Proceedings of the Linux Symposium

Volume Two

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

Contents

Where Linux Kernel Documentation Hides

Rob Landley
Impact Linux
`rob@landley.net`

Abstract

Last year my day job was documenting the Linux kernel. It seemed like a simple task: read through the kernel's Documentation directory, fill in the holes, and make sure it all stays up to date. In reality, I found most kernel documentation lives in web pages, magazine articles, online books, blog entires, wikis, conference papers, videos, standards, man pages, list archives, commit messages, and more. The real problem is editorial: only after finding and indexing the existing documentation can we figure out whether it's up to date and what's missing.

In this talk I'll list the documentation sources I found, show my attempts at organizing them (on `http://kernel.org/doc`), and explain what would be necessary to really get this issue under control.

1 Introduction

In 2007 the Linux Foundation awarded me a fellowship to deal with the ongoing lack of good Linux kernel documentation. Unfortunately, a good problem statement isn't necessarily a good plan of attack. I spent most of the next seven months just trying to figure out what "fixing it" actually meant, and how to go about it. The results may be found at `http://kernel.org/doc`.

My first big surprise was that the kernel's Documentation/ directory is just the tip of the iceberg. Most kernel documentation lives in web pages, magazine articles, online books, blog entries, wikis, conference papers, audio and video recordings of talks, standards, man pages, list archives, commit messages, and more. The real problem isn't a LACK of documentation, it's that no human being can ever hope to read more than a tiny fraction of what's said, written, and recorded about the kernel on a daily basis. Merging all this raw data into the kernel tarball would be like trying to burn the internet to CD.

Coping with such an enormous slush pile is fundamentally an editorial task. Only after finding and indexing the existing mass of documentation can anyone figure out whether what's out there for a given topic is complete and up-to-date. You can't fill in the holes without first knowing where they are.

This paper is not an attempt to summarize seven months of reading about why UTF-8 is a good internationalization format or how to load firmware out of initramfs for a statically linked device driver. It's about turning the dark matter of kernel documentation into something you can browse.

2 The editorial task

Google is great at finding things, but it doesn't tell you what to search for. Google is not a reference work that allows one to see available topics and home in on an area of interest, moving from more general to more specific. But there's more to teaching someone English than handing them a dictionary: a good reference work is not necessarily a good tutorial. Adding a reference index to a tutorial is fairly easy, turning a reference into a tutorial is harder. But creating a reference from scratch is also easier than creating a tutorial from scratch, a reference can be little more than a collection of sorted links, while a tutorial has to make sense.

Indexing the web's Linux kernel documentation just to provide a comprehensive reference is a huge undertaking. Even keeping an index up to date *after* its creation would be a project on par with any other major kernel subsystem. But without knowing here the holes are, writing new documentation to try to fill in those holes tends to reinvent the wheel.

In the absence of an obvious place to go on the web to find the most up-to-date existing documentation, newly created documentation tends to be repetitive and overlapping. Half-hearted attempts to collate what's available into a single new comprehensive version often just add one more variant to the pile. A well-meaning editor who isn't already an expert on a given topic probably won't create a new category killer document attracting patches instead of competition. If they didn't feel like contributing to someone else's existing document, why would the next well-meaning editor be different?

More to the point, author and editor are different jobs. Researching and writing new documentation isn't really an editorial task. An editor collects and organizes the submissions of others. A real editor spends most of their time wading through a slush pile and saying "no" to most of it, or saying "no" to things their assistant editors pass up to them.

If this sounds familiar, it's because fighting off sturgeon's law is what editors do. Open source project maintainers perform an editorial job, publishing regular editions of source code anthologies. Putting together Linux distributions is another layer of editorial filtering. Open source developers are already familiar with this process. Applying it to documentation, providing a brief summary and a pile of links to existing documents is more effective than trying to become an expert on every single topic, and has the advantage of not making the clutter any *worse*.

The other big editorial problem is keeping documentation up to date. When code is a living thing, its documentation must also be. The best documentation about the innards of the 2.4 kernel is only passably accurate about early 2.6, and in many ways the first 2.6 release has more in common with 2.4 than with 2.6.25. But the "many eyeballs" effect of open source can be diluted by having many targets. In a maze of twisty documents, all different, fixes that don't naturally funnel to a central integration and redistribution point get eaten by a grue.

3 Why won't it all fit in the kernel tarball?

It took me a while to realize the editorial nature of the kernel documentation problem, and that it was not primarily a question of new development. The obvious place to start when looking for Linux kernel documentation may be Google, but the next most obvious place is the Documentation directory in the kernel source tarball. And that comes with some enormous built-in assumptions.

The kernel tarball is the central repository for kernel source code, so it's easy to assume that Documentation/ is the central repository for all kernel documentation, or could easily be turned into such. This mistaken assumption cost me about 3 months.

First of all, the Documentation directory isn't even the only significant source of documentation within the kernel tarball itself. The kerneldoc entries in the source code (used by `make htmldocs`) and the kconfig help entries (used by the help option of `make menuconfig`) are each significant and completely separate sources of documentation. The files in Documentation/ seldom if ever refer to htmldocs or menuconfig help, and those seldom refer back to it (or to each other).

This other information cannot easily be migrated to the Documentation directory. The other sources of documentation in the kernel source are usually located near the things they document, to benefit from locality of reference. There's a reason they live where they do, as do over two dozen README files in the source code, the output of `make help`, references to IETF RFC documents in source comments, and so on.

In addition, the data formats are different. Documentation/ consists primarily of flat text files, htmldocs uses structured source code comments to generate docbook (and from that HTML or PDF output), and kconfig is in its own format which has dedicated viewer programs (such as menuconfig).

None of these is really an obvious choice for indexing the others. The flat text of Documentation/ does not lend itself to linking out the way HTML does, so at first glance htmldocs seems a better choice for an index. But the format of htmldocs is highly constrained by its origins as structured source comments; it's designed to do what it's currently doing and not much else. As a potential index, the kconfig help entries have both sets of disadvantages; they're flat text without hyperlinks and they're highly structured for a specific purpose.

On a second look, the Documentation directory seems the least bad choice for indexing the other documentation content in the kernel tarball, so it's worth a closer look here.

4 Organized based on where passing strangers put things down last

Documentation/ does not compile, give warnings, or break the build. It cannot easily be profiled, benchmarked, or regression tested. Because of this, the normal kernel build process doesn't naturally organize it very well. Here are a few of the files in the top level Documentation directory of the 2.6.25 kernel:

- **IRQ.txt**: Introductory file answering the question, "What is an IRQ?"
- **unicode.txt**: a standards document mirrored from lanana.org.
- **stallion.txt**: documentation for an unmaintained multiport serial card, last updated in 1999. (see also sx.txt, riscom8.txt, computone.txt…)
- **unshare.txt**: just under 300 lines of documentation about a single system call, unshare(2).
- **cli-sti-removal.txt**: Guide for migrating away from cli/sti locking, circa 2.5.28.
- **feature-removal-schedule.txt**: an important, regularly updated file about ways the Linux kernel plans to break binary compatability without necessarily involving sysfs.
- **zorro.txt**: Documentation for the Amiga "Zorro" bus.
- **spinlock.txt**: A post Linus made to the linux-kernel mailing list back in 1998 about spinlocks, with some almost intelligible notes at the top about which portions of the original message are obsolete or deprecated.
- **mono.txt** and **java.txt**: instructions on how to configure BINFMT_MISC to run Microsoft and Sun's bytecode languages. (No wine.txt to do the same for Windows binaries, though.)
- **README.cycladesZ**: file containing a URL to firmware for the Cyclades-Z card. (It does not say what a Cyclades-Z card is.)
- **logo.gif** and **logo.txt**: the Tux graphic, and a URL to Larry Ewing's page on it.
- **email-clients.txt**: Notes about sending unmangled patches to the linux kernel mailing list with alpine, evolution, kmail, lotus notes, mutt, pine, sylpheed, thunderbird, and tkrat.
- **IPMI.txt**: docs about the Intelligent Management Platform Interface driver. It's over 600 lines long but links to an Intel website in the introduction because "IPMI is a big subject and I can't cover it all here!"
- **dontdiff**: data file for use with diff's "-X" option.
- **tty.txt**: This file starts "The Lockronomicon: Your guide to the ancient and twisted locking policies of the tty layer and the warped logic behind them. Beware all ye who read on."

This is a small subset of the ~140 files at the top level, and doesn't include anything in the ~75 different subdirectories for busses, architectures, foreign language translations, subsystems, and so on.

A token attempt at organizing Documentation/ can be found in the 00-INDEX files in each subdirectory, containing a one line description of each file (example: "device-mapper/ - directory with info on Device Mapper."). Some directories have this file, some don't. Some files are listed, some aren't.

00-INDEX is better than nothing, but it mirrors a filesystem hierarchy without symlinks. A file like filesystems/ramfs-rootfs-initramfs.txt belongs both in "filesystems" and in "early-userspace", but it has to pick one.

Even the perennial question "where do I start?" has at least three answers in the kernel tarball's existing documentation: the oldest and in some ways still the best is the "README" file at the top of the kernel (not in Documentation), the next oldest is Documentation/kernel-docs.txt, and the newest is Documentation/HOWTO. None of them really provide a good introduction to the kernel's source code. For that I recommend Linux Kernel 2.4 Internals (`http://www.moses.uklinux.net/patches/lki.html`), which is woefully out of date and x86-specific but still the best I've found. It is not in the kernel tarball. (Neither is `http://en.wikibooks.org/wiki/Inside_Linux_Kernel` which seems to be another unrelated attempt at doing the same thing. There are plenty more out there.)

It is possible to clean up Documentation/ (albeit a fairly large undertaking), and I pushed a few patches to this effect (which were generally greeted with a strange combination of indifference and bikeshedding). It's also possible to convert the Documentation directory to HTML (an even larger project, of dubious value). But ultimately, there's a larger philosophical problem.

Documentation/ is based on the assumption that everything of interest will be merged into the kernel tarball. It already copies standards documents and HOWTOs with defined upstream locations, because having potentially out-of-date copies in the kernel tarball is considered superior to having a single cannonical location for this information out on the web. The philosophy of Documentation/ is the same as for code: if out of tree drivers are bad, out of tree documentation must also be bad.

This is a difficult philosophy to apply to indexing documentation that lives on the web. The web has many formats (from pdf to flash) and Documentation has one. Web content has many licenses, the kernel is GPLv2 only. How does one apply CodingStyle to Linus Torvalds' Google video about the origins of git? The kernel source tarball is currently just under 50 megabytes, the mp3 audio recordings of OLS talks just for the year 2000 total a little under 90 megabytes.

Unfortunately, the belief that the internet can or should be distilled into Documentation/ is pervasive among kernel developers. My interview process for the Linux Foundation fellowship consisted of writing Documentation/rbtree.txt. Before doing so I pointed out that there was already an excellent article on Red Black Trees in the Linux Weekly News kernel archives, and another article about it on Wikipedia. But they weren't in the kernel tarball and thus (I was told) they didn't count, so I reinvented the wheel to get the job. Three months later, I regretted adding to fragmentation.

Exporting kernel tarball docs to `http://kernel.org/doc`

The kernel-centric Documentation/ in the kernel tarball created a reciprocal problem: Not only did Documentation/ suck at indexing the web, but the web wasn't doing that great at indexing the kernel's built-in documentation either.

I personally encountered this effect in early 2007, when my Google search for ext2 filesystem format documentation which didn't bring up Documentation/filesystems/ext2.txt in the first five pages of hits. I didn't even notice that file until a month later (after all if its Google rank sucks how good can it be), because there was no cannonical uncompressed location at which to find it on the web, and things like gitweb or the most recent release tarball were too transient to work up much of a ranking for any specific version. (Similarly, there was no standard web location for the current htmldocs, despite those being HTML!)

So the first well-defined problem I needed to tackle was exporting the documentation already in the kernel tarball somewhere Google could find it. I requested a page on kernel.org, and received `http://kernel.org/doc`. I copied the kernel's Documentation/* to `http://kernel.org/doc/Documentation`, set up the `make htmldocs` tools on my laptop, and posted the results to `http://kernel.org/doc/htmldocs`. Then I created a script to periodically update this documentation from the kernel repository (`http://kernel.org/hg/linux-2.6`) and checked this script into a new mercurial repository on my website (`http://landley.net/hg/kdocs`).

Over the months that followed, I improved my export script to harvest and export much more information from the kernel source. (See `http://landley.net/hg/kdocs/file/tip/make/` for the scripts that do all this. If you check out the mercurial repository and run "make/make.sh --long" it'll try to reproduce this directory on your machine. You need mercurial, wget, pdftk, xmlto, and probably some other stuff.)

`http://kernel.org/doc/Documentation/`

The way a web server shows a directory full of files isn't very informative, so I wrote a Python script to turn the 00-INDEX files in each Documentation subdirectory into a simple HTML index. This had the unfortunate side effect of hiding files in any directory with a 00-INDEX that doesn't list everything, so I wrote the script `make/doclinkcheck.py` which compares the generated HTML indexes against the contents of the directories and shows 404 errors and extra files. I sent lots of 00-INDEX patches to linux-kernel trying to fill in some of the gaps, but as of April 2008 doclinkcheck.py shows about 650 files still improperly indexed.

`http://kernel.org/doc/htmldocs`

On the htmldocs front, the top level book index created by `make htmldocs` was unfortunate, so I had my script create a better one. I also wrote a quick script to create "one big html file" versions of each "book", and used the old trick that if "deviceiobook.html" is the one big ("nochunks") version, "deviceiobook/" at the same location is a directory containing the many small pages ("chunks") version. The top level index lists both versions.

`http://kernel.org/doc/menuconfig/`

The kconfig help text is the third big source of kernel documentation, and the only human readable documentation on several topics, so I wrote `make/menuconfig2html.py` to parse the kconfig source files and produce HTML versions of the help text.

The resulting web pages organize information the same way menuconfig does. The first page selects architecture, the later pages show config symbols with one line descriptions. The symbol names link to help text extracted from the appropriate Kconfig file.

I attempted to organize the result to reduce duplication to produce a "single point of truth" for Google to find easily, and hopefully rank high. There are several index pages (since menuconfig shows different menus for different architectures), but each Kconfig file is translated to a single page of help text, and the indexes link to the same translated Kconfig files. Each HTML file is named after the source file it's generated from.

`http://kernel.org/doc/rfc-linux.html`

Many comments in the Linux kernel source code reference Internet Engineering Task Force Request For Comments (IETF RFC) standards documents, which live at `http://tools.ietf.org/html`. I put together a script to grep the source code for RFC mentions, and put a link to that RFC together with links to each source file that mentions it. (It seemed like a useful thing to do at the time.)

`http://kernel.org/doc/readme`

The kernel source contains over two dozen README files outside of the Documentation directory. My export scripts collect them together into one directory.

`http://kernel.org/doc/makehelp.txt`

If you type `make help`, kbuild emits a page of documentation, and my export scripts put that on the web too.

5 Indexing kernel documentation on the internet

With the kernel's existing internal documentation exported to the web, the next task was adding documentation from the net. Mining the internet for Linux kernel documentation and trying to put it in some coherent order is a huge undertaking, and I barely scratched the surface. What I did find was overwhelming, and had some common characteristics.

There are lots of existing indexes of documentation. Linux Weekly News has an index of all the kernel articles it has published over the years (at `http://lwn.net/Kernel/Index/`). Linux Journal magazine has online archives going back to its first issue (`http://www.linuxjournal.com/magazine`). The free online Linux Device Drivers book has an index (`http://lwn.net/Kernel/LDD3/`). Kernel Traffic has an index (`http://kerneltraffic.org/kernel-traffic/archives.html`). My own mirror of the OLS papers has an index (`http://kernel.org/doc/ols`).

The common theme of these indexes (and many more like them) is that they index only their own local content, because the aim of most of these repositories is to create new local documentation rather than index existing external documents. These indexes are valuable, but collating them together is a nontrivial task. When indexed by topic they don't necessarily use the same topic names, while other indexes are only by date. And this glosses over any actual overlap in the article contents.

Some indexes (such as Documentation/kernel-docs.txt) do link to a number of external sources. Others (such as the Linux Documentation Project `http://tldp.org`) attempt to organize existing documentation by mirroring it. These are valuable resources, but most tend to give up after a certain point, either finding natural boundaries or realizing the enormity of the task of indexing the entire internet and deciding against it. Once promising indexing efforts, such as the Open Source Writer's Group (at www.oswg.org), the Linux Kernel Documentation Project (`http://www.nongnu.org/lkdp/`), and on "The Linux Kernel: The Book" (`http://kernelbook.sourceforge.net/`), stalled and died.

The Linux Documentation project (`http://tldp.org`) is the largest and most well known of the existing documentation collection projects, but its primary focus is userspace and its method is mirroring. Its efforts go to collecting and mirroring as many documents as possible, not into cross-referencing or deep linking into them.

6 To mirror or not to mirror

The decision whether or not to mirror web resources is tricky, and has no good answer. On the one hand, mirrors get out of synch with their original sources, take up potentially gigabytes of storage, dilute the Google page rank of the original source, raise licensing concerns, often have an inferior user interface to an original page with a style sheet, and so on. On the other hand, resources that aren't mirrored can go 404 and vanish from the net.

