidth and latency that I need for *my* application from the cloud today? Does it really have the ability to provide the services I need with the security, performance, bandwidth, latency, and availability that I need from my provider of ubiquitous computing?

Of course, the answer to that last question is a bit “It Depends.” For some workloads today, clouds as a flexible, elastic provider of utility computing will do just fine. The same is true for Grids today, and that is why

they are heavily used by some workloads, most commonly scientific workloads. Clouds may provide a bit more flexibility in terms of the workload supported today, but there is a long way for clouds to evolve before they are ready to support the needs of all consumers of the cloud computing resources.

Of course, the answers to some of those security, availability, performance, latency, and bandwidth questions might change if an enterprise could effectively build its own, in-house cloud. With direct access to their local SAN, with access to some of the business services that may not be hosted in the cloud, such as their print services, their LDAP services, their nearby connections to desktops, etc., some of these problems that aren't resolved globally may be addressed in a more localized implementation of clouds. We'll come back to that problem more in a little bit.

Finally, a well designed cloud based environment can enable a variety of scenarios for the end users of the cloud, including test and development configurations; the ability to deploy SaaS; the ability to deploy HaaS, aka Platform as a Service (PaaS); the ability to deploy SOA components; the ability to deploy virtual worlds or gaming environments on demand; and many more common workloads.

4.3 What does it take to *build* a cloud?

This question gets a little harder to answer, and most of the common cloud implementations today are managed by top-notch IT staff, explicitly hiding the details of what goes into making a cloud so that the consumers don't have to deal with it at all. But if someone wants to build their own cloud, they'll have to have a firmer grasp of exactly what goes into a cloud and what components they will need to build or assemble, along with some idea of what the cost for managing that infrastructure is going to be.

For simplification, I'm going to suggest a rough blueprint for what components go into a Cloud. It is definitely possible to vary from that blueprint, and to optimize within the blueprint and potentially still be a cloud computing environment. However, this blueprint should enable you to decide what components you may need to have on hand to build a cloud or to evolve your own computer center into a cloud configuration.

4.3.1 Virtualization

At the very core of cloud computing, I'm going to start with what could be a contentious choice but I'll spend more time justifying that choice later in the paper. That first choice is that any good cloud in this point in time should be built on top of a virtualized platform and that all resources in the environment should be virtualized. This includes not just the platform, but storage and networking as well. While it is possible to get the appearance of having a cloud without virtualization, I'm going to go out on a limb and suggest that non-virtualized solutions have limitations in the flexibility that will in time become a hallmark of cloud computing. This means that the base platform in the case of, say, an Intel® or AMD®-based processor should be virtualized by something like VMware®'s ESX®, Microsoft®'s Windows Server 2008®Hyper-V®, some version of Xen™, or any similar hypervisor. For non-Intel/AMD based machines, such as IBM®'s POWER®family of processors, PowerVM®⁴ would be an appropriate choice and the IBM mainframe provides virtualization in several forms, such as z/VM® or z/OS®.

Today, most cloud-like deployments are based on a single underlying class of platform, although most enterprises are made up of highly heterogenous environments. In an ideal world, the cloud will include all of those platforms as the basis for cloud computing, and the greater vision clearly postulates that support for heterogenous platforms over time. But for our initial blueprinting activity, we'll start with the simplifying assumption that all of the machines are of relatively the same type, and, more importantly, can all run the same hypervisor.

4.3.2 Virtual Appliances

Next, we need a repository of applications to deploy in our cloud. And, since our cloud need not be restricted to the applications that someone else has created, we will need tools to somehow package those applications for deployment within our cloud. For that, I'd recommend that we start with virtual appliances, which are essentially a packaged version of software and an operating system, ready to run on a hypervisor. A number of companies have started down this path, including

⁴http://en.wikipedia.org/wiki/Advanced_Power_Virtualization

VMware and their appliance marketplace⁵ or companies like rPath™⁶ and their rBuilder™ tool⁷ and their appliances.⁸

Since these are virtual appliances, they need to be built for a particular type of hypervisor, be it a Windows/VMware image or a Linux/Xen image. This is where a simpler environment makes it easier to build your own prototype—the more hypervisors that your virtual appliance needs to support, the more complex it is to create a cloud environment. Constructing these virtual images is one of the more challenging aspects, although luckily there is a lot of experience in this space now, with more emerging all the time. Most cloud-like environments today are building multiple versions of virtual appliances from the same sources, such as a Windows/VMware and Linux/Xen at the same time, which generally ensure that any of the appliances built at that time are as close to identical as is reasonably possible.

Another aspect to building these appliances is to understand what format they will be created in. A glance through the rBuilder appliances mentioned earlier shows that today it is possible to build in at least a dozen formats, and that just covers Intel/AMD platforms! If you wanted to build for other processor types or hypervisor technologies, that number will go up from there. Luckily the market will likely resolve this issue one way or the other before long—either a few key virtualization technologies will emerge, or the tools will evolve to support builds for multiple environments at build time.

The VMware and Xen communities are looking into using the Open Virtualization Format as a wrapper format for virtual appliances and that format is being broadly standardized by the Distributed Management Task Force (DMTF).⁹ There is a proposed project to create open source tools for managing OVF files that will hopefully be under way by the time this paper is published.¹⁰

⁵<http://www.vmware.com/appliances>

⁶<http://www.rpath.com>

⁷<http://wiki.rpath.com/wiki/rBuilder>

⁸http://wiki.rpath.com/wiki/Virtual_Appliances

⁹http://www.dmtf.org/newsroom/pr/view?item_key=3b542cbc5e6fc9ede97b9336c29f4c342c02c4e9

¹⁰<http://code.google.com/p/open-ovf/>

4.3.3 Provisioning

Once you have built a virtual appliance, you also need to have some understanding of how to provision that virtual appliance. Provisioning in this context means that the appliance will need to be able to be installed on your virtualization platform. Luckily, the OVF format described above has a place within the file which allows the builder to store some basic information about how to set up, configure, and deploy the virtual appliance. This might include things like configuring virtual LANs, configuring access to local or shared storage and any other configuration related to the virtual environment (such as how the domU¹¹ is configured in Xen, or how to boot the virtual machine).

Provisioning can also be expanded beyond the basic deployment of a single virtual appliance containing a single application stack to deploying either multiple copies of the same virtual appliance or more complex sets of virtual appliances. As an example, it would be possible to provision a three-tier application, such as a database appliance, a middleware appliance, and a web front end appliance. This allows for appliances to be created as building blocks and deployed in sets based on a specific need. This allows for a level of customization without the need to build a large number of highly specific virtual appliances. Also, it allows for some elasticity in the number of appliances deployed—for instance, in the database, middleware, and web front end appliances—it would be easier to deploy additional web front-end appliances as the workload increases, or additional middleware appliances depending on the type of workload.

OVF also has the ability to store multiple virtual appliances in the same wrapper, including all of the instructions to deploy the full set of appliances. To date, there is no solid provisioning tool which generally decodes that information, although the open source OVF tool will help extract that information soon. There are also some tools such as TPM which could take that input and convert it into a provisioning flow. There are limited tools today which provision virtual machines, although one interesting one comes from 3tera's¹² AppLogic.¹³ AppLogic allows for the provisioning of complex workloads onto a physical or virtual machine environment.

To date, there are very limited open source projects in this area, though, and it is an interesting area for additional development work.

4.3.4 Virtual Appliances Catalog

Once we have a set of virtual appliances, we need a place to put them. While a small set of appliances can be kept on a laptop or other location, ideally we would like to create a repository of these applications which can be provisioned by end users on request. Also, we would like for our end users to be able to store their own applications in the virtual appliance catalog. This repository of appliances could be something as simple as a directory with virtual appliances in it, sorted by name. Or, it could be a complex hierarchy of images built for a variety of virtualization environments. Again, for simplicity, I'm going to recommend the flat directory, possibly with a simple web-based front end to view those images. Ideally, that web front would allow end users to select one or more images to deploy to the set of machines in your cloud. These machines would all have a hypervisor installed on them already, and your provisioning software would scatter the virtual appliances intelligently amongst your physical resources.

With some additional intelligence in the deployment software, your virtual appliance catalog could contain images that worked on multiple hypervisors or multiple machine configurations. In particular, the OVF format contains information that identifies the environments to which that virtual appliance can be deployed. That would allow you to deploy virtual appliances to a variety of hardware, provided that you had either built each virtual appliance for multiple platforms.

Now, for simplification, I proposed the virtual appliance catalog as something containing just virtual appliances. However, it would be possible to also have that catalog contain base virtual appliances, perhaps a distribution trimmed down to just what is necessary for supporting a software stack, and a set of applications which could be streamed to the pre-built, pre-defined software appliances. This configuration would provide a bit more flexibility in terms of building blocks and perhaps reduce the number of specialized virtual images slightly while increasing the flexibility of those environments. That may be as simple as today's ability to use a package manager or install tool on a running image. Or it

¹¹domU is Xen's name for a guest operating system.

¹²<http://www.3tera.com>

¹³<http://download2.3tera.net/demo/applogic20demo.html>

might be something more like a pre-installed application built as a union filesystem image, which could simply be mounted on top of an existing appliance. The latter would allow more rapid provisioning of images and ideally leave less configuration to the end user.

4.4 Cloud computing is really just that simple?

Is that all there is? Some hardware, virtual appliances, a catalog, and an ability to provision? From a user perspective, yes, that is one view and it is sufficient to provide a basic cloud configuration. And, this is the basis for Amazon.com's Elastic Compute Cloud (EC2)¹⁴ and IBM's internal Research Compute Cloud (RCC). This is also a solution which is being rolled out to a number of other sites by an IBM team.¹⁵¹⁶

However, there is another view of cloud computing as described by Google and IBM.¹⁷¹⁸ In this view, the focus is on a shift in the programming paradigm to solve more problems using a *huge* number of computers, a la Google's infrastructure. Again, this starts with some hardware, in this case, lot of it; one or more applications, although not necessarily in the form of appliances this time; an ability to provision a parallel style of workload; and, more uniquely, a variant of Google's internal MapReduce¹⁹ algorithm, potentially based on the open source Hadoop²⁰ code. This code enables the workload to be divided quickly among hundreds or thousands—or even tens of thousands of machines—enabling some forms of complex, data intensive processing to be smashed into thousands of very small workloads which can return results in a fraction of the time.

In some ways, this model is a variation on distributed computing, which allows workloads to be partitioned into smaller workloads and distributed to a large number of machines. But in this variation, the work is par-

titioned among a set of machines where multiple machines may compute overlapping results. A “reduce” step winnows out the duplicates, providing a single set of results to the end user. Many believe that this workload will be one of many which take advantage of the forthcoming cloud environments.

4.5 So why all the hype if that's all there is?

So far, we've covered the basis of what make up a localized cloud environment. This definition relies on technologies that mostly exist today and integrates them in a way that makes the deployment of some workloads substantially simpler. But this simplified view is only a subset of the grand vision that many believe is the real direction for cloud computing. In particular the grand vision implies substantially more capability and sharing between the relatively smaller clouds proposed here. However, there are a number of technology gaps between this relatively modest proposal and the grand vision. We'll look at a couple of those gaps here.

4.5.1 Managing thousands of machines

One of the short term challenges for the most basic clouds, as well as a challenge to expanding towards the grand vision of cloud computing comes down to the relatively simple issue of just how to manage all of the computers in a data center. While many view cloud computing by its usage model of effectively providing utility computing, relying on just a few providers with excellent systems administrators to provide all of that capacity is not a very scalable model. And, while the predictions of only five computers in the world have run from the 1950's or 1960's²¹ to the current day,²² the reality is that, by at least some estimates, there are over 25 million servers in the market as of 2005²³ and even if there were a widespread conversion overnight to use one of five mega-datacenters, the time (not to mention the cost) for conversion would be overwhelming. And there is a psychological factor at work as well: most companies aren't ready to trust their core intellectual property

¹⁴<http://www.amazon.com/ec2>

¹⁵<http://www-03.ibm.com/press/us/en/pressrelease/23426.wss>

¹⁶<http://www-03.ibm.com/press/us/en/pressrelease/23710.wss>

¹⁷<http://www.ibm.com/ibm/ideasfromibm/us/google/index.shtml>

¹⁸http://www.google.com/intl/en/press/pressrel/20071008_ibm_univ.html

¹⁹<http://labs.google.com/papers/mapreduce.html>

²⁰<http://hadoop.apache.org/core/>

²¹http://en.wikipedia.org/wiki/Thomas_J._Watson#Famous_misquote

²²<http://www.guardian.co.uk/technology/2008/feb/21/computing.supercomputers>

²³<http://www.itjungle.com/tlb/tlb030607-story04.html>

in the form of data or applications to a third party to host and manage.

So, for the near term at least, the proliferation of clouds will evolve, in my own estimation, from existing data centers into a more global cloud computing IT vision. This migration is akin to the growth of the Internet from its rather humble beginnings at the end of the 1970's through the end of the 1980's, where data centers—mostly in the form of university computing centers and some DARPA²⁴ sponsored sites—were among the first to connect computers locally in a data center with high speed networks, providing the benefits of well connected intranets long before the Internet was available. In fact, the evolution started by creating slow speed connections (based on store and forward technologies like UUCP²⁵ or BITNET²⁶) which provided loose connectivity between well connected, locally managed compute resources. Another parallel between the expected evolution of cloud computing and the Internet can be seen in the handling of security concerns. The first UUCP capabilities were used as a simple mechanism for authenticated users to move files around or to exchange data via email. Later, when ARPANET²⁷ and its successor, NSFNET²⁸, allowed more direct access, a number of secured services such as remote shell (rsh), remote copy (rcp), or unsecured services such as finger and fingerd, whois, date and time servers, and basic domain name services, the Internet allowed select services to be offered, typically at no charge to arbitrary users. The evolution of TCP and IP helped bring the Internet to the point that it was merely a conduit for any type of data that any user might want to publish or consume.

Today any number of providers enable end users to consume compute resources along the lines of specific services, from SaaS providers or even services such as Wikis or customizable home pages. However, the visible emergence of Amazon.com's EC2 and some forms of hosting providers starts to show how shared resources can be made available, potentially with charge back. The percentage of compute resources currently available through such providers is still extremely small—some projections suggest only a few thousand machines

²⁴US Defense Advanced Research Projects agency, <http://www.darpa.mil/>

²⁵<http://en.wikipedia.org/wiki/UUCP>

²⁶<http://en.wikipedia.org/wiki/BITNET>

²⁷<http://en.wikipedia.org/wiki/ARPANET>

²⁸<http://www.nsf.gov/about/history/nsf0050/internet/launch.htm>

are available through Amazon.com, for example. And, one of the key reasons for this, in my estimation, is that the management software has not evolved to efficiently and effectively managing large groups of machines to enable sites to develop cloud computing internally, and the sheer quantity of machines that would be needed to support the entire compute capacity of the world—or even some large percentage of that capacity.

While the full gamut of management complexities are too extreme to dive into in detail here, a few of the highlights include such simple things as hardware management, operating system management, network and storage connectivity management, application management, security management, availability management and energy management, to name a few. Often, each of these are managed independently for each machine and each workload in a data center. Or, in some of the best practices, machines are grouped for ease of management, applications are grouped for simplicity, security policies are centralized for consistency, and often emerging capabilities such as energy management are added as an afterthought. Without substantially improved practices for enterprise and data center management, cloud computing as a grand vision will remain as just a dream.

4.5.2 Provisioning Challenges

Earlier we mentioned provisioning as one of the key components of cloud computing. And, honestly, provisioning is really in its infancy, despite a few companies and projects which have started to address the problems. And, many of those companies and projects have focused on relatively specialized solutions which will not grow up to Internet-scale solutions. I firmly believe that the emergence of cloud computing as a buzz word and as a vision is strongly propelled by the predominance and rising eminence of open source software, including Linux and a variety of key open source components. However, until there is a world-class open source project for generalized, complex provisioning, clouds will evolve from within enterprises and from within a set of localized utility computing providers using only proprietary provisioning technologies.

Today, 3tera's AppLogic or IBM's TPM provide some highly evolved mechanisms for provisioning and deployment. However, as licensed applications, they will be focused on deployments within enterprises and larger

data centers. These and similar products will aid in the evolution of clouds within the data center over the next several years, but the evolution of a project like SystemImager,²⁹ the new xCat³⁰ or improvements to RPM (the RPM Package Manager) or APT (the Advanced Packaging Tool) may help increase the ability to provision virtual machines or deploy virtual appliances. However, provisioning includes the ability to configure networking, perhaps as VLANs or VPNs, and includes the set up and creation of storage. And, being able to deploy those for any possible user-defined workload is still too complex for wide-spread adoption. And, generally, they don't deal with the security implications for proper isolation of workloads that are needed to make sure that I, as a user, am unable to access your proprietary data or applications.

4.5.3 Security Challenges

As alluded to above, security is another of the major inhibitors to true cloud computing today. While some workloads encourage sharing of data, such as Wikis, the actual installation and management of the software application typically needs to be restricted to the administrator of that software. But with utility providers' compute resources being completely accessed through the Internet, any applications typically have Internet access as well. Setting up and configuring VPNs or VLANs requires custom administration by the creator of the virtual appliance or via custom systems management by the person deploying the virtual appliance. Today, enterprises and even small business owners employ as much physical security and firewalling technology as possible to protect their business from crackers.³¹ Setting up equivalent security at a remote internet site is not a well-practiced art today. It may be a several years until best practices emerge and those practices have withstood the test of time.

In addition to the networking component of security, an earlier postulate was that virtual appliances would be instantiated on hypervisors or virtualized platforms. However, today the leading hypervisors do not approach the level of security and isolation between guests that

is present in physical hardware isolation. While there is work going on in both VMware and Xen, for instance, to improve the security isolation between guests, the ability for a guest to "escape" to the hypervisor and from there have access to other guests is a concern that is likely to prevent competitors from sharing the same physical hardware. That level of isolation will increase over the next few years, making increasing levels of multi-tenancy—the ability for diverse guests to share the same hardware—more secure over time.

4.5.4 Other Challenges

While there are a number of other challenges to address, such as how to handle licensing and cost recovery in a virtualized environment, or improving the management of virtual appliances, or making the virtualized environment more dynamic through the capability of live guest migration, those issues are likely to get worked out as utility computing becomes more common, as organizations built their own internal clouds, and virtual appliances become more common.

5 Conclusions and Outlook

Cloud computing clearly has a lot of hype now, and the vision as represented here can be very compelling. The ubiquitous access to computing resources, much as the Internet has provided us with ubiquitous connectivity, is a powerful vision for our future. Clearly, most of the technology to achieve this vision exists in some form today, which is what makes the vision so catchy in the press today. However, as shown here, we can implement subsets of cloud computing today and we can begin to migrate data centers towards a model which will allow the free flowing access of compute resources within a cloud in the next few years. In the meantime, there are a number of areas that need some additional focus to make this vision a reality.

Those challenges include dramatic simplification in the management of physical compute resources, improvements in virtualization and the management of virtual environments, the creation of a pervasive and accessible set of tools to deploy virtual appliances and workloads within a cloud of virtualized resources, improvements in provisioning of networks and storage, and continued work on improving the security provided by hypervisors

²⁹http://wiki.systemimager.org/index.php/Main_Page

³⁰<http://www.xcat.org/>

³¹[http://en.wikipedia.org/wiki/Hacker_\(computer_security\)](http://en.wikipedia.org/wiki/Hacker_(computer_security))

at a minimum. As those are evolving, there is the ability to improve workload management, energy management and availability management. Improvements in all of these areas should also improve the efficiency and utilization of computers resources. Resources should ultimately be more effectively shared, and additional resources would be available on demand for those workloads which dynamically need access to more hardware than they would otherwise have at hand.

The era of cloud computing is just beginning.

Legal Statement

© 2008 IBM Corporation Permission to redistribute in accordance with Linux Symposium sub-mission guidelines is granted; all other rights reserved.

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

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

Introducing the Advanced XIP File System

Jared Hulbert

Numonyx

jaredeh@gmail.com

Abstract

A common rootfs option for Linux mobile phones is the XIP-modified CramFS which, because of its ability to eXecute-In-Place, can save lots of RAM, but requires extra Flash memory. Another option, SquashFS, saves Flash by compressing files but requires more RAM, or it delivers lower performance. By combining the best attributes of both with some original ideas, we've created a compelling new option in the Advanced XIP File System (AXFS).

This paper will discuss the architecture of AXFS. It will also review benchmark results that show how AXFS can make Linux-based mobile devices cheaper, faster, and less power-hungry. Finally, it will explore how the smallest and largest of Linux systems benefit from the changes made in the kernel for AXFS.

1 Filesystems for Embedded Systems

1.1 Why not use what everyone else uses?

Embedded systems and standard personal computers differ a great deal in how they are used, designed, and supported. Nevertheless, Linux developers tend to think of the Flash memory in an embedded system as the equivalent of a hard disk. This leads many embedded Linux newbies to ask, “Can I mount ext2 on the Flash?” The simple answer is yes. One can mount ext2 on a mtd-block partition in much the same way that one can bathe in a car wash. However, for both the car wash and the MTD, this is not what the system was designed for and there are painful consequences—however, they get the job done... well, sort of.

There are a few key differences between filesystems used in a personal computer and those used in embedded systems. One of these differences is compression. Many filesystems used in embedded systems support

compression of file data in 4KB–128KB blocks. Cost, power, and size limitations in embedded systems result in a scarcity of resources. Compression helps to relieve some of that scarcity by allowing the contents of a usable rootfs image to fit into a reasonably sized Flash chip. Some embedded filesystems make use of the MTD device API rather than a block device API. Using an MTD device allows the filesystem to take advantage of special characteristics of Flash and possibly avoid the overhead of the block device drivers. A third difference is that a read-only filesystem is perfectly acceptable for many embedded systems. In fact, a read-only filesystem is sometimes preferred over a writable filesystem in embedded systems. A read-only filesystem can't be corrupted by an unexpected loss of power. Being read-only can offer a bit of security and stability to an embedded system, while allowing for a higher performance and space-efficient filesystem design.

Linux filesystems that are well suited to the needs of embedded systems include CRAMFS, JFFS2, SQUASHFS, YAFFS2, LOGFS, and UBIFS. Only two of these filesystems have been included in the kernel. CRAMFS is a very stable read-only filesystem that supports compression and mounts from a block device. There is an out-of-tree patch set for CRAMFS which enables it to run XIP with no block device. If you need to write data on Flash, the only option in the kernel is the mature JFFS2. Like JFFS2, YAFFS2 requires an MTD device, but does not provide compression like JFFS2. YAFFS2 has been around for several years, but getting it into the Linux kernel does not seem to be a priority for the developers. SQUASHFS is over 5 years old and is included in nearly every distribution. While the developer has made attempts to get it pushed to mainline, those attempts have been sidelined by a surprising amount of resistance. As an improvement over CRAMFS, SQUASHFS is capable of creating larger filesystems at higher compression ratios. LOGFS and UBIFS are projects with the same goal, providing more scalable, writable Flash filesystems for growing NAND

storage in embedded systems. Both support compression and both are trying hard to be included in the kernel by 2.6.26.

1.2 The inconvenient filesystem

The linear XIP CRAMFS patches have proved useful in many small Linux systems for over 8 years. Unfortunately, the patched CRAMFS contains calls to low-level VM functions, a mount option to pass in a physical address, and modifications to the virtual memory code. The main cause of this hubris is that this patched CRAMFS doesn't fit the filesystem paradigm and therefore doesn't fit the infrastructure. The result is an ugly hack and a maintenance nightmare. The patch set broke badly almost every year due to some change in the kernel, because it messed with code which filesystems have no business touching. Not only is XIP CRAMFS hard to maintain and port, it also has serious limitations. CRAMFS only supports ~256MB image sizes and the maximum file size is 16MB. Notwithstanding these limitations, the linear XIP patches to CRAMFS have been included in most embedded Linux distributions for years. Variations on linear XIP CRAMFS ship in millions of Linux-based mobile phones every year.

Embedded Filesystem Summary					
filesystem	compress	MTD/block	writable	XIP	in-kernel
CRAMFS	✓	block	✗	✗	✓
JFFS2	✓	MTD	✓	✗	✓
YAFFS2	✗	MTD	✓	✗	✗
SQUASHFS	✓	block	✗	✗	✗
LOGFS	✓	MTD	✓	✗	soon
UBIFS	✓	MTD	✓	✗	soon
XIP CRAMFS	✓	✗	✗	✗	✗

2 Changing the Filesystem Paradigm

2.1 The current filesystem paradigm

Performing any real operations on data requires the data to be in memory. Executing code also requires that it be in memory. The role of the filesystem in Linux is to order data into files so that it can be found and copied into memory as requested. In a typical personal computer, a Linux filesystem copies data from a hard disk to RAM. The Linux kernel filesystem infrastructure assumes that data must be copied from a high-latency block device to the low-latency system RAM. While there are a few minor exceptions, this is the rule. This is the basic filesystem paradigm driving the architecture of the Linux virtual filesystem.

A typical embedded system today might have an ARM processor with some embedded Flash memory and some RAM. The filesystem would copy data from Flash memory to RAM in this case. While Flash memory has much faster read latency than a hard disk, the basic paradigm is the same. Linux developers tend to think of the Flash memory in an embedded system as the equivalent of a block device like a hard disk. This fits the filesystem paradigm built into the kernel, therefore few kernel developers care to investigate whether the paradigm fits the realities of the hardware.

2.2 Why XIP?

What is so useful about the XIP-patched CRAMFS that has prompted it to be haphazardly maintained out-of-tree for nearly a decade? What is it about these patches that make them so hard to reconcile with the kernel? The answer to both is eXecute-In-Place, or XIP. Executing a program from the same memory it is stored in is usually referred to as XIP. As code must be in memory to be executed, XIP requires a memory-mappable device such as a RAM, ROM, or a NOR Flash. NAND Flashes are not memory-mapped and thus not suitable for XIP; they are more like block devices than RAM.

XIP is common in RTOS-based embedded systems where the memory model is very simple. In a RTOS, to add an application from Flash into the memory map, the designer need only specify where in the Flash the application is, and link the application there during the build. When the system is run, that application from Flash is simply there in memory, ready to be used. The application never needs to be copied to RAM.

A Linux application in Flash is a file that would be contained in a filesystem stored on the Flash. To get this application from Flash into a memory map, individual pages of the application would be copied into RAM from Flash. The RTOS system would only require the Flash space necessary to contain the application. The Linux system would require the Flash space necessary to store the application, and RAM space to contain the application as it gets copied to RAM. The Linux filesystem paradigm treats the Flash as a block device. The Flash-as-block-device paradigm overlooks the memory aspect of the Flash *memory*. The result is wasted resources and higher cost.

If we have in our embedded system a Flash that can be used as memory, why not use it as such? When such

Flash is used as memory, the system can use less memory by removing redundancy as explained above. Using less memory results in reduced cost, which is always a priority for consumer electronics. Reduced memory also reduces power, which increases battery life. Performance can also be improved with XIP. If an application does not need to be copied to RAM nor decompressed—only pointed to—to be used, the paging latency is drastically reduced. Applications launch faster with XIP. Where it can be used, XIP is a great way of improving on system cost and performance. For years the only option was to depend on the limited and hacked linear XIP CRAMFS patches.

2.3 Expanding the paradigm

The first step toward consolidating the XIP features of CRAMFS used in the smallest of Linux system into the kernel came from an unlikely source, one of the largest of Linux systems. As part of the 2.6.13 merge, the s390 architecture tree introduced a block driver for dcss memory that extended the block device to include a `.direct_access()` call. This new interface returns an address to a memory region that can be directly accessed. It allows for XIP from memory which is posing as a special block device. To complete the system modification to ext2 were made, and the functions in `/mm/filemap_xip.c` were introduced to allow data on this dcss memory to be used directly from where it was stored. The s390 architecture users find this feature very useful because of the way their systems allow for Linux virtualization. Because many virtual systems were sharing a root filesystem, requiring each system to maintain copies of important files and code in a page cache when it was already accessible in a shared memory device would be a huge waste of resources.

With these changes to the kernel, the filesystem paradigm changed a bit. Data no longer had to be copied from a block device into RAM before being used; the data that is stored in a special memory device can be mapped directly. While the embedded Linux world continued to fumble with the hacked CRAMFS patches, the mainframe Linux developers laid the foundation for merging XIP into the kernel.

3 Why a new filesystem?

3.1 The Problems

In order to take advantage of the memory savings and performance benefits that XIP has to offer, Linux needed a few more tweaks and a filesystem. Although the `/mm/filemap_xip.c` infrastructure was a step in the right direction, it did not address all the problems with adding XIP functionality for embedded systems. The changes introduced by `/mm/filemap_xip.c` added a new function, `get_xip_page()`, to struct `address_space_operations` that a filesystem was supposed to use to pass a `struct page` for the memory that was to be inserted into a process's memory map directly. In an embedded system, the memory that is to be passed is Flash, not RAM, and has no page associated with it. The way the XIP CRAMFS patches handled this was to call `remap_pfn_range()`. This was one of the causes of the maintenance problems with the patches. Because the API for doing this was intended for limited use in drivers and internal memory management code, not for filesystem interfacing, it changed relatively often. A solution would need to be found that modified the infrastructure from `/mm/filemap_xip.c` with the functionality enabled by calling `remap_pfn_range()` directly.

With no `struct page` to leverage in mapping memory, the kernel would need the physical address of the memory to be mapped. The XIP CRAMFS patches solved this by requiring a `-o physaddr=0x...` at mount. While this approach works, it violates some of the layering principles Linux developers try to enforce. This approach required the filesystem to deal with hardware details, the physical address of a memory device, which are supposed to be handled by driver levels. There were also conflicts with the `ioremap()` call in the filesystem which mapped these physical addresses into kernel addressable virtual addresses. There were some rather important architecture-specific `ioremap()` optimizations controlled by `#ifdef`, creating more confusion and calling more attention to the layer violation.

Analyzing and comparing systems with and without the XIP CRAMFS patches lead to a discovery. Under some circumstances, XIP CRAMFS would indeed save RAM, but it would do so spending more space in extra Flash than was saved. One secondary reason for this mismatch was that XIP CRAMFS had a rather inefficient

way of mixing XIP and non-XIP files. XIP files must be aligned on Flash at page boundaries in order for the memory to be directly inserted into the memory map. XIP CRAMFS left possible alignment holes at the beginning and end of each XIP file. The major cause of this skewed exchange rate was that XIP CRAMFS uncompressed entire files even if only a small part of that file was ever mapped. In a non-XIP system, only the pages that actually did get mapped would be copied into the RAM. To realize true memory savings, a solution would need to be able to identify and XIP at a page granularity rather than a file granularity. Unfortunately the kernel internals capable of inserting physical pages, not backed by a `struct page`, did not allow page-by-page granularity.

If any effort was to be expended on creating a mainlineable XIP filesystem solution, one could not ignore the limitations of CRAMFS. 256MB is painfully close to today's largest NOR Flash chip and is many sizes smaller than today's NAND Flash chips. Even SQUASHFS (which supports 4GB filesystems) was criticized as being "limited" by kernel developers. Simply rearchitecting the CRAMFS patches would not produce a sufficiently scalable solution. Even using SQUASHFS as a starting point might be viewed as not scalable. SQUASHFS would also need to be modified to use MTD devices. JFFS2 could also be viewed as not scalable. It should also be noted that JFFS2, having a writable architecture, would introduce many additional complexities if used as the basis for an XIP filesystem. We decided the best solution was to create a filesystem designed from the ground up to support XIP.