The wayback machine at archive.org aims to preserve the entire internet for posterity, and for the most part I chose to rely on that project to preserve information rather than mirroring it. Some things, such as the OLS papers, I chose to mirror in order to present them in a different format (or at a different granularity) than the source material, or because (like the 2006 OLS slides) the sources were already decaying after a relatively short period of time. But where original sources were established and stable, I linked directly to them. (Mirroring the Linux Weekly News archives would be silly.)

7 On old material and editorial ignorance

Deciding whether or not a reference is obsolete requires domain expertise. This is something that an editor often won't have much of, nor time to acquire it. This is another facet of the author vs editor dichotomy.

An editor must accept that there is material they don't understand, and that attempting to become an expert on every topic is not a viable strategy. New content is generated faster than any human being can absorb it without specializing.

As an editor I found myself fielding documentation that I did not have more than the most superficial understanding of. It was not possible for me to improve this documentation, tell if it's up to date, evaluate its accuracy or thoroughness. (In extreme cases, such as foriegn translations, I couldn't even *read* it.)

What can an editor do about this? Pile it up in a heap. Summarize the topic as best they can (which may just be a title statement cribbed from somewhere), link to the documentation they found in whatever order seems appropriate (if all else fails, there's always alphabetical), and wait for people who *do* understand it to complain. If the editor's brief summary plus pile of links does attract relevant domain experts: delegate to them.

In cases where I could tell that a reference was obsolete (such as devfs), I often wanted to link to it anyway. Why? Because it provides historical insight into the current design. "Here's how the kernel used to do things, and here's what was wrong with that approach." A relevant domain expert can avoid reinventing the wheel if they see what was learned from the previous approaches.

So noting the date of older resources in the index can be valuable, but excluding them just because they're old isn't. These days everyone takes for granted that Linux uses ELF executables, but they have to read old articles from 1995 (such as `http://www.linuxjournal.com/article/1139`) to learn why. (Or at least they did until recently, now there's `http://people.redhat.com/drepper/dsohowto.pdf`. Which provides better coverage of the topic? Since I haven't piled up enough links on that topic to worry about pruning them yet, I don't currently have to make that decision.)

8 What's out there?

Here's a quick survey of some of the more prominent kernel documentation sources out on the web. This is not an attempt to be exhaustive, the internet is too big for that.

linux-kernel mailing list archives

The Linux kernel mailing list is the main channel of discussion for Linux kernel developers. The early history of Linux started on usenet's comp.os.minix and moved to its own "linux-activists" mailing list (archived at `http://www.kclug.org/old_archives/linux-activists/` and with a few interesting early posts summarized at `http://landley.net/history/mirror/linux/1991.html` and `http://landley.net/history/mirror/linux/1992.html`). It then moved to `linux-kernel@vger.rutgers.edu`, and eventually to `vger.kernel.org` when the rutgers machine died.

Numerous archives of linux-kernel are available. The most well known is probably the one at `http://www.uwsg.iu.edu/hypermail/linux/kernel/`. But the sheer volume of this mailing list is such that few if any kernel developers actually read all of it, and the archives are overwhelming. Drinking from this firehose yields a poor signal to noise ratio; it is a valuable source of raw data, but extensive filtering and summarizing is required to extract useful documentation from it.

The linux-kernel mailing list is one of of almost a hundred mailing lists hosted on vger.kernel.org (see `http://vger.kernel.org/vger-lists.html`), and many other kernel-relevant mailing lists live on other servers. Although linux-kernel@vger.kernel.org is the big one, others provide plenty of relevant information.

During the course of the documentation fellowship, I posted regular status updates to linux-doc@vger.kernel.org. That list was mostly moribund, so the archives at `http://marc.info/?l=linux-doc` provide a relatively concise summary of my activities during the project.

`http://kernel-traffic.org/`

To understand the magnitude of the kernel documentation slush pile, ponder the fate of kerneltraffic.org. This popular, widely-read website provided weekly summaries of discussions on the linux kernel mailing list from January 2000 to November 2005. But eventually the volume overwhelmed editor Zack Brown, who brought the project to an end:

From `http://kerneltraffic.org/kernel-traffic/latest.html`

Kernel Traffic has become more and more difficult over the years. From an average of 5 megs of email per week in 1999, the Linux kernel mailing list has gradually increased its traffic to 13 megs per week in 2005. Condensing that into 50 or 100 K of summaries each week has started to take more time than I have to give.

Kernel Traffic was an extremely valuable resource due to the sheer volume of material it condensed, and its loss is still strongly felt. These days, most kernel developers consider it impossible for anyone to read all messages on linux-kernel, certainly not on a regular basis.

In 2007 I hired a research assistant named Mark Miller, in hopes of bringing Kernel Traffic up to date. He reproduced the existing site from its XML source files (see `http://mirell.org/kernel-traffic`), and experimentally summarized a few more weeks. The result was that summarizing each week of posts took him longer than a week, and the amount of expertise necessary to select and summarize interesting threads, plus the sheer number of hours required, made doing just this and nothing else a full time job for someone (such as Zach Brown) who is already a domain expert. The job was simply too big.

Linux Weekly News kernel page

The other systematic summarizer of the linux-kernel mailing list, and the only one to continue a regular publication schedule to this day, is the Linux Weekly News kernel page. Each week, Jonathan Corbet does excellent in-depth analysis of several kernel topics discussed on the list. Since 1999, this has resulted in several hundred individual articles.

The LWN Kernel Index page (`http://lwn.net/Kernel/Index/`) collects the individual articles and organizes them by topic: Race Conditions, Memory Management, Networking, and so on. Individual articles are linked from mulitple topics, as appropriate.

A second index, the LWN Kernel Page (`http://lwn.net/Kernel/`), links to article series such as Kernel Summit coverage, 2.6 API changes, and Ulrich Drepper's series on memory management.

The Linux Weekly News kernel page is published regularly, but it does not attempt to be as thorough as Kernel Traffic was. Kernel Traffic provided brief summaries of up to two dozen mailing list threads each week, while LWN Kernel coverage provides in depth articles about 3-5 topics in a given week. The two complemented each other well, and one is not a substitute for the other.

http://kerneltrap.org/

The Kernel Trap website cherry picks occasional interesting threads from the Linux Kernel Mailing List (among other sources) and reproduces them for interested readers. It falls somewhere between Linux Weekly News and Kernel Traffic in content, with an update frequency avereaging less than one article per day.

Kernel Trap takes little time to follow, and the material it highlights is consistently interesting. Some articles, such as "Decoding Oops" (http://kerneltrap.org/Linux/Decoding_Oops) every would-be kernel developer should read. But in terms of complete coverage of the Linux Kernel Mailing List, Kernel Trap is the next step down after Kernel Traffic and Linux Weekly News.

In addition to summarizing, KernelTrap also generates new content such as event coverage and interviews with prominent developers (see http://kerneltrap.org/features).

Ottawa Linux Symposium proceedings (http://kernel.org/doc/ols)

This conference has produced a wealth of high quality kernel documentation over the years. (So much so that despite many hours devoted to absorbing this material, I've personally never managed to read even half of it.)

Due to the high quality of the OLS papers, and my personal familiarity with them, I devoted significant effort to them. After confirming they were freely redistributable, I took the published PDF volumes and broke them up into individual papers (using the make/splitols.py script, which uses pdftk). This resulted in over 300 individual PDF files, mirrored on kernel.org.

The next step was to index them. For each year, I created an index file, such as the one at http://kernel.org/doc/ols/2002. For the first 25 papers of 2002, I read each one and wrote a brief summary of the contents of each paper, but this was surprisingly exhausting. After that, I just listed the title and authors of each paper. The collective index of the OLS papers links to audio and video recordings of panels, as well as the presentation slides for 2006 (mirrored locally on kernel.org due to their original index pointing into a number of 404 errors after less than a year).

The 2001 papers are no longer available from the OLS website,[1] but Linux Weekly news had copies mirrored. For 2000 I found audio recordings, but no papers. I discovered no material from 1999.

Other conferences

OLS used to be one of four main Linux technical conferences. The others were east and west coast versions of LinuxWorld Expo, and Atlanta Linux Showcase. I have the complete set of casette tapes of talks at the 2001 LinuxWorld Expo which I need to secure the rights to digitize and put online. The CD-ROM from ALS 1999 is at http://kernel.org/doc/als1999.

These days, the most important kernel conferences (and the only ones the project's maintainer still regularly attends) are linux.conf.au and the kernel summit. But interesting material can be found at The April 2008 Linux Foundation Summit in Austin, the Consumer Electronic Linux Forum's annual conference in Mountain View, and many others in Europe, Asia, Africa, Japan.... (See http://www.linuxjournal.com/xstatic/community/events, http://elinux.org/Events, and http://www.linuxcalendar.com for lists.)

These other conferences produce lots of material. Videos and presentation files from the 2007 O'Reilly Open Source Convention are at http://conferences.oreillynet.com/pub/w/58/presentations.html. Videos from the Embedded Linux Conference Europe 2007 are linked from http://lwn.net/Articles/266169/. The full text of 28 papers presented at the Ninth Real-time Linux Workshop (held in Linz, Austria November 2-3 2007) are up at http://linuxdevices.com/articles/AT4991083271.html. Things like the Linux Plumber's Conference (http://linuxplumbersconf.org) or Japan Regional Technical Jamboree #20 http://celinuxforum.org/node/82 produce more all the time.

Recently, Usenix released decades of existing material when it opened web access to its conference proceedings. The announcement is at http://blogs.usenix.org/2008/03/12/

[1] http://www.linuxsymposium.org

This doesn't even get into papers and talks presented at regional LUGs.

Man pages

The best existing documentation on the kernel's system calls is section 2 of the man-pages package. (There's some debate on whether man-pages should document the API exported by the kernel or the wrapped versions provided by glibc, but they're mostly the same.)

I was using Eric Raymond's Doclifter project to convert each new release of man-pages to docbook, and from there to HTML, but eventually the man-pages maintainer started doing his own HTML conversions, and put them on the web at `http://kernel.org/doc/man-pages/online_pages.html`.

This may fall under the heading of "don't ask questions, post errors". Michael Kerrisk had meant to put up html versions of the man pages for some time, but my doclifter conversions were probably horrible enough to bump it up on his todo list before too many outside sources linked to them. (I also sent ESR a few patches to doclifter, and converted his old RCS repository to mercurial so it could go up on the web. This got him started on the whole "source control" kick. Did I mention this project spawns endless tangents?)

Developer blogs and web pages

Prominent kernel developers often have web pages. Many of them (such as `valhenson.com`, `selenic.com/linux-tiny`, and `http://people.netfilter.org/rusty/unreliable-guides/` are documentation resources in their own right.

Many kernel developers also blog. The blog aggregator Kernel Planet `http://kernelplanet.org` does a reasonable job of collecting many developer blogs into a single page, where lots of excellent posts documenting obscure subjects float by... and scroll off the end again, unrecorded.

My own blog for 2007 `http://landley.net/notes-2007.html` documents a lot of my own struggle with the kernel documentation issue. (If you can dig those comments out from the noise about cats and food.)

Wikis and Wikipedia

Distributed, user generated content naturally scales in volume with the size of the userbase. Unfortunately, the editorial task of coordinating, filtering, and integrating the resulting material does not. Doing that takes work.

Drew Curtis, of the news aggregator Fark, recently spoke about "the wisdom of crowds" in an interview:

> We're the only news aggregator out there which is edited, which I think is the next step in social networks because right now everybody is talking about the wisdom of crowds, and all that--which is complete horse ****, and I think the next step is realizing that what crowds pick is pretty much pornography and Internet spam, and as a result you've got to have some editing involved there somewhere.

(Curtis went on to note that the record-holding top story of "Digg" had been puppies playing.)

Wikis are a perfect example of this. The Linux Kernel is the subject of dozens of wikis. Some random examples (with varying degrees of kernel focus) include rt.wiki.kernel.org, elinux.org, linux-mm.org, kernelnewbies.org, unix-kernel-wiki.wikidot.com/linux-kernel-wiki, linux-ntfs.org, wiki.linuxquestions.org, gentoo-wiki.com, wiki.ubuntu.com/KernelTeam, fedoraproject.org/wiki, slackwiki.org, wiki.debian.org, and so on.

The lack of integration leads to multiple wikis emerging even for individual Linux distributions. For example, SuSE has en.opensuse.org, susewiki.org, wiki.linuxquestions.org/wiki/SuSE, suseroot.com, wikipedia's pages on SuSE, the SuSE pages on wiki.kollab.org, linux.ittoolbox.com, www.linuxformat.co.uk/wiki…

The biggest wiki of all is wikipedia, which has hundreds of pages on topics related to the Linux kernel. Unfortunately, the way to find stuff in Wikipedia is to use Google. Wikipedia has both a reluctance to link to anything other than itself, and a general lack of indexing.

Wikipedia's Linux index `http://en.wikipedia.org/wiki/Wikipedia:WikiProject_Linux/index` is mostly oriented towards userspace, containing a single link to the `http://en.wikipedia.org/wiki/Linux_Kernel` page. That page does not attempt to index pages on numerous kernel-relevant topics (such as Red Black Trees or IPSec).

The related Wikibooks project has several pages devoted to the Linux kernel, of which `http://en.wikibooks.org/wiki/The_Linux_Kernel` functions as a reasonable index of external resources. It's actually quite nice, and a good source of further links for anyone interested in finding such. However, at the time of writing this paper, it contains exactly three links to wikipedia articles.

None of these wikis really focuses on indexing external content. Few have complete or well-organized indexes even of their own content. They throw data up in the air and leave sorting it up to Google.

Google Tech Talks

I made a small page linking to a few interesting Google Tech Talks (`http://kernel.org/doc/video.html`) but didn't manage to index even 1% of what's there. And that's just a single video series from one source in California, not a serious attempt to trawl Youtube for kernel content.

I broke down and wrote some

What can I say, I'm weak?

The most popular one hosted on kernel.org/doc would probably be the the Git Bisect HOWTO at `http://kernel.org/doc/local/git-quick.html` which provides just enough background for somebody to use git bisect without forcing them to learn about things like branches first.

Others (such as `Documentation/make/headers_install.txt`) went upstream.

(Don't ask about sysfs. It's a heisenberg system: attempting to document it changes the next release. Documenting it by examining its implementation is explicitly forbidden by its developers; you're supposed to read their minds. You think I'm joking…)

Online books

Linux Device Drivers (`http://lwn.net/Kernel/LDD3/`) is a complete education in the Linux kernel by itself. (I've read maybe 3 chapters.) Linux Kernel in a Nutshell (`http://www.kroah.com/lkn/`) is also online. "Linux Kernel 2.4 Internals" (`http://www.tldp.org/LDP/lki/`) remains an excellent if somewhat dated introduction, and the Linux Documentation Project has its own book called "The Linux Kernel" (`http://tldp.org/LDP/tlk/tlk.html`) based on the 2.0 source. faqs.org has The Linuxx Kernel HOWTO (`http://www.faqs.org/docs/Linux-HOWTO/Kernel-HOWTO.html`)

I got permission from Mel Gorman to mirror the published version of his book "Understanding the Linux Virtual Memory Manager" at `http://kernel.org/doc/gorman`. (It's an excellent resource, and I haven't made it past the introduction yet. It's on my todo list.)

Finding new stuff

Jonathan Corbet tracks changes to the Linux kernel on a Linux Weekly News page `http://lwn.net/Articles/2.6-kernel-api/` and in the Linux Weather Forecast `http://www.linux-foundation.org/en/Linux_Weather_Forecast`. Kernel newbies has its own pages `http://kernelnewbies.org/LinuxChanges` and `http://kernelnewbies.org/Linux26Changes`). The elinux.org website has its own version of the Linux Weather Forecast (`http://elinux.org/Technology_Watch_List`).

Many other websites (such as `http://www.linuxdevices.com`) track changes to their own areas of interest.

And so on

The kernel git archive has some very informative commit messages (browsable at `http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git`) going back to 2.6.12-rc2. Thomas Gleixner converted the old bitkeeper repository (covering 2.5.0 through 2.6.12-rc2) into git (browsable at `http://git.kernel.org/`

?p=linux/kernel/git/tglx/history.git). That's just as much an unsummarized firehose as the linux-kernel mailing list, and I never got around to even trying to deal with it.

On http://kernel.org/doc are links to free online copies of the Single Unix Specification version 3, the C99 standard, the ELF and DWARF specs, and more. Go there. Read it.

9 Organizing this heap

If I sound tired just listing all those resources, imagine what it's like trying to collate them.

My approach was to create a large topic index, using nested html "span" tags and a simple python script to create an index from that, and check the lot into a mercurial repository (http://landley.net/hg/kdocs). I could have used wiki software, but kernel.org dislikes active content for security reasons.

Another reason to avoid wiki software is that the public face of the index is HTML, thus the source format (http://kernel.org/doc/master.idx) should be as close to pure HTML as possible. I expected to receive and merge patches against the generated HTML, and those patches needed to apply to the source with as little work on my part as possible.

Once I had a decent topic index (based on examinations of Linux Device Drivers, Linux Kernel Internals, and so on), the next step was to go through the individual sub-indexes (such as the ones for Linux Weekly News kernel articles, Documentation/, htmldocs, kernel traffic, the Ottawa Linux Symposium papers, and so on) and slot in links to each resource as appropriate. Many resources needed a link from more than one topic, so this was an interative and extremely time consuming process.

I also attempted to summarize each topic briefly, in addition to providing a stack of links. The line between "writing new documentation" and indexing existing documentation was never more blurry than when doing that.

When prioritizing, I kept in mind the rate of churn. Every kernel release breaks out of tree drivers, to the point that even widely used patches such as squashfs or kgdb, applied by every major kernel vendor, are a pain to use with a current vanilla kernel. Documenting interfaces with an expected lifespan of 3 months is a Red Queen's race.

The interface between the kernel and userspace is the most stable part of the kernel, and has some of the best existing documentation. The easiest approach is to start there and work in.

10 Unfinished business

The file http://kernel.org/doc/pending/todoc.txt was my working todo list when the Linux Foundation decided to discontinue the documentation fellowship. After six months, they admitted they hadn't had a clear idea what "solving" the kernel documentation problem meant, and they were going to pull back and reconsider their options. They praised the work I'd done and gave me one more month to finish it up, but did not wish to continue funding it for the full year.

After seven months of drinking from the firehose, I was actually kind of happy to stop. As with the maintainer of Kernel Traffic: it was fun, but I was tired.

Tux meets Radar O'Reilly—Linux in military telecom

Grant Likely
Secret Lab
grant.likely@secretlab.ca

Shawn Bienert
General Dynamics Canada
shawn.bienert@gdcanada.com

Abstract

Military telecom systems have evolved from simple two-way radios to sophisticated LAN/WAN systems supporting voice, video and data traffic. Commercial technologies are being used to build these networks, but regular off the shelf equipment usually isn't able to survive the harsh environments of military field deployments.

This paper discusses the use of Linux in General Dynamics' vehicle mounted MESHnet communication system. We will discuss how Linux is used in the system to build ad-hoc networks and provide reliability for the soldiers who depend on it.

1 Introduction

To say that good communication is critical to the military is never an understatement. Wars have been won and lost on the basis of who had the most accurate and timely information. It is understandable then that the military takes its telecom gear very seriously.

Just like the private sector, in order to survive, the military has had to keep up with advances in technology. Current military communications networks carrying both voice and data traffic bear little resemblance to the original analog radio systems of the past.

In most cases, however, regular commercial equipment is not suitable for a military environment. Aside from the obvious fact that most equipment isn't painted green, the military puts high reliability expectations on its equipment. When your life depends on the proper operation of your equipment reliability tends to be an important concern. For example, tank drivers typically do not have a very good field of view from where they are located in the vehicle, and thus are dependent on directions provided by the commander via the intercom system to accurately direct the vehicle. An intercom failure can very quickly result in problems like running over a small car or driving through a building.

There is a natural conflict between reliability and the increasingly complex services required by military users. As complexity increases, so do the number of potential failure points which tends to reduce the stability and reliability of the entire system. As new technologies such as WiFi, VoIP and Ad-Hoc mobile networking are added, system engineers need to analyze the impact on the rest of the system to ensure overall reliability is maintained.

Within this environment, Linux and other Free and Open Source Software (FOSS) components are being used as building blocks for system design. As we discuss in this paper, FOSS is proving to be a useful tool for solving the conflicting requirements inherent in large military telecom system designs.

2 Typical Military Telecom Network

For illustration purposes, Figure 1 is an example of a typical military telecom deployment based on the next generation of General Dynamics Canada's MESHnet platform. MESHnet equipment provides managed network infrastructure for voice and data traffic within and between military vehicles and provides voice services which run on top of it. For example, a MESHnet user in a vehicle has an audio headset and control panel that give him intercom to his other crew mates, direct telephone service to other users, and access to two-way radio channels. He also has an Ethernet port for attaching his laptop or PDA to the network for configuration, status monitoring, email, and other services. In the background, GPS and other sensors attached to each vehicle use the radio service to automatically transmit position and status reports back to headquarters [1].

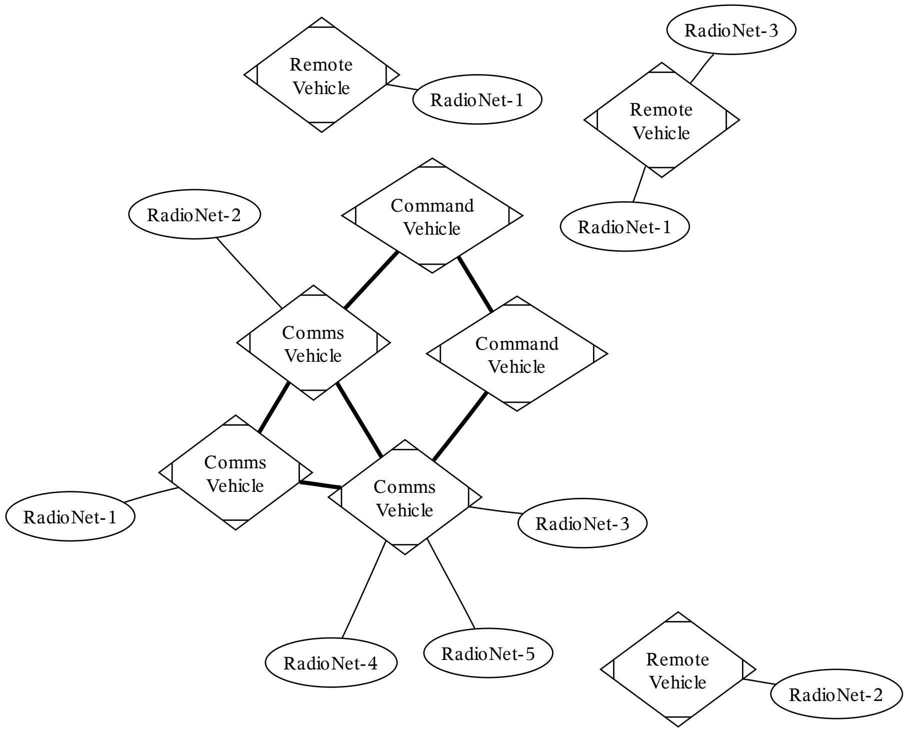


Figure 1: Example MESHnet Deployment with Headquarters and Remote Vehicles

2.1 Vehicle Platforms

At the heart of the MESHnet system are the vehicle installations. MESHnet equipped vehicles provide a user terminal and headset for each crew member in the vehicle, a set of interfaces to two-way radios, and a network router for wired and wireless connections to external equipment. The whole vehicle is wired together with an Ethernet LAN. Figure 2 shows an example vehicle harness which includes 2 radios and a GPS sensor. You'll notice that the Ethernet topology (shown by the bold lines) is a ring. The ring is for redundancy in the event of a cable or equipment failure.

The system is designed to immediately provide intercom service between crew members when the system is initially powered on. Each radio is detected and configured as an abstract *radio net*. Solders can select from a variety of radio nets for monitoring with a single key press and use their Push-to-Talk (PTT) key for transmitting.

Data equipment like Laptops, PDAs, and sensors can be connected to the system via Ethernet, USB, and serial ports, allowing the equipment to work together and providing access to digital data radios for low bandwidth network traffic back to headquarters.

2.2 Headquarters

When a battlefield headquarters is established, several vehicles are parked together and Ethernet is again used to connect them in a mesh topology.[1] The on-vehicle router keeps internal and external Ethernet segments separate. Services provided by each vehicle are bridged onto the inter-vehicle LAN so that user terminals have access to all radios and other equipment within the headquarters.

3 System Design Issues (and how Open Source can solve them)

In this section we discuss the various aspects of the communication system that need to be addressed by system

[1]Hence the name MESHnet.

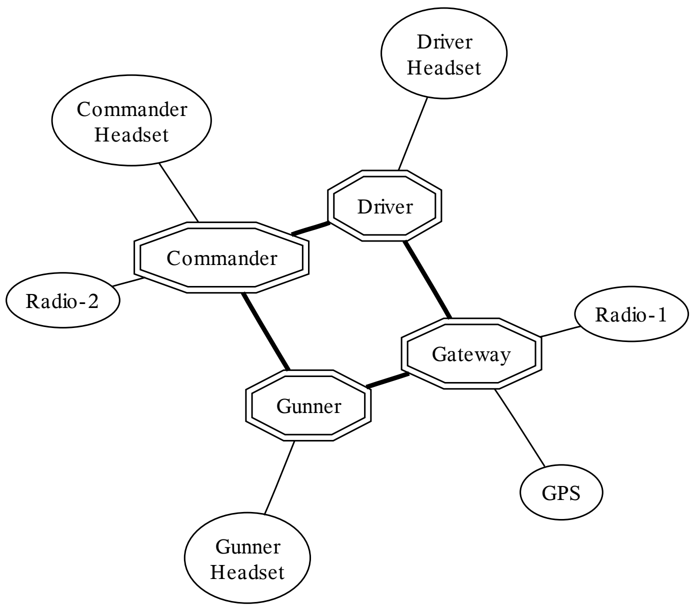


Figure 2: Example MESHnet Equipped Vehicle Installation

designers. Most of these issues bear close resemblance to issues also faced in the commercial environment, but there are some unique characteristics.

3.1 Open Architecture

Telecom systems have become so complex and varied over the years that it is no longer feasible for a single vendor to be capable of designing and supplying all equipment used in a tactical network. Many different vendors supply equipment which is expected to interoperate with the rest of the network. Military customers are also wary of solutions which lock them into a single vendors solution or which force the replacement of existing equipment. As such, there is significant pressure by military customers to use common interfaces wherever possible.

It is now taken for granted that Ethernet and 802.11 wireless are the common interconnect interfaces. The historical proprietary interfaces used in military networks are rapidly giving way to established commercial standards. Not only does it make interoperability easier, it also allows military equipment to use readily available commercial components which in turn reduces cost and complexity.

Additionally, the protocols used by tactical applications are rapidly moving toward common protocols published by international standardization bodies like the Request for Comments (RFC) documents published by the Internet Engineering Task Force (IETF). The entire network is based on the Internet Protocol (IP). Dynamic Host Configuration Protocol (DHCP), Domain Name System (DNS) and the Zeroconf protocol suite simplify and automate network configuration. SNMP is used for network management. Session Initiated Protocol (SIP) and Real Time Protocol (RTP) are natural choices for audio services like radio net access, intercom, and of course, telephony. Similarly, HTTP, SSL, SMB, and other common protocols are not just preferred, but demanded by military customers.

Just like in the commercial world there is little economic sense in developing all the software infrastructure to support these protocols from scratch when the same functionality is easily obtainable from third parties. In particular, since development effort on FOSS operating systems is measured in thousands of man years [2], the functionality and quality FOSS offers is far broader than any company can hope to develop over the course of a single product development cycle.

Of course, custom engineering is still required when standard protocols are not suitable for a particular application. Even so, it is still preferable to start with an existing protocol and build the extra functionality on top of it; ideally while maintaining compliance with the original protocol.

This is where the power of open architecture comes into play. If a custom protocol is what is required, then rather than creating an entirely new protocol from scratch, an existing, standard protocol can be easily tweaked to accommodate the designer's needs. Or, better yet, multiple standard protocols can be combined into a single, custom protocol which still maintains backward compatibility to each original protocol. For example, communicating over a remote radio net involves several tasks: knowing that the radio exists and understanding the path to get there, initiating a pressel-arbitrated session with that radio, and finally transmitting digital audio over the network to that radio. Using a combination of such standard protocols like Zeroconf, SIP, and RTP as building blocks, a custom protocol can be created which will allow the user to perform this complicated operation in a single, simple step with little additional engineering required.

3.2 Segmentation of Functionality

To manage complexity and keep system designers from driving themselves insane with their own designs, it is important to establish boundaries between areas of functionality. Once again, this is not much different from the issues faced by commercial system designers, but the high reliability requirement brings the issue to the forefront.

In the military environment, equipment failures are not just planned for, they are expected. It is important that when equipment does fail that the impact is minimal. For example, imagine a network with 3 user terminals named *commander*, *driver*, and *gunner* and connected in a ring. If the *gunner* terminal fails, then it is expected that any headsets or radios directly connected to the *gunner* node would no longer work. However, that same failure should not affect intercom between the *commander* and *driver* nodes. In this case the system is designed so that no service depends on a particular unit of hardware that is otherwise unrelated to the service. So, while the intercom between *commander* and *driver* may pass through the *gunner* node, the system is designed to bypass it in the event of a failure.

In this example, several boundaries in functionality work together to ensure the desired behaviour. First, at the physical layer the units are at the very least connected in a ring topology which insures that any single point of failure will still retain a connection path between the remaining nodes. A Rapid Spanning Tree Protocol (RSTP) agent runs on each node and controls the network fabric to eliminate Ethernet connectivity loops in real time and ensures the layer 2 network environment is usable without any need for manual intervention.

At layer 3, Zeroconf is used to provide a fallback mechanism for assigning IP addresses in the absence of a properly configured DHCP server. Any two nodes are able to establish a network connection in the absence of any other hardware.

Similarly, the radio net service is logically separate from the network layer and any other services in the network. The radio server runs as a user space application on the node to which a radio is attached and accepts connections from radio clients across the network. It also advertises itself via Multicast DNS (mDNS) which isolates it from depending on a properly configured DNS server. In turn, the radio clients are also user space processes running on end user nodes. If the hardware hosting the radio server fails, then only access to that particular radio is affected and will not bring down the rest of the comm system.

Services and protocols that are decentralized, or are designed to provide automatic failover are greatly preferred over services which do not. The intercom service is provided by *intercom agents* running on each end user devices. The intercom agents do not depend on a single server node to mix all audio streams, but instead use mDNS to discover each other and determine how to mix the intercom traffic between them. Any one intercom node failure shall not take down the entire intercom service.

The Linux kernel plays a significant role here too as the system is designed to tolerate software failures also. With several applications executing on the same hardware device, the failure of a single application must not cause other unrelated applications to fail. The enforced process separation provided by the Linux kernel prevents a bug in one application from affecting the environment of another. This feature in particular is a benefit over traditional flat memory model RTOS systems when providing complex services within a single hardware platform.[2]

Finally, it can be argued that open source projects have tended towards well defined boundaries between components. There is the obvious boundary between kernel and user space, but there are also well defined boundaries between the libraries composing the protocol and application stacks. For example, the MESHnet user devices make use of the GStreamer framework for processing audio streams. The GStreamer core library defines a strict API for plugins but implements very little functionality inside the core library itself. The plugins use the API to communicate with each other over a well defined interface which isolates the two sides from internal implementation details. Separation at this level doesn't directly improve reliability and fault tolerance, but a well defined and predictable API with few side effects makes the system design easier to comprehend and therefore easier to audit for bugs.

3.3 Capability and Quality

Military customers are also unapologetic in their appetite for rich features in their equipment. They want all the features available in comparable commercial equipment, without sacrificing the reliability requirements discussed above. As already stated, the budgets of most military equipment development projects are not large enough to fund the development of all the required components from scratch, so system designers must look to third party software to use as the starting point. This means either licensing a proprietary application or selecting a FOSS component.

When evaluating third party components, quite a few questions tend to be asked: How well does it work? Does it support all the features we need? Is it under active development? Do we get access to the source code? Can it be customized? How much does it cost? How much work is required to integrate it into the rest of the project?

FOSS components do not always come out on top in the tradeoff analysis. For some of the questions, FOSS has a natural advantage over its proprietary counterparts; access to source code and favorable licencing terms being the most significant. For others a lot depends on the type of application and what FOSS components exist in that sphere.

[2]Granted with the tradeoff that embedded Linux system are typically larger and more resource-hungry than the equivalent RTOS implementation. There are no claims of something for nothing here.

For example, within the operating environment sphere there are the Linux and BSD kernels, several other open source RTOSes, and various commercial offerings like vxWorks and QNX. In most cases, Linux comes out on top; active development is high which inspires confidence that the kernel will be around for a long time. It boasts a large feature set, the code quality is excellent, and is easy to port to new embedded platforms.

However, there are still areas where Linux is not chosen as the solutions for good reason. Systems with tight memory constraints are still the domain of traditional RTOSes or even bare metal implementations. Despite large strides being made in the area of Linux real time, some applications require guarantees provided by traditional RTOSes. The existence of legacy code for a particular environment is also a significant factor.

Emotional and legal influences also have an impact on what decision is made. Emotions come into play when designers already have a bias either for or against a particular solution. Misunderstanding or mistrust of the open source development model can also dissuade designers from selecting a FOSS component. And there is always ongoing debate over legal implications of using a GPL licensed work as part of an embedded product.

No single aspect can be pinpointed as the most important factor in selecting a component for use. However, if a FOSS component does provide the functionality needed, and it is shown to be both reliable and actively used in other applications, then there is a greater chance that it will be selected. Well established projects with a broad and active developer base tend to have an advantage in this regard. Not only does this typically indicate that development will continue over the long term, it also suggests (but doesn't guarantee) that it will be possible to obtain support for the component as the need arises. It is also often assumed that a large existing user base will contribute to code quality in the form of bug fixes and real world testing.

Smaller and less popular FOSS projects are at a bit of a disadvantage in this regard and can be viewed as a bit of a risk. If the project developers stop working on it for one reason or another, then the system vendor may be forced to assume the full burden of supporting the software in-house. That being said, the risk associated with small FOSS components is still often lower than the risk associated with a proprietary component being dropped with no recourse by its vendor.

3.4 Maintenance

Unlike the consumer electronics market with high volumes and short product life cycles, the military equipment market tends towards low volume production runs and equipment which must be supported for decades after being deployed. Equipment vendors, especially of large integrated systems, often also enter into long term support contracts for the equipment they have supplied.