Obstacles to extending existing filesystem for XIP

1. No `struct page` for `/mm/filemap_xip.c`
2. Physical address not provided by drivers
3. XIP/compression on page granularity not supported
4. Existing filesystems "limited" or poor fit to application

3.2 Removing Barriers

As we looked at our options to enable XIP with a sustainable solution, it became obvious that we needed to

address the remaining issues in the kernel infrastructure. In order to remove the `struct page` dependency in `/mm/filemap_xip.c` we worked with virtual memory developers as well as the developers of the s390 architecture. Amazingly, the s390 developers were as excited as we were to remove the `struct page` dependencies from the `/mm/filemap_xip.c` for much the same reasons we had. In both the s390 architecture and the classic ARM-plus-Flash embedded system, the XIP memory is memory, but really doesn't want to be thought of as system RAM. Adding a `struct page` increases RAM overhead, but did not deliver any true benefit to our systems. Only in the Linux community do you find "big iron" and embedded systems developers working toward the same goal. The end result is a set of patches that is in the `-mm` tree as of this writing, hoping for a 2.6.26 merge.

Mapping non-`struct page` memory into memory maps requires that the filesystem be able to get the physical address that the virtual memory layers require. The target system, an ARM processor with Flash memory, would be able to have an MTD partition for the filesystem image to reside in. Using the little-used `mtd->point()` would give the filesystem a kernel-addressable virtual address to the image on Flash. While it is tempting to try a `virt_to_phys()` conversion, this simple approach doesn't work for our target architecture. The only place that reliable information about the physical address of Flash resides is in the MTD subsystem. Mounting to an MTD and then getting the physical address from the MTD seems reasonable. However, the MTD interface didn't provide an interface to get the physical address. The MTD developers decided the best way to get the physical address was to extend the `mtd->point()` to include a virtual and a physical address. There is a patch that has been signed off by many key MTD developers and will hopefully be merged in the 2.6.26 window.

Allowing control over the decision to XIP or compress pages at a page granularity requires both a new filesystem architecture and a change to the kernel. The patches required to do this are included with the page-less XIP recently added to the `-mm` tree. At issue were the mechanisms available to insert physical addresses in the form of a page frame number, or `pfn`, into process memory maps. Inserting a `pfn` with no `struct page` required the use of the `VM_PFNMAP` flag. The `VM_PFNMAP` flag makes assumptions about how the

pfns are ordered within a map. These assumptions are incompatible with enabling a page granularity for XIP. The VM_MIXEDMAP patch allows a process's memory to contain pfn-mapped pages and ordinary struct page pages in an arbitrary order. Allowing pfn-mapped and struct page-backed pages to coexist in any order allows a filesystem to have control over what parts of what file are XIP, and which are copied to the page cache.

4 Architecture

4.1 Design Goals

Once we decided that none of the existing Linux filesystems was likely to be easily extended to have the feature set we required, development of the architecture for the Advanced XIP File System began. The target application we had in mind was in mobile phones. Many phones use CRAMFS or SQUASHFS; therefore, many of the features will overlap with these existing filesystems. Being read-only and having limited time stats is acceptable. We need to improve on the size limitations built into CRAMFS and SQUASHFS by creating a 64-bit filesystem. Compressing in greater than 4KB chunks like SQUASHFS should also be enabled. The new filesystem should be able to mount from block devices like the legacy filesystems, but it should also mount directly from a MTD device. One new thing that we needed to add was the ability to mount with part of the image on a memory-mapped NOR Flash chip, while the rest of the image in on a NAND-style Flash chip.

4.2 Feature List

1. Basic Attributes

- 64-bit
- read-only
- designed with embedded needs in mind

2. Compression

- 4KB–4GB compression block size
- page-by-page uncompression map for XIP

3. Flexible Mount

- MTD (NAND/NOR)

- block device
- split across XIP NOR and non-XIP NAND

4. Tools

- GPL mkfs.axfs
- Supported image builder available

4.3 Profiling

Having the capability to decide whether to use XIP on individual pages allows the system designer to make a more cost-effective system, in theory. The obstacle in making this work in practice is deciding the right pages to XIP. The way we decided that made sense to us was to measure which pages in a filesystem are actually paged in. We chose to have a profiler built into the AXFS filesystem driver. The profiler records each time a page from a file in an AXFS filesystem is faulted into a non-writable map. After important use cases, the result can be read out of a /proc file and then the profile buffer can be reset by writing to that same /proc entry. There is a Kconfig option to allow the profiler to be compiling in or out of the driver. To profile a system, the system designer takes the following steps:

1. profiler is compiled in
2. system is booted with an AXFS image
3. important use cases are run
4. profile is extracted from /proc
5. profile is fed back into the image builder
6. profiler is compiled out
7. optimized AXFS image is loaded into system

4.4 Mount Options

Figure 1 shows how the same image can be mounted either on a single device or split across two devices. This is to allow a system designer maximum flexibility in optimizing systems for cost. A typical use for this device-spanning capability is to allow an image to span a NOR-type Flash and a NAND Flash. Device spanning is only permitted if the first device is directly memory-mappable. Any XIP regions of the AXFS image would

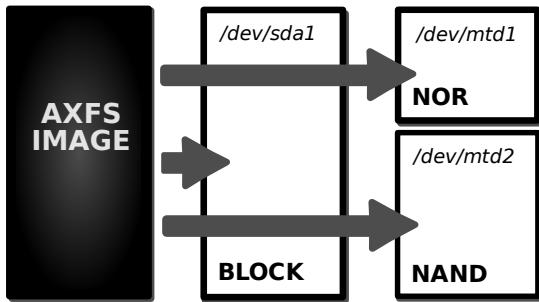


Figure 1: Mounting Details

need to reside on a NOR Flash. However, there is no reason why the compressed page regions would need to be in NOR. As NOR Flash is usually more expensive per bit compared to NAND Flash, placing compressed page regions in NAND Flash makes economic sense. It is not uncommon to have a large amount of NAND Flash in a mobile phone today. The amount on NAND Flash is often driven by media file storage demands rather the size of the root filesystem.

XIP CRAMFS would require a large enough NOR Flash be added to the system for the entire root filesystem image. Such a system could not take advantage of two facts: first, only a part of the image requires the XIP capabilities of the NOR Flash; and second, there is a large NAND Flash available. With AXFS, the NOR Flash can be as small as the XIP regions of the AXFS root filesystem image, with the rest of the image spilling over into the NAND. The reason this can lead to cost savings is that each memory component—the RAM, NOR Flash, and NAND Flash—can be sized to minimum-ration chip sizes. If RAM usage is just over a rational chip size, but there is room in the NOR Flash, the designer can choose to XIP more pages. If the NOR Flash usage is over a chip size, but there is free RAM, pages can be compressed and moved to NAND Flash to be executed from RAM.

This flexibility also lowers risk for the designer. If applications need to be added or turn out to be larger than planned late in the design cycle, adjustments can be made to the contents of RAM and NOR Flash to squeeze the extra code and data into the system while retaining performance. With a traditional system, unexpected code size means unexpectedly high page cache requirements. This leads to more paging events. As system performance is sensitive to paging latencies, more code

will certainly lead to lower system performance. When this happens with an AXFS system, any free space in NOR Flash can be exploited to absorb the extra code or to free up RAM for data.

4.5 Format

The AXFS on-media format is big-endian and has three basic components: the superblock, RegionDescriptors, and Regions. Essentially the superblock points to the RegionDescriptors, which in turn point to Regions (as shown in Figure 2). There is, of course, a single superblock, many RegionDescriptors, and many Regions.

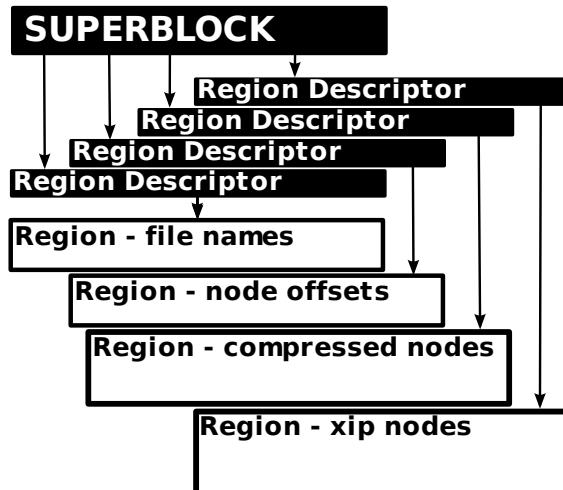


Figure 2: On-media Format

The superblock contains filesystem volume-specific information and many offsets, each pointing to a separate RegionDescriptor. Each RegionDescriptor then contains an offset to its Region. A Region can contain many different kinds of data. Some Regions contain the actual file data and one contains the filename strings, but most Regions contain ByteTables of metadata that allow the files to be reconstructed.

ByteTables are how most numbers are stored in AXFS. It is simply an array of bytes. Several bytes must be “stitched” together to form a larger value. This is how AXFS can have such low overhead and still be a 64-bit filesystem. The secret is that in the AXFS driver, most numbers are unsigned 64-bit values, but in the ByteTables, each value is only the minimum number of bytes required to hold the maximum value of the table. The

number of bytes used to store the values is called the depth. For example, a ByteTable of offsets into the Region containing all the file name strings could have many depths. If the strings Region is only 240 bytes long, the depth is 1. We don't need to store offsets in 8-byte-wide numbers when every offset is going to be less than the 255 covered by a single byte value. On a strings Region that is 15KB in size, the depth would be 2, and so on. The ByteTable format makes it easy to support large volumes without punishing the small embedded systems the design originally targeted.

A Region in AXFS is a segment of the filesystem image that contains the real data. The RegionDescriptors on media representations are big-endian structures that describe where a given Region is located in the image, how big it is, whether it is compressed, and for Regions containing ByteTables information, information about the depth and length of that ByteTable.

Several ByteTable regions are dedicated to inode-specific data such as permissions, offset to the filename, and the array of data nodes for files or child inodes for directories. Those are fairly straightforward. The data nodes prove a little more complex. A file inode points to an array of page-sized (or smaller) data nodes. Each node has a type (XIP, compressed, or byte-aligned) and an index. The XIP case is the simplest. If the node type is XIP, the node index becomes the page number in the XIP region. Multiplying the index by PAGE_SIZE yields the offset to the XIP node data within the XIP Region.

A node of type *byte-aligned* contains data that doesn't compress and is a little more complicated to find. This exists because data that doesn't compress is actually larger when run through a compression algorithm. We couldn't tolerate that kind of waste. The node index becomes the index into a ByteTable of offsets. The offset points to the location within the byte-aligned Region where the actual data is found.

The compressed node type is the most complicated to find. The node index for a compressed node is used as the index to two separate ByteTable values, the *cblock* offset and the *cnode* offset. A cblock is a block of data that is compressed. The uncompressed size of all cblocks is the same for a given filesystem, and is set by the image builder. A cnode is a data node that will be compressed. One or more cnodes are combined to fill a cblock and then compressed as a unit. The compressed

cblocks are then placed in the compressed Region. The cblock offset points to the location of the cblock containing the node we are looking for. The cblock is then uncompressed to RAM. In this uncompressed state, the cnode offset points to where the node's data resides in the cblock.

5 Benchmarks

5.1 Benchmark Setup

The benchmarks were run on our modified “Mainstone 2” boards. The kernel was only slightly modified to run on our platform and to include AXFS. The root filesystem was a build of Opie we created from OpenEmbedded about a year ago. It is a full PDA-style GUI, and we added a few applications like Mplayer and Quake.

- PXA270 Processor
 - 520 MHz (CPU)
 - 104 MHz (SDRAM bus)
 - 52 MHz (NOR flash bus)
- Linux-2.6.22
 - xipImage
 - CONFIG_PREEMPT=y
 - MTD updated to Sept 25 git pull
 - mem=24MB in kernel commandline
- Opie root filesystem
 - OpenEmbedded
 - about one year old

5.2 Performance

Rather than demonstrate meaningless raw throughput numbers, we wanted to use a real-life scenario that shows a difference. The easiest way to show how AXFS can improve performance is by showing how fast it can be at launching applications. We reduced the available RAM to simulate the memory pressure present in a cost-sensitive consumer electronic device. Figure 3 shows the combined time it took to start up several applications in seconds for each filesystem. Once these applications (video player, PDF viewer, web browser) launched, they

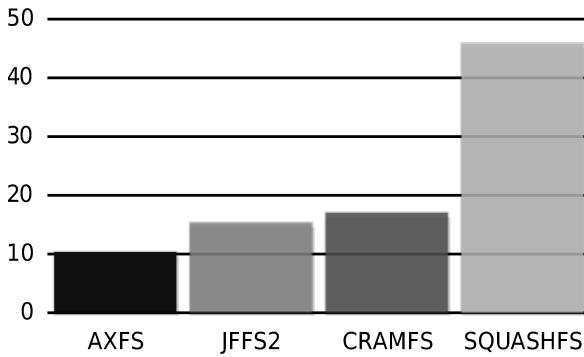


Figure 3: Aggregate application launch (s)

performed the same on each platform as far as we could measure.

Figure 4 shows the effect of memory pressure on the various filesystems. This shows the time it took to boot to the Opie splash screen with the amount of RAM varied via `mem=`. As the memory pressure builds, you can see that the performance is very steady on AXFS, while it really effects the performance of SQUASHFS.

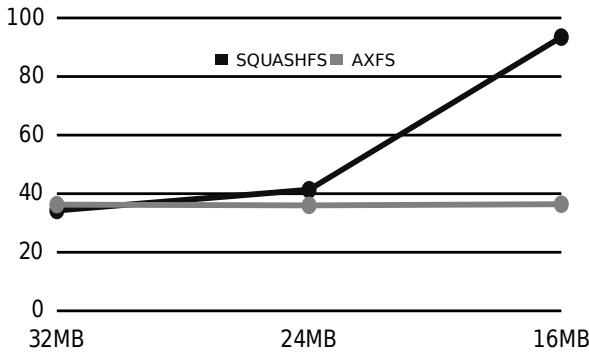


Figure 4: Impact of memory pressure on boot time (s)

5.3 Size

Comparing the image sizes for each filesystem, as we do in Figure 5, shows how AXFS compresses better than all the others, although it isn't much smaller than SQUASHFS. It also shows how optimized XIP AXFS is much smaller than the best we could do with XIP CRAMFS; both save the same amount of RAM.

Comparing the size of the XIP AXFS to the SQUASHFS image in Figure 5 makes it look as though the SQUASHFS is smaller. That is only part of the equation. If we take the amount of RAM used into consideration, it is clear an XIP AXFS image uses less total

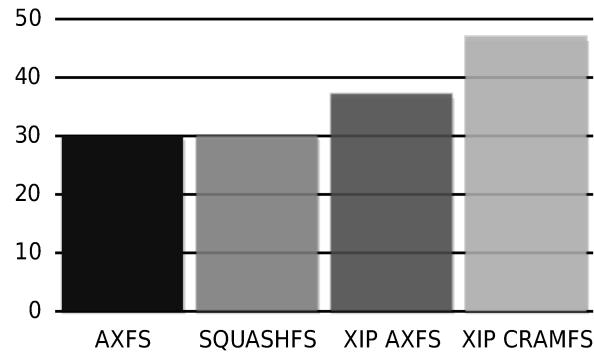


Figure 5: Image sizes (MB)

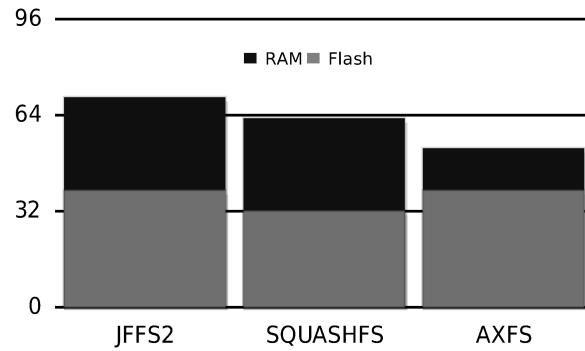


Figure 6: total used memory (MB)

memory, as shown in Figure 6. As all of these system use more than 32MB of Flash, a 64MB Flash chip would likely have to be used for all. However, while the JFFS2 and SQUASHFS systems both need 32MB of RAM, the AXFS system would only need 16MB. In fact, the AXFS system would still have more free RAM than the others, even with a smaller RAM chip.

6 Summary

AXFS was designed to create a filesystem with a chance of being merged into the kernel that provided the XIP functionality long used in the embedded world. It can conserve memory, saving power and cost. In the right circumstances it can make systems quicker and more responsive. Hopefully it can soon be merged and enjoyed by all—especially by those that have long been struggling with the XIP CRAMFS patches. We are also hopeful that other users will find new uses for its unique capabilities, such as the root filesystem in a LiveCD.

Low Power MPEG4 Player

Joo-Young Hwang
Software Labs
Samsung Electronics Co. Ltd.
jooyoung.hwang@samsung.com

Woo-Bok Yi
Software Labs
Samsung Electronics Co. Ltd.
woobok.yi@samsung.com

Sang-Bum Suh
Software Labs
Samsung Electronics Co. Ltd.
sbuk.suh@samsung.com

Jun-Hee Kim
Seoul National University
goldlion@davinci.snu.ac.kr

Ji-Hong Kim
Seoul National University
jihong@davinci.snu.ac.kr

Abstract

In this paper, design and implementation of a dynamic power management for a MPEG-4 player is described. We designed two dynamic voltage scaling algorithms (feedback-based and buffering-based) to adapt CPU voltage dynamically according to the variable bit rate of a movie. We experimented the algorithms with the open-source XviD player. Our modified XviD player can save up to about 50% power without performance degradation. We describe practical lessons learned in optimization of the algorithms.

Power management (PM) is an important issue for battery-powered Linux devices, particularly for video playing devices. Our work shows how a conventional open-source video player can be modified to save power significantly on a CPU with dynamic voltage scaling capability.

1 Introduction

Power consumption is one of the big issues facing mobile embedded devices. Multimedia playing is one of the key functions of recent mobile devices, and the multimedia player is running for most of the time while using those devices. Power management for efficient multimedia playing in mobile devices is an important issue.

There have been numerous research papers published on power management for server systems [10], soft-real-time systems [6], and up to real-time systems [12, 1, 11].

Some of them are focused on per-component power management or system-level power management. In this paper, we focus on power management of the CPU component for multimedia playing devices, which is a soft-real-time system.

Many modern CPUs provides dynamic changing of clock frequencies and voltages in order to reduce power consumption. When a CPU becomes idle, it normally enters idle mode, a low-power mode provided by most current processors. However, it is rather efficient to reduce voltage levels as much as possible without violating deadlines of tasks, because power consumption of digital CMOS circuits is quadratically proportional to the supply voltages ($P = \alpha C f V_{DD}^2$)[2, 7].

Conventional DVS algorithms adjust CPU voltage/clock frequency dynamically according to workload variations. There are two DVS (Dynamic Voltage Scaling) approaches to exploit CPU slack time: inter-task and intra-task. Inter-task DVS is to exploit the slack time obtained from one task when the task completes prior to its planned execution time for the next task to schedule. The OS scheduler determines the CPU voltage level for the task to schedule, and changes the CPU voltage as determined.

Intra-task DVS is to change voltage levels during the execution of a task. Power Control Points (PCP) are inserted into a conventional program in order to change CPU voltage level at those points. PCP can be inserted automatically by the compiler, or manually by program-

mers. PCP should be inserted appropriately to avoid unnecessary frequent voltage switches, which will lead to performance degradation. There is no general rule of thumb on where to insert PCP into a program, but it may varies from application to application. The workload of an application should be analyzed statically or profiled dynamically at run time in order to pick the proper position for PCP.

Dynamic voltage scaling for multimedia players is an intra-task DVS method. There have been many papers on dynamic voltage scaling multimedia playback [3, 4, 8, 5, 9, 6]. In this paper, we describe a case study on practical issues in deployment of DVS algorithms on Linux and legacy multimedia player. We describe implementation issues of two DVS methods (feedback-based and buffer-based) and show detailed performance analysis. We also indicate possible further optimization directions.

This paper is organized as following: In Section 2, we overview currently known DVS algorithms and describe our implementation of those DVS algorithms in the open source XviD MPEG4 player, and discuss implementation issues in Section 3. Experimental performance results are shown in Section 4, and we summarize this paper in Section 5.

2 DVS Algorithms for Multimedia Playing

2.1 Feedback-based DVS Algorithm

Using conventional feedback-based algorithms [3, 4, 8, 5, 9], workload of the current Video Object Plane (VOP) is predicted considering the previous VOP decoding workloads, and the CPU voltage/clock is adjusted accordingly. In our media player, the VOP workload is estimated based on a simple moving average scheme; the workloads measured in the unit of the number of CPU cycles for previous frames are averaged over a window (typically 3 frames). Target CPU clock for the current frame decoding is calculated as the estimated workload divided by the decoding period. Then we select the power state with the minimum CPU clock among the CPU clocks higher than the calculated target frequency. The algorithm uses a different time window size for different VOP types (I, P, or B) of a frame.

2.2 Profile-based DVS Algorithm

This algorithm is the ideal case of a feedback-based algorithm. By profiling actual VOP decoding times in off-line and using the profile at run-time, the feedback-based algorithm can adjust voltage/clock according to correctly predicted workloads for all the frames. The VOP decoding times are measured for each CPU clock frequency because decoding time is affected not only by CPU clock frequency, but also by off-chip memory latencies. The performance of the profile-based feedback scheme gives the ideal performance which can be obtained by feedback-based algorithms.

2.3 Buffer-based DVS Algorithm

This algorithm, originally proposed in [6], can be used for input buffering or output buffering. The paper described only the input buffering case in detail, but we are interested in output buffering in this paper.

When there is no output buffer and the decoder should output decoded stream to the frame buffer directly, the decoder should wait until the next period to update the frame buffer. VST (Workload Variation Slack Time), which is generated by early completion of decoding of a frame, cannot be exploited for decoding of the next frame. If there are output buffers to save the decoded stream, the decoder can begin decoding of the next frame, exploiting the VST generated by the previous frame. This is illustrated in Figure 1.

When decoding of the j -th VOP is complete prior to its deadline, the slack whose amount is VST_j occurs. Without output buffers, the decoder should wait for this slack interval. Assuming that the workload required for decoding the $j+1$ -th frame is PEC, the CPU clock frequency for decoding the frame is PEC/Period. With output buffers, the decoder can start decoding of $j+1$ -th frame without waiting for the next period. At the moment, CPU clock frequency is adjusted to be PEC / (VST_j + Period), so power consumption is reduced. At the deadline of $j+1$ -th frame, the current frame buffer address is switched to the memory, which contains the decoded data of the $j+1$ -th frame.

According to [6], maximum buffer size can be estimated by the ratio of worst case execution time (WCET) to best case execution time (BCET). The VST increases as the actual execution time becomes shorter than the

deadline. Assuming the steady state worst-case scenario where VST is saturated to a maximum value, (BCET / WCET) becomes equivalent to $(T / T + VST)$ where T is the period, then $VST = T(WCET/BCET - 1)$. To fully exploit the VST, output buffers should be available. Therefore, the buffer size h should satisfy the following.

$$h \geq \lceil \frac{VST}{T} \rceil = \lceil \frac{WCET}{BCET} - 1 \rceil$$

To determine BCET and WCET for a media clip, the clip is played once at the full speed of a CPU. The authors of [6] supposed that the execution times of the applications follow a normal distribution and took 3σ variations around the mean of the distribution as boundary values. In our experiment, we simply take the actual maximum and minimum execution times as WCET and BCET, respectively.

3 Implementation

3.1 Voltage Scaling Function in Linux Kernel

In our experimental platform, CPU supply voltage is regulated via the LTC1663 chip, and it takes non-negligible time to change CPU voltage. So, the multimedia player should not block waiting for completion of CPU voltage change. A voltage scaling daemon, running as a kernel thread, is responsible for changing CPU voltage to a new value, specified by the multimedia player's voltage scaling request. The VS daemon writes the request to the LTC1663 chip and sleeps on interrupt from the chip, which arrives when the actual voltage change is complete.

3.2 Media Player SW Architecture

Our low power media player consists of four modules as described in the following.

- AVI I/O Library

Most MPEG-4 video data are contained in a separate container format such as AVI or MOV. Among them, AVI format is generally used, and we use it for our experiments. AVI I/O library extracts MPEG-4 VOP stream data from AVI file and sends it to XviD decoder. We use Transcode AVI I/O library for this module.

- XviD Decoder Library

This is a open source GPL licensed MPEG-4 decoder library. It is not official reference software for MPEG-4, but it is fully compatible with MPEG-4 and its performance is well optimized, so we choose it for our baseline media player.

- Low Power Player Module

This is the core module of our media player. It performs basic hardware initialization and sends frame data decoded by the XviD decoder library to the LCD frame buffer, in synchronization with frame deadlines. This also includes implementation of feedback-based and buffer-based DVS algorithms.

- User-level VS Library

The VS library provides user-level DVS APIs for applications. These invoke system calls to get service of DVS functions from Linux kernel.

DVS APIs provided by the user-level VS library to the media player are summarized in the following.

- `unsigned int getCurrentSpeed()`
returns current relative speed
- `void setScaledClock(unsigned int newSpeed)`
set CPU clock frequency to a value corresponding to a given relative speed value of "newSpeed"
- `void setScaledSpeed (unsigned int newSpeed)`
set both CPU clock frequency and voltage at the same time corresponding to a given relative speed value of "newSpeed"
- `unsigned int getCurrentVoltage()`
returns current voltage value multiplied by 100.
- `void setVoltage(unsigned int newVoltage)`
set CPU voltage to the given "newVoltage" which is a voltage value multiplied by 100.

3.3 Power States Selection

PXA-255 provides various CPU voltage/clock combinations, as shown in Table 1. We selected combinations with the same memory clock frequency of 99.5 MHz to avoid ambiguity, as described in the following. When we consider two combinations with different memory clocks and different CPU clocks, it is not deterministic which one consumes more power and gives higher performance, because the actual power consumption depends on workload characteristics. For memory intensive workloads, a combination with a higher memory clock may give higher performance, even though it has lower CPU clock frequency. It is also difficult to compare the total power consumption including CPU and memory power, because they may vary from device to device, and an accurate power consumption specification for processor and memory is required for correct comparison. For implementation simplicity and portability, we selected the four combinations in Table 1.

PXA-255 processor has an idle mode, where power consumption is minimal. It is generally entered by operating systems when the system is idle. When an interrupt arrives, processor mode is immediately changed to active mode. Even though the system is idle, the Linux kernel still typically processes a timer tick every ten milliseconds. To reduce the power consumption for processing, the periodic timer ticks when the system is idle; we adjust the CPU voltage/clock to the lowest power state on entry to slack interval.

3.4 Deadline Misses Handling

In the feedback-based method, a deadline miss may occur when the workload prediction is incorrect. If the CPU voltage is adjusted too low, the decoding of a frame will not be complete until the deadline of the frame. When such deadline miss occurs for a frame, the VOP deadline of the next frame is shortened accordingly not to increase the total play time.

4 Performance results

Our experimental platform, TynuxBox-Xe, is equipped with an Intel PXA255 CPU operating at 400MHz, 32MB SDRAM, and 32MB NOR flash memory. To measure the power consumption of the CPU, a data

acquisition instrument is used to collect voltage samplings. Small serial resistance is inserted between the supply and the CPU to measure the current flow. Voltage level is sampled at the points (A) and (B) in Figure 2. Instantaneous power consumption of the CPU is calculated as $V_B \frac{V_A - V_B}{R}$, where V_A and V_B are the voltages at (A) and (B) points, respectively. Voltages are sampled at 1000 HZ frequency. We used a trailer for the Matrix Revolutions as an input, in which length is 63 seconds, video frame rate is 12 frame per second, and screen resolution is 240x160.

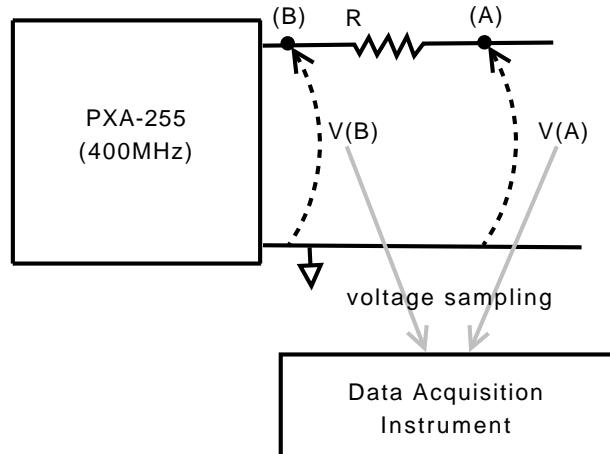


Figure 2: Power consumption measurement

We compare the following four cases:

- Normal case without DVS methods. CPU voltage is set to the highest value when system is active, and CPU enters idle mode when system is idle.
- Feedback-based method. It is to adjust CPU voltage frame by frame according to the predicted current frame's CPU workload, which is estimated based on history of actual workloads of the previous frames. Moving average window size is 3.
- Profile-based method. It is to adjust CPU voltage frame by frame according to the actual workload which is known a priori.
- Buffer-based method. It is to adjust CPU voltage frame by frame according to the worst-case execution time of each frame. The number of buffers is set to 4, which is calculated as described in 2.3.

Figure 3 shows instantaneous power consumption of the three DVS methods for the first 500 milliseconds time

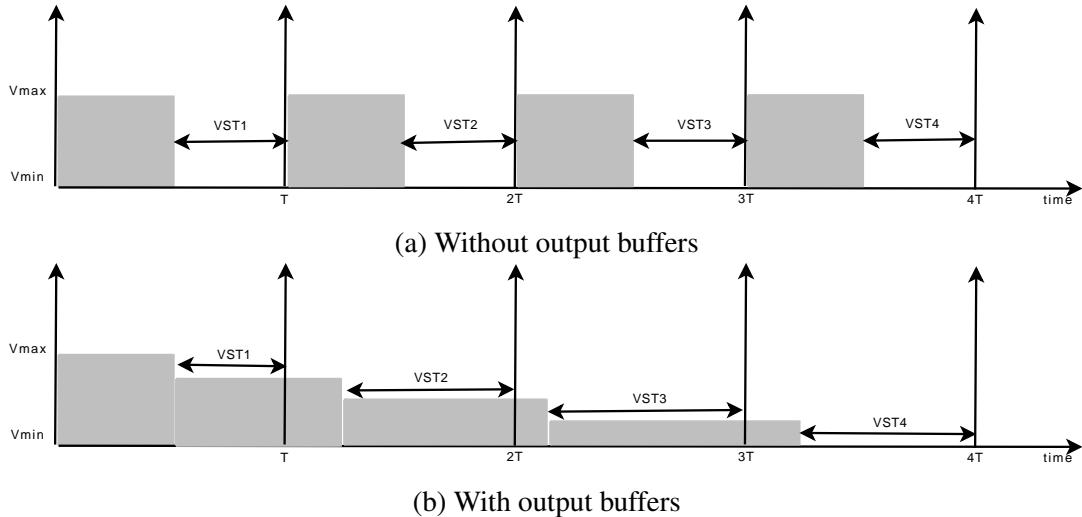


Figure 1: Working behaviour comparison between w/o buffer and w/ buffer cases.

CPU Clock (MHz) corresponding to PXA255's CCCR setting (N)				SDRAM Clk(MHz)
N = 1.00	N = 1.50	N = 2.00	N = 3.00	
99.5@1.0V	-	199.1@1.0V	298.6@1.1V	99.5
132.7@1.0V	-	-	-	66
199.1@1.0V	298.6@1.1V	398.1@1.3V	-	99.5
265.4@1.1V	-	-	-	66
331.8@1.3V	-	-	-	83
398.1@1.3V	-	-	-	99.5

Table 1: PXA255 CPU voltage/clock combinations table with SDRAM clock frequency. CCCR is Core Clock Configuration Register of PXA255 CPU.

interval, during which the video changes smoothly and decoding workload is low. As shown in Figure 3 (a), the decoder in the normal case is active only for approximately half of the period. For the workload, all those DVS methods work well and none of them outperforms the others. Total energy consumption over the time interval are 11.49, 7.37, 6.23, and 7.26 mJ for normal, feedback-based, profile-based, and buffer-based methods, respectively. It should be noted that the buffer-based method adjusts the CPU voltage higher than the profile-based method for the first frame, because it uses the worst-case execution time for workload estimation, while the profile-based method uses the exact workload of the frame. For that reason, total energy consumption of the profile-based method is the lowest among the three methods.

Figure 4 shows instantaneous power consumption of the DVS methods for the time interval from 3 - 3.5 seconds of the video during which the scene changes quickly, and decoding workload is high. The feedback-based method failed to reduce power for this workload because the moving average-based workload estimation is wrong for most cases. The buffer-based method works well for the workload, and even better than the profile-based method. In the method, CPU voltages are kept low for most of time and media player hardly enters Idle mode, which is owing to the efficient exploitation of VST. Unlike other methods, in the buffer-based method, CPU voltage change is not synchronized with frame deadline as observed during the $(i+4)$ -th frame in Figure 4 (c). Energy consumptions during the time interval are 14.84, 14.91, 10.28, and 7.33 mJ for normal, feedback-based, profile-based, and buffer-based methods, respectively.

The current buffer-based method has more optimization opportunities. It is possible that CPU voltage is conservatively set because the buffer-based method uses the worst-case execution time of a video clip, instead of using feedback from actual execution times. In case of playing a video whose frame decoding complexity variation is very high, setting the CPU voltage considering WCET may lead to buffer shortage while decoding low complexity frames. Using actual execution times can be a solution of this problem. Since this may also cause deadline misses as the feedback-based method, either appropriate deadline miss handling or deadline miss avoidance is necessary, which is one of our future works.

5 Summary

In this paper, we described our implementation of DVS algorithms designed for low-power multimedia playback. Feedback-based and buffer-based methods are implemented using the XviD MPEG4 player. The feedback-based method performs well for smoothly changing video. However, it could not correctly predict frame decoding workloads for quickly changing video, which led to non-significant power reduction. We also implemented the profile-based DVS method, which uses correct workload information profiled at off-line to show the upper bounds of the performance, which can be obtained by ideal feedback-based methods.

The buffer-based method using multiple output buffers performs better than the profile-based method, owing to efficient exploitation of VST (Workload Variation Slack Time). Without output buffers, the decoder cannot decode the next frame, even though the current frame decoding is completed earlier than its deadline. In contrast, with output buffers, the decoder can continue work by queuing output to buffers. The number of buffers does not have to be large, and 4 buffers were enough to get significant power reduction for our test video sequence.

We showed detailed analysis of voltage scaling behaviour for typical DVS methods. We also indicate the possibility of further optimization of the buffer-based DVS method for handling video sequences with varying frame complexity.

References

- [1] H. Aydin, R. Melhem, D. Mosse, and P. M. Alvarez. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *Proceedings of IEEE Real-Time Systems Symposium*, 2001.
- [2] T. Burd and R. Brodersen. Processor design for portable systems. In *Journal of VLSI Signal Processing*, Aug. 1996.
- [3] K. Choi, K. Dantu, W. Cheng, and M. Pedram. Frame-based dynamic voltage and frequency scaling for a mpeg decoder. In *Proceedings of International Conference on Computer Aided Design*, pages 732–737, November 2002.

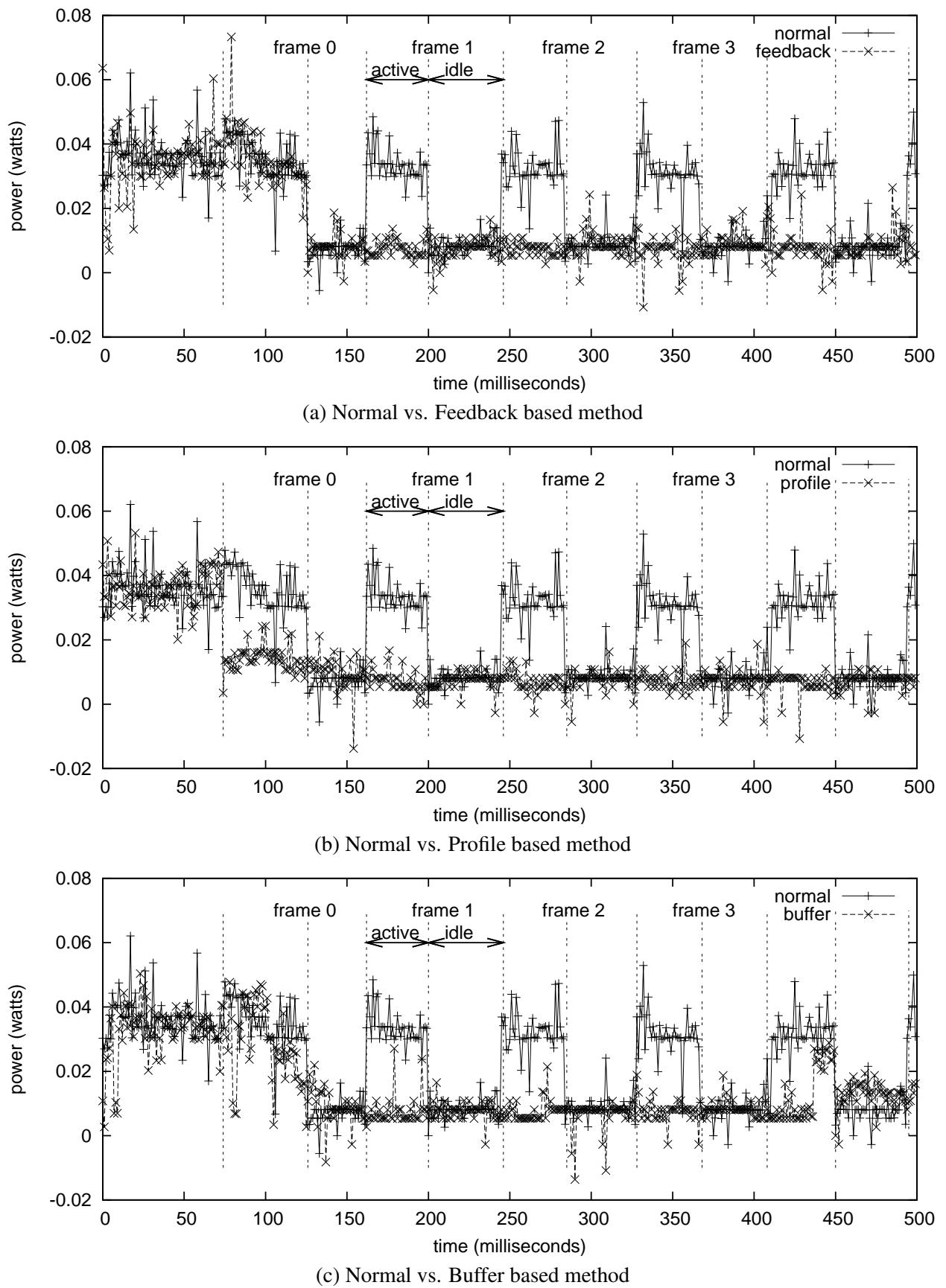


Figure 3: DVS results for smoothly changing video part.

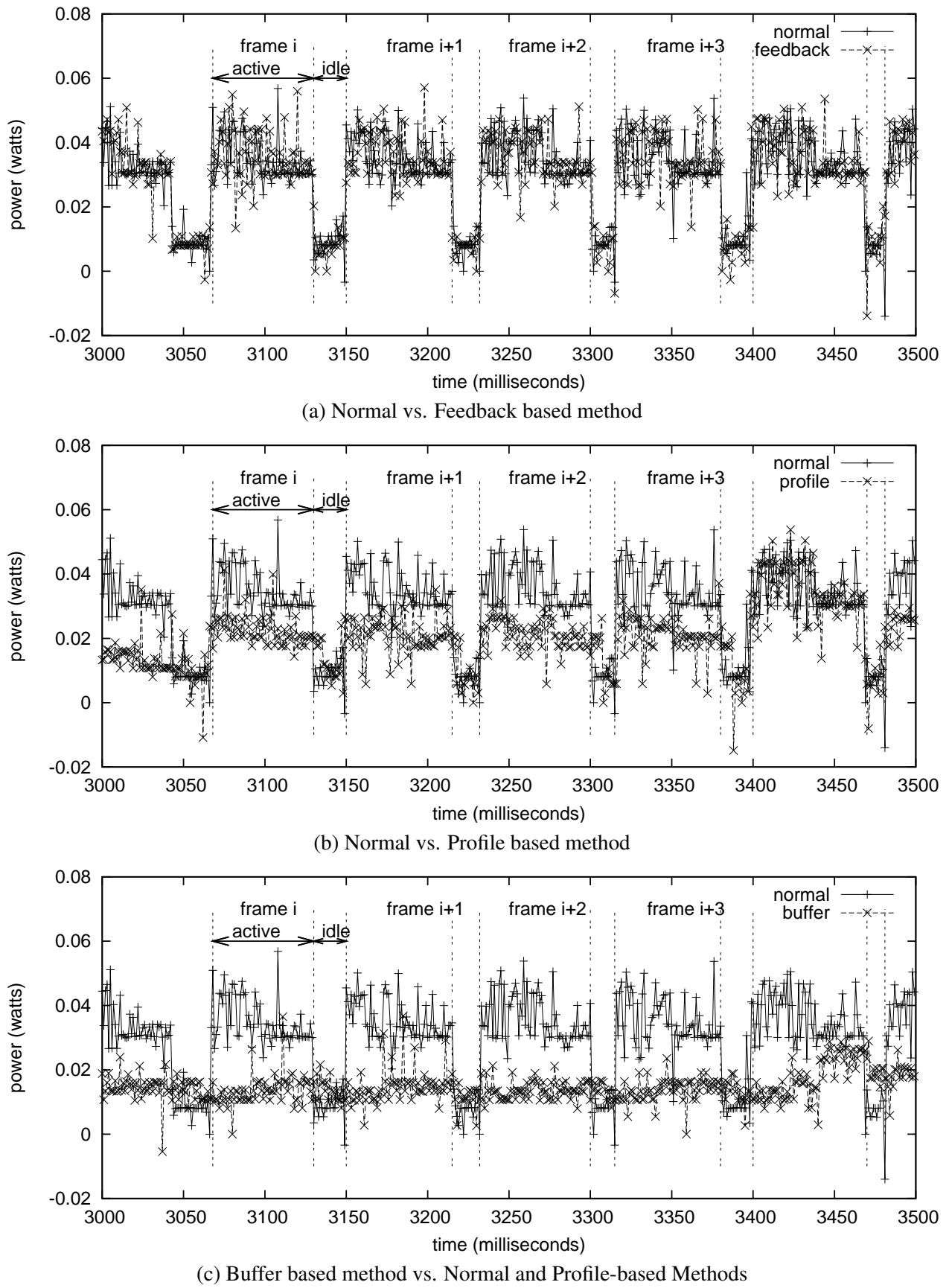


Figure 4: DVS results for quickly changing video part.

- [4] K. Choi, R. Soma, and M. Pedram. Off-chip latency-driven dynamic voltage and frequency scaling for an mpeg decoding. In *Proceedings of 41st Design Automation Conference*, pages 544–549, June 2004.
- [5] C. Im and S. Ha. Dynamic voltage scheduling with buffers in low-power multimedia applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(4):686–705, November 2004.
- [6] C. Im, H. Kim, and S. Ha. Dynamic voltage scheduling technique for low-power multimedia applications using buffers. In *Proc. Int'l Symp. on Low Power Electronics and Design*, pages 34–39, 2001.
- [7] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings of ISLPED (International Symposium on Low Power Electronics and Design)*, Aug. 1998.
- [8] Z. Lu, J. Lach, M. Stan, and K. Skadron. Reducing multimedia decode power using feedback control. In *Proc. of International Conference on Computer Design*, pages 489–496, October 2003.
- [9] Z. Lu, J. Lach, M. Stan, and K. Skadron. Design and implementation of an energy efficient multimedia playback system. In *Signals, Systems and Computers, 2006. ACSSC '06. Fortieth Asilomar Conference on*, pages 1491–1497, Oct.-Nov. 2006.
- [10] T. Pering, T. Burd, and R. Broderson. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proc. Int'l Symp. on Low Power Electronics and Design*, pages 76–81, 1998.
- [11] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of 18th ACM Symposium on Operating Systems Principles*, pages 89–102, 2001.
- [12] Y. Shin, K. Choi, and T. Sakurai. Power optimization of real-time embedded systems on variable speed processors. In *Proceedings of the International Conference on Computer-Aided Design*, pages 365–368, 2000.

VESPER (Virtual Embraced Space ProbER)

Sungho Kim

Hitachi, Ltd., Systems Development Lab

sungho.kim.zd@hitachi.com

Satoru Moriya

Hitachi, Ltd., Systems Development Lab

satoru.moriya.br@hitachi.com

Satoshi Oshima

Hitachi, Ltd., Systems Development Lab

satoshi.oshima.fk@hitachi.com

Abstract

This paper describes VESPER (Virtual Embraced Space ProbER), the framework that gathers guest information effectively in a virtualized environment. VESPER is designed to provide evaluation criteria for system reliability and serviceability, used in decision-making for system switching or migration by a cluster manager. In general, the cluster manager exchanges messages between underlying nodes to check their health through the network. In this way, however, the manager can not discover a fault immediately and get detailed information about faulty nodes. VESPER injects Kprobes into the guest to gather the guest’s detailed information. By communicating with the guest in kprobes through the VMM infrastructure, VESPER can provide the manager with prompt fault information much more quickly.

In this paper, we explain how VESPER injects Kprobes into a guest and clarify the benefits by showing a use case on Xen. As VESPER is not strongly coupled to a specific VMM, we also show its portability to KVM and lguest.

1 Introduction

Recently, the trend of applying virtualization technology to enterprise server systems is getting much more noticeable, such as in server consolidation. The technology enables a single server to execute multiple tasks which would usually run on multiple physical machines. From that point, applying virtualization technology to cluster computing is very attractive technology and worth considering as well, in terms of efficient resource utilization and system dependability. Furthermore, use of virtualized environments for cluster systems allows us to make improvements in several areas

of general clustering technology—especially, fail-over response latency in high-availability clusters.

Even in a virtualized environment, a cluster manager such as Heartbeat [1][2] software delivers messages between underlying nodes to check their health through network periodically (known as heartbeat). If any node fails to reply in a certain time, the manager will assign the service which the faulty node was providing to another node. This amount of time before switching to another node is called the deadtime, key to ascertaining node death in Heartbeat. With this approach, however, the manager can not immediately determine faults, nor get detailed information about faulty nodes; this results in fail-over response latency. So we have focused on improving upon this latency, and on the failure analysis the manager should facilitate in clustering virtual machines, by considering features of virtualization technology.

One solution to the latency and failure analysis issues is to add dynamic probing technology into virtual machines. This allows the cluster manager to have:

- Dynamic probe insertion to guarantee service availability while inserting a probe.
- Arbitrary probe insertion to any process address to hold its versatile probing capability.
- Prompt notification when corresponding events happen around a probe.

Utilizing this featured probe technology to examine the health of a virtual cluster member machine could lead to faster and more efficient evaluation criteria for system switching or migration than a simple, periodic message delivery mechanism.

Speaking of the probe technology, Kprobes [3] are available in the Linux kernel community. Kprobes can insert a probe dynamically at a given address in the running kernel of a targeted virtual machine. Its execution of the probed instruction in the kernel is capable of dealing with the detailed information on the targeted virtual machine in an event-driven fashion.

From a system management point of view, however, one privileged system running the cluster manager (called **host** hereafter) should obtain all probed data from the targeted virtual machine (called **guest** hereafter). So, there needs a mechanism to insert probes from the host into the guests to improve the manageability of the manager, which Kprobes does not take into consideration.

Xenprobes [4] has already addressed this topic. Xenprobes is newly devised for probing virtual machines, but adopts Kprobes's concept in inserting breakpoints where one needs a probe. However, Xenprobes needs the help of a VMM-like furnished debugging mechanism [5] and should stop the guest to insert the probes. Moreover, every breakpoint causes VMM to give execution control to the host, especially in Xen [6] technology. These could be significant problems in service availability.

In this paper, therefore, we propose the framework named VESPER (Virtual Embraced Space Prober) which gathers guest information effectively in a virtualized environment, taking advantage of the full features of Kprobes, adopted for its probing component. In contrast to Xenprobes, VESPER never gets involved with probe handlers; this acts to avoid unnecessary probing overhead and to improve service availability. VESPER simply transfers Kprobes generated in the host to the targeted guest. The transferred Kprobes do all the necessary probing work themselves in the guest, and then VESPER simultaneously obtains the result of Kprobes through shared buffers (such as relayfs) built across the host and guest.

We will describe how VESPER injects the probes into guest and provides the solution to fail-over response latency with failure analysis in a virtualized environment by showing a use case with Xen, in the following sections of this paper.

Section 2 briefly describes design requirements in VESPER; Section 3 presents the architecture of VESPER. Section 4 presents implementation details of VESPER,

while Section 5 shows some benefits of VESPER for simple web services. Finally, we conclude this paper in Section 6.

2 Design overview

In designing VESPER, we took special interest in simplicity of implementation and robustness against disasters in userland. We thus set up some design requirements on VESPER as follows, to reflect our concepts.

1. No modifications on host or guest kernels. Lightweight implementation as a kernel module could assure better usability and availability of VESPER in systems due to dynamic loading and unloading features of kernel modules.
2. Only the host can insert probes into the guest, and guest itself loads them using guest kernel space only. As mentioned in the previous section, the cluster manager running on the host might as well control probes into the guest from a management point of view. In addition, the guest itself loads the probes inserted by the host into its kernel space dynamically to keep its serviceability. Furthermore, if some disaster should mangle user space but not kernel space in the guest, VEPSEN should still be available to find out what the problem is.
3. All probed data from the guest is sent to host as quickly as possible. To receive prompt alerts via probed data is the main purpose of VESPER, and thus to improve fail-over response latency.

To satisfy the requirements above, VESPER thinks of Xen and KVM mainly as target VMMs because of the popularity of Xen and KVM [7] in the OSS (Open Source Software) community in this writing. Firstly, to address Requirement 1 above, VESPER is, by preference, developed as a virtual device driver. Requirements 2 and 3 dictate that VESPER communicate with host and guest. As a matter of fact, Xen and KVM technology provide a well-defined device driver model, split device driver, and communication infrastructure between host and guest. Xenbus is infrastructure specific to Xen technology, whereas `virtio` is applicable to Xen and KVM as well. Especially, `virtio` is available in 2.6.24 Linux kernel. VESPER uses hierarchical layers in its

software structure to accommodate various infrastructures, making it more available and portable.

The layer dependent on a certain infrastructure prepares the set of functions and data items for the use of the layers independent of the VMM architecture. Details on that structure and implementation will be discussed in the upcoming sections.

3 Architecture and Semantics of VESPER

For probing the guest, VESPER uses Kprobes to hook into the guest Linux kernel, and uses `relayfs` to record probed data in the probe handler of Kprobes. In this section, we take a brief look at the VESPER architecture featuring a 3-layered structure, then we present its semantics.

3.1 VESPER Architecture

As mentioned before, VESPER uses Kprobes to hook into the guest kernel. However, because Kprobes is the probing interface for the local system, it can not implant probes into the remote system directly. Besides, in the typical use case of Kprobes, the probe handler in Kprobes comes in the form of a kernel module. Therefore, in using Kprobes to hook into the guest kernel, VESPER should be able to load probing kernel modules, on which the handlers to probe are implemented, from host to guest. This loading capability of VESPER is implemented as split drivers and named *Probe Loader*.

In VESPER, the probing modules use relay buffers to record data in the probe handlers. At this point, probing modules are in the guest; thus, VESPER needs to transfer the buffer data from the guest to the host. This relayed data transfer capability is also implemented as split drivers and named *Probe Listener*.

Figure 1 is the block diagram of the VESPER component.

As just described, VESPER contains two pairs of split drivers. These drivers are implemented for each VMM because they strongly depend on the underlying VMM. So, we divide VESPER into three layers (shown in Figure 2): UI Layer, Action Layer, and Communication Layer, in order to localize VMM-dependent code. This structure lets VESPER run on Xen and KVM by replacing only the VMM architecture-dependent layer—the Communication Layer.

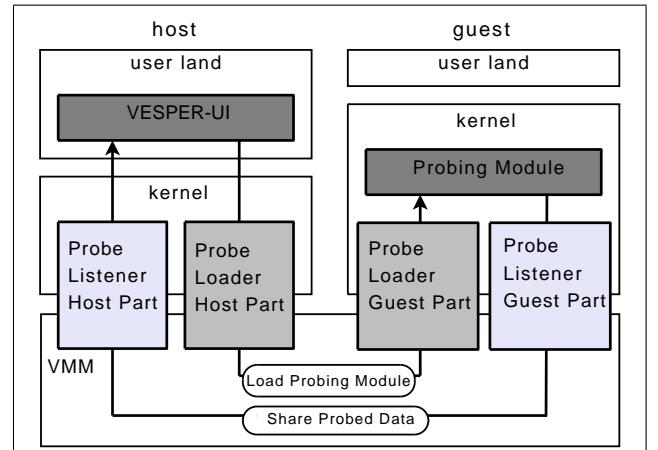


Figure 1: Architecture of VESPER

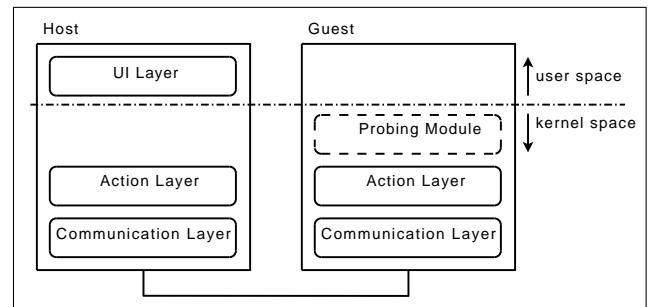


Figure 2: Layer of VESPER

1. UI Layer

This is the interface layer between VESPER and user applications which manage the guest in the host. This layer provides interfaces for loading and unloading probing modules to/from the guest and accessing probed data recorded in the guest. These are presented in detail in Section 5.

2. Action Layer

This is the worker layer which processes requests from the UI and Communication Layer. Concretely speaking, in this layer VESPER does actual work for load/unload probing modules to/from guest and sharing probed data between host and guest.

3. Communication Layer

This is the layer which provides communication channels between host and guest. It strongly depends on VMM architecture. By implementing this layer with respect to each VMM, VESPER confines the differences between VMM environments to this layer.

3.2 VESPER Semantics

In order to implant probes into guests, VESPER must load probing modules from the host to the guest. And then, VESPER sends probed data from the guest to the host.

Figure 3 illustrates the semantics overview of VESPER.

3.2.1 Module Loading

The first step to probe the guest kernel is to load the probing module onto the guest.

0. Make Module

First of all, one should make a probing module which uses Kprobes and relayfs.

A1. Module Load Command

Execute the probing module insertion via interfaces provided by the probe loader. One can also specify module parameters, if needed.

A2. Obtain Module Information

On the host-side Action Layer of the probe loader, from user space, VESPER obtains the module information to insert such as the module's name, its size, and its address with others related to module parameters, if any.

A3. Send/Receive Request

Through the interface provided by the VMM, the probe loader transfers the probing module's information between the host and guest.

A4. Load Module

In the Action Layer, the probe loader on the guest side loads the module without userspace help.

A5. Share Relay Buffer

In the Action Layer, the guest's probe listener gets relayfs buffer information such as the read index, buffer ID, etc., from the probe modules; it then exports the buffer to the host.

A6. Send/Receive Buffer Information

Communication layer of guest probe listener transfer the shared buffer information to host via VMM interface, and then, host probe listener receives it.

A7. Setup Relayfs Structure

The Action Layer of the host's probe listener builds the relayfs structure based on the information received from the guest.

A8. Analyze/Offer Probed Data

One can read probed data through the UI Layer of the probe listener.

3.2.2 Probing

After finishing the procedures in Section 3.2.1, the host and guest probe listeners share relay buffers of the probing module. Consequently, it is not necessary to transfer all the recorded data from guest to host, but it is necessary to transfer index information about shared relay buffers, where the data is, to get the start index for the actual access to relay buffers by host.

B1. Gather Guest Kernel Data

Once loaded, the probing module puts data into the relay buffer in the probe handler.

B2. Get Index Information

When change occurs in the relay sub-buffer, the action layer of the guest probe listener gets the index information and creates a message to notify host.

B3. Send/Receive Message

Through the communication layer, the host probe listener is notified of the index data from the guest.

B4. Update Index

The action layer function of the host probe listener updates the index of relayfs in the host, based on the received message.

3.2.3 Module Unloading

Basically, unloading a probing module below is similar to loading a module. The significant difference is that it takes two steps in the unloading module process, because before removing the relay buffer in the handler in the guest, the exported user interface for the buffer in host should be dropped.

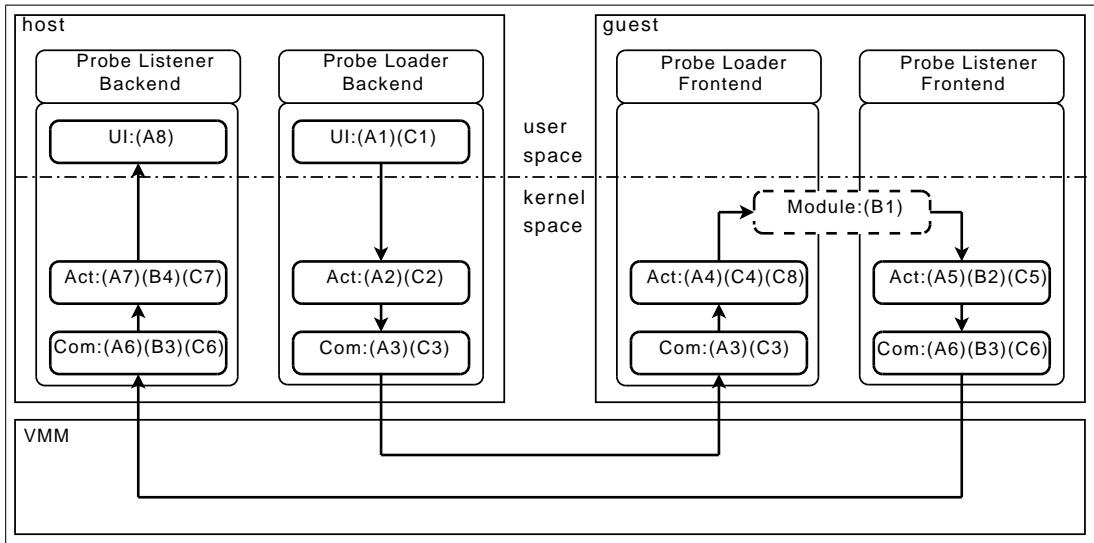


Figure 3: Process Flow of VESPER.

- C1.** Module Unload Command
- C2.** Obtain Module Information
- C3.** Send/Receive Request
- C4.** Unload Module (step1)
- C5.** Stop Sharing Relay Buffer
- C6.** Send/Receive Buffer Information
- C7.** Destroy Relayfs Buffer
- C8.** Unload Module (step2)

In the next section, we describe the detailed implementation of the probe loader and listener.

4 Implementation of VESPER

As previously described, VESPER consists of two components named probe loader and probe listener. Each component is split into three parts, UI Layer, Action Layer, and Communication Layer, to confine the dependency on the underlying VMM. In this section, we present the implementation of VESPER from the viewpoint of the Communication Layer on Xen.

4.1 Probe Loader

Probe Loader loads probing modules from host to guest without using the guest's user space. To make this concept real, probe loader needs to perform two functions. One is to transfer the probing module from the host to guest, and the other is to load the module in the guest without involving user space.

4.1.1 Module Transfer Function

In order to implant the probing module from the host into the guest memory space, VESPER needs to be able to transfer the module image somehow. Generally, VMM provides I/O infrastructure between host and guest using a shared memory mechanism. In the Xen environment, grant table and I/O ring are provided for that. To use them for transferring the module, probe loader is implemented as split drivers—frontend driver in the guest, and backend driver in the host, on Xenbus.

Backend driver allocates shared memory using grant table and write the module image into it. And then, the driver pushes a request onto I/O ring for the frontend driver. After receiving the request through I/O ring, frontend driver maps the shared memory to its virtual memory space.

For security reasons, it is normally the frontend driver that requests I/O from the backend driver. If, instead,

the backend driver issued I/O requests to the frontend driver, because frontend driver should have read/write permissions on the host to accomplish the requests, this would result in the guest being able change the contents of the host's page.

In VESPER, however, (as mentioned above) the host must issue requests to the guest for loading the module. From a security perspective, we must ensure that VESPER's communication protocol between host and guest follows this pattern—that is, that the frontend driver requests, and the backend driver responds. To keep this pattern, the frontend driver first issues a dummy request to backend driver. Then, the backend driver issues the module loading request as a response to the dummy request in the protocol, when the UI layer in host triggers. At this phase, the frontend driver has received all the information about the module to insert and then allocates a grant table for the module image. After grant table allocation, the frontend driver issues the real module loading request. With that request, the backend driver does `copy_from_user` to the grant table with respect to the module for the response. Finally, the result of loading the module is sent as a new dummy request from frontend driver. When next request is triggered from the host, the backend driver only issues a new request as the response to the last dummy request for the next process. The processes above are involved for every module insertion request in the host. The details of the protocol follow.

1. Frontend driver issues a dummy request.
2. Application triggers the module loading into guest.
3. Backend driver issues the module loading request as a response to the dummy request.
4. Frontend driver issues a real request with grant table allocated.
5. Backend driver copies the module into the grant table.
6. Frontend driver loads the module.
7. Frontend driver issues a new dummy request with the last loading module result.
8. Backend driver hands in the result of the module loading to the application.
9. Repeat Steps 2 through 8 for every request from the application.

From the benefit of the protocol, we sacrifice only one dummy request incurred in the initialization phase of the drivers, which is very trivial compared to the security hole.

4.1.2 Module Load Function

Sometimes there is the situation that a user program goes out of control. In such cases, the user program cannot be executed, but a kernel program can. If it is possible to load the module without user space help, one is able to analyze system faults even in the case above. Thus, in VESPER, the module load function is implemented without using a user-space program.