Knowing this, it is prudent to start the design process in the mindset that any chosen components such as CPUs and memory chips must be replaceable for the entire expected lifetime of the equipment. One way to do this is to restrict component choices to ones that have a long term production commitment from the manufacturer. Another way is to do a lifetime buyout for the quantity of chips required over the expected support period.

Similarly, software components have the same support requirement. Each component must be supportable over the long term. FOSS components have a natural advantage in this area. Unlike with proprietary components, a third party FOSS vendor cannot impose restrictions on the use and maintenance of a FOSS component. Nor can business or financial changes with a third party vendor affect the ability to maintain the product.[3]

3.5 Redundancy and Fault Tolerance

In this section and the next we get into the real areas where military equipment vendors differentiate themselves from their competition. Pretty much all of the *functionality* required for tactical telecom systems already exist to a large degree in both the FOSS and proprietary ecosystems. How well the components are *integrated* together onto a hardware platform is a big part of whether or not the system is suitable for military use, and that depends on strong system engineering.

[3]Unless, of course, you've also subcontracted support to said third party vendor, then you could be stuck with a manpower problem.

Fault tolerance is an excellent example. Looking at individual components does not tell you much about the system as a whole. To design a reliable system requires looking at the entire system requirements and designing architectures that provide those functions in a reliable and fault tolerant way. Some of those decisions are simple and only affect a small aspect of the design. For example, a typical requirement is for equipment to stand up to the kind of abuse inflicted by solders. A common solution is to enclose the electronics in cast aluminum chassis and to use MIL-STD-38999 connectors instead of RJ-45 jacks for cable connections.

Other issues affect the design of more than one subsystem. Designing a reliable Ethernet layer has an impact on multiple layers of the system design. It has already been discussed that using a mesh topology of Ethernet connections requires the design of each node to include a *managed* Ethernet switch and requires an RSTP agent to run on each box. In addition, the system must be able to report any relevant changes to the network topology to the system users in a useful form. For example, on a vehicle with three nodes connected in a ring, if the system detects that one of the three links is not connected, then that probably indicates that an equipment failure has already occurred and that there is no remaining backup connections in the event of a second failure. This is information that the soldier needs to know so that a decision can be made about whether or not to continue using the damaged equipment. Therefore, the reliability of the Ethernet layer also has an impact on the design of the user interface so that changes in equipment status can be reported.

While individual components have little influence on the system design as a whole, using well designed components with predictable behaviour simplifies the job of the system designer just by requiring less effort to understand the low level intricacies.

3.6 User Interface and Configuration

Finally, even the most feature rich and capable a system is just an expensive doorstop if nobody is able to understand how to use it. On the whole, solders are mostly uninterested in the details of a telecom system and only care about whether or not the system lets them talk to who they need to talk to and provide the network connections they need. Even if the system is quite complex the system designer should strive to make it have the appearance of simplicity for the vast majority of users.

For example, if the network consists of user devices with attached headsets and two units are wired together and powered up without any kind of network configuration, then it is appropriate for the devices to self configure themselves and enable intercom between the two headsets without any manual intervention. Similarly, selecting basic services should strive to only require a single key press. For the few users who do need greater control, like network managers, it is appropriate to provide a different control interface that doesn't hide the details or complexity of the system.

4 Conclusion

The military sector faces significant challenges when designing a communication system that meets the demand for increasingly complex functionality while still retaining the robustness and reliability required by life-critical equipment. With its open architecture and high level of functionality, quality, availability, and maintainability, Linux and other Free and Open Source Software is often well suited to providing the building blocks on which to base the next generation of sophisticated yet stable and operationally simple military communication systems.

References

[1] Mark Adcock, Russell Heal, A Land Tactical Internet Architecture for Battlespace Communications, `http://www.gdcanada.com/documents/Battlespace%20Networking%20MA%203.pdf`, Retrieved on Apr 10, 2008

[2] David A. Wheeler, More Than a Gigabuck: Estimating GNU/Linux's Size, `http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html`, Retrieved on Apr 10, 2008

A Symphony of Flavours: Using the device tree to describe embedded hardware

Grant Likely
Secret Lab
`grant.likely@secretlab.ca`

Josh Boyer
IBM
`jwboyer@linux.vnet.ibm.com`

Abstract

As part of the merger of 32-bit and 64-bit PowerPC support in the kernel, the decision was to also standardize the firmware interface by using an OpenFirmware-style device tree for all PowerPC platforms; server, desktop, and embedded. Up to this point, most PowerPC embedded systems were using an inflexible and fragile, board-specific data structure to pass data between the boot loader and the kernel. The move to using a device tree is expected to simplify and generalize the PowerPC boot sequence.

This paper discusses the implications of using a device tree in the context of embedded systems. We'll cover the current state of device tree support in *arch/powerpc*, and both the advantages and disadvantages for embedded system support.

1 Background

We could go on for days talking about how embedded systems differ from desktops or servers. However, this paper is interested in one particular aspect: the method used by the operating system to determine the hardware configuration.

In general, desktop and server computers are engineered to be compatible with existing software. The expectation is that the operating system should not need to be re-compiled every time new hardware is added. Standardized firmware interfaces ensure that the boot loader can boot the operating system and pass it important details such as memory size and console device. PCs have the BIOS. PowerPC and Sparc systems typically use OpenFirmware. Commodity hardware is also designed to be probeable by the OS so that the full configuration of the system can be detected by the kernel.

The embedded world is different. Systems vary wildly, and since the software is customized for the system, there isn't the same market pressure to standardize firmware interfaces. You can see this reflected in the boot schemes used by embedded Linux. Often the operating system is compiled for a specific board (platform) with the boot loader providing minimal information about the hardware layout, and the platform initialization code is hard coded with the system configuration.

Similarly, data that is provided by the boot firmware is often laid out in an ad-hoc manner specific to the board port. The old embedded PowerPC support in the kernel (found in the `arch/ppc` subdirectory) uses a particularly bad method for transferring data between the boot loader and the kernel. A structure called `bd_info`, which is defined in `include/asm-ppc/ppcboot.h`, defines the layout of the data provided by the boot loader. `#defines` are used within the structure to add platform-specific fields, but there is no mechanism to describe which `bd_info` layout is passed to the kernel or what board is present. Changes to the layout of `bd_info` must be made in both the firmware and the kernel source trees at the same time. Therefore, the kernel can only ever be configured and compiled for a single platform at a time.

When the decision was made to merge 32-bit (`arch/ppc`) and 64-bit (`arch/ppc64`) PowerPC support in the kernel, it was also decided to use the opportunity to clean up the firmware interface. For `arch/powerpc` (the merged architecture tree), all PowerPC platforms must now provide an OpenFirmware-style device tree to the kernel at boot time. The kernel reads the device tree data to determine the exact hardware configuration of the platform.

```
/ {         // the root node
  an-empty-property;
  a-child-node {
    array-prop = <0x100 32>;
    string-prop = "hello, world";
  };
  another-child-node {
    binary-prop = [0102CAFE];
    string-list = "yes","no","maybe";
  };
};
```

Figure 1: Simple example of the `.dts` file format

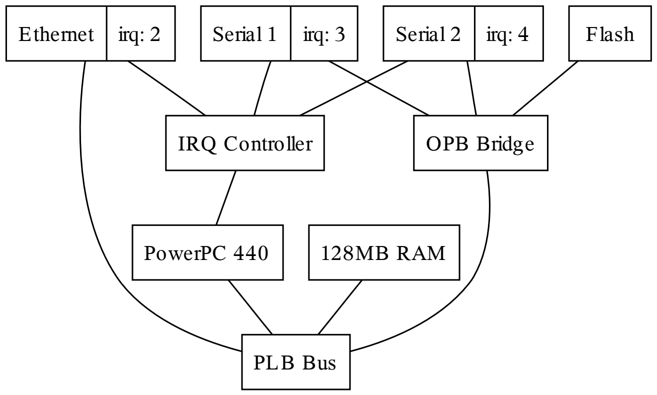


Figure 2: Example PowerPC 440 System

2 Description of Device Trees

In the simplest terms, a device tree is a data structure that describes the hardware configuration. It includes information about the CPUs, memory banks, buses, and peripherals. The operating system is able to parse the data structure at boot time and use it to make decisions about how to configure the kernel and which device drivers to load.

The data structure itself is organized as a tree with a single root node named `/`. Each node has a name and may have any number of child nodes. Nodes can also have an optional set of named property values containing arbitrary data.

The format of data contained within the device tree closely follows the conventions already established by IEEE standard 1275. While this paper covers a basic layout of the device tree data, it is strongly recommended that Linux BSP developers reference the original IEEE standard 1275 documentation and other Open-Firmware resources. [1][2]

The device tree source (`.dts`) format is used to express device trees in human-editable format. The device tree compiler tool (`dtc`) can be used to translate device trees between the `.dts` format and the binary device tree blob (`.dtb`) format needed by an operating system. Figure 1 is an example of a tree in `.dts` format. Details of the device tree blob data format can be found in the kernel's Documentation directory. [3]

For illustrative purposes, let's take a simple example of a machine and create a device tree representation of the various components within it. Our example system is shown in Figure 2.

```
/dts-v1/
/ {
  model = "acme,simple-board";
  compatible = "acme,simple-board";
  #address-cells = <1>;
  #size-cells = <1>;

  // Child nodes go here
};
```

Figure 3: Example system root node

It should be noted that the device tree does not need to be an exhaustive list of all devices in the system. It is optional to itemize devices attached to probable buses such as PCI and USB because the operating system already has a reliable method for discovering them.

2.1 The root Node

The start of the tree is called the root node. The root node for our simple machine is shown in Figure 3. The `model` and `compatible` properties contain the exact name of the platform in the form `<mfg>,<board>`, where `<mfg>` is the system vendor, and `<board>` is the board model. This string is a globally unique identifier for the board model. The compatible property is not explicitly required; however, it can be useful when two boards are similar in hardware setup. We will discuss compatible values more in Section 2.4.1.

2.2 The cpus Node

The `cpus` node is a child of the root node and it has a child node for each CPU in the system. There are no ex-

```
cpus {
  #address-cells = <1>;
  #size-cells = <0>;
  cpu@0 {
    device_type = "cpu";
    model = "PowerPC,440GP";
    reg = <0>;
    // 400MHz clocks
    clock-frequency = <400000000>;
    timebase-frequency = <400000000>;
    i-cache-line-size = <32>;
    d-cache-line-size = <32>;
    i-cache-size = <32768>;
    d-cache-size = <32768>;
  };
};
```

Figure 4: `cpus` node

plicitly required properties for this node; however, it is often good practice to specify `#address-cells=<1>`, and `#size-cells=<0>`. This specifies the format for the *reg* property of the individual CPU nodes, which is used to encode the physical CPU number.

CPU nodes contain properties for each CPU on the board. The unit name for CPU nodes is in the form `cpu@0` and it should have a `model` property to describe the CPU type. CPU nodes have properties to specify the core frequency, L1 cache information, and timer clock frequency. Figure 4 is the `cpus` node for our sample system.

2.3 System Memory

The node that describes the memory for a board is, unsurprisingly, called a *memory node*. It is most common to have a single memory node that describes all of the memory ranges and is a child of the root node. The `reg` property is used to define one or more physical address ranges of usable memory. Our example system has 128 MiB of memory, so the memory node would look like Figure 5.

2.4 Devices

A hierarchy of nodes is used to describe both the buses and devices in the system. Each bus and device in the system gets its own node in the device tree. The node for

```
memory {
  device_type = "memory";
  // 128MB of RAM based at address 0
  reg = <0x0 0x08000000>;
};
```

Figure 5: Memory node

the processor local bus is typically a direct child of the root node. Devices and bridges attached to the local bus are children of the local bus node. Figure 6 shows the hierarchy of device nodes for the sample system. This hierarchy shows an interrupt controller, an Ethernet device, and an OPB bridge attached to the PLB bus. Two serial devices and a Flash device are attached to the OPB bus.

2.4.1 The compatible property

You'll notice that every node in the device hierarchy has a `compatible` property. `compatible` is the key that an OS uses to decide what device a node is describing. In general, compatible strings should be in the form `<manufacturer>,<part-num>`. For each unique set of `compatible` values, there should be a *device tree binding* defined for the device. The binding documents what hardware the node describes and what additional properties can be defined to fully describe the configuration of the device. Typically, bindings for new devices are documented in the Linux Documentation directory in `booting-without-of.txt` [3].

You'll also notice that sometimes `compatible` is a list of strings. If a device is register-level compatible with an older device, then it can specify both its compatible string and the string for the older device, so that an operating system knows that the device is compatible with an older device driver. These strings should be ordered with the specific device first, followed by a list of compatible devices. For example, the flash device in the simple system claims compatibility with `cfi-flash`, which is the string for CFI-compliant NOR flash chips.

2.4.2 Addressing

The device address is specified with the `reg` property. `reg` is an array of *cell* values. In device tree terminol-

```
  plb {
    compatible = "simple-bus";
    #address-cells = <1>;
    #size-cells = <1>;
    ranges;
    UIC0: interrupt-controller {
      compatible = "ibm,uic-440gp",
                   "ibm,uic";
      interrupt-controller;
      #interrupt-cells = <2>;
    };
    ethernet@20000 {
      compatible = "ibm,emac-440gp";
      reg = <0x20000 0x70>;
      interrupt-parent = <&UIC0>;
      interrupts = <0 4>;
    };
    opb {
      compatible = "simple-bus";
      #address-cells = <1>;
      #size-cells = <1>;
      ranges = <0x0 0xe0000000
                0x20000000>;
      serial@0 {
        compatible = "ns16550";
        reg = <0x0 0x10>;
        interrupt-parent = <&UIC0>;
        interrupts = <1 4>;
      };
      serial@10000 {
        compatible = "ns16550";
        reg = <0x10000 0x10>;
        interrupt-parent = <&UIC0>;
        interrupts = <2 4>;
      };
      flash@1ff00000 {
        compatible = "amd,s29gl256n",
                     "cfi-flash";
        reg = <0x1ff00000 0x100000>;
      };
    };
  };
```

Figure 6: Simple System Device Hierarchy

ogy, cells are simply 32-bit values. Array properties like `reg` are arrays of cell values. Each `reg` property is a list of one or more address tuples on which the device can be accessed. The tuple consists of the base address of the region and the region size.

```
reg = <base1 size1 [base2 size2 [...]]>;
```

The actual size of each `reg` tuple is defined by the parent nodes' `#address-cells` and `#size-cells` properties. `#address-cells` is the number of cells used to specify a base address. Similarly, `#size-cells` is the number of cells used to specify a region size. The number of cells used by `reg` must be a multiple of `#address-cells` plus `#size-cells`.

It is important to note that `reg` defines *bus addresses*, not *system addresses*. The *bus address* is local to the bus that the device resides on, or in device tree terms, the address is local to the parent of the node. Buses in turn can map bus addresses up to their parent using the `ranges` property. The format of `ranges` is:

```
ranges = <addr1 parent1 size1 [...]>;
```

Where *addr* is a bus address and is `#address-cells` wide, *parent* is an address on the parent bus and is the parent node's `#address-cells` wide. *size* is the parent node's `#size-cells` wide.

Buses that provide a 1:1 mapping between bus address and parent address can forgo the explicit mapping described above and simply specify an empty `ranges` property:

```
ranges;
```

In this example system, the Flash device is at address 0x1ff00000 on the OPB bus, and the OPB bus specifies that PLB bus address 0xe0000000 is mapped to address 0x0000000 on the OPB bus. Therefore, the Flash device can be found at base address 0xfff00000.

2.5 Interrupts and Interrupt Controllers

The natural layout of a tree structure is perfect for describing simple hierarchies between devices, but is not particularly suitable for capturing complex interconnections.

Interrupt signals are a good example of additional linkages. For example, it is correct to describe the serial device in our sample system as a child of the OPB bus. However, it is also correct to say that it is a child of the interrupt controller device, so how should this be described in the device tree? Established convention says that the natural tree structure should be used to describe the primary interface for addressing and controlling the devices. Secondary connections can then be described with an explicit link between nodes called a *phandle*. A *phandle* is simply a property in one node that contains a pointer to another node.

For the case of interrupt connections, device nodes use the `interrupt-parent` and `interrupts` properties to describe a connection to an interrupt controller. `interrupt-parent` is a phandle to the node that describes the interrupt controller and `interrupts` is a list of interrupt signals on the interrupt controller that the device can raise.

Interrupt controller nodes must define an empty property called `interrupt-controller`. They also must define `#interrupt-cells` as the number of cells required to specify a single interrupt signal on the interrupt controller, similar to how `#address-cells` specifies the number of cells required for an address value.

Many systems, particularly SoC systems, only have one interrupt controller, but more than one can be cascaded together. The links between interrupt controllers and devices form the *interrupt tree*.

Referring back to the serial device node, the property `interrupt-parent` defines the link between the node and its interrupt parent in the interrupt tree.

The `interrupts` property defines the specific interrupt identifier. Its format depends on its interrupt parent's `#interrupt-cells` property and the values contained within are specific to that interrupt domain. A common binding for this property when the parent's `#interrupt-cells` property is 2 is to have the first cell represent the hardware interrupt number for the device in the interrupt controller, followed by its level/ sense information.