Currently the core system of the module loader in Linux, such as `load_module`, calls `copy_from_user` to get module images because it assumes that module loading is executed in user space. Inside `copy_from_user`, `access_ok` is called to verify its memory address. However, it never checks whether the calling function is executed in user space; it only checks whether the address limit is in its process address space. Hence, we implemented the module load function as a kernel thread in the Action Layer of the guest probe loader. This kernel thread calls `sys_init_module`, which calls `load_module`. Because the loaded module is already copied from the host via the module transfer function described above, `copy_from_user` works properly.

Nevertheless, it is impossible to invoke `sys_init_module` and `load_module` from external kernel modules, because they are not exported by `EXPORT_SYMBOL`. To address this problem, in VESPER, we get the address of these symbols from the guest kernel symbol table, and then pass them as parameters to the user interface probe loader provides for loading.

4.2 Probe Listener

The Probe Listener should retrieve the probed data from the guest and make it available to user-space applications in the host. Probing modules use relayfs to record the probed data which describes the behavior of the guest kernel around the probe points.

In fact, relayfs tends to allocate its buffers by pages, and grant table is also a page-oriented mechanism. So, provided that the buffers controlled by relayfs are allocated by grant table in the guest, a copy process is definitely unnecessary between host and guest to share the data, because grant table is transparent to both host and guest. A design decision on using grant table as relayfs buffers could also eliminate the need of other control mechanisms onto buffers than relayfs rchan.

In the case of sharing the probed data in the buffers, VESPER should also share and update some information about the buffers, such as the read index and padding value for relayfs in the host, to access the buffers.

As a result, Probe Listener consists of two components, which are a buffer share function and an index update function, and it is implemented as split drivers, just like Probe Loader.

4.2.1 Buffer Share Function

As mentioned before, because the probing module records probed data into relay buffers, Probe Listener shares them between host and guest. Sharing the buffers is implemented by using grant table like the module transfer function in Probe Loader. Similarly, information about which buffers need to be shared is provided from guest to host by using I/O ring.

Once the relay buffers are exported by the guest and their information is received by the host, the relayfs structure is built on the host to provide probed data in the buffers to user-space applications. At this time, Probe Listener does not call `relay_open` to create a `rchan` structure, which is a control structure of relayfs. This is because Probe Loader does not need to newly allocate the pages for the relay buffers, but should just map the pages exported by the guest. Therefore, Probe Listener sets up the `rchan` structure manually. After `rchan` is set up, the interface to read this relay buffers is created on `/sys/kernel/debug/vesper/domid/modname/` like other subsystems which use relayfs. User applications can read this interface directly, or use APIs abstracted by VESPER.

Finally, to stop sharing the buffer, the probe listener executes the above process in reverse. Removing the relay

structure is done at first, and then exporting relay buffers is stopped.

4.2.2 Index Update Function

When the probe listener shares the relay buffers between host and guest, it must synchronize some buffer information such as the read index between both `rchan` structures. If it does not, the user application on the host cannot read the probed data correctly. Probe Listener uses I/O ring for the information transfer. The guest's probe listener creates a message including the information, pushes it to I/O ring, and then notifies the host's probe listener. The host's probe listener gets the message from I/O ring and updates its own relay buffer information with the message.

Ideally, probe listener should update that information immediately whenever the guest `rchan` is changed. However, message passing by I/O ring is too expensive to update each time due to the intervention of the interrupt mechanism to notify host of the existence of pending messages. Hence, Probe Listener updates the buffer information when switching to sub-buffer occurs. In doing so, probe listener updates the buffer control information, and the user application can get the latest data probed from the guest.

5 VESPER Interface

This section describes the VESPER Interface.

5.1 The VESPER User API

VESPER provides user applications with simple interfaces to insert probing modules to target guests and to obtain probed data from the guests in Figure 4. Arguments of `virt_insmod` and `virt_rmmmod` are simply the same as `insmod` and `rmmmod`, Linux user commands to handle kernel modules, except that they target the guest. In addition, `virt_is_alive` is available for the application to check if some error occurs around the probed point.

5.2 The VESPER Module API

VESPER defines an API to export relay buffers of the probing module to the host. Additionally, VESPER provides a callback function for use when the sub-buffer of

```
int virt_insmod(
    const int target_guest,
    const char *modname,
    const char *opt);

int virt_rmmmod(
    const int target_guest,
    const char *modname,
    const long flags);

bool virt_is_alive(
    const int target_guest,
    const char *modname);
```

Figure 4: Prototype of the VESPER user API

the relay is changed. All probing modules should call the exported function after `relay_open`, and the stop export function before `relay_close`. The callback function also is set up to `subbuf_start`, the member of `struct rchan_callbacks`. Figure 5 shows the prototype of the module API.

```
int relay_export_start(
    struct rchan *rchan,
    const char *modname);

void relay_export_stop(
    const char *modname);

int virtrelay_subbuf_start_callback(
    struct rchan_buf *buf,
    void *subbuf,
    void *prev_subbuf,
    size_t prev_padding);
```

Figure 5: Prototype of the VESPER module API

6 Evaluation

In this section, we will examine the benefits of VESPER with a webserver running on a Xen-based guest, and will explain how VESPER works with Heartbeat to eliminate the latency using the test case we plan to build.

6.1 Test environment

For this experiment, we plan to prepare two physical machines. We set up two guests as resources managed by the LRM (Local Resource Manager) of Heartbeat on each physical machine. In fact, it is a controversial issue on how a guest is treated in the cluster, as one of the cluster nodes, or as a resource like IP. However, we will treat guests as a resource to avoid any complexity of management caused by difficulty in identifying host and guest from the all nodes, in case a guest were treated as a node. On each physical machine, the webserver is on the one guest, we say VM1; it is actively performing web service. On the other hand, the webserver in the other guest, we say VM2, is inactive. Figure 6 depicts the details.

When something wrong happens to VM1, Heartbeat lets VM2 take over all of roles which VM1 was performing. However, one physical machine, P1, has Heartbeat's LRM without involvement of VESPER to show how Heartbeat works in the usual way. However, the other physical machine, P2, has LRM cooperating with VESPER. Here, we treat the guest as a resource so that we plan to add virtual machine resource plugin conforming to OCF (Open Cluster Framework) to LRM to make Heartbeat and LRM recognize a virtual machine as a resource. The plugin has interfaces for start, stop, and monitor (a.k.a. status) the resource. In implementing the resource plugin, the monitor interface of the plugin will invoke VESPER for LRM on P2 only.

6.2 Implementation of virtual machine resource

In Heartbeat, CRM (Cluster Resource Manager) coordinates what resources ought to run where, or which status they are running, working with the resource configuration it maintains. And it commands LRM to achieve all the things. LRM then searches for proper resource to handle by way of the PILS subsystem of Heartbeat. LRM calls exported interfaces by the resource to execute CRM requests, start/stop/monitor, etc. We newly define a virtual machine resource, and it exports a start/stop/monitor interface implemented as follows.

- *start*. The resource invokes the `xm` command (Xen tool) to start a specific guest.
- *stop*. The resource also invokes `xm` (Xen tool) to stop a specific guest.

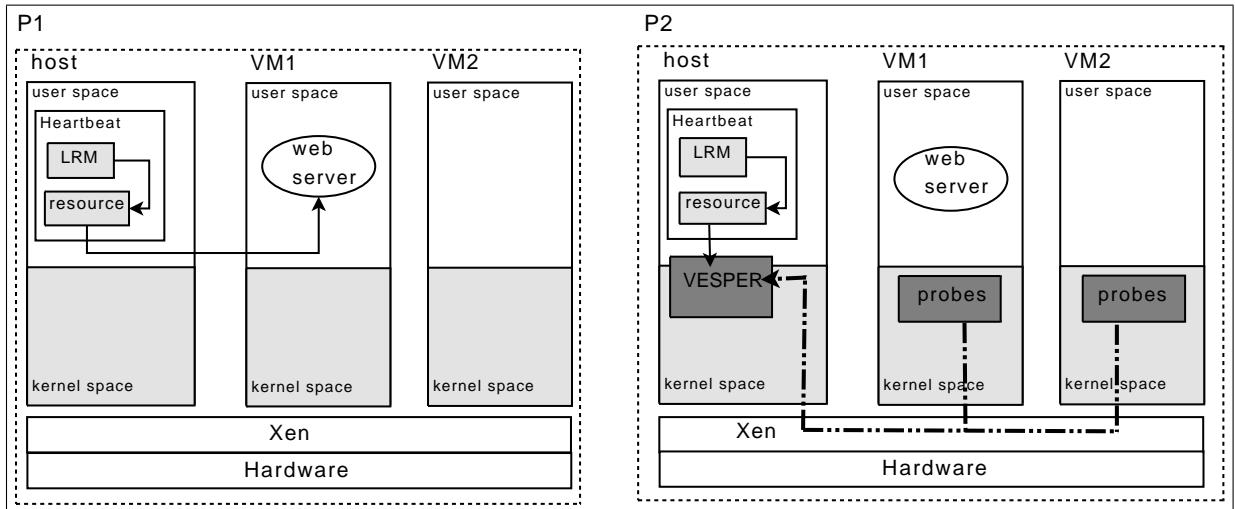


Figure 6: Test environment to demonstrate the usability of VESPER.

- *monitor*. The resource invokes the `xm` command (Xen tool) to get the status of a specific guest as well as monitor procedure, prepared for this test, to get status of the services running on that guest. What is more, the resource on P2 invokes the VESPER interface and does a logical OR on the results of the above two for the return value, only in case of P2.

6.3 Expected result and discussion

For simplicity, we insert Kprobes around the panic path of the guest kernel via VESPER. Kprobes thus inserted will put clues like the callstack to the panic on the relayfs when the panic occurs. Then, we cause a panic intentionally on VM1 and measure the recovery time on P1 and P2. We can easily expect that P2 will recognize what happened to VM1 of P2 as soon as VM1 panics, because of the prompt notification done by VESPER. Obviously service recovery time is supposed to be dramatically improved by as much as deadtime we set for Heartbeat. Moreover, one can find out the reason why VM1 on P2 panicked through the probed data on the relayfs later on.

Through the experiment, we could verify better performance on response latency and usability of failure analysis provided by VESPER.

However, special care should be taken with two issues regarding probes. One is what probe points are suitable for proper monitoring. If the targeted, necessary probe

points miss, no more improvement over usual Heartbeat can be expected. Actually, the problem on where probe points should be inserted seems very tricky to handle, because highly experienced developers or system administrators on kernel context and applications running on the server are required to select optimal probe points. The other is about overhead produced by the execution of probes. One should adjust the overhead according to the required service performance. Both issues exclude each other. More probes inserted to hit fine-grained events cause more overhead in probing, obviously. Some mechanism to help one select optimal probes could be needed. Some suggestions for these issues will be mentioned as future works of VESPER in the next section.

7 Conclusion and future works

In this paper, we proposed VESPER as a framework to insert probes in virtualized environments and discussed what topics VESPER can solve in clustering computing. After that, we described the design and the implementation of VESPER. Then we suggested a test bed to show the performance improvement on fail-over response latency and failure analysis. Finally we discussed some considerations on places and overhead of probing. To address these considerations, we have some plans about future VEPSER developments.

7.1 Probing aid subsystem

For the ease use of cluster manager or other applications, we plan to develop a probing aid subsystem. Probing

points could be classified into several groups based on their functionality. The subsystem thus can pre-define several groups of probe points and abstract them to its clients or application—like memory group, network group, block-io group, etc. The clients just select one of groups, and the subsystem will generate all needed probes relayed to VESPER. Also, fine-grained selection from several groups will be supported by the subsystem.

7.2 SystemTap Enhancement

We have a plan to integrate the feature of VESPERs for virtualization into the SystemTap [8] for its versatile usage in the native kernel as well as the virtualized kernel.

7.3 Virtio support and evaluation on KVM and lguest

Virtio is likely to promise one standard solution to host and guest communication infrastructure on various VMMs. VESPER should support virtio and be evaluated on KVM and lguest [9] to verify its portability and usability.

The next two are not related directly to probing technique but are worth examining as enhancements of functionality of virtual machine resource facilities in clusters in terms of virtualization technology.

7.4 Virtual machine resource support for LRM

Arguably, there is a question on how services running on virtual machines should be treated if a virtual machine is treated as a resource. Should the services be handled at the same level as the virtual machine itself in LRM? When the services go fail-down, only the failed service must be moved to another virtual machine on the same physical machine—better than virtual machine itself should be switched to another virtual machine on a different physical machine. The next version of interface VESPER exports will suggest the solution to that.

7.5 Precaution capability on collapse of the host

If the host collapsed, all services running on the guests would be lost. It is obviously a big problem. Therefore, VESPER should probe the host simultaneously to check whether the host is in good condition. When VESPER catches a sign of the host's collapse, the cluster manager notified by VESPER could take necessary action, such as live migration to other host.

8 Acknowledgments

We would like to thank Yumiko Sugita and our colleagues for reviewing this paper.

9 Legal Statements

Linux is a registered trademark of Linus Torvalds. Other company, product, and service names may be trademarks or service marks of others.

References

- [1] Alan Robertson, “Linux-HA Heartbeat Design,” In *Proceedings of the 4th International Linux Showcase and Conference*, 2000.
- [2] Heartbeat, <http://linux-ha.org>.
- [3] Ananth N. Mavinakayanahalli et al., “Probing the Guts of Kprobes,” In *Proceedings of the Linux Symposium*, Ottawa, Canada, 2006.
- [4] Nguyen A. Quynh et al., “Xenprobes, A Lightweight User-space Probing Framework for Xen Virtual Machine,” In *USENIX Annual Technical Conference Proceedings*, 2007.
- [5] Nitin A. Kamble et al., “Evolution in Kernel Debugging using Hardware Virtualization With Xen,” In *Proceedings of the Linux Symposium*, Ottawa, Canada, 2006.
- [6] The Xen virtual machine monitor, <http://www.cl.cam.ac.uk/Research/SRG/netos/xen/>.
- [7] KVM, <http://kvm.qumranet.com/>.
- [8] SystemTap, <http://sourceware.org/systemtap/>.
- [9] Rusty Russell, “lguest: Implementing the little Linux hypervisor,” In *Proceedings of the Linux Symposium*, Ottawa, Canada, 2007.
- [10] VESPER, <http://vesper.sourceforge.net/>.

Camcorder multimedia framework with Linux and GStreamer

W. H. Lee, E. K. Kim, J. J. Lee , S. H. Kim, S. S. Park
SWL, Samsung Electronics
woonghee.lee@samsung.com

Abstract

Along with recent rapid technical advances, user expectations for multimedia devices have been changed from basic functions to many intelligent features. In order to meet such requirements, the product requires not only a powerful hardware platform, but also a software framework based on appropriate OS, such as Linux, supporting many rich development features.

In this paper, a camcorder framework is introduced that is designed and implemented by making use of open source middleware in Linux. Many potential developers can be referred to this multimedia framework for camcorder and other similar product development. The overall framework architecture as well as communication mechanisms are described in detail. Furthermore, many methods implemented to improve the system performance are addressed as well.

1 Introduction

It has recently become very popular to use the internet to express ourselves to everyone in the world. In addition to blogs, the emerging motion video service provided by such companies as YouTube and Metacafe help us to use internet in this way. The question is, how can we record the video content we want to express?

Digital camcorders, cameras and even mobile phones can be used for making movies. But the quality generated by mobile phones or digital cameras is generally not as good as that of a digital camcorder. If users want to make higher quality content they must use digital camcorders.

In this paper we introduce a camcorder multimedia framework with Linux and GStreamer. We take into account portability and reusability in the design of this framework. To achieve portability and reusability we adopt a layered and modular architecture.

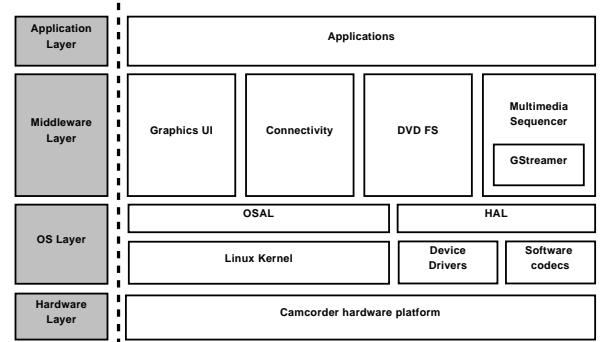


Figure 1: Architecture diagram of camcorder multimedia framework

The three software layers on any hardware platform are application, middleware, and OS. The architecture and functional operation of each layer is discussed. Additionally, some design and implementation issues are addressed from the perspective of system performance.

The overall software architecture of a multimedia framework is described in Section 2. The framework design and its operation are introduced in detail in Section 3. Development environments, implementation, and performance issues are represented and discussed in Section 4. Finally, we present some concluding remarks in Section 5.

2 Multimedia Framework Overview

The three layers of the multimedia framework are application, middleware and OS. The architecture of the multimedia framework is shown in Figure 1.

2.1 Application layer

There are many programs, such as players and recorders for movie and still pictures, User Interface (UI) managers, USB control, navigator and camera manager in

the application layer. The player and recorder are similar to the general media player and recorder present in standard platforms, however they have additional features in order to support DVD media. The UI manager interacts with camcorders through the keypad, sound and display. USB control manages USB connections. Users can browse DVD titles and select a particular clip with the navigator module. All camera specific functions such as Image Stabilizer and Auto Focusing are implemented inside the camera manager. Because the application layer outside of this paper's scope, we will not discuss it further.

2.2 Middleware layer

The Middleware layer is categorized into four functional groups: multimedia, connectivity, UI, and DVDFS (DVD File System). The multimedia module includes the DVD sequencer, GStreamer and many media specific plugins. USB specific functions are implemented in the connectivity module and are further broken into three major functional blocks: USB Mass Storage (UMS), Digital Print Solution (DPS) and PC Camera (PC-Cam). The UI module includes FLTK and Nano-X. DVDFS is the module that controls the DVD disk file system.

2.3 OS layer

The OS layer plays an important role in system management, OS services, and hardware abstraction. It consists of OSAL (OS Abstraction Layer), HAL (Hardware Abstraction Layer), Linux kernel, device drivers and software codecs. The OSAL provides the middleware an abstraction eliminating the OS dependency. The HAL also provides the middleware with an abstraction eliminating the hardware dependency. The role of the Linux kernel is the management of the system and the support of the general OS environment. The device drivers are used to control the hardware. The multimedia codecs not supported by hardware in the platform are implemented by software.

2.4 Hardware layer

In this paper, the hardware layer represents the camcorder hardware platform including a camcorder specific multimedia SoC chip and its supporting board. The

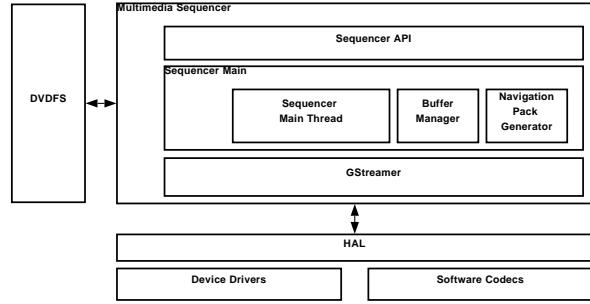


Figure 2: Structure of the multimedia sequencer

SoC chip supports multimedia oriented operations such as MPEG-2 coding, multiplexing and de-multiplexing of the DVD stream, and IO operations to the DVD disc. In this paper, an application product is assumed to be a camcorder. The supporting board has NOR flash, SDRAM, DVD loader, LCD screen, key pad, camera module and so on.

3 Architecture design

3.1 Multimedia subsystem

3.1.1 Sequencer

The multimedia sequencer is a middleware module that has functions related to the multimedia control, such as playback and recording.

The architecture of the multimedia sequencer is described in Figure 2.

The multimedia sequencer is configured using interfaces such as the sequencer API layer, sequencer main module, and GStreamer multimedia engine. The GStreamer plugins call the device drivers or the software codecs through the HAL APIs. A developer can create and maintain the plugin codes easily using the HAL APIs. The DVDFS module is used for reading and writing a DVD disc in the sequencer. It is used in both the DVDSrc plugin and the buffer manager block. For playback, the DVDSrc plugin reads the stream data stored in a DVD disc using the DVDFS. For recording, the buffer manager uses the DVDFS for writing the recorded stream data to the DVD disc.

In the sequencer API layer, there are functions related to creation, initialization and control of the sequencer.

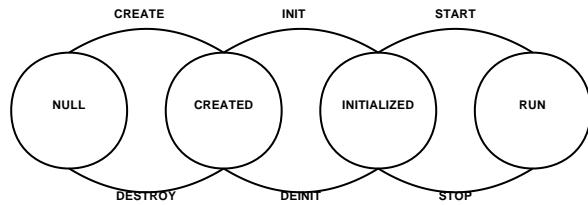


Figure 3: State diagram of multimedia sequencer

The application can create, destroy, initialize, and de-initialize the multimedia sequencer. There are three modes and two types in the multimedia sequencer. The modes are categorized as DVD-Video, DVD+VR and DVD-VR, and the two types are playback and record. For example, if the application initializes the multimedia sequencer using DVD-Video mode and playback type, then the sequencer is set for the DVD-Video player. The sequencer API also supplies control functions. Using these control functions, the application can start, stop, pause and resume the sequencer. And the application can register callback function pointers to the sequencer using the init function. The sequencer can send information to the application at any time using this registered callback function.

In the main layer of the sequencer, the three function blocks are the thread block, buffer manager and navigation pack generator. In the main thread block, the thread routine reads and processes messages in a queue. When the sequencer API is called by the application, the API function is converted to a message. This converted message is sent back to the message queue in the main thread block of the sequencer.

When the sequencer is operating as a DVD recorder, the buffer manager and the navigation pack generator are activated. In the DVD recorder, VOB data has to be recorded on the DVD disc. VOB data contains the presentation data and part of the navigation data. VOB data may be divided into CELL's, which are made up of VOBU's. The video and audio data is packed and recorded as a VOBU unit. The navigation pack is stored at the start of each VOBU data. Because CELL information has to be stored in the navigation packs of VOBU's in the CELL, buffering of CELL data is necessary. The buffering manager controls the VOBU data and calls the navigation pack generator to make navigation pack data of VOBU's in the CELL.

The four states of the multimedia sequencer are NULL, CREATED, INITIALIZED and RUN. The states can

change as described in Figure 3. Each state can be managed and set to in the multimedia sequencer based on the state of the GStreamer core.

3.1.2 Interface between GStreamer and sequencer

In this section, we describe the interface between GStreamer and the multimedia sequencer. The GStreamer pipeline configuration for DVD playback and recording is also discussed.

When the multimedia sequencer is initialized, the `gst_init()` function is called for GStreamer framework initialization.[1] After the GStreamer framework is initialized, shared libraries for plugins are loaded using the `gst_plugin_load_file()` function. After loading plugin libraries, the sequencer creates a pipeline using `gst_pipeline_new()` function.

To monitor the state of the GStreamer framework, the sequencer creates an event loop thread that checks messages from the GStreamer framework. In this thread, the sequencer gets a bus from the created pipeline and checks the `GstMessage` from the GStreamer framework using `gst_bus_poll()` function. To send the `GstMessage` to the application, the sequencer uses the registered callback function.

When the application calls the `start()` function of the sequencer, the loaded GStreamer plugins are registered using `gst_element_factory_make()` function and then the pipeline is configured. After the pipeline is configured, the properties of plugins are set using `g_object_set()` function.

To change the GStreamer state, the sequencer calls the `gst_element_set_state()` function. When the playback or recording operation is started, the state of GStreamer is changed to `GST_STATE_PLAYING`. To pause the GStreamer framework, the state is changed to `GST_STATE_PAUSED`.

`GObject` signals are used to communicate between the sequencer and the plugin. In GStreamer, the `GObject` signal is already defined for notifying the application of plugin events. In order to register the signal in the sequencer, the `g_signal_connect()` function is called in the sequencer.[2] After the plugin sends the signal to sequencer, the signal handler function in the sequencer is called. The detailed structure of GStreamer pipelines for

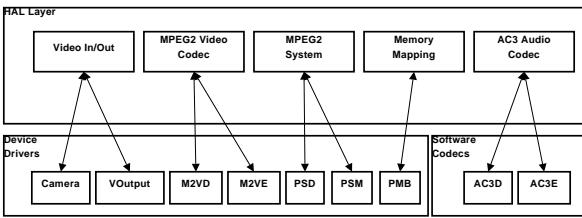


Figure 4: HAL layer with device drivers

playback and record are discussed in sections 3.1.4 and 3.1.5.

3.1.3 Hardware abstraction layer

There are some hardware IP blocks for DVD camcorders in the hardware layer. These hardware IP's can be accessed and controlled by device drivers. These device drivers are wrapped by the HAL layer described in Figure 4. The GStreamer elements related to the device drivers are implemented using the HAL layer API's.

In the HAL layer, there are wrapper functions for video input and output. The device driver functions of camera and VOutput are wrapped in the HAL. The device drivers for MPEG2 video are M2VD and M2VE. These device drivers are wrapped up in the HAL as the MPEG2 video codec. For MPEG2 system layer, there are some hardware IP's such as MPEG2 PS demuxer and muxer. These short names for these device drivers are PSD and PSM. These device drivers are wrapped in the HAL layer as MPEG2 system. Because the hardware IP requires a physical memory buffer for input data, the user space program has to be able to access the physical memory buffer area. For this reason we use memory mapped IO for mapping a physical memory area in device driver into a virtual memory area in user space. An additional benefit of using memory mapped IO is that memory copy operations can be avoided. In device drivers for hardware IP's, the physical memory area can be assigned as the input and output buffer. But, if an element is not connected to such device drivers, the element has to use a device driver that enables the physical memory area mapping. For this reason the PMB device driver is implemented for memory mapping in the HAL layer. The operation of PMB will be described in detail in section 4.2.2. The AC3 audio encode and decoder codecs are needed for a DVD camcorder. These software codecs and named as AC3D and AC3E. The au-

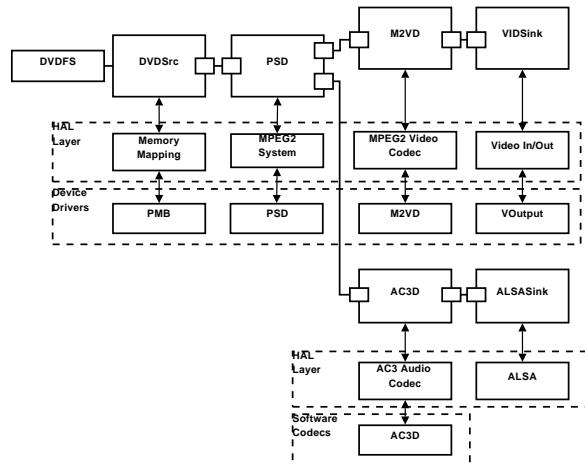


Figure 5: Playback pipeline

dio codecs are wrapped in the HAL layer as AC3 audio codec.

3.1.4 Playback pipeline

In this section, the structure and HAL interface of the playback pipeline are discussed. Its diagram is shown in Figure 5.

The DVDSrc element reads the stream data from DVD disc using DVDFS and stores the stream data in the stream buffer. Because the PSD element requires physical memory for the input buffer, the input stream buffer has to be assigned to a physical memory area. This physical buffer is assigned by the PMB device driver through the memory mapping HAL layer and is mapped into the virtual memory address in the DVDSrc element. The DVDSrc element stores the stream data into the physical memory buffer using the mapped virtual memory address. The base addresses of physical and virtual memory and the memory size are sent to the PSD element using GstCaps. The input stream buffer memory is configured and managed as a ring buffer in the DVDSrc element. The address of the input data is sent to the PSD element through the GstBuffer.

The PSD element parses the input stream data and stores the parsed data into the video and audio stream buffers that assigned by the PSD device driver through the MPEG2 system HAL layer. This parsing operation is processed using the PSD device driver through the MPEG2 system HAL layer. The stream buffer for the parsed data is located at a physical memory area and the

address of this stream buffer is sent to the M2VD element through the GstBuffer.

The M2VD element is used for MPEG2 video decoding. In the M2VD element, the M2VD device driver is connected through the HAL layer of the MPEG2 video codec. The address of the video stream buffer in the physical memory area is sent from PSD element. The M2VD element decodes the input video stream data and stores the decoded frame data into the physical memory buffer. This physical memory buffer is assigned by the M2VD device driver through the HAL layer of MPEG2 video codec. After decoding the video stream into the reconstructed frame, the M2VD element sends the physical buffer address of output buffer to the VIDSSink element through GstBuffer. Then, the reconstructed frame can be displayed on either LCD or TV.

For audio, the stream data is sent from the PSD to the AC3D element. Because the AC3 audio decoder is a software codec, it can access the input buffer by virtual memory address. The PSD sends the memory mapped virtual address to the AC3D element using GstBuffer. The AC3D element decodes the AC3 audio stream data to PCM data by AC3D software codec through the HAL layer of AC3 Audio codec. The decoded PCM data is sent to the ALSASink element. This ALSASink element is provided from the GStreamer base plugin.

Audio and video synchronization is controlled using the system clock provided from GStreamer. AV synchronization is implemented based on the time stamp data embedded in the stream data. The time stamp data, so called PTS (Presentation Time Stamp), is extracted from the navigation pack in VOBU data and converted to the GStreamer time format. All audio packs have the PTS, so the audio decoder can send the PTS to next element without additional calculation. In the case of video pack, however, the PTS only exists in the first video pack of GOP. Hence, the time information of the video packs has to be calculated using the video frame rate. In GStreamer, the calculated time information is embedded in GstBuffer structure and passed through the GStreamer pipeline. And then the audio and video synchronization can be achieved automatically in the GStreamer core.

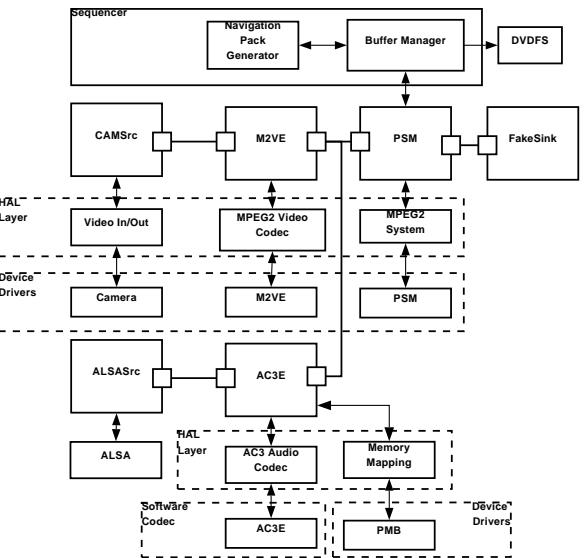


Figure 6: Record pipeline

3.1.5 Record pipeline

In this section, the structure and HAL interface of the record pipeline are discussed. Its diagram is shown in Figure 6.

The CAMSrc element is connected to the camera device driver through the video in/out HAL layer. The input frame data is stored in the physical address area that is assigned by camera device driver. The physical address of the input frame data is sent to the M2VE element.

The M2VE element encodes the input frame data and sends the physical address of the encoded video stream data to the PSM element.

The ALSASrc element sends the captured PCM data to the AC3E element. In the AC3E element, the PCM data is encoded by the AC3E software codec through the HAL layer of AC3 audio codec. Because the PSM device driver requires the physical memory address of the input audio stream buffer, the AC3E element uses the PMB device driver for memory mapping. The encoded audio data is stored in the virtual memory address by the AC3E element and the memory mapped physical address is sent to the next PSM element.

The PSM element packetizes the input video and audio stream data, and generates video and audio packs. In order to satisfy the MPEG2 system layer and the DVD standard, the PSM element uses the buffer manager in the multimedia sequencer. For synchronization

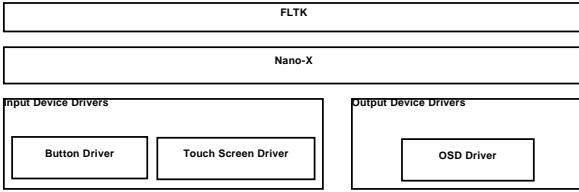


Figure 7: Structure of graphics subsystem

of recorded stream, the PSM element uses the GstCollectPads that is provided from GStreamer.

In PSM, the VOBU information from the packetized data is sent to the buffer manager. The buffer manager generates the navigation pack for the VOBU using the navigation pack generator. After CELL data is ready in the buffer manager, the CELL data is recorded to the DVD disc using DVDFS module. Because the disc write operation is processed in the buffer manger and DVDFS module, we use the FakeSink as a dummy element for the generation of a complete pipeline

3.2 Graphics subsystem

The structure of the graphics subsystem is depicted in Figure 7. The FLTK (Fast Light Tool Kit)[3] is a lightweight version of GTK+, and Nano-X[4] is a lightweight X window system. They are generally used for the graphics subsystem of embedded platforms due to their small size. They are supported by input and output device drivers. The input device can be a keyboard and a pointer like touchpad, and the output device can be on screen display (OSD) or the frame buffer.

3.2.1 Input and output device drivers

To support input devices, the button and the touch screen devices are used in the camcorder platform. The button driver takes care of the press, release and long press events from each key or button. The touch screen driver handles of click, double click and drag events.

The target platform has video output and OSD layers. The video output layer is used for displaying video frame data from camera module, and the OSD is used for displaying camcorder information like icons, numbers and menus. In the graphics subsystem, only the OSD layer is of interest. The OSD implementation is similar to that of the Linux frame buffer, however, it

needs an additional process. It should be set the palette of the specified OSD and its property as followings:

- Width and height
- Bit per pixel
- Screen color number
- Pixel format
- Frame buffer address
- Scale
- Chroma-key information

Additionally, the video output and OSD have different color formats, YUV and RGB respectively. Because the color format to be displayed on LCD or TV is YUV, the RGB data from OSD must be converted to YUV.

3.2.2 Nano-X

Nano-X is a lightweight X windows system developed by the Nano-X open source project. Although it implements a lightweight X window system, it offers lots of functionality so that it is sufficient for embedded devices such as digital cameras and camcorders.

Nano-X is implemented based on MicroGUI, which is a portable graphics engine and composed of Win32-like Microwindows API and X-like Nano-X API.

Nano-X generally supports keyboards and pointers such as mice as input devices, however buttons and a touch screen are used for the input devices of camcorders. The drivers of button and touch screen devices should work like those of keyboard and pointer devices.

3.2.3 FLTK

FLTK is a lightweight version of GTK+. FLTK takes an important role in creating and exploiting widgets, handling various events in the windows and widgets, and supporting OpenGL window to implement OpenGL's applications on the FLTK. Its role is similar to GTK+'s.

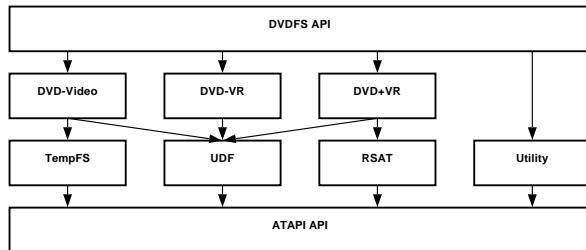


Figure 8: Architecture of DVDFS

FLTK is implemented in C++ so that we can create various types of windows and widgets, and handle the special events by sub-classing the widget and window class. FLTK supports FFLUID (UI Designer Toolkit) to help UI designers create an UI applications easily.

Because all applications in our framework are implemented in C, application programmers can not create or extend the widgets and windows implemented by C++. So the parts of FLTK implemented by C++ should be wrapped in API's for the mixed C/C++ development.

3.3 DVD file system

Nowadays there are various types of DVD-Disc, such as DVD-ROM, DVD-R, DVD-RW, DVD-RAM, DVD+R and DVD+RW. Each of them has different physical characteristics. Moreover, the formats for storing data on DVD-Disc can be categorized into three different specifications: DVD-Video[5], DVD-VR and DVD+VR. The camcorder software has to consider all of the types of media and DVD formats. DVDFS provides simple APIs for DVD recording and playback. The application and middleware can record and play the DVD contents without worrying about what types of DVD media and formats are used.

The architecture of DVDFS is shown in Figure 8.

Modules of DVDFS can be described as follows:

- **DVDFS API**-The unified user API set.
- **DVD-Video**-Module for DVD-Video format
- **DVD-VR**-Module for DVD-VR format
- **DVD+VR**-Module for DVD+VR format
- **Tempfs**-Temporary File System. It manages a temporary table which is used in the finalization on DVD-Video format.

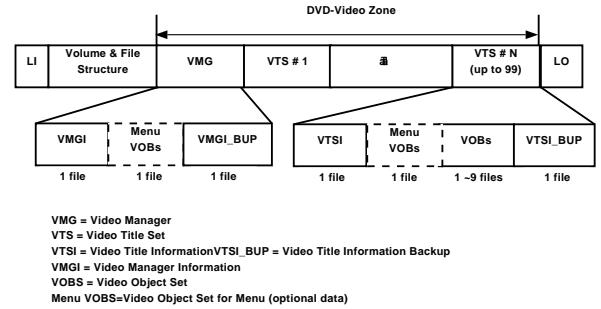


Figure 9: Layout of DVD-Video format

- **UDF-Module** for Universal Disc Format[6]
- **RSAT-Module** for Reserved Sector Allocation Table. It is used by DVD+VR module to translate the mapping information of first reserved area.
- **Utility-Module** for formatting, getting disc info.
- **ATAPI API-Wrapper** of the OS specific ATAPI.

DVD-Video, unlike other formats, does not support real-time recoding, so it needs to support Tempfs. We will focus on how to record through DVD-Video format in following sections.

3.3.1 DVD-Video format

DVD-Video format is a basic specification and most widely used for the distribution of movies. In the beginning it only considered pre-pressed disc like DVD-ROM, so the order of files is physically predetermined as showin in Figure 9. For that reason it is not suitable for real-time recording devices such as camcorders, recorders and so on.

3.3.2 Real-time recording on DVD-R

DVD-R is a write-once media. Basically, it has to be written by sequential-recording methods. This means that recording is only permitted at the next address of the written block. Therefore, DVD-R provides the incremental write mode. In this mode, the media can be divided into several areas and these areas are called RZone. Sequential-recording is performed in each RZone. DVDFS adopts this method to reserved areas

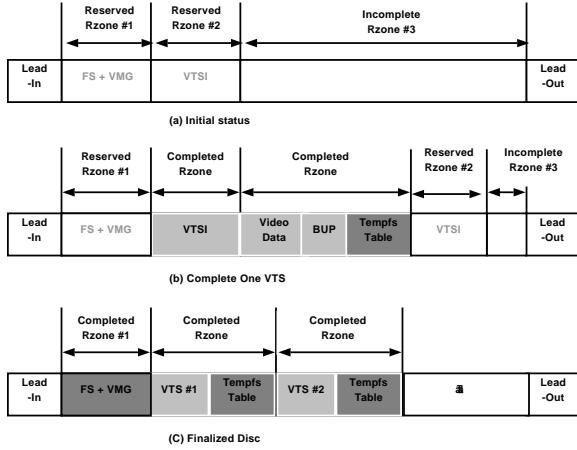


Figure 10: Real-time recording DVD-Video with DVD-R

for file system data and video information files which are determined after recording video data.

The management of RZone is the most important process in recording data on DVD-R. Disc status during the recording operation on DVD-Video and its sequence are described in Figure 10.

(a) depicts the initial status of disc. RZone #1 is reserved for file system and VMG. RZone #2 is reserved for VTSI. The remained area is automatically regarded as RZone #3, and it is used for recording movie data.

(b) depicts the status of the completed VTS. When user closes a movie data recorded, VTSI_BUP is generated and written continuously. Then, Tempfs table is written at the end of VTSI_BUP. It contains all data of file location, file size and time information. Once RZone is closed, it can not be managed any more. Then, DVDFS need to reserve another RZone for new VTSI.

(c) depicts the final status. Firstly, the volume and file structures of UDF are written to RZone #1. At that time, the last Tempfs table is used for constructing the UDF data structures. Then, VMG data is written on the end of UDF. Finally, the border will be closed.

3.3.3 Real-time recording on DVD-RW

In case of the rewritable media, formatting creates addressable blocks which can be overwritten. It takes a long time to format a whole disc. To avoid this problem, quick format is used. Formatting is performed in

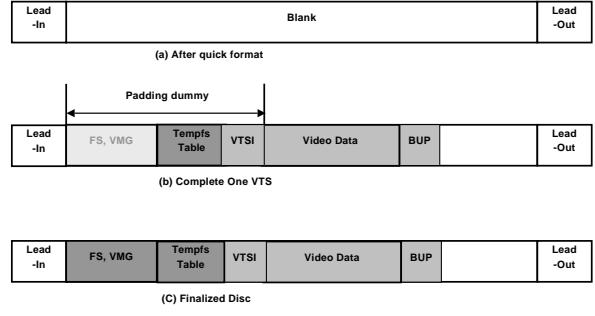


Figure 11: Real-time recording DVD-Video with DVD-RW

the small part of disc, and other parts are only used for sequential recording. Once a block is written, it turns into an over-writable block.

Figure 11 illustrates the sequence of recording operations on DVD-RW. After quick format, DVDFS reserves a space by writing dummy data. This area will contain file system, VMG, VTSI, and Tempfs table. Video data is written after that padded data.

Another difference between recording of DVD-R and that of DVD-RW is the location of Tempfs table. Because the over-writing is possible in DVD-RW, the Tempfs table is written on a fixed area

3.4 Connectivity subsystem

3.4.1 Architecture of USB Connectivity

The Linux kernel supports USB devices using the USB gadget framework.[7] It is made up of the Peripheral controller drivers, Gadget drivers and Upper layers. The Peripheral controller drivers manage and control the USB hardware IP. The Gadget drivers are the logical layers divided by their function. They can be a file system (Gadget file system), a network (Ethernet over USB), a serial or a MIDI. Upper layers are the supporting layers for user applications. They execute the specific functions such as UMS, DPS, Ethernet or serial.

The USB connectivity architecture is depicted in Figure 12. USB controller driver is the Peripheral controller driver in the USB gadget framework. It has to fit well with the API of the USB gadget framework. This makes it easier to implement or to use Gadget drivers. Although there are many kinds of Gadget drivers, only

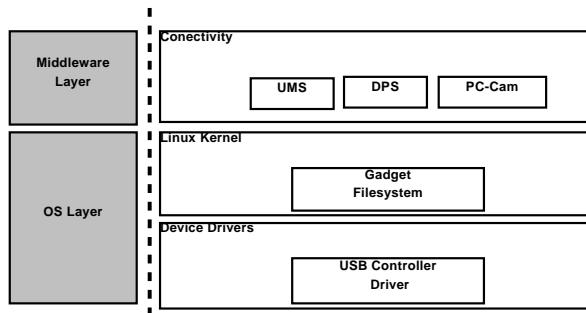


Figure 12: Architecture of USB connectivity

the Gadget file system is used in our camcorder framework. In the middleware layer, the connectivity module supplies UMS, DPS and PC-Cam.

3.4.2 Blocks of USB connectivity

UMS transfers files to computer. This requires some implementations such as:

- **USB Mass Storage Class Bulk-Only Transport**
 - It sends commands through CBW(Command Block Wrapper) and gets the result of CBW by receiving CSW(Command Status Wrapper).[8]
- **SCSI protocol** - It is a protocol included in CBW. It has commands like READ and WRITE.[9]
- **Block device control function** - It enables UMS to get block device information such as media type and number of sectors from block device. And it must be able to read and write from/to the block device.

DPS enables the camcorder to connect to a printer directly and to print images. The connected printer must have the function of PictBridge. This needs some implementations such as:

- **Picture Transfer Protocol**-It is a protocol for digital cameras to send images to other devices like PC and printers.[10]
- **PictBridge**-It is an industry standard written by CIPA (Camera & Image Products Association). It includes the methods for the device discovery and sending the information of images.[11]

PC-Cam enables a camcorder to send the captured images and sounds. Data transfer is divided into two parts - video and audio. Video data is compressed by JPEG and transferred to PC. The methods for transferring audio data observe the definition of USB Audio Class.[12]

3.5 OS

3.5.1 OS abstraction layer

The original purpose of OSAL was to remove the OS dependency from software. If middleware and application have such a dependency on the OS, they can't be reused on different OS's. However, it causes some overhead to cover up the difference between many OS's. Therefore, we designed the OSAL to achieve its original purpose and to reduce overhead simultaneously. To meet this goal, the number of categories and functions of OSAL are limited to the minimum as far as possible. The categories consist of task, semaphore, message queue, mutex, timer and system timer. The functions of OSAL are limited to creating, deleting and a few action functions.

3.5.2 Porting Linux kernel

In order to support the camcorder platform, the Linux kernel needs to be ported with board specific code and device drivers. The first work is to select the appropriate ARM core code from many versions of codes already existing in the Linux kernel source. Because our platform has ARM 11 core, the ARMv6 code of Linux kernel source was selected. The second is to include machine specific code. This code initializes and controls the peripherals of the SoC Chip such as Timer, Clock, DMA, IO mapping and so on. Finally, device drivers not existing in the standard kernel source need to be created. They support many kinds of general and camcorder specific devices. They are almost based on open source and managed by HAL layer and GStreamer elements in middleware.

4 Implementation

4.1 Development environments with emulator

The emulator is a useful tool for development and testing of applications on a PC. It is helpful when a real target is not available. It provides an emulated camcorder

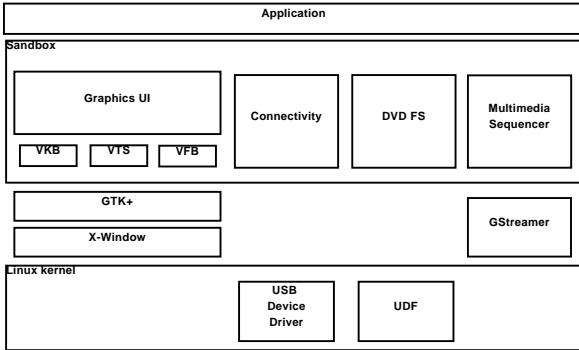


Figure 13: Architecure of emulator

framework. The user interface of the emulator includes keyboards, buttons, menu toolbar, touch screen and so on. It is based on X Window, GTK+ and sandbox modified from scratchbox[13]. The architecture of emulator is shown in Figure 13.

In general, the emulator supports virtual devices for UI interface. The camcorder emulator supports virtual keyboard (VKB), virtual touch screen (VTS) and virtual frame buffer (VFB). Each device works as input and output devices respectively. The emulator can display the camcorder application on the PC using the X window through the VFB device and emulate its actions through VKB and VTS.

The other parts of camcorder framework are also emulated by substituting target platform dependent part. We install USB-device PCI card to the PC for emulation of USB connectivity. In the case of GStreamer emulation, the software codecs are used instead of the hardware codec.

4.2 Performance enhancement methods

4.2.1 Thread based element in GStreamer

When input data is not sufficient to be processed, the input data needs to be buffered more. Especially in the case of DVDs, because the input stream data is divided into packs, the M2VD element has to wait the input data until the input stream data is sufficient to be decoded and reconstructed into a frame. For this buffering operation, the M2VD element is threaded. The structure of the M2VD element based on the thread is similar to that of the queue element of GStreamer. In the thread based element, the size of queue is the main control point of the element. When the chain function of

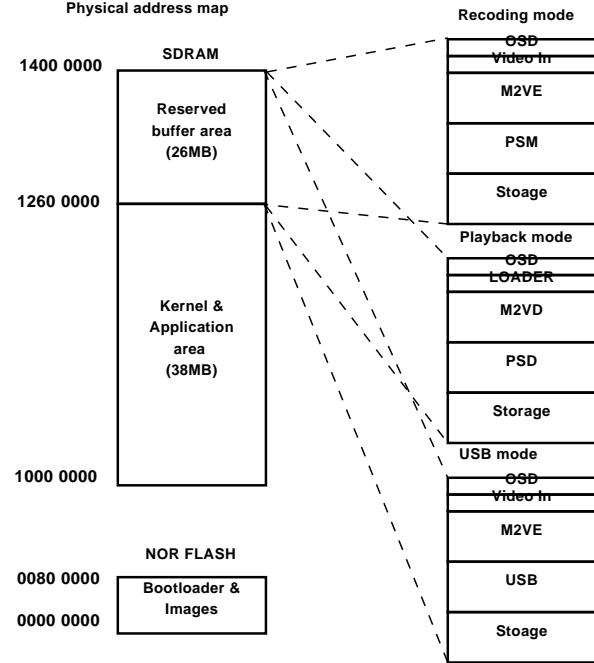


Figure 14: Reserved buffer area

the thread based element is called, the element just increases the size of queue by the input data size and returns immediately. The main operation on the input data starts immediately when the size of queue is sufficient for processing. Because the memory mapped addresses are shared between the elements, this buffer size based control is possible. The immediate return of the thread based element enables the previous element to do another operation. We apply the structure of thread based elements to both M2VD and AC3D elements. These elements are linked with the source pads of PSD element. Because of the immediate response/return of the M2VD and AC3D to the PSD element we can get better decoding performance

4.2.2 Reserved buffer area

The memory allocation routine spends CPU time to use memory efficiently. The dynamic memory allocation, e.g., memory allocation and free in Linux kernel, are very complicated and computationally intensive. In order to eliminate these problems, the reserved buffer area is used for some device drivers.

Figure 14 shows the reserved buffer area in our camcorder framework. It's divided into small parts for the device driver according to execution modes such as

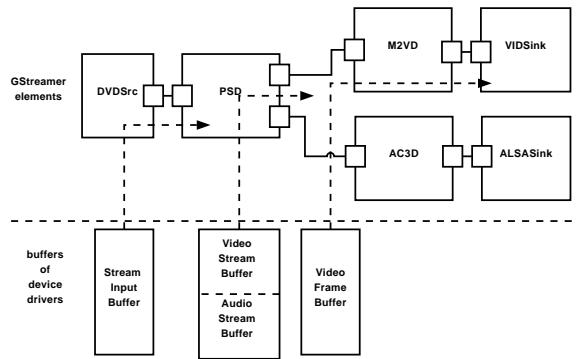


Figure 15: Memory interfaces of elements

recording, playback and USB mode. Each device driver buffer can be accessed by not only the device driver itself, but also user applications or middleware.

4.2.3 Using memory mapped IO

Normally, read and write functions are used for exchanging data between user application and kernel device drivers, but this needs to be reconsidered from a performance point of view. Firstly, they use system calls has a long call path. Secondly, the read and write data are copied by two functions, `copy_from_user()` and `copy_to_user()`, in the device driver. In order to avoid copy operations, the `mmap` is used instead of the read and write operations. The `mmap` permits us to directly access the buffer in the device driver. The camcorder framework benefits from this `mmap` operation a lot because it transfers a huge data frequently.

Figure 15 shows the memory interface for DVD playback. In the previous section, buffers are allocated in the reserved buffer area and accessed by their own device drivers. However, GStreamer elements can not access those buffers directly. Therefore, GStreamer elements get the access permission and virtual address by calling `mmap` functions of device drivers. Each element controls or manages its buffer through the virtual address.

4.2.4 Fast boot

In a PC environment, the Linux system spends more than one minute on the booting process. The booting process includes following steps; loading image, initializing the kernel and device drivers, and starting some

daemons. However, such a long boot time is not suitable for commercial products. Fast boot techniques for Linux have been studied for a long time.[14] We investigated and adapted them to our camcorder framework.

- **Fast memory copy** - Generally, the memory copy routine is implemented by ARM instructions such as LDR and STR. The transfer unit is 4 bytes long in size with those instructions. However, LDM and STM can transfer data in a larger unit longer than 4 bytes. By using these instructions, the fast memory copy can be achieved.
- **Using the uncompressed kernel image** - The zImage is the compressed kernel image. It spends quite a long time to uncompress the image. If the uncompressed image is used, the uncompressing time can be removed.
- **Reducing kernel option** - There are many options in the Linux kernel source. Only minimum options required for booting the system up are turned on to save time.
- **Preset LPJ** - In the kernel initialization, there is a function so called LPJ to calibrate the CPU speed. It is used for busy waiting like `udelay()` and `mdelay()`. The Preset LPJ means that the calibrating code is disabled and its output variable for the delay is set with the pre-estimated value.
- **Invalidating printk** - There are a lot of messages generated in the booting phase, and consequently they affect the booting time. To solve this we can either use the 'quiet' parameter in the kernel, or turn off the `CONFIG_PRINTK` option. The former increases the verbose level so that messages are not printed out. And the latter invalidates the `printk` function completely. In other words, the `printk` function is converted to a null function. We invalidate the `printk` function using the latter.
- **Using prelink** - An application is generally built with shared libraries. Using shared libraries saves the development time and the image size as well. However, it spends a lot of time binding the library symbols at run-time. The prelink technique removes the bind operation, so that the start-up time of the application can be reduced.

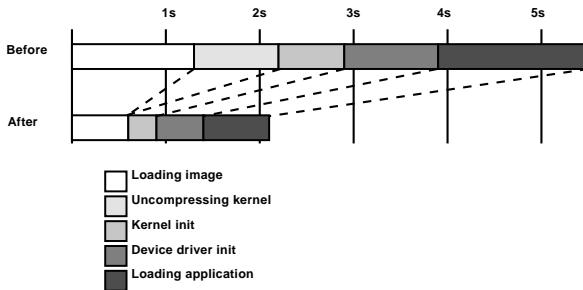


Figure 16: Result of fast boot

To analyze the fast booting process, the booting completion point needs to be defined first. Usually, it correspond to a state of record standby. In this paper, however, the completion point is defined as just before the application start-up because the application development is not completed for integration in the software framework at this time. Figure 16 shows measured data after adopting the fast boot techniques mentioned above. The booting time was about 5.5 seconds before adoption of fast boot techniques. After adoption, it reduced to about 2.1 seconds. Assuming a real product development, it will presumably take an extra time due to added components in the application.

4.2.5 Memory foot print

Because GStreamer depends on many external libraries, the total library size of the GStreamer framework is not suitable for the embedded system. In order to use the GStreamer in an embedded system, the size optimization is a significant part of development. In order to reduce GStreamer size, the external libraries actually used in the operation of GStreamer should be identified, assorbed, and built with the appropriate options accordingly.

The external libraries that have dependencies with the GStreamer core are libxml, libz, libglib, libcheck and libpopt. Of the above, all external libraries except the libglib can be removed. Because the libglib is extensively used in the entire GStreamer core, it needs to be kept.

The result of optimization is described in the following Table 1. As a result of optimization, the total size of GStreamer was reduced from 2.9MB to 1.4MB approximately.

Library name	Before	After
libcheck.so	27,216	0
libpopt.so	26,928	0
libxml2.so	1,110,004	0
libz.so	74,140	0
libglib-2.0.so	622,420	622,420
libgmodule-2.0.so	10,308	10,308
libgobject-2.0.so	243,020	243,020
libgthread-2.0.so	14,684	14,684
libgstreamer-0.10.so	644,160	445,896
libgstbase-0.10.so	147,516	79,280
libgstinterfaces-0.10.so	32,412	32,392
total	2,952,808	1,448,000

Table 1: Foot print of GStreamer optimization

5 Conclusions

In order to support many multimedia devices such as cameras and camcorders using a single hardware platform equipped with a dedicated multimedia SoC, a flexible and extensible software framework is highly preferred from the development cost perspective.

GStreamer is one of the open source based multimedia frameworks. Especially, it is adopted widely in multimedia product because of a variety of its plugin functions. In this paper, we presented a multimedia framework which supports both DVD and memory camcorders. It was designed and implemented by making use of open source middleware such as GStreamer in Linux OS. Based on the GStreamer engine, we developed a multimedia framework with many plug-ins implemented by hardware as well as software codec. The emulator was also introduced as a development environment in the PC platform.

References

- [1] *GStreamer Application Development Manual*, <http://gstreamer.freedesktop.org/data/doc/gstreamer/head/manual>
- [2] *GStreamer Plugin Writer's Guide*, <http://gstreamer.freedesktop.org/data/doc/gstreamer/head/pwg/>
- [3] *Fast Light Tool Kit Open Source Project*, <http://www.fltk.org/>
- [4] *The Nano-X Windows System Open Source Project*, <http://www.microwindows.org/>

- [5] *DVD Specification for Read-Only Disc Part3: Version 1.1*
- [6] *Universal Disc Format Specification Revision 2.50*, <http://www.osta.org/specs/>
- [7] *Linux-USB Gadget API Framework*,
<http://www.linux-usb.org/gadget/>
- [8] *USB Mass Storage Class Bulk-Only Transport*,
http://www.usb.org/developers/devclass_docs/usbmassbulk_10.pdf
- [9] *Information Technology - SCSI Block Commands 2-3*
- [10] *PIMA 15740:2000 - Picture Transfer Protocol for Digital Still Photography Devices*,
<http://www.i3a.org/>
- [11] *White Paper of CIPA DC-001-2003 Digital Photo Solutions for Imaging Devices*
- [12] *Universal Serial Bus Device Class Definition for Audio Devices*
- [13] *Scratchbox*,
<http://www.scratchbox.org/documentation/general/tutorials/explained.html>
- [14] *Boot Time Resources*,
<http://tree.ceLinuxforum.org/pubwiki/moin.cgi/BootupTimeResources>

On submitting kernel patches

Andi Kleen

Intel Open Source Technology Center

ak@linux.intel.com

Abstract

A lot of groups and individual developers work on improving the Linux kernel. Many innovative new features are developed all the time. The best and smoothest way to distribute and maintain new kernel features is to incorporate them into the standard mainline source tree. This involves a review process and some standard conventions. Unfortunately actually getting innovative new features through review can be a rough ride and sometimes they don't make it in at all. This paper examines some common problems for submitting larger changes and some strategies to avoid problems.

1 Introduction

Many people and groups want to contribute to the Linux kernel.

Sometimes it can be difficult to get larger changes into the mainline¹ sources, especially for new contributors. This paper examines some of the challenges in submitting larger patches, and outlines some solutions for them.

This paper assumes that the reader already knows the basics of patch submissions. These are covered in detail in various documents in the Linux kernel source tree (see [4], [3], [5]). Instead of repeating the material in there this paper covers some strategic higher level points in patch submission.

Some of the procedures suggested here are also only applicable to complex or controversial changes. Linux kernel development isn't (yet) a bureaucracy with fixed complicated procedures that cannot be diverged from and there is quite some flexibility in the process.

2 Why submit kernel patches to mainline?

There are many good reasons not to keep changes private but to submit them to Linus Torvalds' mainline source tree.

- A common approach for companies new to Linux is to take a snapshot of one kernel version (and its associated userland) and try to standardize on that version forever. But freezing on old versions for a long time is not a good idea. New features and bug fixes are constantly being added to mainline and some of them will be eventually needed. Freezing on a old version cuts off from a lot of free development.

While some changes from mainline can be relatively easily backported to older source trees, many others (and that will likely be the bug fix or feature you want) can be very difficult to backport. The Linux kernel infrastructure is constantly evolving and new changes often rely on newer infrastructure which is not available in old kernels. And even relatively simple backports tend to eventually become a maintenance problem when they add up in numbers because such a tree will diverge quite a lot from a standard kernel and invalidate previous testing. Then there are also security fixes that will be missed in trees that are frozen too long. It is possible to keep up with the security changes in mainline, but it's quite expensive requiring constant effort. And missed security fixes can be very costly to fix later.

At some point usually, it is required to resync the code base with mainline when updating to a new version. Forward porting private changes during such a version update tend to be hard, especially if they are complicated or affect many parts of the kernel. To avoid this problem submit changes to mainline relatively quickly after developing them,

¹"Mainline" refers to the kernel.org tree as maintained by Linus Torvalds.

then they will be part of the standard code base and still be there after version updates. This will also give additional code review, additional improvements and bug fixing and a lot of testing for free.

- For changes and redesigns done in mainline, usually only the requirements of in-tree code are considered. So, even if an enhancement works first externally with just standard exported kernel symbols, these might change or be taken away at any time. The only sure way to avoid that or at least guarantee an equivalent replacement interface to prevent breaking your code is to merge into mainline.

When the code is in mainline, it will be updated to any interface changes directly or in consultation with the maintainer. And in mainline, the user base will also test on the code and provide free quality-assurance.

- Writing a driver for some device and getting it into mainline means that all the distributions will automatically pick it up and support that device. That makes it much easier for users to use in the end because installing external drivers tends to be complicated and clumsy.

For another perspective on the why to submit changes to mainline, see also Andrew Morton’s presentation [1].

3 Basics of maintenance

All code in the kernel has some maintenance overhead. Once code is submitted the kernel maintainers commit themselves to keeping it functional for a long time.² It is usually expected that the person who submits new code does most of the maintenance for that code at least initially. Some of the procedures described here are actually to demonstrate that the patch submitter is trustworthy enough and they not just plan to “throw code over the fence.”

All code has bugs, so initially when code is submitted, it is assumed contain new defects. Exposing code to mainline also tends to generate a lot of new testing in new unexpected circumstances, which will expose new

²There is a procedure to deprecate functionality too, but it is rarely used and only for very strong reasons.

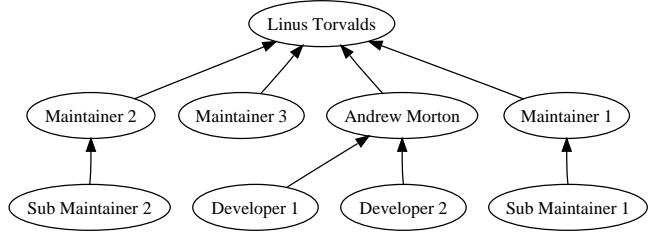


Figure 1: Kernel maintainer hierarchy (example) and patch flow. Andrew Morton is the fallback maintainer taking care of areas with no own maintainer or of patches crossing many subsystems

problems. One important part of patch submission is to make sure these bugs will be handled adequately.

The mainline kernel changes at a very high rate (see [7] for detailed numbers), and it is very important for the overall quality of the Linux kernel to keep the bug rate under control. See [2] for details on the mechanics of QA and bug reporting in mainline Linux. Because new code often has more bugs than old code, the maintainers tend to use various procedures to make sure the bugs in the new code are minimized early on and handled quickly. One common way to do this is extensive code review.

All new kernel code has to go through standard code review to make sure it follows the kernel coding style [6] and avoids deprecated kernel interfaces and some common mistakes. Code review will also look for other functional problems (although that is not the main focus) and include a design review. Coding style is already covered extensively in [6], and I won’t cover it in detail in this paper. In the end, coding style makes only a small part of a successful piece of kernel code anyways and is commonly overrated. Still it is a good idea to follow the coding style in the initial posting so that discussions about white space and variable naming do not distract from the actual functionality of the change.