2.6 Special Nodes

2.6.1 The chosen Node

Firmware often needs to pass non-hardware types of information to the client OS, such as console port, and boot arguments. The node that describes this kind of information is called the `/chosen` node. There are no required properties for this node; however, it is quite useful for defining the board setup. If our example system booted from a USB key and used the serial port as the console, the `/chosen` node might look like Figure 7.

```
chosen {
  bootargs = "root=/dev/sda1 rw ip=off";
  linux,stdout-path =
      "/plb/opb/serial@10000";
};
```

Figure 7: Chosen node

```
  aliases {
    console = "/plb/opb/serial@10000";
    ethernet0 = "/plb/ethernet@20000";
    serial0 = "/plb/opb/serial@0";
    serial1 = "/plb/opb/serial@10000";
  };
```

Figure 8: Aliases node

2.6.2 aliases

In order to ease device lookup in client operating systems, it is often desirable to define an `aliases` node. This allows one to provide a shorthand method for identifying a device without having to specify the full path on lookup. This is typically only done for the more common devices on a board, such as Ethernet or serial ports. Figure 8 provides an example.

The types of nodes and properties that can be contained in a device tree are as varied as the hardware that they describe. As hardware designers invent new and creative ways of designing components, these unique properties will continue to grow. However, the direction is to be as general as possible to allow commonality between the various hardware components across these boards. Given the flexible nature of the device tree concept, the hope is that the client operating systems will be able to adapt to new hardware designs with a minimal amount of code churn and allow the most possible reuse of code.

3 Usage of Device Tree in Linux Kernel

3.1 Early Boot

To understand how the kernel makes use of the device tree, we will start with a brief overview of the `arch/powerpc` boot sequence. `arch/powerpc` provides a single entry point used by all PowerPC platforms. The kernel expects a pointer to the device tree blob in memory to be in register r3 before jumping to the kernel entry point.[1]

The kernel first does some basic CPU and memory initialization, and then it tries to determine what kind of platform it is running on. Each supported platform defines a `machdep_calls` structure. The kernel calls `probe_machine()`, which walks through the `machine_desc` table, calling the `.probe()` hook for each one. Each `.probe()` hook examines the device tree and returns true if it decides that the tree describes a board supported by that platform code. Typically, a probe will look at the `compatible` property on the root node of the tree to make the decision, but it is free to look at any other property in the tree. When a probe hook returns true, `probe_machine()` stops iterating over the table and the boot process continues.

3.2 Device initialization

In most regards, using the device tree has little impact on the rest of the boot sequence. Platform code registers devices into the device model and device drivers bind to them. Probable buses like PCI and USB probe for devices and have no need for the device tree. Sequence-wise, the boot process doesn't look much different, so the real impact is not on the sequence, but rather on where the kernel obtains information from about peripherals attached to the system.

The interesting questions, then, revolve around how the platform code determines what devices are present and how it registers them with the kernel.

3.2.1 `of_platform` bus

Currently, most embedded platforms using the device tree take advantage of the `of_platform` bus infrastructure. Like the `platform` bus, the `of_platform` bus doesn't represent a hardware bus. It is a software construct for manually registering devices into the device model; this is useful for hardware which cannot be probed. Platform code[2] can use the `of_platform_bus_probe()` convenience function to iterate over a part of the device tree and register a `struct of_device` for each device. Device drivers in turn register a `struct of_platform_driver`, and the `of_platform` infrastructure matches drivers to devices.

The core of both the `platform` and `of_platform` buses is almost identical, which begs the question of why do two separate software buses exist in the first place? The primary reason is that they use different methods to match devices to drivers. The platform bus simply matches a device and a driver if they share the same `.name` property. The `of_platform` bus instead matches drivers to devices on the basis of property values in the tree; in particular, the driver provides a match table of `name`, `device_type`, and `compatible` properties' values. When the values in one of the table entries match the values in an `of_device`, then the bus calls the driver's probe hook.

3.2.2 platform bus adapters

While the `of_platform` bus is often convenient, it is by no means mandated or the only way to retrieve device information out of the device tree. Some drivers already exist with platform bus bindings and the developers have decided not to rework the binding to use `of_platform`. Rather, a helper function is used to search the device tree for nodes with the appropriate property values. When interesting nodes are found, the function creates a new `struct platform_device`, populates it with data from the tree node, and registers it with the platform bus. Several examples of this can be seen in `arch/powerpc/syslib/fsl_soc.c`. As of this writing, the Freescale Gianfar, USB host controller, and I2C device drivers work this way.

There is some debate amongst the Linux PowerPC hackers over whether to merge the `of_platform` bus functionality back into the platform bus. Doing so would eliminate a lot of duplicated code between them, but it leaves the question of how to match drivers and devices. The following are some of the options:

[1]Actually, this is not entirely true. If the boot firmware provides an Open Firmware-compatible client interface API, then the kernel first executes the `prom_init()` trampoline function to extract the device tree from the firmware before jumping into the common entry point.

[2]Not to be confused with the *platform bus*. *Platform code* in this context refers to the support code for a particular hardware platform and can be found in the `arch/powerpc/platforms` subdirectory.

Teach platform bus about device tree matching. If the platform drivers could optionally supply an OF match table, it could be used if the platform device also had a pointer to a device node in the tree. The downside is that it increases the complexity of platform devices, but these are intended to be simple constructs. It is uncertain whether this approach would be acceptable to the platform bus maintainers.

Translate between nodes properties and platform bus names. This approach has a minimal amount of impact on existing platform bus drivers. However, it requires the match table to also supply functions for populating `pdata` structures from the data in the device tree. Also, device-tree-to-platform-bus translation must occur at boot time and not at module load time, which means that the binding data must be contained within the driver module. Besides, device registration is supposed to be under the control of platform code. It is poor form for drivers to register their own platform devices.

Make drivers search the device tree directly. This solves the problem of data about matching devices being separate from the device driver, but it doesn't work so well because there is no easy way to prevent multiple drivers from binding against the same node. Once again, it is bad form for drivers to register their own devices.

3.2.3 Other Methods

Of course, not all initialization fits simply within the `platform/of_platform` bus model. Initialization of interrupt controllers is a good example, since such controllers are initialized directly from one of the platform code hooks and do not touch the driver model at all. Another example is devices that are logically described by more than one node within the device tree. For instance, consider an audio device consisting of a node for the I2S bus and another node for the CODEC device. In this case, each node cannot be probed independently by separate drivers. The platform code most likely needs to interpret the tree data to create device registrations useful to the device drivers.

The key here is that the device tree is simply a data structure. Its sole purpose is to describe the hardware layout and it does not dictate kernel architecture. Platform code and device drivers are free to query any part of the tree they desire to make appropriate decisions.

At this point it is worth mentioning that it can be a strong temptation to design new device tree bindings around what is convenient for the device drivers. The problem is that what might seem like a good approach when you start writing a device driver often turns out to be just the first of several bad ideas before you finish it. By keeping the device tree design focused on hardware description alone, it decouples it from the driver design and makes it easier to change the driver approach at some point in the future. There are literally decades of Open Firmware conventions to help you design appropriate bindings.

4 Case Studies

4.1 PowerPC 440 SoCs

The PowerPC 440 chip is a widely used SoC that comes in many different variations. In addition to the PPC 440 core, the chip contains devices such as 16550-compatible serial ports, on-board Ethernet, PCI host bridge, i2c, GPIO, and NAND flash controllers. While the actual devices on the various flavors of the 440 are typically identical, the quantity and location of them in the memory map is very diverse. This lends itself quite well to the device tree concept.

In `arch/ppc`, each PPC 440 board had its own unique board file, and described the MMIO resources for its devices as a set of `#define` directives in unique header files. There were some attempts to provide common code to share among board ports; however, the amount of code duplication across the architecture was quite large. A typical board file was around 200 lines of C code. The code base was manageable and fairly well maintained, but finding a fairly complete view of the interaction among the files was at times challenging.

When contrasted with the `arch/powerpc` port for PPC 440, some of the benefits of the device tree method are quickly revealed. The average board file is around 65 lines of C code. There are some boards that have no explicit board file at all, as they simply reuse one from a similar board. Board ports have become relatively easy to do, often taking someone familiar with device trees a relatively short time to complete base support for a

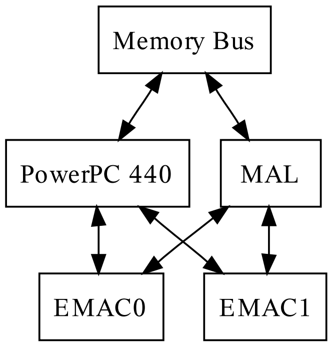


Figure 9: Logical EMAC/MAL connections

board.[3] Additionally, multiplatform support makes it possible to have a single `vmlinux` binary that will run on any of the PPC 440 boards currently supported today. This was virtually impossible before device trees were used to provide the device resources to the kernel.

However, using a device tree does present unique challenges at times. Situations arise that require the introduction of new properties or using different methods of defining the interaction between nodes. For PowerPC 440, one of those cases is the MAL and EMAC nodes. The MAL and EMAC combined comprise the on-board Ethernet. A simplified view of the interconnects is shown in Figure 9.

The MAL device has a number of channels for transmit and receive. These channels are what the various EMAC instances use for their transmit and receive operations. The MAL also has 5 interrupts, but not all these interrupts go to a single interrupt parent. These issues required some more complex concepts and new properties to be applied in the device tree.

To solve the multiple interrupt parent problem for the MAL, an *interrupt map* was used. In this situation, the MAL node's `interrupt-parent` property is set to itself, and the `interrupts` property simply lists interrupts 0–5. Recall that this is possible because the representation of that property is dependent upon the node's `interrupt-parent` property, which in this case is the MAL itself. To properly map the real interrupts to the appropriate controllers, the `interrupt-map` property is used. In this property, each MAL-specific interrupt is mapped to its proper interrupt parent using the interrupt domain for that parent. Figure 10 shows the device tree node for the MAL. Here you can see that MAL interrupt 0 maps to UIC0 interrupt 10, and so on.

[3]The AMCC PowerPC 440EPx Yosemite board support was completed in 5 hours, with the majority of the work coming from rearranging the existing 440EPx support and adapting a new DTS file.

```
MAL0: mcmal {
  compatible = "ibm,mcmal-440gp",
    "ibm,mcmal";
  dcr-reg = <0x180 0x62>;
  num-tx-chans = <4>;
  num-rx-chans = <4>;
  interrupt-parent = <&MAL0>;
  interrupts = <0 1 2 3 4>;
  #interrupt-cells = <1>;
  interrupt-map = <
   /*TXEOB*/ 0 &UIC0 0xa 4
   /*RXEOB*/ 1 &UIC0 0xb 4
   /*SERR*/  2 &UIC1 0 4
   /*TXDE*/  3 &UIC1 1 4
   /*RXDE*/  4 &UIC1 2 4>;
  interrupt-map-mask = <0xffffffff>;
};

EMAC0: ethernet@40000800 {
  device_type = "network";
  compatible = "ibm,emac-440gp",
    "ibm,emac";
  interrupt-parent = <&UIC1>;
  interrupts = <1c 4 1d 4>;
  reg = <40000800 0x70>;
  local-mac-address = [000000000000];
  mal-device = <&MAL0>;
  mal-tx-channel = <0 1>;
  mal-rx-channel = <0>;
};
```

Figure 10: PowerPC 440 MAL and EMAC nodes

You'll also notice in Figure 10 that there are some new MAL-specific properties introduced for the total number of transmit and receive channels. When looking at the EMAC node in Figure 10, you will see that there are new properties specific to the MAL as well. Namely, the `mal-device` property is used to specify which MAL this particular EMAC connects to, pointing back to the phandle of the MAL node. The `mal-tx-channel` and `mal-rx-channel` properties are used to specify which channels within that MAL are used. The device driver for the on-board Ethernet parses these properties to correctly configure the MAL and EMAC devices.

While this is certainly not the most complex interaction between devices that can be found, it does illustrate how a more interesting setup can be accomplished using the device tree. For those interested in further examples of complex setups, including multiple bridges and unique address ranges, the `arch/powerpc/boot/dts/ebony.dts` file found in the kernel would be a good starting point.

4.2 Linux on Xilinx Spartan and Virtex FPGA platforms

Xilinx has two interesting FPGA platforms which can support Linux. The *Spartan* and *Virtex* devices both support the *Microblaze* soft CPU that can be synthesized within the FPGA fabric. In addition, some of the *Virtex* devices include one or more dedicated PowerPC 405 CPU cores. In both cases, the CPU is attached to peripherals which are synthesized inside the FPGA fabric. Changing peripheral layout is a simple matter of replacing the bitstream file used to program the FPGA.

The FPGA bitstream file is compiled from VHDL, Verilog, and system layout files by Xilinx's *Embedded Development Kit* tool chain. Historically, EDK also generated an include file called `xparameters.h` which contains a set of `#define` statements that describes what peripherals are present and how they are configured. Unfortunately, using `#defines` to describe the hardware causes the kernel to be hard coded for a particular version of the bitstream. If the FPGA design changes, then the kernel needs to be recompiled.

Just like other platforms, migrating to `arch/powerpc` means that Virtex PowerPC support must adopt the device tree. Fortunately, the device tree model is particularly suited to the dynamic nature of an FPGA platform. By formalizing the hardware description into the device tree, the kernel code (and therefore the compiled image) is decoupled from the hardware design—particularly useful now that hardware engineers have learned software's trick of changing everything with a single line of source code.

On the other hand, the burden of tracking changes in the hardware design is simply shifted from making changes in the source code to making changes in the device tree source file (`.dts`). For most embedded platforms, the `.dts` file is written and maintained by hand, which is not a large burden when the hardware is stable and few changes are needed once it is written. The burden becomes much greater in the FPGA environment if every change to the bitstream requires a manual audit of the design to identify device tree impacts.

Fortunately, the FPGA tool chain itself provides a solution. The FPGA design files already describe the system CPUs, buses, and peripherals in a tree structure. Since the FPGA tool chain makes significant use of the TCL language, it is possible to write a script that inspects EDK's internal representation of the system design and emits a well formed `dts` file for the current design. Such a tool has been written; it is called *gen-mhs-devtree* [**?**] and it is in the process of being officially integrated into the EDK tool chain.

Figure 11 shows an example of a device tree node generated by an instance of version 1.00.b of the `opb_uartlite` ipcore. As you can see, the node includes the typical `compatible`, `reg`, and `interrupt` properties, but it also includes a set of properties with the *xlnx,* prefix. These properties are the ipcore configuration parameters extracted from the FPGA design. The device tree has made it possible to provide this data to the operating system in a simple and extensible way.

```
RS232_Uart_1: serial@40400000 {
  compatible =
      "xlnx,opb-uartlite-1.00.b";
  device_type = "serial";
  interrupt-parent = <&opb_intc_0>;
  interrupts = < 4 0 >;
  port-number = <0>;
  reg = < 40400000 10000 >;
  xlnx,baudrate = <2580>;
  xlnx,clk-freq = <5f5e100>;
  xlnx,data-bits = <8>;
  xlnx,odd-parity = <0>;
  xlnx,use-parity = <0>;
} ;
```

Figure 11: node generated by `gen-mhs-devtree`

With the success of the device tree for Xilinx PowerPC designs, Xilinx has decided to also adopt the device tree mechanism for Microblaze designs. Since many of the Xilinx peripherals are available for both PowerPC and Microblaze designs anyway, it was the natural choice to use the same mechanism for describing the hardware so that driver support can be shared by both architectures.

5 Tradeoffs and Critique

5.1 kernel size

One of the often-heard concerns of using the device tree method is how much larger the kernel binary will be. The common theory is that by adding all the device tree probing code, and any other "glue" code to make the board-specific drivers function with the generic kernel infrastructure, the overall `vmlinux` binary size will drastically increase. Combine this with having to store the dtb for the board and pass it in memory to the kernel, and one could see why this might be a concern for embedded targets.

While it is certainly true that the size of the `vmlinux` binary does grow, the actual differences are not as large as one may think. Let's examine the sizes of an `arch/ppc` and an `arch/powerpc` `vmlinux` binary using feature-equivalent kernel configs for minimal support with a 2.6.25-rc9 source tree.[4] Table 1 shows the resulting binary size for each arch tree.

Arch	Text	Data	BSS	Total
ppc	2218957	111300	82124	2412381
powerpc	2226529	139564	94204	2460297

Table 1: Section sizes of `vmlinux` binaries

As you can see, the overhead for the device tree method in this case is approximately 47KiB. Add in the additional 5KiB for the dtb file, and the total overhead for a bootable kernel is approximately 52KiB.

While to some this may seem like quite a bit of growth for such a simple configuration, it is important to keep in mind that this brings in the base OpenFirmware parsing code that would be required for any `arch/powerpc` port. Each device driver would have some overhead when compared to its `arch/ppc` equivalent; however, this would be a fairly small percentage overall. This can be seen when examining the `vmlinux` size for a multiplatform `arch/powerpc` config file. This config builds a kernel that runs on 6 additional boards, with support for 5 additional CPU types, and adds the MTD subsystem and OF driver. The resulting `vmlinux` adds approximately 130 KiB of overhead when compared to the single-board `arch/ppc` config. A comparison with a similar multiplatform config in `arch/ppc` cannot be done, as there is no multiplatform support in that tree.

[4]Essentially, the kernel was configured for BOOTP autoconf using an NFS rootfilesystem for the PowerPC 440GP Ebony evaluation board.

5.2 Multiplatform Kernels

Another question that is often heard is "But why do I care if I can boot one `vmlinux` on multiple boards? I only care about one board!" The answer to that, in short, is that most people probably don't care at all. That is particularly true of people who are building a single embedded product that will only ever have one configuration. However, there are some benefits to having the ability to create such kernels.