This paper takes a higher level look on the mechanics of merging larger functionality, and assumes the basic Linux code requirements (like coding style, using standard kernel interfaces etc.) are already covered.

4 Types of submissions

4.1 Easy cases

- The easiest case is a clear bug fix. The need for a bug fix is obvious, and the only argument might

be how the bug is actually fixed. But getting clear bug fixes merged is usually no problem. Sometimes, the maintainers might want to fix a particular bug differently than with the original patch, but then the bug will usually be fixed in a different way by someone else. The end result is that the bug is fixed.

Occasionally, there can be differences on what is considered to be a bug and what is not. In this case, the submitter has to argue for its case in the review mail thread.

- Then there are cleanups. Cleanups can range from very simple as in fixing coding style or making code sparse³ clean to removing unused code to refactoring an outdated subsystem. Getting such cleanups in is not normally a problem, but they have to be timed right to not conflict with other higher priority changes during the merge windows.⁴ The maintainers can normally coordinate that.
- Optimizations are usually welcome, and not too hard to merge, but there are some caveats. When the optimization applies only to a very special case, it is important that it does not impact other more common cases. And there should be benchmark numbers (or other data) showing that the optimization is useful. The impact of the optimization on the code base should also be limited (unless it is a major advantage for an important use-case). Generally optimizations should not impact maintainability too much. Especially when the optimization is not a dramatic improvement or does apply only to some special cases, it is important that it is clean code and its impact is limited. Cleaning up some code while doing the optimization will make the optimization much more attractive.

4.2 Hardware drivers

The need can be easily established for submitting hardware drivers for standard devices like network cards or SCSI adapters. The hardware exists and Linux should

³Sparse is a static code analysis tool written for the Linux kernel. See [8]. But like most static-checking tools, it needs a large amount of work initially to eliminate false positives.

⁴Merge window is the two week period after each major releases where maintainers send major features to Linus Torvalds tree. Most features and code changes go in at that time.

support it and the driver (if correctly written) will in most cases not impact any other code.

The increasing the bug rate argument in Section 3 is fortunately not a serious problem in this special case. If the hardware is not there, the driver will not be used, and can then also not cause bugs.⁵ Luckily, this means that because most of the kernel source base are drivers, the effective bug rate is not raising as quickly as you would expect from the raw source size growth. Still, this is a problem maintainers are concerned about, especially for core kernel code that is used on nearly⁶ all hardware and on drivers for hardware used very widely in the user-base.

There are well-established procedures to get new drivers in, and doing so is normally not a problem. In some cases, depending on the code quality of the driver, it can take a long time with many iterations of reviews and fixes.

One more difficult issue are special optimizations for specific drivers. Most drivers won't need any, for example, a standard NIC or block driver is usually not a problem to add because it only plugs into the well established network driver interface. There will be no changes on other code.

On the other hand, if a hardware device does for example RDMA,⁷ and needs special support in the core networking stack to support that, merging that will be much more difficult because these code changes could impact other setups too. One recommended strategy in this case is to first get basic support in while minimizing changes to core infrastructure

Sometimes, there is first a rough consensus in the kernel community that particular optimizations should not be supported for various reasons. One example of this is stateful TCP/IP stack off-loading⁸ or native support for hashed page tables in the core VM. Of course such consensus can be eventually re-evaluated when the facts

⁵This ignores the case of non-essential drivers for common hardware. Adding them could risk increasing the bug rate.

⁶Nearly because there are some special cases like devices without a block driver or MMU-less devices that disable significant parts of a standard kernel.

⁷Remote DMA—DMA directly controlled by a remote machine over the network.

⁸Partial stateless offloads like Large-Receive-Offload (LRO) or TCP Segmentation Offload (TSO) on the other hand are already supported.

change (or it is demonstrated that the optimization is really needed), but it is typically difficult to do so.

Once the basic support is in, and you need some specific changes to optimize for your special case, one reasonable way to get this done is to do clean-up or redesign that improves the standard subsystem code (not considering your changes), and then just happens to make your particular optimization easier. The trade-off is here that offsetting the maintainability impact on the subsystem by cleaning it up and improving it first, later re-adding some complexity for special optimizations can be justified.

Assuming your proposed change does not fall into one of these difficult areas, it should be relatively easy to get it included in mainline once the driver passes the basic code review.

If it is in a difficult area, it is usually better to at least try to merge it, but will require much more work. In a few extreme cases the actual merge will be very hard to impossible too, for instance when you're planning to submit a patch supporting a full TCP offload engine. On the other hand, if the arguments are good maintainers sometimes reconsider.

4.3 New core functionality

An especially touchy topic is adding new hooks into the core kernel, like new notifier chains or new functions calling into specific subsystems. Very generic notifiers and hooks tend to have a large maintenance impact because they have the potential to alter the code flow in unexpected ways, lead to lock order problems, bugs, and unexpected behavior, and generally making code harder to maintain and debug later. That is why maintainers tend to be not very happy about adding them to their subsystems. If you really need the hooks anyways trading cleanups for hooks as described in Section 4.2 is a reasonable (but not guaranteed to be successful) strategy

Usually, there will be a discussion on the need for the hooks on the mailing list, with commenters suggesting design alternative if the case is not very clear. This may result in you having to redesign some parts if you cannot convince the maintainer of the benefits of your particular design. A redesign might include moving some parts to userland or doing it altogether differently.

To avoid wasting too much work, it is a good idea to discuss the basic design publicly before spending too

much time on real production code. Of course doing prototypes first to measure basic feasibility is still a good idea. Just do not expect these prototypes to be necessarily merged exactly as they are. As usual prototype code tends to require some work to make it production ready.

5 Splitting submissions into pieces

It is also fairly important to submit larger changes in smaller pieces so that reviewers and maintainers can process the changes step-by-step. Normally this means splitting a larger patch series into smaller logical chunks than can be reviewed together. There are exceptions to this. For example a single driver source file that is completely new is normally reviewed together, even when it is large. But this protocol is fairly important for any changes to existing code.

These changes must be bisectable:⁹ that is applying or unapplying any patch in the sequence must still produce a buildable and working kernel.

Usually, you will need to revise patches multiple times based on the feedback before they can be accepted. Avoid patch splitting methods that cause you a lot of work each time you post. Ideally you keep the split patches in change set oriented version control system like git or mercurial or in a patch management system like quilt [9]. Personally, I prefer quilt for patch management which has the smoothest learning curve. [10] has a introduction on using quilt.

It is also recommended to time submissions of large patch kits. Posting more than 20 patches at a time will overwhelm the review capacity of the mailing lists. Group them into logical groups and post these one at a time with at least a day grace time in between.

Patches should be logical steps with their own descriptions, but they don't need to be too small. Only create very small (less than 10 lines change) patches if they are really a logically an self-contained change. On the other hand, new files typically do not need to be split up, except when parts of them logically belong to other changes in other areas.

When you post revised patches, add a version number to the subject, and also maintain a brief change log at the end of the patch description.

⁹Typically used with the “git bisect” command for semi-automated debugging.

6 A good description

Submitting a Linux kernel patch is like publishing a paper. It will be scrutinized by a sceptical public. That is why the description is very important. You should spend some time writing a proper introduction explaining what your change is all about, what your motivations were and what the important design decisions and trade-offs are.

Ideally you will also already address expected counter arguments in the description. It is a good idea to browse the mailing list archives beforehand and to study a few successful submissions there.

Hard numbers quantifying improvements are always appreciated, too, in the description. If there is something to measure, measure it and include the numbers. If your patch is an optimization, measure the improvement and include the numbers.

The quality of a description can make or break whether your kernel patch is accepted. If you cannot convince the kernel reviewers that your work is interesting and worthwhile, then there will be no code review and without visible code review the code will likely not be merged.

The linux-kernel mailing list (and other kernel project mailing lists) are an attention economy, and it is important to be competitive here.

7 Establishing trust

If your patch is larger than just a change to an existing system (like a new driver or similar), you will be expected to be the maintainer of that code for at least some time. A maintainer is someone who takes care of the code of some subsystem, incorporates patches, makes sure they are up to standard, does some release engineering to stabilize the code for releases, and sends changes bundled to the next level maintainer.

This relationship involves some level of trust. The next level maintainer trusts you to do this job properly, and keep linux code quality high. If the person is not known yet from other projects, the only way to get such trust is to do something publically visible. That could be done by submitting some self-written code that is high quality or by fixing bugs.

The maintainers in general favor people who do not only care about their little piece of code but who take a little larger view of the code base, and are known to improve, and fix other areas of the kernel too. This does not necessarily mean a lot of work, and could be just an occasional bug fix.

8 Setting up a community

For larger new kernel features like a file system or a network protocol, it is a good idea to have a small user base first. This user base is needed for testing, and to make sure there is actually interest in the new feature.

It is recommended to at least initially (while your community is still young) keep most of the technical discussion on the linux-kernel mailing list. This way the maintainers can see there is an active group working on the particular feature, fixing bugs, and caring about user's needs which then builds up trust for an eventual merge.

9 When to post

Publishing a patch is a very important event in its lifetime. There is a delicate balance between posting too early and posting too late.

First, once the code basically works, it is a good idea to post it as a Request-for-Comments (RFC), clearly marking it as such. For more complicated changes or you're not sure yet what will be acceptable to the maintainers you can also post a rough prototype or even just a design overview what you're planning as RFC. This would not be intended to be merged, but just to invite initial comments, and to make other developers aware you have something in the pipeline. There should already be valuable feedback in that stage, and when the feedback includes requests for larger changes, you do not need to throw as much work away when you redo code, as you would had posted a finished patch. For simple changes the RFC stage can be skipped, but for anything more complex it is typically a good idea. For very complex or very controversial changes you will likely go through multiple RFC stages.

Also, conducting more of the development process visible on the mailing list is good for building trust, and for establishing a community as discussed earlier.

On the other hand, actually merging (submitting to a upstream maintainer) too early when you still know the code is unstable is a bad idea. The problem is that even when some subsystem is marked experimental, people will start using it, and if it does not work or only works very badly or worse corrupts data, it will get a bad name. Getting a bad name early on can be a huge problem later for a project and it will be hard to ever get rid of that taint again.¹⁰

So there should be some basic stability before actually submitting it to merge. On the other hand, it definitely does not need to be finished. Feature completeness or final performance is not a requirement.

Patches take some time to travel through review and then the various maintainer trees. If you want a change in a particular release it is really too late when you only post it during the two week merge window. Rather when the merge window opens the patch should be already through review and traveled up the maintainer hierarchy. That will take some time. And during the merge window reviewer capacity tends to be in short order and there might be none left for you.

10 Dealing with reviews

Code review is an integral part of the Linux code submission process. It proceeds on public mailing lists with everyone allowed to comment on your patches. Most of the comments will be useful and help to improve the code. This is usually fairly obvious, in this case just make the changes they request.

Sometimes even when the comments are useful, they might cause you excessive work: for example when they ask for a redesign. Sometimes there can be very good reasons for the redesign, sometimes not. It is your judgment. Or sometimes the reviewer just missed something and the suggestion wouldn't actually improve the code or not handle some case correctly. If the reason is convincing, you should just make the changes if feasible.

In some cases, the reviewer might not realize how much work it would be to implement a particular change. If you are not convinced of the reasons for the redesign or it is just not feasible because it would take too long, explain that clearly with pure technical arguments on the

¹⁰There are several high quality subsystems in the kernel like JFS or ACPI who suffer from this problem.

mailing list (but do not get yourself dragged into a flame war).

Then if even after discussion the reviewer still insist on you making such a change you don't like, you have to judge the request: If the the maintainer of the subsystem or a upstream maintainer requests the change, and you cannot convince them to change their mind, you'll have to implement the change or drop the submission. If someone other than the maintainer requests the change, it is useful to ask the maintainer for their opinion in a private mail before embarking on large projects addressing review comments.

A reasonable rule of thumb (but there are exceptions of course!) for how serious to take a reviewer is to check how much code the person contributed. As a first approximation, if someone never contributed any patches themselves their comments are less important,¹¹ and you could ignore them partly or completely after describing why on the mailing list. You can look up the contributions of a particular person in the git version history. But you really should do that only in extreme cases when there is no other choice. Linux kernel development unfortunately suffers from a shortage of reviewers so you should consider well before you ignore one and only do that for very strong reasons.

And sometimes you will notice that the comments from a particular reviewer are just not constructive. This comes from the fact that review is open to all on the internet, and there are occasionally bad reviewers too. These cases tend to be usually clear, and it is reasonable not to address such unproductive comments.

11 Merging plans

For complicated patch kits, especially when they depend on other changes, and after the basic reviews are done, it is also a good idea to negotiate a merging plan with the various stake holders. This is especially important if changes touch multiple subsystems, and might need in theory to go through multiple maintainers (which can be quite tedious to coordinate). Merging plan would be a simple email telling them in what order and when the patches will go in and get their approval. If a change touches a lot of subsystems you don't necessarily need the approval from all maintainers. Such tree wide sweeps are normally coordinated by the upstream maintainers.

¹¹Some people call Linux a “codeocracy” because of that.

12 Interfaces

Reviewers and maintainers focus on the interface designs for user space programs. This is because merging an externally visible interface commits the kernel to the interface because other code will depend on it. Changing a published interface later is much harder and often special backwards compatibility code is required. This usually leads to very close scrutiny of interfaces by reviewers and maintainers before merge. To avoid delays, it is best to get interface designs right on the first try. On the other hand this first try should be as simple as possible and not include ever feature you plan on the first iteration.

For many submissions, like device drivers for standard devices or a file system, user space interfaces will not be relevant because they don't have user interfaces of their own.

- There are various interface styles (e.g. file systems, files in sysfs, system calls, ioctls, netlink, character devices) which have different trade offs and are the right choice for different areas. It is important to chose the interface style that fits the application best.
- There should be some design documentation on the interface included with the submission.
- It is recommended to make any new interfaces as simple as possible before submission. This will make reviewing easier and they can be still later extended.
- 64bit architectures typically use a compat layer to translate system calls from 32bit application to the internal 64bit ABI of the kernel. The needs of the compat layer needs to be considered for new interfaces.
- For new system calls or system call extensions there should be a manpage and some submittable test code.
- Remove any private debug interfaces before submission if it's not clear they will be useful long term for code maintenance. Alternatively maintain them as a separate add-on patch.

On the other hand internal kernel interfaces are considered subject to change and are much less critical.

13 Resolving problems

Sometimes a submission gets stuck during the submission process. In this case, it is a good idea to just send private mail to the maintainer who is responsible and ask advice on how to proceed with the merge. Alternatively, it is also possible to ask one of the upstream maintainers (in case the problem is with the maintainer)

There is also the linux-mentors [11] program and the kernelnewbies project [12] who can help with process issues.

14 Case studies

14.1 dprobes

Dprobes [13] was a dynamic tracing project from IBM originally ported from OS/2 and released as a kernel patch in 2000. It allowed to attach probes written in a simple stack based byte code language to arbitrary code in the kernel or user space, to collect information on kernel behavior non intrusively. Dynamic tracing is a very popular topic now,¹² but back then, dprobes was clearly ahead of its time. There was not much interest in it.

The dprobes team posted various versions with increasing functionality, but could not really build a user community. dprobes was a comprehensive solution, covering both user space and kernel tracing. The user space tracing required some changes to the Virtual Memory subsystem that were not well received by the VM developers. Putting a byte-code interpreter into the kernel was also unpopular, both from its code and because the kernel developers as potential user base preferred to write such code in C. That lead to the original dprobes code never being merged to mainline and never getting a real user base.

After a few years of unsuccessful merging attempts, maintaining the code out of tree and existing in relative obscurity the project reinvented itself. One part of dprobes was the ability to set non intrusive breakpoints to any kernel code. They extracted that facility and allowed it to attach handlers in kernel modules written in C to arbitrary kernel functions. This new facility was called kprobes [15]. kprobes became quickly relatively

¹²Especially due to the marketing efforts of a particular operating system vendor for a much later similar project

popular with the kernel community and was merged after a short time. It attracted also significant contributors from outside the original dprobes project. This was both because it was a much simpler patch, touching less code, none of its changes were controversial, and that other competing operating systems had made dynamic tracing popular by then, so there was generally much more interest.

Kprobes then was the kernel infrastructure used by the [14] project started shortly after the merge. systemtap is a scripting language that generates C source for a kernel module that then uses kprobes to trace kernel functions. systemtap and kprobes are popular Linux features now, but it took a long time to get there. Also, significant features of the original dprobes (attaching probes to user space code) are still missing. Support for user probes is currently developed as part of the utrace project, but unfortunately utrace development proceeds slowly and has unlikely prospects for merge because utrace is a gigantic “redesign everything” patch developed on a private mailing list.

The lessons to learn from the dprobes story are:

- When building something new and radical, it is needed to “sell” it at least a little, to get a user base and interest from reviewers. dprobes, while being technically a very interesting project, did not compete well initially in the attention economy.
- Don’t try to put in all features on the first step. Submit features step-by-step.
- When a particular part of a submission is very unpopular, strip it out to not let it delay submission of the other parts.
- Don’t wait too long to redesign if the original design is too unpopular.

14.2 perfmon2

perfmon1 was a relatively simple architecture specific interface for the performance counters of the IA64 port. That code was integrated into the mainline Linux source. Later, it was redesigned as perfmon2 [16] with many more features, support for more performance monitoring hardware, support for flexible output formats using plug-ins, and its cross architecture support.¹³

¹³perfmon2 is generally considered to be showing some signs of the “Second System effect.”

perfmon2 was developed outside the normal Linux kernel, together with its user land components at [17]. While it was able to attract some code contributions and made some attempts to be merged into the mainline kernel, it has not succeeded yet. This is mainly because it has acquired many features on its long out-of-tree development path that are difficult to untangle now.¹⁴

This lead to the perfmon2 patch submissions being very large patches with many interdependencies which are very difficult to understand and review and debug later. It also didn’t help that its very flexible plug-in-based output module architecture was a design pattern not previously used in Linux. Kernel programmer’s are relatively conservative regarding design patterns. And the subsystem came with a very complicated system call-based interface that was difficult to evaluate. Also, many perfmon2 features were relatively old and it was sometimes impossible to reconstruct why they were even added. In turn, the patches were unable to either attract enough reviewers or to satisfy the comments of the few reviews it got. That in turn, didn’t lead to maintainer confidence in the code and there was no mainline merge so far, except for some trivial separable changes.

- Be conservative with novel design patterns (like the perfmon2 output plug-ins) Kernel programmers are conservative regarding coding style and not very open to novel programming styles.
- Don’t combine too many novelties into a single patch. If you have a lot of new ideas do them step by step.
- Don’t make the code too flexible internally. Too much flexibility makes it harder to evaluate.
- Don’t do significant follow-on development outside the mainline kernel tree. Concentrate on developing the basic subsystem without too many bells and whistles and merge that quickly into the mainline. Then any additional features requested by users should be submitted to the mainline incrementally.
- If any optional features are already implemented up front before merging, develop them as incremental patches to the core code base, not in an integrated source tree.

¹⁴Often programmers don’t remember ever detail on why they did some code years before.

In the author's opinion, the only promising strategy for a perfmon2 merge now would be similar to the evolution from dprobes to kprobes in Section 14.1. Go back to the core competencies: define the core functionality of a performance counter interface, or develop a *perfmon3* which only implements the core functionality in a very clean way, and submit that. Then port forward features requested by the user step-by-step from perfmon2, each time you reevaluate the design of a particular feature and its user interface. This process would likely lead to a much cleaner perfmon2 code base. There is some work underway now by the perfmon2 author to do this.

15 Conclusion

Submitting code to the mainline Linux kernel is rewarding, but also takes some care to be successful. With the rough guidelines in this paper, I hope some commons problems while submitting Linux kernel changes can be avoided.

References

- [1] Andrew Morton
kernel.org development and the embedded world
<http://userweb.kernel.org/~akpm/rants/elc-08.odp>
- [2] Andrew Morton
Production process, bug tracking and quality assurance of the Linux kernel
<http://userweb.kernel.org/~akpm/rants/hannover-kernel-qa.odp>
- [3] *Documentation/SubmittingDrivers* in the Linux kernel source from
<http://www.kernel.org>
Living document. Always refer to the version from the latest kernel release.
- [4] *Documentation/SubmitChecklist* in your Linux kernel source from
<http://www.kernel.org>
Living document. Always refer to the version from the latest kernel release.
- [5] *Documentation/SubmittingPatches* in the Linux kernel source from
<http://www.kernel.org>
Living document. Always refer to the version from the latest kernel release.
- [6] *Documentation/CodingStyle* in the Linux kernel source from <http://www.kernel.org>
Living document. Always refer to the version from the latest kernel release.
- [7] Kroah-Hartmann *et al.*,
<https://www.linux-foundation.org/publications/linuxkerneldevelopment.php>
- [8] Sparse, the semantic parser,
<http://www.kernel.org/pub/software-devel/sparse/>
- [9] Quilt, the patch manager,
<http://savannah.nongnu.org/projects/quilt>
- [10] Grünbacher,
How to survive with many patches or Introduction to Quilt,
<http://www.suse.de/~agruen/quilt.pdf>
- [11] <http://www.linuxmentors.org>
- [12] <http://www.kernelnewbies.org>
- [13] <http://dprobes.sourceforge.net>
- [14] Prasad
Locating System Problems using Dynamic Instrumentation
Proceedings of the Linux Symposium, Ottawa, Canada, 2005.
- [15] Keniston
Documentation/kprobes.txt in the Linux kernel source from <http://www.kernel.org>
- [16] Eranian
Perfmon2: A flexible performance monitoring interface for Linux
Proceedings of the Linux Symposium, Ottawa, Canada, 2006.
- [17] <http://perfmon2.sourceforge.net>

This work represents the opinion of the author and not of Intel.

This paper is (c) 2008 by Intel. Redistribution rights are granted per submission guidelines; all other rights reserved.

* Other names and brands may be claimed as the property of others.

Ext4 block and inode allocator improvements

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

Mingming Cao
IBM Linux Technology Center
cmm@us.ibm.com

Jose R Santos
IBM Linux Technology Center
jrs@us.ibm.com

Andreas Dilger
Sun Microsystems, Inc
adilger@sun.com

Abstract

File systems have conflicting needs with respect to block allocation for small and large files. Small related files should be nearby on disk to maximize disk track cache and avoid seeking. To avoid fragmentation, files that grow should reserve space. The new multiple block and delayed allocators for Ext4 try to satisfy these requirements by deferring allocation until flush time, packing small files contiguously, and allocating large files on RAID device-aligned boundaries with reserved space for growth. In this paper, we discuss the new multiple block allocator for Ext4 and compare the same with the reservation-based allocation. We also discuss the performance advantage of placing the meta-data blocks close together on disk so that meta-data intensive workloads seek less.

1 Introduction

The Ext4 file system was forked from Ext3 file system about two years ago to address the capability and scalability bottleneck of the Ext3 file system. Ext3 file system size is hard limited to 16 TB on x86 architecture, as a consequence of 32-bit block numbers. This limit has already been reached in the enterprise world. With the disk capacity doubling every year and with increasing needs for larger file systems to store personal digital media, desktop users will very soon want to remove the limit for Ext3 file system. Thus, the first change made in the Ext4 file system is to lift the maximum file system size from 2^{32} blocks(16 TB with 4 KB blocksize) to 2^{48} blocks.

Extent mapping is also used in Ext4 to represent new files, rather than the double, triple indirect block mapping used in Ext2/3. Extents are being used in many

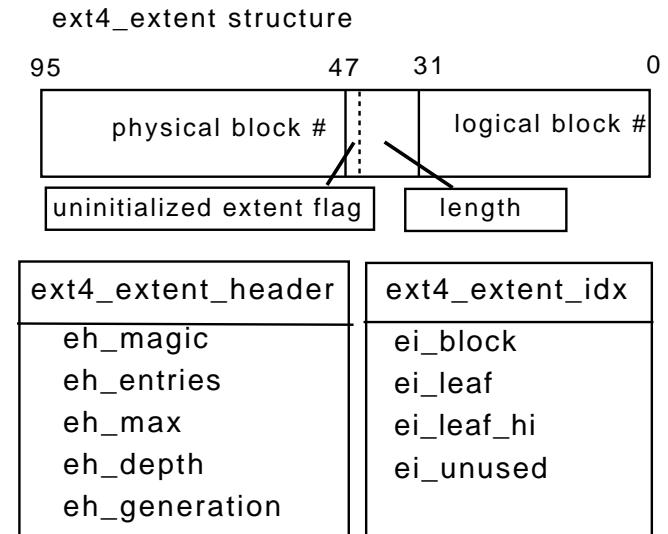


Figure 1: Ext4 extents, header and index structures

modern file systems[1], and it is well-known as a way to efficiently represent a large contiguous file by reducing the meta-data needed to address large files. Extents help improve the performance of sequential file read/writes since extents are a significantly smaller amount of meta-data to be written to describe contiguous blocks, thus reducing the file system overhead. It also greatly reduces the time to truncate a file as the meta-data updates are reduced. The extent format supported by Ext4 is shown in Fig 1 and Fig 2. A full description of Ext4 features can be found in [2].

Most files need only a few extents to describe their logical-to-physical block mapping, which can be accommodated within the inode or a single extent map block. However, some extreme cases, such as sparse files with random allocation patterns, or a very badly fragmented file system, are not efficiently represented

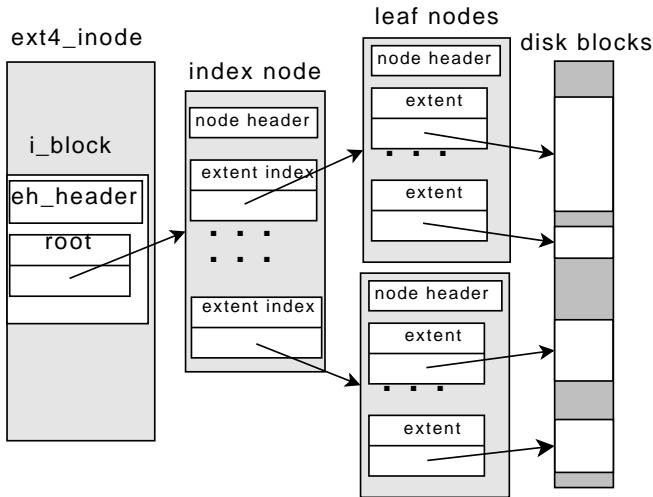


Figure 2: Ext4 extent tree layout

using extent maps. In Ext4, it is more important to have an advanced block allocator to reduce file fragmentation. A well-designed block allocator should pack small files close to each other for locality and also place large files contiguously on disk to improve I/O performance.

The block allocator in the Ext3 file system does limited work to reduce file fragmentation and maintains group locality for small files under the same directory. The Ext3 allocator tries to allocate multiple blocks on a best effort basis but does not do a smart search to find a better location to place a large file. It is also unaware of the relationship between different files, thus it is quite possible to place small related files far apart, causing extra seeks when loading a large number of related files. Ext4 block allocation tries to address these two problems with the new multiple block allocator while still making it possible to use the old block allocator. Multiple block allocation can be enabled or disabled by mount option `-o mballoc` and `-o nomballoc`, respectively. The current development version of Ext4 file system enables the multiple block allocator by default.

While a lot of work has gone into the block allocator, one must not forget that the inode allocator is what determines where blocks start getting allocated to begin with. While disk platter density has increased dramatically over the years, the old Ext3 inode allocator does little to ensure data and meta-data locality in order to take advantage of that density and avoid large seeks. Ext4 has begun exploring the possibilities of compacting data to increase locality and reduce seeks, which ultimately leads to better performance. Removal of re-

strictions in Ext4's block groups has made it possible to rethink meta-data placement and inode allocation in ways not possible in Ext3.

This paper is organized into the following sections. In Section 2.1 and 2.2, we give some background on Ext3 block allocation principles and current limitations. In Section 2.3, we give a detailed description of the new multiple block allocator and how it addresses the limitations facing the Ext3 file system. Section 2.4 compares the Ext3 and Ext4 block allocators with some performance data we collected.

Finally, we discuss the inode allocator changes made in Ext4. We start with Section 4.1, describing the Ext3 inode allocation policy, the impact that the inode allocation has on overall file system performance, and how changing the concept of a block group can lead to performance improvements. In Section 4.3, we explore a new inode allocator that uses this new concept in meta-data allocation to manipulate block allocation and data locality on disk. Later, we examine the performance improvements in the Ext4 file system due to the changing of block group concept and the new inode allocator. Finally, we will discuss some potential future work in Section 4.5.

2 Ext4 Multiple Blocks Allocator

In this section we give some background on the Ext2/3 block allocation before we discuss the new Ext4 block allocator. We refer to the old block allocator as the Ext3 block allocator and the new multiple block allocator as the Ext4 block allocator for simplicity.

2.1 Ext3 block allocator

Block allocation is the heart of a file system design. Overall, the goal of file system block allocation is to reduce disk seek time by reducing file system fragmentation and maintaining locality for related files. Also, it needs to scale well on large file systems and parallel allocation scenarios. Here is a short summary of the strategy used in the Ext3 block allocator:

To scale well, the Ext3 file system is partitioned into 128 MB block group chunks. Each block group maintains a single block bitmap to describe data block availability inside this block group. This way allocation on different block groups can be done in parallel.

When allocating a block for a file, the Ext3 block allocator always starts from the block group where the inode structure is stored to keep the meta-data and data blocks close to each other. When there are no free blocks available in the target block group it will search for a free block from the rest of the block groups. Ext3 always tries to keep the files under the same directory close to each other until the parent block group is filled.

To reduce large file fragmentation Ext3 uses a goal block to hint where it should allocate the next block from. If the application is doing a sequential I/O the target is to get the block following the last allocated block. When the target block is not available it will search further to find a free extent of at least 8 blocks and starts allocation from there. This way the resulting allocations will be contiguous.

In case of multiple files allocating blocks concurrently, the Ext3 block allocator uses block reservation to make sure that subsequent requests for blocks for a particular file get served before it is interleaved with other files. Reserving blocks which can be used to satisfy the subsequent requests enable the block allocator to place blocks corresponding to a file nearby. There is a per-file reservation window, indicating the range of disk blocks reserved for this file. However, the per-file reservation window is done purely in memory. Each block allocation will first check the file's own reservation window before starts to find unreserved free block on bitmap. A per-file system red-black tree is used to maintain all the reservation windows and to ensure that when allocating blocks using bitmap, we don't allocate blocks out of another file's reservation window.

2.2 Problems with Ext3 block allocator

Although block reservation makes it possible to allocate multiple blocks at a time in Ext3, this is very limited and based on best effort basis. Ext3 still uses the bitmap to search for the free blocks to reserve. The lack of free extent information across the whole file system results in poor allocation pattern for multiple blocks since the allocator searches for free blocks only inside the reservation window.