One group that obviously benefits from this are the upstream kernel maintainers. When changing generic code or adding new features, it is much simpler to build a multiplatform kernel and test it across a variety of boards than it is to build individual kernels for each board. This is helpful for not only the architecture maintainers, but anyone doing wider cross-arch build testing.

It's important to note that for most of the embedded boards, there is nothing that precludes building a single board config. Indeed, there are approximately 35 such defconfigs for the Freescale and IBM/AMCC embedded CPUs alone. However, doing a "buildall" regression build of this takes quite a long time, and the majority of it is spent building the same drivers, filesystems, and generic kernel code. By using a multiplatform defconfig, you only need to build the common bits once and the board-specific bits are built all together.

So while not everyone agrees that multiplatform embedded kernels are useful, the current direction is to put emphasis on making sure new board ports don't break this model. The hope is that it will be easier for the maintainers to adapt existing code, perform cleanups, and do better build test regressions.

References

[1] `http://playground.sun.com/1275/home.html`, Accessed on June 24, 2008.

[2] `http://www.openfirmware.org`, Accessed on June 24, 2008.

[3] Benjamin Herrenschmidt, Becky Bruce, *et al.* `http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=blob;f=Documentation/powerpc/booting-without-of.txt`, Retrieved on June 21, 2008.

Tux on the Air: The State of Linux Wireless Networking

John W. Linville
Red Hat, Inc.
`linville@redhat.com`

Abstract

"They just want their hardware to work," said Jeff Garzik in his assessment on the state of Linux wireless networking in early 2006. Since then, more and more of "them" have their wish. Lots of hardware works, and most users have little or no trouble using their Linux laptops at their favorite cafe or hotspot. Wireless networking no longer tops the list of complaints about Linux. Of course, some problems persist... and new things are on the horizon.

This paper will discuss the current state of Linux wireless networking, mostly from a kernel perspective. We will explore where we are, some of how we got here, and a little of why things are the way they are. We will also preview some of what is to come and how we plan to get there.

1 Where have we been?

The original wireless LAN devices were what are now called "full MAC" devices. These devices go to a great deal of effort to present themselves to the host processor very much like an ethernet device. They tend to have large firmware images that create that illusion, and only add the requirement of configuring parameters particular to dealing with wireless LANs such as specifying a Service Set IDentifier (SSID).

Devices with "full MAC" designs are not particularly prone to being forced into modes that do not comply with governmental regulations. As a result, vendors of such devices tend to be more open to supporting open source drivers. Examples include the old Prism2 and the early Prism54 devices, which are well supported by drivers in the Linux kernel. Unfortunately, the requirement for large firmware images tends to increase the costs associated with producing this kind of device.

Newer consumer-oriented wireless LAN devices tend to utilize what is known as a "half MAC" or "soft MAC" design. These devices minimize the work done using firmware on the devices themselves. Instead, only critical functions are performed by the device firmware, and higher functions like connection management are transfered to the host processor. This solves problems for hardware manufacturers, but makes life more difficult for open source software in more ways than one.

The chief problem created by the shift to "soft MAC" designs is the need for software to perform those functions on the host processor that had previously been performed by firmware on the wireless LAN device. The early Intel Centrino wireless drivers used a component called "ieee80211" to perform these functions. The `ieee80211` component used code adapted from the earlier `hostapd` driver for Prism2 devices.

Unfortunately, the early Centrino wireless hardware designs were not sufficiently general to apply the `ieee80211` code directly to other drivers. In response to that need, Johannes Berg developed an extension to that code to support true "soft MAC" designs. The original `bcm43xx` and `zd1211rw` drivers used this "ieee80211softmac" code successfully, as did a few more drivers that never got merged into the mainline Linux kernel. However, many developers (including Johannes Berg) felt that this combination of code was poorly integrated and the source of many problems.

About this time, Devicescape released source code to what would eventually become the `mac80211` component. Most of the active wireless developers rallied around this code and so all new development for "soft MAC" devices was shifted to this codebase. Unfortunately, many core Linux networking developers quickly identified systemic problems with the code from Devicescape. Thankfully, Jiri Benc adopted the Devicescape code and began working to resolve these problems. After many months, a great deal of code pruning, and some help from Michael Wu, Johannes Berg, and others, the new `mac80211` component was initially merged into the Linux kernel version 2.6.22.

The remaining drivers were merged later, and finally there was much rejoicing.

Now things are much better. Of course, the Linux kernel continues to support "full MAC" designs, and now it has infrastructure to support "soft MAC" devices as well. Thanks to this new infrastructure, it is possible to add new features to a whole set of related drivers with a minimal set of changes. Linux is well on its way to being a first-class platform for all forms of wireless LAN networking.

2 Where are we now?

The past is the past. What is the situation now? What drivers are available? Which ones are coming soon? How fast is wireless LAN development progressing? Where can I get the latest code? Where can I find current information? And what is coming next?

2.1 Driver status

Several wireless LAN drivers have been developed and merged into the mainstream kernel over the past few years. Still more are currently under development. Sadly, a few may never appear.

2.1.1 Current Drivers

Table 1 represents the 802.11 wireless LAN drivers available in the 2.6.25 kernel which were originally merged in 2006 or later. As you can see, nearly all of them are based upon the `mac80211` infrastructure. The notable exception is `libertas`. That driver was developed for use in the One Laptop Per Child's XO laptop, which relied on extensive firmware both for power management and for implementing a pre-standard version of mesh networking. The other exception is `rndis_wlan`, which is primarily an extension of the existing `rndis_host` ethernet driver to also support configuration of wireless devices which implement the RNDIS standard. Aside from these two, all of the other drivers are for "soft MAC" devices.

All of these drivers support infrastructure mode (a.k.a. "managed mode") and most of them support IBSS mode (a.k.a. "ad-hoc mode") as well. These drivers can reasonably be expected to work well with NetworkManager and similar applications as well as the traditional wireless configuration tools (e.g., `iwconfig`).

Driver	Hardware Vendor	Uses mac80211?
`adm8211`	ADMTek	Y
`ath5k`	Atheros	Y
`b43`	Broadcom	Y
`b43legacy`	Broadcom	Y
`iwl3945`	Intel	Y
`iwl4965`	Intel	Y
`libertas`	Marvell	N
`p54pci`	Intersil	Y
`p54usb`	Intersil	Y
`rndis_wlan`	Various	N
`rt2400pci`	Ralink	Y
`rt2500pci`	Ralink	Y
`rt2500usb`	Ralink	Y
`rt61pci`	Ralink	Y
`rt73usb`	Ralink	Y
`rtl8180`	Realtek	Y
`rtl8187`	Realtek	Y
`zd1211rw`	ZyDAS	Y

Table 1: New drivers since 2005

2.1.2 Drivers In Progress

A number of drivers have been started but are not yet completed or mergeable for one reason or another. Reasons include questionable reverse-engineering practices, incomplete specifications, or simply a lack of developers working on producing a mergeable driver.

The `tiacx` driver for the Texas Instruments ACX100 chipset has been around for a long time. At one time this driver was working well and had been successfully ported to the `mac80211` infrastructure. Unfortunately, questions were raised about the reverse engineering process used to create the initial version of this driver. It is possible that TI could clarify this driver's legal status. Otherwise, work similar to that done by the SFLC to verify the provenance of the `ath5k` driver will be required to remove the legal clouds currently preventing this driver from being merged into the mainstream Linux kernel.

The `agnx` driver for the Airgo chipsets[1] is based upon a set of reverse-engineered specifications[2] as well. To date, this team has maintained a rigid separation between reverse engineers and driver authors. This is the same technique used with good results to implement the `b43` and `b43legacy` drivers. Unfortunately, the main

[1]`http://git.sipsolutions.net/?p=agnx.git`
[2]`http://airgo.wdwconsulting.net/mymoin`

reverse engineer has had to leave the project for personal reasons. Worse, the driver is still not fully functional. Until the `agnx` reverse engineering team is reconstituted, this driver remains in limbo.

The `at76_usb` driver supports USB wireless chipsets from Atmel. This driver has been ported to the `mac80211` infrastructure and it works reasonably well. It is possible that it will be merged as early as the 2.6.27 merge window.

A driver has appeared recently for the Marvell 88w8335 chipset. These are PCI (and CardBus) devices manufactured a few years ago, and Marvell has shown little interest in supporting an upstream driver for them. While these devices were marketed under the "Libertas" brand, these devices bear little or no resemblance to those covered by the `libertas` driver. The driver for these devices is called `mrv8kng` and it has been posted for review. Hopefully that will lead to it being mergeable as early as the 2.6.27 merge window as well.

Marvell has shown interest in supporting a driver for their current TopDog chipset. They have released a driver based on the `net80211` infrastructure from the Madwifi project, and they have provided some minimal documentation on the firmware for their device. Unfortunately, no one is known to be actively working to port this driver to `mac80211` or to make it mergeable into the mainstream kernel.

2.1.3 Unlikely Drivers

Hardware vendors come and go, and some products are more successful than others. Nowadays most Linux wireless drivers are reverse engineered, and reverse engineering takes lots of both motivation and skill. While it is certainly possible that a motivated and skilled reverse engineer will apply his craft to produce a driver for an uncommon hardware device, the odds are against such an occurrence. So wireless devices that enjoyed limited market penetration and are no longer in production are unlikely to ever get a native Linux driver. One example of such a device is the InProComm IPN2220. Unfortunately, NDISwrapper will likely remain the only viable solution for making this hardware work under Linux.

2.2 Patch Activity

Take it from the author, someone who knows: wireless LAN is currently one of the fastest developing segments of the Linux kernel. New drivers or new features arrive nearly every month, and the large portions continue to undergo extensive refactoring as lessons are learned and functionality is extended. In fact, some might say that too much refactoring continues to occur! Nevertheless, the wireless developers continue to be prolific coders.

This is not simply anecdotal—Linux Weekly News (LWN) regularly documents patch activity in the kernel as it nears each release. Because of the role the author plays as wireless maintainer, the number of Linux kernel Signed-off-by's for the author provides a good proxy for the level of activity around wireless LAN in the kernel. For 2.6.24, LWN placed the author as the #5 "gatekeeper" for patches going into the linux kernel.[3] This represented over 4% of the total patches in that release, or more than 1 out of every 25. For 2.6.25, it was a full 5%, or 1 out of every 20 patches.[4] This is all the more impressive when one considers that no wireless vendor other than Intel provides direct developer services.[5] Given the amount of work remaining, I see no reason to believe that this level of production will change significantly in the near future.

2.3 How does someone get the code?

Given the quick pace which continues around the Linux kernel in the area of wireless LAN development, not all distributions are shipping with current wireless code. All of this progress does one no good if you do not have the code. So how is a user to go about getting it?

2.3.1 Development Trees

Ongoing development is done using `git`, now the standard revision control system for the Linux kernel. Multiple trees are used in order to accomodate various needs relating both to distributing patches to other kernel maintainers and to facilitating further development.

[3] `http://lwn.net/Articles/264440/`

[4] `http://lwn.net/Articles/275954/`

[5] In fact, nearly all of the prominent wireless developers are university students!

Most of these trees are of no interest either to end users or to casual developers.

The tree that is of interest is the wireless-testing tree, `git://git.kernel.org/pub/scm/linux/kernel/git/linville/wireless-testing.git`. This tree is generally based on a recent "rc" release (e.g., 2.6.25-rc9) from Linus. It also includes any patches destined for the current release that have not yet been merged by Linus, as well as any patches queued for the next release. This tree might also include drivers that are still in development and are not yet considered stable enough for inclusion even in the next kernel release. Consequently, this tree is intended as the base for any significant wireless LAN development. Users or developers seeking an up-to-date wireless LAN codebase should use this tree.

2.3.2 The compat-wireless-2.6 project

Not everyone is interested in running a "bleeding edge" kernel. The compat-wireless-2.6 project[6] was created to fill this need. This project provides a means to compile very recent wireless code in a way that is compatible with older kernels. At the time of this writing, kernels as old as 2.6.21 are supported. Even older kernels may be supported by the time you read this.

The compat-wireless-2.6 project maintains a set of scripts, patches, and compatibility code. These are combined with code taken from a current wireless-testing tree. The resulting code is compiled against the user's running kernel, resulting in modules that can be loaded to provide current wireless bugfixes and updated hardware support. Users of enterprise distributions or others who need to run older kernels may find this project to be quite useful.

2.3.3 Fedora

Obviously the easiest way to get kernels is through a distribution. This is especially true for users who may not be kernel hackers or even software developers at all. Perhaps unfortunately (or possibly by design), distributions have varying policies regarding kernel updates. Consequently, not all distributions have an easy means for users to run kernels with current wireless LAN code. One distribution that does provide such kernels is Fedora.

The Fedora build system is called Koji,[7] and all official Fedora packages are built there. As a Fedora contributor, the author ensures that current wireless fixes and updates make their way into the Fedora kernel on a timely basis. The normal Fedora update process typically makes kernel updates available within a few weeks. Those too impatient to wait can retrieve later kernels directly from Koji. Picking the latest kernel built by the author is usually the best way to find the kernel with the latest wireless bits:

```
http://koji.fedoraproject.org/koji/
userinfo?userID=388
```

2.4 Website

Those simply wanting a starting point for information about current wireless LAN developments would do well to vist the Linux Wireless wiki,[8] graciously hosted by Johannes Berg. This site has information organized for users, hardware vendors, and potential developers. Since it is a wiki, the site is easily updated as old information becomes obsolete, and it is open to a wide variety of potential contributors who may or may not be actual software developers. The Linux Wireless wiki is a good first stop for anyone seeking more information about wireless LAN support in Linux.

3 What is coming?

The road behind us was a hard slough, and now we stand on steady ground. Yet we are far from home! What new features are coming to Linux wireless LANs? A new and better means for configuring wireless devices is on the way, and new ways for using those devices to communicate are coming as well.

3.1 Replacing Wireless Extensions with CFG80211

Traditionally Linux wireless interfaces have been configured using an Application Programming Interface (API) known as Wireless Extensions (WE). This API

[6] `http://www.linuxwireless.org/en/users/Download`

[7] `http://koji.fedoraproject.org`

[8] `http://wireless.kernel.org`

is based on a series of `ioctl` calls, each of which specifies the parameters of a specific attribute for wireless LAN configuration. This API mapped sufficiently well to the designs of wireless LAN devices that were prevalent when WE was produced, and it continues to remain at least minimally serviceable for modern designs. However, WE has many shortcomings. Chief of these is that it fails to specify a number of details about what default behaviors should be, what the proper timing should be, or order of configuration steps, or even what the exact meaning is intended to be for a number of parameters. Further, reliance on individual configuration actions for what otherwise might be considered atomic operations introduces the possibility of race conditions when configuring devices. Also, WE has proven to be difficult to extend without breaking the kernel's pledge of userland Application Binary Interface (ABI) consistency between releases. Finally, the in-kernel WE implementation is mostly transparent, forcing individual drivers to reimplement a number of features of the API which might otherwise be shared.[9] All of this, coupled with the general disdain for `ioctl`-based interfaces which is now prevalent amongst kernel developers, makes it difficult to extend or even adequately maintain WE going forward.

In order to address this, work has begun on `cfg80211`. This is a component intended to replace WE for configuration of wireless interfaces. The `cfg80211` component should provide a much cleaner API both to userland applications and to drivers. The userland interface (as implemented in the Netlink-based `nl80211` component) will provide a logical grouping of configuration parameters so that logically atomic operations are actually handled atomically within the kernel. On the driver side, `cfg80211` will provide an interface that minimizes the amount of configuration handling that drivers need to do on their own.[10] Finally, the designers of `cfg80211` have attempted to observe the API design lessons learned over a decade of continuing Linux development. The `cfg80211` component should remain both extensible and maintainable for a long time to come.

[9]This leads to differing behaviors between drivers and is a potential source of bugs that might otherwise be avoided.

[10]This should serve to provide much more consistent wireless driver behavior observable by userland applications like NetworkManager.

3.2 "Access point" mode

Most people interested in wireless LAN technology know that there is a difference between a wireless client device and a wireless access point. The latter is usually a small box with antennas on it that plugs into the wired LAN (e.g., the Ethernet jack on the wall or the back of a cable modem or DSL modem). Access points are wireless infrastructure devices that coordinate wireless LAN traffic, and they require somewhat different software to implement this coordination behavior. Further, many early wireless device designs made it impossible to implement an access point no matter if you had the required software or not. This is why the list of devices traditionally supported by the `hostapd` software is rather short.

The dirty little secret is that there is not really anything special about the physical wireless LAN hardware used in an access point. Usually only the software and/or firmware controlling it prevents or enables it to be used as an access point. With older designs, this meant that access points needed devices with firmware which allowed access point functions to work. With the "soft MAC" designs which are now prevalent, this means that software in the kernel can allow a wide variety of devices to be used to implement an access point.

It turns out that the `mac80211` component already contains most of the code needed to enable this access point behavior. However, it is currently disabled in stock kernels. This is because that behavior needs a stable API for control by userland software (e.g., `hostapd`), and such an API has not yet been agreed upon. Work is in progress (and far along) on implementing support for such an API in the `cfg80211` and `nl80211` components. The current maintainer of the `mac80211` component, Johannes Berg, has a series of patches for both the kernel and for `hostapd` that enables using Linux as an access point. This support may be merged into the mainline kernel as early as version 2.6.27.

3.3 Mesh networking

Many people realize that there are currently two common modes of communication on wireless LANs: a) *access point* or infrastructure mode, where the wireless client talks to an access point that directs traffic to the rest of the LAN; and b) *ad-hoc* or independent BSS mode, where wireless clients coordinate wireless traffic amongst each other, but with traffic limited to stations that are physically in range of one another. In recent years a new mode has been under development. This mode, commonly called *mesh networking*, is a bit like a mixture of the two previous modes. Wireless stations coordinate wireless traffic amongst each other within a limited range, but also stations can pass traffic between stations that cannot otherwise reach each other. This enables communication over much larger ranges without requiring lots of infrastructure, and is therefore ideal for underdeveloped or disaster-stricken areas. This mode of communication has seen popular use by the OLPC project in their XO laptops.

The people that developed the wireless mesh firmware for the adapters on the XO laptop have also contributed a pre-standard implementation of mesh networking for the `mac80211` component. The developers from Cozybit seem to be committed to maintaining and improving this code until the 802.11s specification for mesh networking is finalized. This gives Linux a cutting-edge wireless capability not currently seen in other mainstream operating systems. Surely this will prove useful to a great number of people all over the world.

3.4 Multi-Queue Support

The 802.11e wireless standard defines Quality of Service (QoS) mechanisms based on classifying traffic into four queues.[11] All pending traffic in the highest priority queue is transmitted before traffic in the next highest priority, and so on. The `mac80211` component implements support for this by using a custom queueing discipline associated with the physical wireless device.

Modern wired LAN devices have evolved designs which also have multiple queues for supporting QoS applications. Consequently, the Linux kernel's networking infrastructure has been extended to support the concept of multiple hardware queues attached to a single network interface. Now that this exists, it seems sensible to convert the `mac80211` component to make use of this new infrastructure. Work is currently underway to achieve just that.

4 What else is needed?

So, it seems that things are firming up reasonably well. Further, the wireless LAN developers already have the next round of work cut out for them. But surely that is not all that is lacking? There certainly are areas that need to be addressed. These include better power management and taking advantage of performance-enhancing features available to drivers within the Linux kernel's networking layer.

[11]The queues are designated for voice traffic, video traffic, best-effort traffic, and background traffic.

4.1 Better Power Management

Power management is an important issue. Not only are mobile devices continuing to proliferate, but also energy efficiency has become increasingly important even with desktop computers and other fixed-location devices. There are both economic and ecological reasons behind this trend, and it is unlikely to significantly decrease in importance anytime soon—just the opposite is likely. So it behooves wireless LAN devices to be good citizens regarding power usage.

4.1.1 Suspend and Resume

Drivers receive notifications of suspend and resume events from the core kernel. Drivers are expected to save or restore the state of their associated hardware as appropriate for the specific notification. This approach suffices for the vast majority of LAN adapters. Since "full MAC" devices implement the connection management functionality themselves, this approach should work for those devices as well.

In the case of `mac80211`-based devices, the actual hardware driver does not implement the connection management functionality. Since the wireless LAN environment may change radically between when a device is suspended and when it is resumed, there is no way for a `mac80211`-based driver to reliably resume operational state by itself after a suspend.

Unfortunately, the `mac80211` component is currently completely unaware of suspend and resume events. Drivers work around this by unregistering themselves from `mac80211` upon suspend and re-registering themselves upon resume. This method works reasonably well in many circumstances, but it is unreliable and it tends to increase the wait required after a resume before the wireless interface is again usable. The `mac80211` component needs to be aware of suspend and resume events and it needs to handle them appropriately without forcing driver work-arounds.

4.1.2 Power Saving Mode

The 802.11 specification includes a mechanism for a device in infrastructure mode to notify its associated access point that it is entering power-saving mode. The access point then queues frames intended for the power-saving station. Periodically the station returns to full power state long enough to ask the access point if it is holding traffic for the station, and to accept delivery of any such traffic. This can enable a device to save a great deal of power if it is not actively transmitting traffic.

The `mac80211` component currently makes no use of this mechanism. The potential for power savings makes this seem like a "must have" feature. On the other hand, implementing it may not be as simple as it sounds. Still, this would be a welcome addition to the `mac80211` feature set.

4.2 NAPI Interrupt Mitigation

NAPI is a mechanism used by network drivers to mitigate the costs of processing interrupts on a busy network link. The basic idea is to disable interrupts after the first one and schedule a polling routing to keep processing incoming frames. In that way, the kernel only incurs the cost of the first interrupt in a burst of traffic rather than processing interrupts for individual frames.

Originally, NAPI implicitly assumed that a given interrupt source was associated with a single network interface. Because the `mac80211` component supports multiple kernel network interfaces on a single physical wireless interface, `mac80211` drivers were implicitly excluded from using NAPI. This has not been a huge problem, because the transfer speeds used on wireless networks has been relatively slow. However, 802.11n is bringing much faster speeds to wireless LANs. Fortunately, NAPI has been changed to disassociate interrupt sources from specific network interfaces. The `mac80211` component should take advantage of NAPI in order to maximize wireless LAN performance.

5 Other issues

Perhaps more than any other section of the kernel, wireless LAN support is held hostage to non-technical concerns. Coping with these legal and political issues is key to maintaining and improving good support for wireless LANs in the Linux kernel.

5.1 Firmware Licensing

Wireless LAN protocols have stringent timing requirements for a number of operations. Consequently, many adapters have an embedded microcontroller with firmware to handle a variety of operations. This is true even for many "soft MAC" designs, even though they rely on the host processor for most higher-level operations. These devices simply do not work without their required firmware.

In the Windows and OSX worlds, drivers typically include the firmware as data embedded in the driver binaries. In most cases developers have learned how to separate the firmware from those binaries so that they can be loaded by Linux drivers as well. Unfortunately, the copyright status of the resulting firmware images is at best uncertain.

Many vendors have provided liberal licenses for their firmware images which allow those images to be freely distributed for use with Linux drivers. Intel and Ralink are two examples of good citizens in this regard. Other vendors have proven unwilling to cooperate on this issue, with Broadcom as the most clear example of an uncooperative vendor. Making wireless LAN hardware purchasing decisions in support of cooperative vendors is advised as the best approach to resolving this issue.

5.2 Vendor Participation

Many vendors have proven unwilling to provide either support or programming documentation for their wireless LAN devices. This is true even for vendors like Broadcom who have grown accustomed to providing such support for their wired LAN devices. Some vendors cite the spectre of governmental regulation as a reason they cannot participate in the creation of open source drivers. Other vendors (such as Realtek) find ways to provide community developers with information and still others (particularly Intel) provide developers dedicated to the cause. Unfortunately, too many vendors continue to depend on support from reverse engineers and unsupported community developers. Again, economic pressure is advised as the best approach to resolving this situation.

5.3 Regulatory Issues

As noted above, vendors often cite the spectre of governmental regualtors as a reason to provide poor support for Linux or no support for Linux at all. Unfortunately, these fears are not altogether unfounded. The regulations governing wireless LAN communications are determined by geographical location and political realities. There are literally dozens or even hundreds of different regulatory jurisdictions covering the planet. Each of these jurisdictions can have its own set of rules about which channels are available, what practices (e.g., active scanning) are acceptable on each channel, what power output is acceptable on each channel, whether the rules differ between indoor and outdoor operation, and many other variables.

Complying with all these regulations can be troublesome at best. Add to that the fact that failure to ensure compliance with local regulations can result in local officials refusing to let a vendor conduct business in their jurisdiction. One can understand how a vendor may be hesitant to embrace the loss of total control over their products. Still, some vendors have found ways to overcome these fears. Once again, economic pressure is advised to persuade hesitant vendors to find ways to satisfy regulators while supporting open source drivers, as well as to reward those vendors which have already done just that.

6 Conclusion

Hopefully it is clear to the reader that a great deal of progress has been made for Linux in the wireless LAN arena over the past few years. Not so long ago Linux was a wireless LAN ghetto, with only a few devices working reliably enough for general use. Now most consumer devices are either already supported, or support already exists for a similar device, with reverse engineering efforts continuing in hope of supporting new device versions. Better still, we now see a number of wireless LAN vendors offering some form of Linux support even if that is only providing liberal licensing terms for device firmware. Also, development of fixes and new features continues to make Linux wireless LAN even better. Wireless LAN is no longer the biggest problem for Linux.

AUGEAS—a configuration API

David Lutterkort
Red Hat, Inc.
dlutter@redhat.com

Abstract

One of the many things that makes Linux configuration management the minefield we all love is the lack of a local configuration API. The main culprit for this situation, that configuration data is generally stored in text files in a wide variety of formats, is both an important part of the Linux culture and valuable when humans need to make configuration changes manually.

AUGEAS provides a local configuration API that presents configuration data as a tree. The tree is backed directly by the various config files as they exist today; modifications to the tree correspond directly to changes in the underlying files. AUGEAS takes great care to preserve comments and other formatting details across editing operations. The transformation from files into the tree and back is controlled by a description of the file's format, consisting of regular expressions and instructions on how to map matches into the tree. AUGEAS currently can be used through a command line tool, the C API, and from Ruby, Python, and OCaml. It also comes with descriptions for a good number of common Linux config files that can be edited "out-of-the-box."

1 Introduction

Configuration management of Linux[1] systems is a notoriously thorny subject. Problems in this space are numerous, from making large numbers of machines manageable by mere mortals, to the sheer mechanics of changing the configuration files of a single system programmatically. What makes the latter so difficult is the colorful variety of configuration file formats in common use, which has historically prevented any form of system-wide configuration API for Linux systems.

AUGEAS lays the foundation for such an API by focusing on the most basic and mundane task in this area: changing configuration files in a way that abstracts away the formatting details that are irrelevant to programmatic configuration changes. While formatting details may be irrelevant in this context, they are still important, and AUGEAS goes through a lot of trouble to preserve comments, whitespace, etc.

Logic for configuration changes is embedded in many tools, and the same logic is reinvented multiple times, often for no other reason than a difference in implementation language. As an example, `Webmin` [**?**] can edit a wide variety of configuration files; that editing logic, and the hard work for writing and maintaining it, is of no use to other configuration tools like `Puppet`, `Bcfg2`, or `cfengine`, since none of them is written in Perl. The prospect of reimplementing large parts of this logic in Ruby for use by `Puppet` was so unappealing that it became clear that a language-agnostic and tool-independent way to achieve this had to be found.

The lack of a simple local configuration API also prevents building higher-level services around configuration changes. Without it, there is no way to set system-wide (or site-wide) policy for such changes that can be enforced across multiple tools, from simple local GUI tools to remote administration capabilities.

AUGEAS tackles this problem in the simplest possible way and focuses solely on the mechanics of modifying configuration files.

2 Design

There is no shortage of attempts to simplify and unify Linux system configuration. Generally, they fall into one of three categories: *keyhole* approaches targeted at one specific purpose; *greenfield* approaches to solve modifying configuration data once and for all; and *templating*, popular in homegrown config management systems when plain file copying becomes unsatisfactory, and often used as a middle ground for the first two.

[1]Most of this paper applies to any Unix-like system, though we will only talk about Linux here.

A careful look at these three types of approaches was very instructive in setting AUGEAS' design goals, and determining what exactly it should do, and, even more importantly, what it should *not* attempt to do.

2.1 Keyhole Approaches

The most direct and obvious approach to scripting configuration changes is to use simple text editing tools like `sed` and `awk` or the equivalent facilities in the scripting language of choice. Since changing configuration data is usually only part of a larger task, like writing an installer, the resulting scripts are good enough for their purpose, but not general enough to be of use in other situations, even if language barriers are not a concern.

All popular open-source config management systems follow this approach, too, resulting in unnecessary duplication of logic to parse and write configuration files, and a healthy mix of common and unique bugs in that logic. Even seemingly simple file formats hold surprises that make it all too easy to write a parser that will fail to process all legal files for that format correctly. As an example, one simple and popular format uses setting of shell variables in a file sourced into a larger script. Since comments in shell scripts start with a `#` and extend to the end of the line, a parser of such files should strip this pattern from every line it reads before processing it further. Unless, of course, the `#` appears inside a quoted string. But even that will trip up on unquoted uses of `#` that do not start a comment, such as `V=x#y`.

2.2 Greenfield Approaches

Recognizing that the state of the art of modifying Linux/Unix configuration is less than ideal, various proposals have been put forward to improve it, such as Elektra and Uniconf. Since the variations in config file formats are the biggest roadblock to treating configuration data in a unified manner, they generally start by proposing a new scheme for storing that data. The exact storage scheme varies from project to project, from LDAP to relational databases and files in their *own* favorite format. Such an approach is of course unrealistic, since it requires that the upstream consumers of configuration data modify their applications to use the new API. From the perspective of an upstream maintainer, such changes are without reward while the API is new and unproven. And the only way to prove that an API is worth upstream's effort is to get upstream projects to use it.

A side effect of introducing completely new storage for configuration data is that the configuration files that administrators are used to, and that a multitude of tools knows about, are either no longer used at all, or are no longer the authoritative store for configuration data. This is undesirable, as system administrators have to get used to a whole new way of making local configuration changes, and scripts have to be changed to use the new configuration API.

Greenfield approaches generally also aim much higher than just modifying configuration data. It is tempting to model other aspects of configuration data and build more capabilities into the new unified API, ranging from fine-grained permissioning schemes to hiding differences between Linux distributions and Unix flavors. Each of these addresses a problem that is hard in its own right, often not just because of technical difficulties, but also because modeling it in a way that is suitable for a wide variety of uses is hard.

2.3 Templating

Templating, just as the greenfield approaches, introduces a new "master" store for all configuration data, which makes it impossible to change it in its "native" location, either manually (for example, during an emergency), or with other programs than the template engine.

2.4 Design Goals

With the successes and limitations of these approaches in mind, AUGEAS focuses on the problem at the heart of all of them: editing configuration files programmatically. Above all else, AUGEAS limits its scope to a handful of goals around that task.

As we have seen for greenfield approaches, it is unlikely that the current situation of configuration data scattered across many files in many formats can be addressed by radically breaking with history and custom. At the same time, as shown by templating approaches, the penalty for generating these files from a new authoritative source is high, and rarely ever appropriate. AUGEAS therefore uses the existing config files as its sole store of configuration data and does not require additional data stores.

The multitude of producers and consumers of configuration data makes it imperative that AUGEAS be useful without the support of these producers and consumers. In other words, AUGEAS must be useful without any changes to other code which handles configuration data—in particular, without any support from the primary users of that data like system daemons. Similarly, there are a vast number of tools that modify configuration data, and it should be possible to use these tools side-by-side with AUGEAS. As a consequence, AUGEAS should not rely on such tools preserving any AUGEAS-specific annotations (for example, in comments), while making sure that such annotations added by other tools are preserved across edits with AUGEAS.

We would like AUGEAS to handle as wide a variety of configuration files as possible. Since every format needs some form of intervention by a person, AUGEAS should make it as easy as possible to describe file formats, not only in terms of the notation of the description, but also in the checks that it can perform to spot errors in the description.

How a change to the tree is translated into a change in the underlying file should be intuitive, and should correspond to a reasonable expectation of "minimal" edits. For example, changing the alias of one host in `/etc/hosts` should only lead to a change on the line containing the host entry, and leave the rest of the file untouched.

Finally, configuration changes have to be made in many situations and with tools written in many languages. AUGEAS therefore must be "language neutral" in the sense that it can be used by the widest variety of programming languages. In practice, this means that AUGEAS has to be implemented in C. Furthermore, the public API relies solely on strings as data types, where some strings denote paths in the tree.

3 Using AUGEAS to change files

AUGEAS can be used in a number of ways: a C library API, the `augtool` shell command, and from a number of other programming languages. Currently, bindings are available for Python, Ruby, and OCaml. The following discusses the usage of `augtool`, which closely mirrors the other interfaces.

3.1 The tree and path expressions

In the tree that AUGEAS maintains, each node consists of three pieces of information: a string *label* that is part of the path to the node and all its children, an optional string *value*, and a list of child nodes. Since files are inherently ordered data structures, the AUGEAS tree is also ordered; in particular, the order of siblings matters. Multiple siblings can have the same label—for example, a host entry in `/etc/hosts` can have multiple aliases, each of which is stored in a separate `alias` node.

Because of this structure, AUGEAS' tree is conceptually similar to an XML parse tree. When multiple siblings have the same label, it is of course necessary to distinguish between them. For that, and for simple searches, AUGEAS adapts some of the conventions of XPath [2]. In particular, a label by itself in a path, for example, `alias/`, matches *all* children of a node with label `alias`. The *n*-th child with that label can be picked out with `alias[n]`, and the last such child, with the special notation `alias[last()]`.

Wildcard searches using `*` as a path component are also supported. The path `/p/*/g` matches all grandchildren with label `g` of the node `p`. Searches with `*` are not recursive, and the above pattern does not match a node `p/a/b/g`.

3.2 Tree manipulation

When `augtool` starts, it reads *schema* descriptions out of a set of predefined directories, and parses configuration files according to them. The result of this initialization is the tree that is presented to the user. The user can now query the tree, using `match` to search nodes that match a certain path expression, and `get` to retrieve the value associated with a node.

The tree is modified using `ins` to insert new nodes at a specific position in the tree—for example, to insert another `alias` node after the first such node, and `rm` to delete nodes and whole subtrees. The value associated with a node can be changed with `set`.

Files are not modified while the tree is being changed, both so that files with possibly invalid entries are not produced while multi-step modifications are under way, and to enable more extensive consistency checks when files are finally written. Writing of files is initiated with the `save` command, which writes new versions of all files whose tree representation has changed; files that have no modifications made to them are not touched.