Another disadvantage of the Ext3 block allocator is that it doesn't differentiate allocation for small and large files. Large directories, such as /etc, contain large numbers of small configuration files that need to be

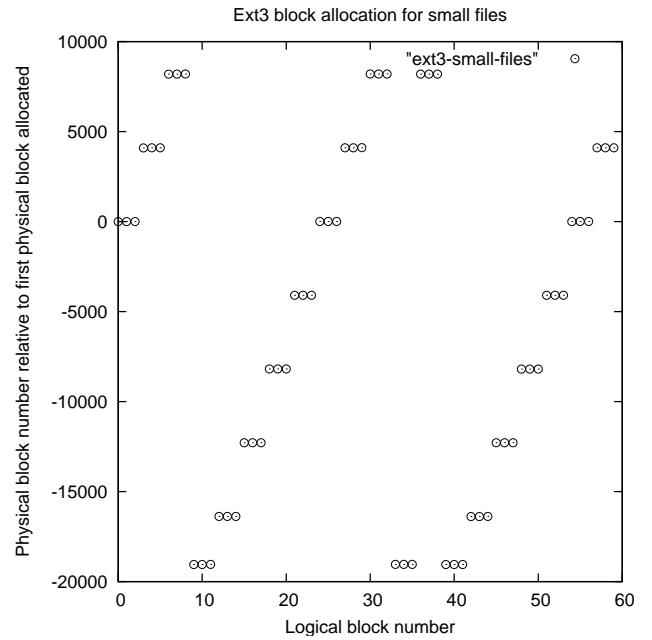


Figure 3: Ext3 block allocator for small files

read during boot. If the files are placed far apart on the disk the bootup process would be delayed by expensive seeks across the underlying device to load all the files. If the block allocator could place these related small files closer it would be a great benefit to the read performance.

We used two test cases to illustrate the performance characteristic of Ext3 block allocator for small and large files, as shown in Fig 3 and Fig 4. In the first test we used one thread to sequentially create 20 small files of 12 KB. In the second test, we create a single large file and multiple small files in parallel. The large file is created by a single thread, while the small files are created by another thread in parallel. The graph is plotted with the logical block number on the x-axis and the physical block number on the y-axis. To better illustrate the locality, the physical block number plotted is calculated by subtracting the first allocated block number from the actual allocated block number. With regard to small files, the 4th logical block number is the first logical block number of the second file. This helps to better illustrate how closely the small files are placed on disk.

Since Ext3 simply uses a goal block to determine where to place the new files, small files are kept apart by Ext3 allocator intentionally to avoid too much fragmentation in case the files are large files. This is caused by lack

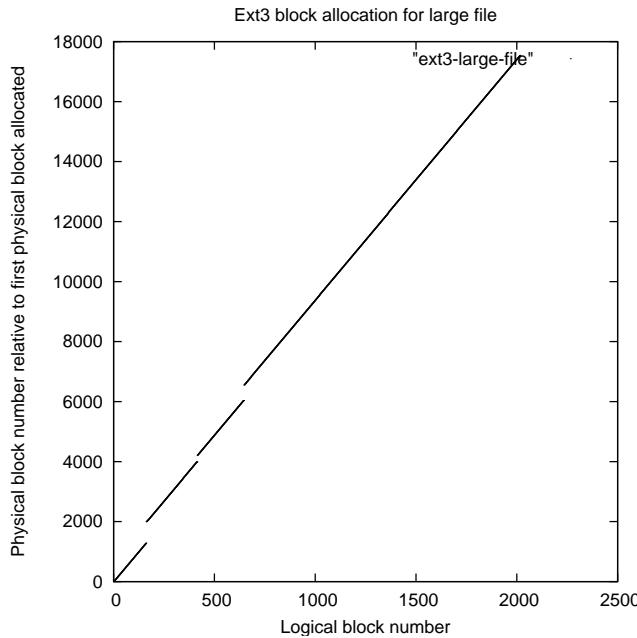


Figure 4: Ext3 block allocator for large file

of information that those small files are generated by the same process and therefore should be kept close to each other. As we see in Fig [3], the locality of those small files are bad, though the files themselves are not fragmented.

Fig [4] illustrates the fragmentation of a large file in Ext3. Because Ext3 lacks knowledge that the large file is unrelated to the small files, the allocations for the large file and the small files are fighting for free spaces close to each other, even though the block reservation helped reduce the fragmentation to some extent already. A better solution is to keep the large file allocation far apart from unrelated allocation at the very beginning to avoid interleaved fragmentation.

2.3 Ext4 Multiple block allocator

The Ext4 multiple block allocator tries to address the Ext3 block allocator limitation discussed above. The main goal is to provide better allocation for small and large files. This is achieved by using a different strategy for different allocation requests. For a relatively small allocation request, Ext4 tries to allocate from a per-CPU locality group, which is shared by all allocations under the same CPU, in order to try to keep these small files close to each other. A large allocation request

is allocated from per-file preallocation first. Like Ext3 reservation, Ext4 maintains an in-memory preallocation range for each file, and uses that to solve the fragmentation issues caused by concurrent allocation.

The Ext4 multiple block allocator maintains two preallocated spaces from which block requests are satisfied: A per-inode preallocation space and a per-CPU locality group preallocation space. The per-inode preallocation space is used for larger request and helps in making sure larger files are less interleaved if blocks are allocated at the same time. The per-CPU locality group preallocation space is used for smaller file allocation and helps in making sure small files are placed closer on disk. Which preallocation space to use depends on the total size derived out of current file size and allocation request size. The allocator provides a tunable `/proc/fs/ext4/<partition>/stream_req` that defaults to 16. If the total size is less than `stream_req` blocks, we use per-CPU locality group preallocation space.

While allocating blocks from the inode preallocation space, we also make sure we pick the blocks in such a way that random writes result in less fragmentation. This is achieved by storing the logical block number as a part of preallocation space and using the value in determining the physical block that needs to be allocated for a subsequent request.

If we can't allocate blocks from the preallocation space, we then look at the per-block-group buddy cache. The buddy cache consists of multiple free extent maps and a group bitmap. The extent map is built by scanning all the free blocks in a group on the first allocation. While scanning for free blocks in a block group we consider the blocks in the preallocation space as allocated and don't count them as free. This is needed to make sure that when allocating blocks from the extent map, we don't allocate blocks from a preallocation space. Doing so can result in file fragmentation. The free extent map obtained by scanning the block group is stored in a format, as shown in Fig 5, called a buddy bitmap. We also store the block group bitmap along with the extent map. This bitmap differs from the on-disk block group bitmap in that it considers blocks in preallocation space as allocated. The free extent information and the bitmap is then stored in the page cache of an in-core inode, and is indexed with the group number. The page containing the free extent information and bitmap is calculated as shown in Fig 6.

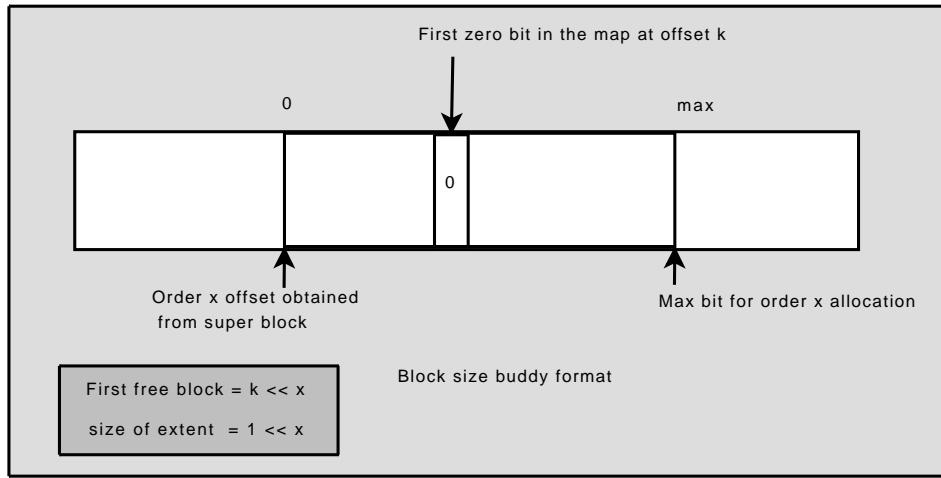


Figure 5: Ext4 buddy cache layout

```
/*
 * the buddy cache inode stores the block bitmap
 * and buddy information in consecutive blocks.
 * So for each group we need two blocks.
 */
block = group * 2;
pnum = block / blocks_per_page;
poff = block % blocks_per_page;

page = find_get_page(inode->i_mapping, pnum);
.....
if (!PageUptodate(page)) {
    ext4_mb_init_cache(page, NULL);
.....
```

Figure 6: Buddy cache inode offset mapping

Before allocating blocks from the buddy cache, we normalize the request. This helps prevent heavy fragmentation of the free extent map, which groups free blocks in power of 2 size. The extra blocks allocated out of the buddy cache are later added to the preallocation space so that the subsequent block requests are served from the preallocation space. In the case of large files, the normalization follows a list of heuristics based on file size. For smaller files, we normalize the block request to stripe size if specified at mount time or to `s_mb_group_prealloc`. `s_mb_group_prealloc` defaults to 512 and can be configured via `/proc/fs/ext4/<partition/group_prealloc`

Searching for blocks in buddy cache involves:

- Search for requested number of blocks in the extent

map.

- If not found, and if the request length is same as stripe size, search for free blocks in stripe size aligned chunks. Searching for stripe aligned chunks results in better allocation on RAID setup.
- If not found, search for free blocks in the bitmap and use the best extent found.

Each of these searches starts with the block group in which the goal block is found. Not all block groups are used for the buddy cache search. If we are looking for blocks in the extent map, we only look at block groups that have the requested order of blocks free. When searching for stripe-size-aligned free blocks we only look at block groups that have stripe size chunks of free blocks. When searching for free blocks in the bitmap, we look at block groups that have the requested number of free blocks.

2.4 Ext4 multiple block allocator performance advantage

The performance advantage of the multiple block allocator related to small files is shown in Fig 7. The blocks are closer because they are satisfied from the same locality group preallocation space.

The performance advantage of multiple block allocator related to large files is shown in Fig 8. The blocks are closer because they are satisfied from the inode-specific preallocation space.

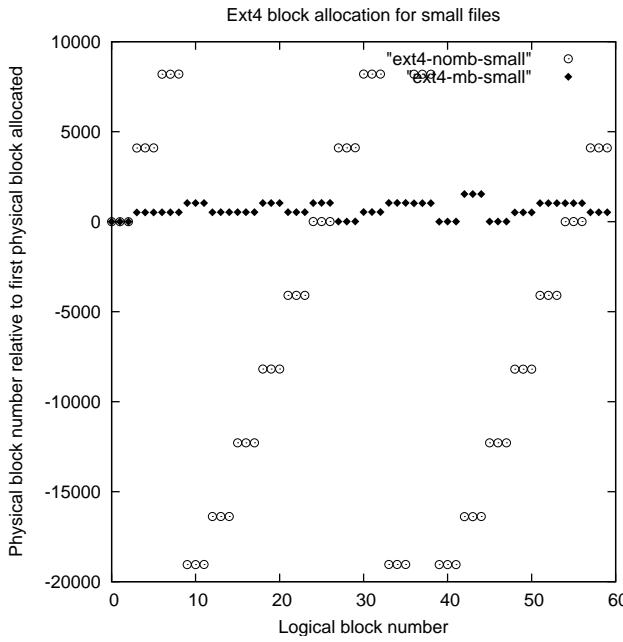


Figure 7: Multiple block allocator small file performance

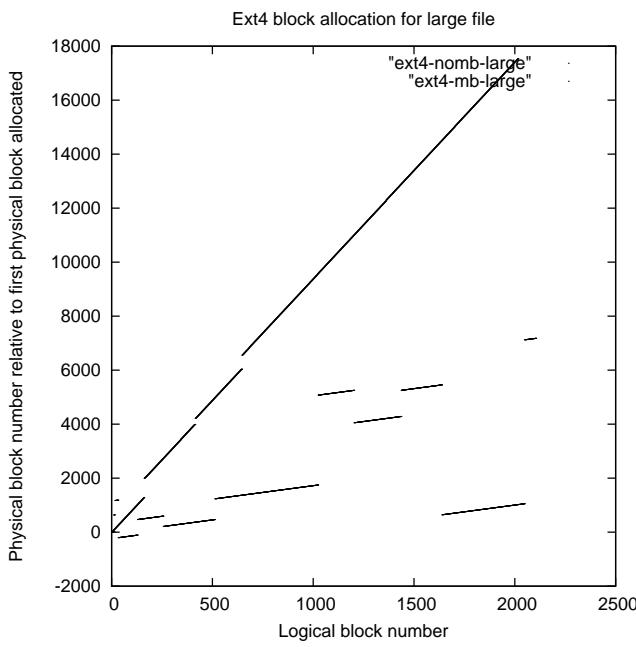


Figure 8: Multiple block allocator large file performance

Table 1 shows the compilebench[5] number comparing Ext4 and Ext3 allocators with the `data=ordered` mount option. Compilebench tries to age a file system by simulating some of the disk IO common in creating, compiling, patching, stating and reading kernel trees. It indirectly measures how well file systems can maintain directory locality as the disk fills up and directories age

Test	Ext4 allocator	Ext3 allocator
initial create	20.44 MB/s	19.64 MB/s
create total	11.81 MB/s	8.12 MB/s
patch total	4.95 MB/s	3.66 MB/s
compile total	16.66 MB/s	12.57 MB/s
clean total	247.54 MB/s	82.57 MB/s
read tree total	7.06 MB/s	6.99 MB/s
read compiled tree total	9.06 MB/s	10.40 MB/s
delete tree total	7.06 seconds	16.66 seconds
delete compiled tree total	9.64 seconds	22.21 seconds
stat tree total	5.31 seconds	13.39 seconds
stat compiled tree total	5.63 seconds	14.70 seconds

Table 1: Compliebench numbers for Ext4 and Ext3 allocator

2.5 Evolution of an Allocator

The mballoc allocator included in Ext4 is actually the third generation of this allocation engine written by Alex Tomas (Zhuravlev). The first two versions of the allocator were focused mainly on large allocations (1 MB at a time), while the third generation also works to improve small files allocation.

Even at low I/O rates, the single-block allocation used by Ext3 does not necessarily make a good decision for inter-block allocations. The Linux VFS layer splits any large I/O submitted to the kernel into page-sized chunks and forces single-block allocations by the file system without providing any information about the rest of the outstanding I/O on that file. The block found for the first allocation is usually the first free block in the block group, and no effort is made to find a range of free blocks suitable for the amount of data being written to the file. This leads to poor allocation decisions, such as selecting free block ranges that are not large enough for even a single `write()` call.

At very high I/O rates (over 1 GB/s) the single-block allocation engine also becomes a CPU bottleneck because

every block allocation traverses the file system allocator, scans for a free block, and locks to update data structures. With mballoc, one test showed an improvement from 800 MB/s to 1500 MB/s on the same system by reducing the CPU cost per allocated block.

What is critical to the success of mballoc is the ability to make smart allocation decisions based on as many blocks of file data as possible. This necessitated the development of delayed allocation for normal I/O. When mballoc has a good idea that a file is small or large, and how many data blocks to allocate, it can make good decisions.

The first version of mballoc used the buddy allocator only to allocate contiguous chunks of blocks, and would align the allocation to the start of a free extent. While this still leads to performance gains due to avoided seeks and aggregation of multiple allocations, it is not optimal in the face of RAID devices that have lower overhead when the I/O is properly aligned on RAID disk and stripe-wide boundaries.

The second version of mballoc improved the buddy allocator to align large allocations to RAID device boundaries. This is easily done directly from the buddy bitmap for RAID geometries that use power-of-two numbers of data disks in each stripe by simply stopping search for free bits in the buddy bitmap at the RAID boundary size.

Avoiding the read-modify-write cycle for RAID systems can more than double performance because the costly synchronous read is avoided. In RAID devices that have a read cache that is aligned to the stripe boundaries, doing a misaligned read will double the amount of data read from disk and fill two cachelines.

The Ext4 superblock now has fields that store the RAID geometry at mke2fs time or with tune2fs. It is likely that the complex RAID geometry probing done at mkfs time for XFS will also be adopted by mke2fs in order to populate these fields automatically.

During testing for the current third version of mballoc the performance of small file I/O was investigated, and it was seen that the aggregate performance can be improved dramatically only when there are no seeks between the blocks allocated to different files. As a result the group allocation mechanism is used to pack small allocations together without any free space in between. In the past there was always space reserved or left at the end of each file “just in case” there were more blocks to

allocate. For small files, this forces a seek after each read or write. With delayed allocation, the size of a small file is known at allocation time and there is no longer a need for a gap after each file.

The current mballoc allocator can align allocations to a RAID geometry that is not power-of-two aligned, though it is slightly more expensive.

More work still remains to be done to optimize the allocator. In particular, mballoc keeps the “scan groups for good allocations” behaviour of the Ext2/3 block allocator. Implementing an in-memory list for more optimal free-extent searching, as XFS does, would further reduce the cost of searching for free extents.

Also, there is some cost for initializing the buddy bitmaps. Doing this at mount time, as the first version of mballoc did, introduces an unacceptable mount delay for very large file systems. Doing it at first access adds latency and hurts performance for the first uses of the file system. Having a thread started at mount time to scan the groups and do buddy initialization asynchronously among other things, would help avoid both issues.

3 Delayed allocation

Because Ext4 uses extent mapping to efficiently represent large files, it is natural to process a multiple block allocation together, rather than one block allocation at a time as done in Ext3. Because block allocation requests for buffered I/O are passed through the VFS layer one at a time at the `write_begin` time, the underlying Ext3 file system cannot foresee and cluster future requests. Thus, delayed allocation is being proposed multiple times to enable multiple block allocation for buffered I/O.

Delayed allocation, in short, defers block allocations from `write()` operation time to page flush time. This method provides multiple benefits: it increases the opportunity to combine many block allocation requests into a single request reducing fragmentation and saving CPU cycles, and avoids unnecessary block allocation for short-lived files.

In general, with delayed allocation, instead of allocating the disk block in `write_begin`, the VFS just does a plain block look up. For those unmapped buffer heads, it calculates the required number of blocks to reserve (including data blocks and meta-data blocks),

reserves them to make sure that there are enough free blocks in the file system to satisfy the write. After that is done, it marks the buffer heads as delayed allocated(BH_DELAY). No block allocation is done at that moment. Later, when the pages get flushed to disk by `writepage()` or `writepages()`, these functions will walk all the dirty pages in the specified inode, cluster the logically contiguous ones, and attempts to perform cluster block allocation for those buffer heads marked as BH_DELAY all together, then submit the page or pages to the bio layer. After the block allocation is complete, the unused block reservation is returned back to the file system.

The current implementation of delayed allocation is mostly done in the VFS layer, hoping that multiple file systems, such as Ext2 and 3, can benefit from the feature. There are new address space operations added for Ext4 for delayed allocation mode, providing call back functions for `write_begin()`, `write_end()` and `writepage()` for delayed allocation, with updated block allocation/reservation/look-up functions. The current Ext4 delayed allocation only supports `data=writeback` journalling mode. In the future, there are plans to add delayed support for `data=ordered` journalling mode.

4 FLEX_BG and the Inode allocator

4.1 The old inode allocator

The inode allocator in an Ext2/3 file system uses the block groups as the vehicle to determine where new inodes are placed on the storage media. The Ext2/3/4 file system is divided into small groups of blocks with the block group size determined by the amount of blocks that a single bitmap can handle. In a 4 KB block file system, a single block bitmap can handle 32768 blocks for a total of 128 MB per block group. This means that for every 128 MB, there will be meta-data blocks (block/inode bitmaps and inode table blocks) interrupting the contiguous flow of blocks that can be used to allocate data.

The Orlov directory inode allocator [6] tries to maximize the chances of getting large block allocations by reducing the chances of getting an inode allocated in a block group with a low free block count. The Orlov allocator also tries to maintain locality of related data

(i.e. files in the same directory) as much as possible. The Orlov allocator does this by looking at the ratio of free blocks, free inodes, and number of directories in a block group to find the best suitable placement of a directory inode. Inode allocations that are not directories are handled by a second allocator that starts its free inode search from the block group where the parent directory is located and attempts to place the inodes with the same parent inode in the same block group. While this approach works very well given the block group design of Ext2/3/4, it does have some limitations:

- A 1 TB file system has around 8192 block groups. Searching through that many block groups on a heavily used file system can become expensive.
- Orlov can place a directory inode in a random location that is physically far away from its parent directory.
- Orlov's calculations to find a single 128 MB block group are expensive in this multi-terabyte world we live in.
- Given that hard disk seek times are not expected to improve much during the next couple of years, while at the same time seeing big increases in capacity and interface throughput, the Orlov allocator does little to improve data locality.

Of course the real problem is not Orlov itself, but the restrictions imposed on it by the size of a block group. One solution around this limitation is to implement multi-block bitmaps. The drawback with this implementation is that things like handling bad blocks inside one of those bitmap or inode tables becomes very complicated when searching for free blocks to replace the bad ones. A simpler solution is to get rid of the notion that meta-data need to be located within the file system block group.

4.2 FLEX_BG

Simply put, the new FLEX_BG feature removes the restriction that the bitmaps and inode tables of a particular block group MUST be located within the block range of that block group. We can remove this restriction thanks to e2fsprogs being a robust tool at finding errors through fsck. While activating the FLEX_BG feature flag itself

doesn't change anything in the behavior of Ext4, the feature does allow mke2fs to allocate bitmaps and inode tables in ways not possible before. By tightly allocating bitmaps and inode tables close together, one could essentially build a large virtual block group that gets around some of the size limitations of regular block groups.

The new extent feature benefits from the new meta-data allocation by moving meta-data blocks that would otherwise prevent the availability of contiguous free blocks on the storage media. By moving those blocks to the beginning of a large virtual block group, the chances of allocating larger extents are improved. Also, having the meta-data for many block groups contiguous on disk avoids seeking for meta-data intensive workloads including e2fsck.

4.3 FLEX_BG inode allocator

Given that we can now have a larger collection of blocks that can be called a block group, making the kernel take advantage of this capability was the obvious next step. Because the Orlov directory inode allocator owed some design decisions to the size limitations of traditional block groups, a new inode allocator would need to employ different techniques in order to take better advantage of the on disk meta-data allocation. Some of the ways the new inode allocator differs from the old allocator are:

- The allocator always tries to fill a virtual block group to a certain free block ratio before attempting to allocate on another group. This is done to improve data locality on the disc by avoiding seeks as much as possible.
- Directories are treated the same as regular inodes. Given that these virtual block groups can now handle multiple gigabytes worth of free blocks, spreading directories across block groups not only does not provide the same incentive, it could actually hurt performance by allowing larger seeks on relatively small amounts of data.
- The allocator may search backwards for suitable block groups. If the current group does not have enough free blocks, it may try the previous group first just in case space was freed up.

- The allocator reserves space for file appends in the group. If all the blocks in a group are used, appending blocks to an inode in that group could mean that the allocation could happen at a large offset within the storage media increasing seek times. This block reservation is not used unless the last group has been used past its reserved ratio.

The size of a virtual group is always a power-of-two multiple of a *normal* block group in size, and it is specified at mke2fs time. The size is stored in the super block in to control how to build the in-memory free inode and block structure that the algorithm uses to determine the utilization of the virtual block group. Because we look at the Ext4 block group descriptors only when a suitable virtual group is found, this algorithm requires fewer endian conversions than the traditional Orlov allocator on big endian machines.

One big focus of this new inode allocator is to maintain data and meta-data locality to reduce seek time when compared to the old allocator. Another very important area of focus is reduction in allocation overhead. By not handling directory inodes differently, we remove a lot of the complexity from the old allocator while the smaller number of virtual block groups makes searching for adequate block groups easier.

Now that a group of inode tables is treated as if it were a single large inode table, the new allocator also benefits from another new feature found in Ext4. Uninitialized block groups mark inode tables as uninitialized when they are not in use and thus skips reading those inode tables at fsck time providing significant fsck speed improvements. Because the allocator only uses an inode table when the previous table on the same virtual group is full, fewer inode tables get initialized resulting in fewer inode tables that need to be loaded and improved fsck times.

4.4 Performance results

We used the FFSB[4] benchmark with a profile that executes a combination of small file reads, writes, creates, appends, and deletes. This helps simulate a meta-data heavy workload. A Fibre Channel disk array was used as a storage media for the FFSB test with 1 GB of fast write cache. The benchmark was run with a FLEX_BG virtual block group of 64 packed groups and it is compared to a regular EXT4 file system mounted with both

mballoc and delalloc. Note that 64 packed groups is not the optimum number for every hardware configuration, but this number provided very good overall performance for workloads on the hardware available. Further study is needed to determine the right size for a particular hardware configuration or workload.

Op	Ext4	Ext4(flex_bg)
read	67937 ops	73056 ops
write	98488 ops	104904 ops
create	1086604 ops	1228395 ops
append	31903 ops	33225 ops
delete	59503 ops	70075 ops
Total ops/s	4477.37 ops/s	5026.78 ops/s

Table 2: FFSB small meta-data FibreChannel(1 thread)-FLEX_BG with 64 block groups

In Table 2, the single threaded results show a 10% overall improvement in operations-per-second throughput. The 16 threaded results in Table 3 show an even better improvements of 18% over the regular EXT4 allocation. The results also show that there is 5.6% better scalability from 1 to 16 threads when using FLEX_BG grouping compared to the normal allocation.

Op	Ext4	Ext4(flex_bg)
read	96778 ops	119135 ops
write	143744 ops	174409 ops
create	1584997ops	1937469 ops
append	46735 ops	56409 ops
delete	93333 ops	113598 ops
Total ops/s	6514.51 ops/s	7968.24 ops/s

Table 3: FFSB small meta-data (16 threads)- FLEX_BG with 64 block groups

To see the overall effect of the new allocator on more complex directory layouts, we used Compilebench[5], which generates a specified number of Linux kernel tree like directories with files that represent file sizes in the actual kernel tree. In table 4, we see the overall results for the benchmark are better when using grouping and the new inode allocator. One exception is both the "read tree" and the "read compiled tree" which show slightly slower results. It shows that there is still some room for improvement in the meta-data allocation and the inode allocator.

Test	Ext4	Ext4(flex_bg)
intial create	73.38 MB/s	81.09 MB/s
create total	69.90 MB/s	73.32 MB/s
patch total	21.73 MB/s	21.85 MB/s
compile total	125.52 MB/s	127.26 MB/s
clean total	921.01 MB/s	973.91 MB/s
read tree total	18.73 MB/s	18.43 MB/s
read compiled tree total	35.59 MB/s	33.91 MB/s
delete tree total	4.18 seconds	3.70 seconds
delete compiled tree total	4.11 seconds	3.90 seconds
stat tree total	2.68 seconds	2.59 seconds
stat compiled tree total	3.20 seconds	2.86 seconds

Table 4: Compliebench FiberChannel - FLEX_BG with 64 block groups

4.5 Future work

The concept of grouping meta-data in EXT4 is still in its infancy and there is still a lot of room for improvement. We expect to gain performance out of the EXT4 file system by looking at the layout of the meta-data. Because the definition of the FLEX_BG feature removes meta-data placement restrictions, this allows the implementation of virtual groups to be fluid without sacrificing backward compatibility.

Other optimizations that are worth exploring are placement of meta-data in the center of the virtual group instead of the beginning to reduce the worst case seek scenario. Because the current implementation just focuses on inodes as a way to manipulate the block allocator, future implementations could go further still and make the various block allocators FLEX_BG grouping aware. Because the right size of groups depends on things related to the disk itself, making mke2fs smarter to automatically set the virtual group size base on media size, media type and RAID configuration will help users deploy this feature.

Shorter-term fixes for removing file system size limitations are in the works. The current code stores the sum of free blocks and inodes of all the groups that build a virtual block group in an in-memory data structure. This means that if the data structure exceeds page size, the allocation would fail, in which case we revert back to the old allocator on very large file systems. While this is a performance feature, storing all this informa-

tion in memory may be overkill for very large file systems. Building this on top of an LRU scheme removes this limitation and saves kernel memory.

5 Conclusion

The main motivation for forking the Ext4 file system from the Ext3 file system was to break the relatively small file system size limit. This satisfies the requirements for larger files, large I/O, and a large number of files. To maintain the Ext4 file system as a general purpose file system on a desktop, it is also important to make sure the Ext4 file system performs better on small files.

We have discussed the work related to block allocation and inode allocation in the Ext4 file system and how to satisfy the conflicting requirements of making Ext4 a high performance general purpose file system for both the desktop and the server. The combination of pre-allocation, delayed allocation, group preallocation, and multiple block allocation greatly help reduce fragmentation issues occurring on large file allocation and poor locality issues on small files that have been seen in Ext3 file system. By tightly allocating bitmap and inode tables close together with `FLEX_BG`, one could essentially build a large virtual block group that increases the likelihood of allocating large chunks of extents and allows Ext4 file system to handle better on meta-data-intensive workload. Future work, including support for delayed allocation for ordered mode journalling and online defragmentation, will help to future reduce file fragmentation issues.

Acknowledgements

We would like to thank many Ext3/4 developers for their contribution to Ext4 file system, especially grateful to Alex Thomas, Theodore Tso, Dave Kleikamp, Eric Sandeen, Jan Kara, Josef Bacik, Amit Arora, Avantika Mathur, and Takashi Sato.

We owe thanks to Jean-Noël Cordenner and Valérie Clément, for their help on performance testing and analysis, and development and support of the Ext4 file system.

We owe thanks to Eric Sandeen for his careful review on an earlier draft of this manuscript.

Legal Statement

Copyright © 2008 IBM.

Copyright © 2008 Sun Microsystems, Inc.

This work represents the view of the authors and does not necessarily represent the view of IBM or Sun Microsystems.

IBM and the IBM logo are trademarks or registered trademarks of International Business Machines Corporation in the United States and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

References in this publication to IBM products or services do not imply that IBM intends to make them available in all countries in which IBM operates.

This document is provided “AS IS,” with no express or implied warranties. Use the information in this document at your own risk.

References

- [1] BEST, S. JFS overview
<http://jfs.sourceforge.net/project/pub/jfs.pdf>.
- [2] MATHUR, A., CAO, M., BHATTACHARYA, S., DILGER, A., TOMAS, A AND VIVER, L. The New ext4 filesystem: current status and future plans. In *Ottawa Linux Symposium* (2007).
<http://ols.108.redhat.com/2007/Reprints/mathur-Reprint.pdf>
- [3] BRYANT, R., FORESTER, R., HAWKES, J. Filesystem Performance and Scalability in Linux 2.4.17 . In *USENIX Annual Technical Conference, Freenix Track* (2002). http://www.usenix.org/event/usenix02/tech/freenix/full_papers/bryant/bryant_html/
- [4] Ffsb project on sourceforge. Tech. rep. <http://sourceforge.net/projects/ffsb>.
- [5] Compilebench Tech. rep. <http://oss.oracle.com/~mason/compilebench>.
- [6] COBERT, J. The Orlov block allocator.
<http://lwn.net/Articles/14633/>.

Bazillions of Pages

The Future of Memory Management under Linux

Christoph Lameter

Silicon Graphics, Inc.

christoph@lameter.com

Abstract

A new computer system running Linux is likely equipped with at least 4GB of memory. The Linux VM manages this memory in 4KB chunks. So the Linux VM has to manage 1 million memory chunks. There are some people who already run configurations with tens or hundreds of gigabytes of memory. As time progresses these large memory sizes are going to become more and more common. The Linux VM will have to manage more and more pages. For effective memory reclaim these pages may have to be repeatedly scanned in order to determine the least recently used pages.

It is not surprising that the VM starts to struggling with the increasing amount of work. At 8 to 16GB one can observe live lock situations with certain loads. A number of possible solutions to this problem are considered: One is Rik van Riel's work of optimizing the way pages are handled in the VM, another is Andrea Arcangeli's increase in the base page size. And yet another is to make the page size dynamic in order to allow subsystems to choose the page size that is most beneficial for a given load.

1 Introduction

1.1 4 megabytes are lots of memory

My first Linux installation was done using a set of floppy disks containing Slackware (1.0.4) and a strange kernel version that was numbered 0.99 followed by some trailing letters and numbers that I cannot remember. It was 1993 and I had to download 12 disk images through a dial up line which took almost a week. The download was using an advanced file transfer protocol named Z-modem. The Slackware software was installed on a machine that had a 386SX processor (no floating point unit)

and 4 megabytes of memory. The machine was already a big step forward from my first computer, a PET 2001 (Commodore Business Machines) that I gained access to in 1978. The PET had 4KB of memory. The 386SX machine was great and I thought really had plenty of memory especially since MS-DOS could only use 640 kilobytes.

1.2 More and more memory

Fast forward to today and now I work at Silicon Graphics on making Linux run well on Supercomputers. This adds another strange twist since I get to work with machines that have thousands of times more memory than an average computer. SGI has customers with machines equipped with several petabytes of RAM. And given the way the capacities develop: It may take less than a decade until we get to machines that have memory sizes in the exabyte range.¹

So there is the chance of seeing how Linux handles super large memory sizes years before they are available in smaller computers for everyone. The time delay is about a decade or so before these become available in an average computer. Linux servers with one terabyte of RAM will likely become available around 2010, memory sizes of a petabyte may be possible by 2018. Machines like that will also have a couple of hundred processors (cores?) accessing that memory.²

I see how Linux runs with large memory sizes years before the same memory sizes are available to the general public. If there is a hard issue related to memory then it frequently ends up on my desk. The advantage for Linux is that we have a chance to prepare the Linux kernel for the upcoming memory issues years before they become a problem for the general users of Linux.

¹One exabyte has a million gigabytes and a petabyte a thousand gigabytes.

²See Intel's plans for large numbers of cores[6].

One of the areas of concern is that memory sizes keep growing while processor speeds and memory access speeds are mostly stagnating. Memory management in Linux occurs by managing a small piece of meta data (the `struct page`) for each 4KB chunk of memory (*a page*.) Whenever the kernel needs to perform I/O, when a page fault occurs, or when the kernel is handling memory in some other way then the meta data in `struct page` is used for synchronization and for tracking the state of the page.

2 The memory problem

The available computing resources to manage the meta data in the `struct page` are shrinking because memory sizes grow faster than processing speeds. Each new generation of hardware grows the number of 4KB pages that have to be managed. The VM therefore has to deal with an ever growing number of pages (bazillions of pages because bazillions is an undetermined large number.) It looks like the trend will continue for the foreseeable future.³

As a result we see critical OS activities like memory reclaim taking a larger and larger portion of computing resources. For memory reclaim, the page expiration is based on examining all the the meta data structures (in `struct page`) at some point. I/O bottlenecks also develop because each 4KB page has to be handled separately for DMA transfers.⁴

The larger the dataset that is streamed through the system becomes, the larger the scaling issues that we will encounter due to the meta data that has to be kept for each 4KB page that is processed by the I/O layer.

Table 1 shows the development of memory sizes, processor capabilities and memory access speed for an average computer (simplified.) The number of pages increases faster than the number of cores and the speed of memory. Various hardware tricks are used in memory subsystems to improve speed because we cannot change

³Maybe there will be a plateau when the limits of what is addressable within a 64 bit address space (at 8 exabytes) is reached. Not likely to occur until 2024 or so for the average Linux server but I would expect memory sizes like that to be reached by 2014 in the supercomputer area.

⁴Supercomputer I/O is expected to be able to saturate the links to the storage subsystem which contains large RAID configurations (thousands of disks.) Being able to just saturate the bandwidth of a single hard disk is certainly not acceptable.

Year	Mem	Pages	ClckFreq	Cores	MSpeed
1993	4MB	1024	16Mhz	1	70ns
2001	32MB	8192	300Mhz	1	60ns
2005	1GB	256000	1-2 Ghz	1	55ns
2008	4GB	1 million	2-3Ghz	2	50ns
2010?	64GB	16 million	2-3Ghz	4-8	45ns?
2020?	128TB	32 billion	3-4Ghz	128	40ns?

Table 1: Memory sizes and processors

the basic physical limitations of how DRAM works. Many of the optimizations are based on the assumption of linear memory accesses, other optimizations are increasing the number of bits that can be fetched simultaneously. Processors have to manage larger and larger CPU caches in multiple layers (L1, L2, L3, maybe L4 soon?) to avoid the penalty of memory accesses. Memory accesses become more expensive as the distance between the processor and memory increases. The optimizations make a relatively small amount of memory accessible without long latencies and favor linear accesses to memory.

In such an environment memory locality becomes an important consideration for improving the speed of the computer system as a whole. The management of ever larger lists of `struct page` referring to pages which over time become distributed seemingly randomly all over the memory of the system does not have a beneficial effect on performance.

The ratio of cache to memory size is also falling following a similar trajectory. The cost of a random memory access will become more and more expensive over time since the chance of hitting an object in the CPU cache by chance is reduced.

The situation for Supercomputer configurations is worse (the following data is for SGI Altix 3700/4700) because the memory sizes are much larger while the processor and memory are similar to other contemporary hardware. On the other hand the number of processors is larger but the number of processors also grows slower than the total amount of memory. There is the additional complexity of scaling the synchronization methods:

Table 2 shows the development of the sizes for Supercomputers. To some extent the massive amount of page meta data was reduced on Itanium by increasing the page size. A 16KB page size means that there are only one fourth of the pages to worry about. The I/O subsystems can perform linear DMA transfers for 16KB

Year	Mem	Procs	PageSize	Pages
2004	8TB	512	16K (ia64)	512 million
2006	16TB	1024	16K	1 billion
2007	4PB	2048	16k	256 billion
2008	4PB	4096	64k	64 billion
2008	16TB	4096	4k(x86)	4 billion
2014?	1EB?	16384?	4k(x86)	256 trillion

Table 2: Supercomputer memory sizes and processors

chunks which reduces the number of scatter gather entries that have to be managed by the storage subsystem. The increase to a page size of 64KB in 2008 decreases the management overhead again but the overall number of pages still stays comparatively high. We noticed that configurations with over 2 thousand processors only run reliably with 64KB pages. Otherwise the VM regularly gets into fits while handling large queues of pages even if running a HPC load that requires minimal I/O.

The successor to the Altix series will no longer be based on Itanium processors but on x86 architecture. We can no longer utilize large page sizes, but have to use the only page size that x86 supports which is 4KB. Ironically these systems are intended to support even larger memory sizes but the current physical addressing capabilities of the Xeon line limit the amount of physical memory that a single Linux instance will be able to access. The address size limitations effectively reduce the number of pages to be managed by a Linux instance but will force the construction of large shared memory machines running a number of Linux instances that each live in their own 16TB memory segment. Then we need specialized hardware that bridges between the Linux instances. Not a nice picture but the limit on the physical memory that can be addressed by a processor also limits the number of page structs that have to be managed by the kernel.

The address space sizes are likely to be increased as processor development continues. The future generations of processor will then be able to reach similar memory capacities as the Itanium based systems are capable of today. The number of page structs that have to be managed by the VM makes a jump of several orders for similar memory sizes on Itanium versus Xeon. There are at least 4 times more page structs compared to an Itanium system with a 16k page size or even 16 times more page structs for systems currently running 64KB page size.

2.1 The VM heart attack

Let's consider a typical scenario that can lead to a system appearing to live lock. The more processors and memory are involved, the more likely these are to occur.

The Linux VM uses lists of pages in order to determine which 4KB pages contain information that is no longer worth keeping in memory. These lists establish the least used memory in the system and then the Linux VM can reclaim that memory for other uses. For each page it has to be determined if it was used since the last scan by checking a referenced bit. The VM **must** therefore visit all pages regularly in order to correctly expire pages from memory. The more memory there is, the larger these lists become. The more allocations are performed, the more aggressive the VM has to scan and then trim pages via these lists.

As long as there is no lock contention and reclaim passes only take a small amount of time everything will be okay. The system is not under much memory pressure and has a reasonable chance to find freeable memory with short scans.

If memory allocations continue to occur and multiple processes start to expire memory then extensive scanning may result. Since we keep on adding more processors (due to the increasing use of multi-core technology) lock contention may also result because multiple processors attempt to reclaim memory simultaneously from the same memory range. Each has to wait for the lock in order to be allowed to scan the list. These lists are protected by spin locks and so other processors are waiting by spinning on the lock. The more processors the more likely live lock scenarios can develop due to starvation or simple slowdown because processors have to wait for locks that are held for a long time.

On NUMA it is typical that things take a turn for the worse when direct reclaim is beginning to occur concurrently. At that point the VM is walking down potentially long zone lists. Most of those are remote and memory accesses are especially expensive. If concurrent reclaim occurs within the same zone from multiple remote processors then excessive latencies will slow reclaim down further. At the end a majority of the processors may be in reclaim and only minimal processor time may become available for user processes to make progress with data processing.

2.2 Randomized memory references

The larger the memory the higher the chances of TLB misses which may also slow the machine. TLBs can map 4KB or 2MB sections of memory. The number of TLBs that a processor can cache is limited. If the processor can handle 512 4KB TLB entries (like in the upcoming Nehalem processor) then the processor can access only 2M of memory without a TLB miss. The number of supported TLBs for 2 megabyte entries is only 64 which allows the access to 128 megabyte of memory. TLB misses can be fast if the page table entries are held in the processor cache but a cache miss of the page table entry can introduce significant latencies, and the larger the amount of memory the higher the chance of these misses.

With increased memory the accesses to data locations in memory will be more sparse if data is not allocated closely together. It is therefore becoming expensive to follow pointers to memory that has not recently been accessed, and TLB misses become more and more likely to occur. There is the increasing chance of CPU cache misses, and the memory architectures are optimized for neighboring memory accesses which cannot handle sparse memory accesses effectively.

There are therefore multiple reasons why it is advantageous to use memory that is near other memory that was recently accessed. However, the arrays of pointers to struct page that we currently use for various purposes in the VM have a problem here because they have no locality.⁵ Pointers may go to arbitrary pages all over memory. For all practical purposes these pointer lists may degenerate to accesses that are seemingly random, defeating the highly developed logic put in the processor to prefetch memory. In the worst case these lists may cause a TLB miss for each page struct on the list.

2.3 I/O fragmentation

The result of degeneration of page lists to access to random locations in memory has another important consequence for I/O. The pages that are sent down to the I/O subsystem cannot be coalesced into larger linear chunks

⁵The situation for page structs is still better than references to objects in the pages because page structs are placed in a special memmap area whereas other objects can be placed anywhere in memory.

(which is typically possible for some time after boot because pages that follow each other are allocated in order.) And therefore the I/O devices have to do I/O via scatter/gather entries to bazillion of pages in seemingly random locations in memory. I/O devices suitable for Linux must support an ever increasing number of scatter gather entries. The scatter gather list complexity becomes a potential I/O performance bottleneck.

3 Solutions for handling large amounts of memory

The main problem here is that the VM has to sift through too much meta data for key operations like memory allocation, reclaim, and I/O. Plus the meta data is seemingly randomly distributed over all of memory reducing optimizations that the memory subsystem could make. The following solutions are focusing on reducing the scanning effort, reducing the amount of memory references for key VM operations, and increasing the locality of access in order for CPU caches, TLBs, and memory subsystems to be able to optimize memory accesses.

3.1 Reclaim improvements

One approach is to look at the problems that are emerging with memory reclaim in the VM. Rik van Riel has, over the last years, investigated a variety of methods to improve memory reclaim. These methods result in a better determination of which pages to evict from memory utilizing new reclaim algorithms developed in an academic setting. Among them are: ARC, ClockPro, CAR, and LIRS.⁶ These improvements would allow faster expiration of pages by reducing scan time and allow a more accurate prediction of pages that may be needed in the future.

The other aspect of Rik and other developers work on reclaim is that methods are developed to exclude pages that are unreclaimable from the reclaim lists and lists are created that contain easily reclaimable pages. These methods reduce the scanning overhead and may reduce the problems that we currently see. They are particularly good for specialized loads that result in large numbers of unreclaimable pages. The current reclaim in the Linux VM has to scan unreclaimable pages again and again which is obviously good to avoid.

⁶See <http://linux-mm.org> for more information about these approaches.

These are interesting approaches that enhance page reclaim and allow us to manage even more page structs in a better way. However, they do not address the fundamental issue that there is too much meta data for the VM to handle. The advanced reclaim methods still require eventually scanning through all the pages. The number of pages is not reduced but keeps on growing while we make minimal progress in getting these advanced reclaim algorithms working in the Linux VM.

3.2 Increasing the default page size

The solution that we have adopted for Itanium is to change the default page size. On Itanium this is supported by the hardware, so it's easy to do and the increase in page size has a significant effect in reducing the VM overhead for those large machines.

The x86 platform only supports 4KB and 2MB (64-bit) page table entries. We could work with that and increase the default page size by installing multiple 4KB page table entries in order to simulate e.g. a 16KB or 64KB “page” like is done on IA64.

But it is not clear that one actually would want a larger default page size. The 4KB page size is appropriate for the executables and small files that are needed by the operating system. The main use of larger sized pages are for applications that either perform a large amount of I/O (databases, enterprise applications) or need large amounts of memory (HPC applications.)

The binary format is also affected. Current binaries are formatted to have data aligned on 4KB boundaries. If that is no longer the case then we either have to change the binary format or provide some sort of layer that allows 4KB aligned access although the default page size is larger. The easy solution out of that may be to simply redefine the binary format and rebuild a completely new distribution. But that would require some work on binutils, the linker, and the loader.

Another idea that avoids multiple 4KB PTEs per large page is to set the default page size to the next higher page size which is 2 megabytes on x86. Essentially we are using a PMD for a PTE. Such a specialized version of Linux could perhaps run as a guest inside a virtualized environment (KVM, Lguest) in order to allow the special HPC or Enterprise class applications to run. The applications would have to be compiled for an environment that has a 2MB base page size and we would need

a minimal distribution to create essential binaries that are necessary for the application.

3.3 Optional Support for larger page sizes

Optional support for larger page sizes means that the binary format can be left intact. User space works as it always has. Without enabling additional options the behavior of the kernel does not change. Optional large page size support means that all existing kernel APIs to user space stay as they are. Large page support can be switched on for special purposes by—for example—formatting a disk with a larger block size than 4KB. Or one could create a pseudo file system in memory with a larger page size that is then mapped into a processes memory.

However, larger pages means that the VM must now support more frequent allocations of contiguous memory larger than a 4KB. Requests for contiguous memory may vary in size. Memory fragmentation in Linux may increase.

Linux already has fragmentation avoidance logic implemented by Mel Gorman and one active defragmentation method (lumpy reclaim.) Both are measures to keep contiguous memory available. Both increase the chance of being able to obtain contiguous memory beyond the size of a single page but neither can guarantee that a large contiguous memory chunk is available at any point in time. Any user of contiguous memory beyond 4KB must implement fallback measures that kick in if large contiguous memory is not available. In fallback we loose the localization of memory accesses and increase the cache footprint. Fallback usually is realized by managing lists of small 4KB pages.

These fallbacks are currently rare but with the increase of demand for larger allocations the fallbacks may become more frequent. Additional defragmentation measures could be needed to produce more contiguous memory to avoid fallbacks to lists of pages.

4 Virtualizable Compound Pages

Virtualizable Compound Pages are a first step to support allocations of varying sizes of larger pages in the kernel. An allocation request for a Virtualizable Compound Page will first attempt to allocate linear physical

memory of the requested size (a real Compound Page.) Typically these allocations will be successful resulting in a large page useful to allow the localization of multiple objects, the use of large stacks, or temporary storage for various purposes.

If the page allocator does not have sufficient linear memory available then the fallback logic will allocate a series of 4KB pages and use the vmalloc functionality to allow the memory provided to be used as virtually contiguous memory. This increases the overhead because the processor needs to use a page table to look up the memory location of each 4KB chunk but the page table handling logic is usually integrated in the processor and therefore fast and entries are cached. However, memory may be physically dispersed which means the optimizations for linear access of the memory subsystem and I/O subsystems may not be triggered. An array of pointers to the pages has to be maintained as well.

Virtualizable Compound Pages have the disadvantage that the user of these pages must always be aware that the linear memory that was provided by the allocator may be virtual and that actual physical memory may not be contiguous. If for example, a device needs to operate on the memory of a Virtualized Compound Page then the device needs to perform scatter gather operations to the physical pages that constitute the Virtualized Compound Page.

Additional problems result if the processor needs to have TLB entries loaded via a processor trap (like on IA64.) In that case access to the virtualized memory requires the ability to handle the faults in the contexts that the virtualized compound is used. In the case of IA64 the use of Virtualizable Compounds for stacks is impossible because the trap mechanism itself depends on the availability of the stack. If the processor implements TLB lookups in hardware (like x86) then the use of Virtualizable Compounds for stack areas is possible.

Virtualizable Compound Pages allow to optimize two typical usage scenarios in the kernel:

4.1 Avoid vmalloc

The use of Virtualizable Compound Pages allows the reduction of the use of vmalloc'ed memory. If a Virtualizable Compound is used instead of vmalloc then the page allocator will typically be able to provide a contiguous physical memory area. No vmalloc is necessary

unless memory is significantly fragmented and the antifragmentation measures have not produced enough linear memory. Therefore overall vmalloc use of the kernel is reduced. The need to go through a page table can be avoided and access to memory becomes more effective.

4.2 Fallback for higher order allocations

The other use of Virtual Compounds is to avoid higher order allocations that may fail. If higher order allocation requests are converted to Virtual Compounds then the kernel code can transparently handle situations in which memory is fragmented and no higher order pages are available. A typical use case are large buffers and stacks (would e.g. allow the use of significantly larger stack areas than currently possible.)

5 Variable order slab caches

It is easy to make slab allocations use various sizes of pages if the maximum number of objects is stored with each page. A variable order slab cache therefore does not need virtual mappings like provided through Virtualizable Compound Pages. Slab allocations can then be tuned to use more or less large pages depending on their availability. Allocation sizes can be cut back to lower sizes if memory fragmentation demands this. Allocations of large units that fail can be retried with a smaller allocation unit. This means that in extreme cases the effectiveness of the antifragmentation and defragmentation methods determines the extent to which large allocation units can be used. Therefore the speed and the locality of slab object allocations may depend on effective defragmentation.

The size of the allocation units also increases the likelihood that slab objects are allocated near one another. The resulting object locality reduces the TLB pressure commonly coming from pointer chasing because objects are distributed all over memory. An optimal configuration can be obtained if the allocation unit is made to fit the TLB size used for the kernel data segment. On x86_64 this is 2 megabytes, so if the allocation unit is set to 2 megabytes then most objects taken out from a given slab on one processor will come from the same 2 megabyte area that can be covered with a single TLB entry.

A larger allocation unit is particularly useful for slab caches with larger object sizes (1–3 kilobyte objects)

because a larger allocation allows more effective placement with less wasted memory. Objects may not fit well into smaller sized allocations. In addition to more efficient placement, allocation requests are also able to use the fast path for a higher percentage of allocations. As a result, calls to the page allocator become less frequent.

The allocation using varying page sizes puts more stress on the antifragmentation and defragmentation methods of the page allocator. If a high number of allocations fail and requires a retry with a smaller allocation size then this method is not effective and it would be better to switch back to smaller allocation units.

6 Larger I/O buffers

One of the limitations under which Linux file systems suffer is that I/O must be managed in 4KB chunks because the maximum buffer size is constrained by the page size of the operating system. This size is 4KB on x86 and as a result file systems are basically on their own if they want to manage data in larger chunks of memory. Some file systems implement a layer that allows the management of larger buffers (as in XFS) using virtualized mapping which means managing a list of pages for each larger buffer. The additional overhead reduces the potential performance gain and results in potentially non localized memory accesses.

The small I/O buffers also limit the amount of contiguous I/O that can be reliably submitted via DMA transfers to devices. Small 4KB chunks of memory require the management of large scatter gather lists which may be a limiting factor in the I/O throughput of a device.

Some file systems (like the ext file systems) are limited in the size of the volumes they currently support because they track allocations in page size chunks. If the chunk size can be increased through the use of larger I/O buffers then the size of volumes can be increased. The meta data that has to be managed for a given volume size is reduced significantly which accelerates the operation of the file system. The scaling possible with larger buffers can breathe new life into old file systems that can now overcome their size and performance limitations.

Large page support requires modification to the way that page size is handled in functions that provide page cache operations. The page size is stored in the mapping structure that exists for each opened file in the system. The

mapping structure must then be consulted during each page cache operation to determine the block size to use for a particular operation. The page size or buffer size can then be configured dynamically for each open file.

However, the use of large pages should not change the user API. In particular `mmap` semantics that are exposed to user space should not change. It needs to be possible to continue the mapping in 4KB chunks. This becomes possible if we allow mapping of 4KB segments of larger pages into an address space. The semantics of `mmap` are then preserved. One has to realize though that state for multiple of these 4KB chunks is kept in a single page struct. Only the large page as a whole can be dirtied or locked. Write and read operations must be performed using the block size set in the mapping.

Typically large pages up to 64KB can be supported by Linux file systems since the file systems already support platforms (IA64, Powerpc) that allow a 64KB base page size. The file system meta data structures are therefore already prepared to support block sizes up to 64KB. Support for larger sizes will require modifications to the file systems. The ability to set the page size per mapping may allow the design of entirely new file systems with support for a block size that can be configured per directory or per file.

The changes necessary for large page support are typically transparent for file systems. The `set_blocksize()` function will allow setting larger sizes than 4KB pages which will affect the raw block device.

Fallback occurs using Virtualizable Compound Pages. File systems can access the buffer data via linear access through the page tables in the fallback case. However devices may have to check if a virtually contiguous page is passed to the driver and then set up DMA with a scatter gather list of the physical pages that constitute the Virtualized Compound Page.

7 Memory Fragmentation

Memory fragmentation has been an issue for a long time for the Linux kernel. Over time the 4KB pages used for processes tend to get randomly distributed over memory which also has the effect of making allocations of large contiguous chunks of memory impossible. There is no problem as long as only 4KB page allocations are performed because memory of the same size is freed

and allocated. However, larger allocations than 4KB have always been performed for the stacks on x86 (8KB, so two contiguous pages are required) and for certain slab caches where objects would have caused too much memory wastage if they would have been placed in a 4KB page. The risk of failing such an allocation was judged to be negligible. If such an 8KB allocation cannot be satisfied then the page allocator will continue reclaiming until two consecutive pages are available.

The larger the chunk of memory becomes, the larger the risk becomes that an allocation cannot be satisfied. The page allocator will typically attempt to continue page reclaim in order to generate contiguous pages for allocations up to order 3 (32KB.) Even larger allocations will fail after a single reclaim pass that failed to generate a sufficiently sized page.

7.1 Antifragmentation Measures

Antifragmentation measures avoid fragmentation by classifying the allocation according to object lifetimes and reclaimability. One result of antifragmentation measures is that reclaimable pages are allocated in special memory areas. A series of neighboring pages can then be reclaimed to obtain large contiguous regions. So there is some level of guarantee that reclaim will be able to open up contiguous areas of memory because there are no unreclaimable pages in the way.

Antifragmentation measures were added to support allocation of huge pages even after the system has been running for awhile and after system memory has become fragmented. Huge pages are becoming more important for applications since they allow the localization of memory accesses and the reduction of TLB pressure by the use of a single TLB entry for 2MB of memory. Huge pages are a crude way to reach optimal speed of a machine as long as we have no large page support in the VM.

Problems still exist with allocations that are marked *unmovable*. These allocations are mostly page tables and certain slab allocations. The situation could be improved by making page table pages *movable*.⁷ The slab defrag functionality allows making slab objects *movable* but a support function must be provided for each slab cache to provide such functionality for each slab. A significant reduction of the number of *unmovable* allocations would be possible with these two measures.

⁷See Ross Biro's work on relocatable page table pages.

Somewhat fewer problems exist with allocations marked *reclaimable*. Reclaimable allocations are mainly used for slab allocations that can be reclaimed using shrinkers. The slab defrag measures add methods to these slabs that allow the targeted reclaim of objects in these caches. So these are already movable to some extend and the movability of these objects will increase as the reclaim methods for reclaimable slabs mature.

Antifragmentation measures cannot guarantee the availability of larger allocations. Antifragmentation measures only increase the likelihood that a large allocation will be successful. In the worst case the situation can degenerate into a state in which the categorization of allocations fails. Antifragmentation will never move pages but only sort the allocations according to its reclaimability which means that antifragmentation can be supported with minimal overhead.

7.2 Defragmentation

Defragmentation measures move pages or target specific pages that are in the way of generating a large contiguous section of memory. Defragmentation therefore involves more overhead than antifragmentation measures.

The one defragmentation method currently implemented in the kernel is *lumpy reclaim*. Lumpy reclaim works with *movable* pages, during reclaim we check if the neighboring pages could be freed. The freeing of adjacent pages then allows the merging of free pages to large contiguous chunks that could be used for large page allocations for the page cache, etc. However, lumpy reclaim does not apply to *unmovable* allocations and *reclaimable* allocations. Some additional work could make lumpy reclaim like methods work for *reclaimable* allocations.

Mel Gorman has a patch set that implements full fledged defragmentation. Defragmentation has much in common with memory hot plug. In both cases an area of memory is scanned and memory is then moved elsewhere. We could have a defragmentation solution in the kernel if we wanted to or needed to have such support.

8 The need for a better page allocator

The development of new functionality in the page allocator has been slow since the merge of the antifragmentation measures which was a controversial decision that took years to make.

There are a number of known problems:

8.1 Slow 4KB page allocations

The current buffering mechanism for 4KB pages (which one would expect to be of superior speed given the importance of 4KB allocations to the VM) is suffering from bloat and is inferior to the allocation speed of the slab allocators by some orders of magnitude. The result is that 4KB allocations frequently use the slab allocators instead of the page allocator. Various subsystems compensate by having their own buffering schemes to avoid the page allocator. All of that code could be avoided if the page allocator fast path could be made competitive in performance to what the slab allocators can do.

Ironically 4KB page allocations are often about 5% slower (uncontended case) than 8KB sized allocations. 8KB allocations bypass the 4KB buffering mechanism and therefore can avoid the list management overhead. Performance wise it seems to be best for a subsystem to allocate a large chunk of memory from the page allocator and then cut it into 4KB pieces on its own.

8.2 Issues with lock contention from multiple processors

Higher order allocations have a disadvantage: Access to the buddy free lists requires taking a zone lock which is—for most systems—a global lock. So multiple processors cannot simultaneously allocate memory from the page allocator. For 4KB sizes we have a buffering mechanism that avoids the locking (but creates overhead that hurts elsewhere.)

If multiple processors allocate memory continuously from the page allocator then we may end up with bouncing cache lines for the zone locks. This contention can even be observed with 4KB allocations if they are frequent because even the 4KB buffering scheme needs to go to the free lists once in a while to check out a new batch of pages.

8.3 More effective support for order N allocations

If the page allocator is presented with varying orders of allocations then it would be best if these would be satisfied from several different areas. If allocations of the same order came from the same memory area then

fragmentation would be reduced. Such a scheme is an extension of the antifragmentation method of sorting the allocations according to their lifetime. We would also sort them by size.

8.4 Scaling memory reclaim

One important aspect of larger page support is that it addresses the reclaim issue. If the pages on the reclaim lists have a larger size then there are fewer of these pages for a given amount of memory. The number of page structs that have to be processed is reduced and therefore reclaim works in a more effective way.

Reclaim is currently problematic on a multitude of platforms. Even desktop loads can start to suffer from reclaim scaling if applications are pushed into heavy reclaim. Swapping of large applications can make the system feel sluggish permanently. One wonders if it would not be better to simply fail if there is not enough memory rather than have the system become so sluggish that it takes a long time even if one attempts to simply reboot the system to get rid of the memory reclaim problems.⁸ In the HPC area it is already fairly common to abort an application if heavy reclaim occurs because the applications becomes unacceptably slow.

9 Transparent Huge and Giant page support

Support for varying sizes of pages for the page cache would allow transparent support for huge pages with minimal effort. Most of the page cache functions could be used directly by the huge page subsystem. The VM could be optimized to install PMDs instead of PTEs if the PTEs would fill the complete page table page at the lowest layer.

Ultimately it would be possible to get rid of the current huge page support. A small skeleton could be retained for backward compatibility. Having transparent huge page support would clean up special casing in the VM and make it easy for applications to use huge pages for various purposes without the use of special libraries.

Giant pages are 1GB sized mappings that are currently only supported by the most recent AMD processors. Transparent huge page support could be extended to

⁸The problem is in no way unique to Linux

also support the 1GB PUDs that these processors provide without the need to add yet another subsystem with special reservations. 1GB support would be a way to effectively manage memory for applications that may use several terabytes of memory.

10 Conclusion

Memory sizes are going to continue to increase, while processor speeds will continue to not make much headway. Further parallelization will occur by processor manufacturers increasing symmetric (multi core) and asymmetric (coprocessors) parallelism on the die.

Concurrency issues will therefore continue to dominate the development of operating systems. It is likely that we will see a ratio of over 4GB of memory per core. A single processor may have to handle about 1 million pages for reclaim or for I/O if we stay with the current scheme of handling memory in the VM. We will have to deal with this situation in some way. Either we need to develop ways to handle bazillions of pages or we need to reduce their number. However, optimal performance will only be reached through an effective reduction of the number of entities that the kernel has to handle.

The trend to processor specialization will continue since binding a task to a processor will allow effective use of the CPU caches and speed the operation of actions necessary repeatedly. This means that limiting I/O submission for a given cached dataset to a few processors makes sense. Also it may be useful to dedicate certain processors to the operating system for reclaim, defragmentation and similar memory intensive operations that would contaminate the caches of other processors executing mostly in user context.

11 References

References

- [1] Brim, Michael J. & James D. Speirs, *The Processor-Memory Gap: Current and Future Memory Architectures*. 2002. <http://pages.cs.wisc.edu/~mjbrim/personal/classes/752/report.ps>.
- [2] Mahapatra, Nihar R. & Balakrishna Venkatrao, “The Processor-Memory Bottleneck: Problems and Solutions.” *Crossroads*, Volume 6, Issue 3es. ACM: New York, 1999.

- [3] Marathe, Jaydeep P. *METRICS: Tracking Memory Bottlenecks via Binary Rewriting*. Master Thesis: North Carolina University, 2003. <http://www.lib.ncsu.edu/theses/available/etd-07132003-161530/unrestricted/etd.pdf>.
- [4] “Optoelectronic Integration Overcoming Processor Bottlenecks” in *Science Daily*, August 4th, 2005. <http://www.sciencedaily.com/releases/2005/08/050804053723.htm>.
- [5] Sutter, Herb, “The Free Lunch is Over: A Fundamental Turn toward Concurrency in Software.” *Dr. Dobb's Journal*, (30)3, March 2005.
- [6] Bokar, Shekhar, Pradeep Dubey, Kevin Kahn, David Kuck, Hans Mulder, Steve Pawlowski and Justin Rattner ”Platform 2015: Intel Processor and Platform Evolution for the Next Decade.” *Technology Intel Magazine*, Intel Corporation: March 2005.