What files are written, and how the tree is transformed back into files, are again governed by the schemas that `augtool` read on startup. Schemas are written in a domain-specific language, and the primitives of the language ensure that the transformation from file to tree and the reverse transformation from tree to file match and follow certain consistency rules designed to make the round trip from file through tree to modifed file safe and match users' expectations. The mechanisms performing the transformation need to know two pieces of information: which files to transform, and how to transform them. The first is given through a file name filter, described as shell globs specifying which files to include or exclude; the second is done by writing a *lens* that is applied to the contents of each file matching the filter.

4 Lenses and bidirectional programming

AUGEAS needs to transform file contents (strings) into a tree and that tree back into file contents. Rather than having users specify these two mappings separately, and running the risk that they might not be compatible with one another, AUGEAS uses the idea of a *lens* to combine the two mappings in a way that guarantees their compatibility in a very precise sense.

The term *lens* was coined by the `Harmony` project [4], and originates from the "view update" problem: given a concrete view of data (configuration files in AUGEAS' case) and an abstract view of the same data (the tree), construct suitable mappings between the two views that translate changes to the abstract view into intuitively minimal changes of the concrete data. Generally, the mapping from concrete to abstract view leaves out some information, for example, formatting details or comments that are of no use in the abstract view. Conversely, the mapping from abstract to concrete view must restore that data. With that, lenses are not bijective mappings between the concrete and the abstract domains: multiple concrete views, namely all the ones that differ only in unimportant details such as formatting, map to the same abstract view. `Harmony` uses lenses to construct mappings between tree-structured data [3], for example, for synchronization of calendar files that essentially contain the same information, but use different XML-based formats. Similarly, `Boomerang` [1] performs mappings between unstructured text data. AUGEAS uses the same approach for its mapping between text data and trees.

Formally, a lens consists of two functions, *get* and *put*.[2] If C is the set of all concrete data structures (in AUGEAS' case, strings), and A is the set of all abstract data structures (in AUGEAS' case, trees), a lens l consists of

$$l.get : C \to A$$
$$l.put : A \times C \to C$$

The *get* function is used to transform concrete views into abstract ones. The *put* function, which maps abstract views back to concrete views, receives the original concrete view as its second argument and consults that to restore any information left out by the *get* direction. The two directions are tied together by two requirements that express intuitive notions of how lenses should behave when we make a roundtrip from concrete to abstract view and back: for every $c \in C$ and $a \in A$, a lens l must fulfill

$$l.put\ (l.get\ c)\ c = c \qquad \text{(GETPUT)}$$
$$l.get\ (l.put\ a\ c) = a \qquad \text{(PUTGET)}$$

Put into words, GETPUT expresses that transforming a string c into a tree, and then transforming the unmodified tree back into a string, should yield the string c we started with, and ensures that the *put* direction restores any information not captured in the tree faithfully when the tree is not modified. The PUTGET law states that transforming any tree back into a string using an arbitrary concrete string c as the second argument to *put* and transforming the result back into a tree must yield exactly the tree we started with, limiting how *put* uses its second argument c when transforming a tree: it can only use it for those parts of a string that are abstracted away by the *get* direction.

The lens laws are weaker than requiring that *get* and *put* be inverses of one another. That would require that both be bijective, and keep lenses from doing what makes them so useful in practice: abstracting away unimportant information like comments or how many spaces are used to separate two values. Since all lenses contain a *get* and *put* direction that are compatible in the sense laid

[2]Strictly speaking, there is a third function, *create*, involved to create new concrete data from abstract data alone; since we are not proving anything about lenses here, there's no need to distinguish between *put* and *create*.

down by the lens laws, building complex lenses from simpler ones is called *bidirectional programming*, since every lens expresses how to get from input *c* to output *a*, and at the same time, how to get from an output *a* back to an input *c*.

Lenses are built from simple builtin lenses, called *lens primitives*, by combining them with a few builtin *lens combinators*. This mechanism of building complex lenses from simpler ones forms the backbone of AUGEAS' schema descriptions. The two lens laws, GETPUT and PUTGET, restrict how lenses can be combined; in AUGEAS, these restrictions are enforced by the typechecker described below.

Typically, a complex lens that describes the processing of a whole file is broken up in smaller lenses, each of which processes a small portion of the file, for example the aliases of a host in `/etc/hosts`. To support this mode of working, where complex lenses are gradually built from simpler ones, AUGEAS has a builtin unit test facility which makes it possible to verify that a "small" lens applied to a text fragment or a partial tree produces the desired result.

4.1 Matching

Behind the scenes, when a lens is applied either to a string (in the *get* direction) or to a tree (in the *put* direction), it has to be *matched* to the current input for a variety of reasons, the most basic being to check whether a lens applies to the input at all.

In the *get* direction, this poses little difficulty and matching of a lens to a string boils down to matching a string to a regular expression. Regular expressions are restricted to the ones familiar from formal language theory, not the ones popular in various languages such as Perl, as those introduce extensions that leave the realm of regular languages. Some of the computations that the typechecker has to perform can be easily done with regular languages but become uncomputable when a broader class of languages, such as context-free languages, are considered. In practical terms, AUGEAS uses the notation of extended POSIX regular expressions, but does not support backreferences.[3]

[3]The implementation currently also lacks support for a few other features, such as named character classes, but unlike backreferences, those are supportable in principle.

Matching a tree in the *put* direction to a lens is more complicated than string matches for the *get* direction. To avoid implementing a mechanism that matches trees against a full tree schema, AUGEAS defines tree matching solely in terms of matching the labels at one level of the tree against a regular expression. For example, a lens that produces any number of nodes labelled `a` followed by a node labelled `b`, matches any tree that has such a sequence of nodes at its root level, regardless of the structure of the trees underneath each `a` and `b` node. This simplification makes it possible to reduce tree matching to matching regular expressions against strings. A tree with two nodes labelled `a` followed by a node labelled `b` and a node labelled `c` at its root level is converted into the string `a/a/b/c/` for purposes of matching, and the lens mentioned above is converted to the regular expression `(a/)*b/`. Clearly, this lens does not match the tree `a/a/b/c/`.

4.2 Lens primitives

AUGEAS has a handful of lens primitives; strictly speaking, the builtins are functions that, given a regular expression, indicated as *re*, or a string, indicated by *str*, or both, produce lenses.

Tree nodes are constructed by the *subtree* lens combinator discussed in the next section. The lens primitives lay the groundwork for the *subtree* lens: they mark which parts of the input to use as a tree label, which to store as the node's value, and which to omit from the tree.

- *key re* matches the regular expression *re* in the *get* direction and tells the *subtree* lens to use that as the label of the tree node it is constructing. In the *put* direction, it outputs the label of the current tree node.
- *label str* does not consume any input in the *get* direction, nor does it produce output in the *put* direction. It simply tells the *subtree* lens to use the *str* as the label of a tree node.
- *seq str* is similar to *label*; in the *get* direction, it sets the label of the enclosing *subtree* to a number. When that subtree is used in an iteration, the numbers are consecutive, starting from 1. The *str* argument is used to distinguish between separate sequences. In the *put* direction, the *seq* lens expects a tree node labelled with any positive number. There

is also a *counter str* lens whose sole effect it is to reset the counter with the given name back to 1 in the *get* direction.

- *store re* matches the regular expression *re* in the *get* direction and tells the *subtree* lens to use that as the value of the tree node it is constructing. In the *put* direction, it outputs the value of the current tree node.
- *del re str* matches the regular expression *re* in the *get* direction and suppresses any matches from inclusion in the tree. In the *put* direction, it restores the match in the output, if the current tree node corresponds to preexisting input, or outputs the default *str* if the current tree node was added to the tree and does not have any counterpart in the original input.

4.3 Lens combinators

Besides the *subtree* lens, there are a few more lens combinators that make it possible to build complicated lenses from the five lens primitives listed above. In the following, l, l_1, and l_2 always refer to arbitrary lenses:

- The *subtree* lens, written as $[l]$, applies l in the *get* direction to the input and constructs a new tree node based on the results of $l.get$. It uses whatever l marked as label and value for the new tree node; if l contains other *subtree* lenses, the trees constructed by them become the children of the new tree node.
- Lens concatenation, written as $l_1 \cdot l_2$, applies first l_1 and then l_2. In the *get* direction, the tree produced by l_1 is concatenated with that produced by l_2; similarly, in the *put* direction, the current tree is split and the first part is passed to the *put* direction of l_1, and the second part to the *put* direction of l_2.
- Lens union, $l_1|l_2$, chooses one of l_1 or l_2 and applies it. Which one is chosen depends on which one matches the current text in the *get* direction, or the current tree in the *put* direction.
- Lens iteration, l^* and l^+, applies l as long as it matches the current text in the *get* direction and the current tree in the *put* direction.

When AUGEAS processes a file with a lens l, it expects that the lens for that file processes the file in its entirety: that means that $l.get$ has to match the whole contents of the file. If it only matches partially, AUGEAS flags that as an error and refuses to produce the tree for that file. Similarly, when AUGEAS writes the tree back to a file, it expects that the entire subtree for that file, for example, everything under `/files/etc/hosts`, gets written out to file. It is an error if any nodes in that subtree are not written, or if required nodes (such as the canonical name for an entry in `/etc/hosts`) are missing from the tree.

5 Writing schemas

The description of how files are to be mapped to the tree, and the tree back into files are defined in AUGEAS' domain-specific language. The language is a functional language, following the syntactic conventions of ML. Figure 1 shows the definitions needed to process `/etc/hosts`.

Schema descriptions are divided into modules, one per file. A module can contain `autoload` directives and names defined with `let`. AUGEAS' language is strongly typed, and statically typechecked; this ensures that as many checks as possible are performed without ever transforming a single file. The available types are `string`, `regexp`, `lens`, `filter`, and `transform`—the last two are only needed for describing which lens is applied to what file.

String literals are enclosed in double-quotes, and can use the escape sequences familiar from C. Regular expression literals are enclosed in forward slashes and use extended POSIX syntax.

The most important part of the listing in Figure 1 is line 16, which defines the lens used to transform a whole `/etc/hosts` file. Strictly speaking, lenses are only ever applied to strings; finding files and reading and writing their contents is done by transforms. The transform `xfm` combines the lens `lns` and a filter that includes the one file `/etc/hosts`. Transforms are used by `augtool` when it starts up to find all the files it needs to load; to this end, it looks for all transforms in modules on its search path that are marked for autoload, as the transform `xfm` is on line 2 in the example.

The `/etc/hosts` file is line-oriented, with lines further subdivided in fields separated by whitespace. The fields are the IP address, the canonical name, and an

```
 1: module Hosts =
 2:   autoload xfm

 4:  let sep_tab = del /[ \t]+/ "\t"
 5:  let sep_spc = del /[ \t]+/ " "
 6:  let eol = del "\n" "\n"

 8:  let comment = [ del /#.*\n/ "# " ]
 9:  let word = /[^# \n\t]+/
10:  let host = [ seq "host" .
11:                [ label "ipaddr" . store  word ] . sep_tab .
12:                [ label "canonical" . store word ] .
13:                [ label "alias" . sep_spc . store word ]*
14:              . eol ]

16:  let lns = ( comment | host ) *

18:  let xfm = transform lns (incl "/etc/hosts")
```

Figure 1: The definition of the lenses used for mapping `/etc/hosts` into the tree and back.

```
127.0.0.1       localhost
192.168.0.2     server
# A comment
192.168.0.3     ns
```

(a) Restoring comments by position

```
127.0.0.1       localhost
192.168.0.1     router
# A comment
192.168.0.2     server
192.168.0.3     ns
```

(b) Initial `/etc/hosts`

```
127.0.0.1       localhost
# A comment
192.168.0.2     server
192.168.0.3     ns
```

(c) Restoring comments by key

Figure 2: Two possibilities of restoring comments in a changed file. After removing the tree node for `192.168.0.1` from the tree for the initial file *(middle)*, the tree can either be transformed so that the comment is restored at the same position *(left)*, or so that the comment is restored by its key *(right)*.

arbitrary number of aliases for a host. Lines starting with # are comments. Accordingly, the lens `lns` on line 16 processes any combination of matches for the `comment` and `host` lens.

The `comment` lens on line 8 deletes any line matching the regular expression `/#.*\n/`, i.e. anything from a starting # to the end of the line. Since AUGEAS requires that a file in its entirety is matched, there is no need to anchor regular expressions at the start and end of lines with `^` or `$`. The `del` primitive is enclosed in a subtree construct `[...]`; that causes the tree to contain a node with `NULL` label and value for every comment in the file. The reason for doing this has to do with how the *put* direction of *del* restores text: conceptually, the *get* direction of lenses produces a skeleton of the parsed text consisting of all the text deleted by the *del* lens with "holes" to fill in the parts stored in the tree. The *put* direction traverses the tree and fills the holes. The skeletons are associated with the parent node in the tree. If the comments were not their own tree node, AUGEAS would treat the whole `/etc/hosts` file as consisting of some comments with a fixed number of host entries between the comments. As an example, consider the initial file in Figure 2. After the initial file shown in the middle is read into the tree and the tree node corresponding to `192.168.0.1` is deleted, there are two ways in which the comment can be preserved when the tree is saved back to file: either by putting the comment after the second host entry (by position) as shown in Figure 4.3 or as coming after the entry for `127.0.0.1` but before the entry for `192.168.0.2` (by key). The former behavior results from *not* enclosing the *del* for `comment` in a subtree, the latter is the behavior of the lens in Figure 1.

The `host` lens on lines 10–14 in Figure 1 is straightforward in comparison: it stores host entries in separate subtrees, labelled with the number of the host entry. Each such subtree consists of a node labelled `ipaddr`, followed by a node labelled `canonical`, followed by zero or more nodes labelled `alias`. The value for each of the nodes is taken from splitting the line along spaces. The only difference between the `sep_tab` and `sep_spc` lenses that consume the whitespace between tokens on a line is how they behave when the tree is modified so that a brand new host entry is written to the file: `sep_tab` produces a tab character in that case, whereas `sep_spc` produces a space character.

The example in Figure 1 also illustrates two different ways to transform array-like constructs into the tree: the whole `/etc/hosts` file can be viewed as an array of lines of host entries (ignoring comments for the moment), and the aliases for each host are an array of space-separated tokens. For the former, we used the `seq` lens to produce a tree node for each host entry, wherease for the latter, we simply produce a new node with label `alias` for each alias. The reason for this is again connected to how formatting is preserved when entries are deleted from the tree or added to it. The former construct, using `seq` restores spacing by key, the number of the host entry in this case, whereas the latter restores it by position. When a new host entry is inserted into the tree under a new key, e.g. `10000`, all existing entries keep their spacing, since the skeletons for each entry are restored using that key. On the other hand, when a new alias for a host is inserted into the tree as the first alias for the host, the spacing is restored by position, so that the space between the (new) first and second alias is the same as the space that was in the initial file between the (old) first and second alias. Generally, it is preferrable to map arrays into the tree by using the same fixed label repeatedly, as is done for aliases here, since it makes the tree easier to manipulate, but often, considerations of what it means to preserve formatting in an "intuitive" way require that constructs using `seq` be used.

5.1 Lens development

When developing a lens for a new file format, the needed lens is gradually built up from simpler lenses and tested against appropriate text or tree fragments. For example, with the definitions from Figure 1, we can add a test

```
test lns get
  "127.0.0.1 localhost.localdomain localhost" = ?
```

to that same file. Running the modified file through `augparse` prints the tree resulting from applying the *get* direction of `lns` to the given string; `augparse` is a companion to `augtool` geared towards lens development.

There are two different kinds of tests: `test LNS get STR = RESULT` applies the *get* direction of `LNS` to `STR` and compares the resulting tree to `RESULT`, or prints it if `RESULT` is `?`. Conversely, `test LNS put STR after COMMANDS = RESULT` first applies the *get* direction of `LNS` to `STR`; it then changes the resulting tree using the `COMMANDS`, which can change the tree similar to `augtool`'s `set`, `rm`, and `ins` commands, and transforms the modified tree back to a string using the *put* direction of `LNS`. The test succeeds if this string equals the given `RESULT` string.

5.2 The typechecker

When configuration files are modified programmatically, ensuring that the changed configuration files are still valid is a major concern. AUGEAS contains a typechecker, very closely modelled on `Boomerang`'s [1] typechecker, that helps guard against common problems that could lead to invalid configuration files. Typechecking is performed statically—in other words, based solely on the schema description, to help weed out problems before any file is ever transformed according to that schema.

Typechecking happens in two phases: the first phase performs fairly standard checks that arguments to functions and operators have the type required by those functions and operators, for example, to ensure that the *subtree* operator `[...]` is only applied to lenses, and not to strings or regular expressions.

The second phase checks lenses for certain problems as they are constructed from simpler lenses. The details of those checks are based on the theoretical foundation laid by `Boomerang` [1] and make heavy use of computations on the regular languages matched by those lenses. In essence, the checks ensure that the lens laws GETPUT and PUTGET hold for any lens that is constructed.

Explaining these checks in detail would triple this paper in size; instead, let us just look at one of them to provide a taste of what the typechecker does. For two lenses l_1 and l_2, call the regular expressions they match in the *get* direction r_1 and r_2. The *get* direction of the concatenation $l = l_1 \cdot l_2$ of these two lenses matches the concatenation $r = r_1 r_2$ of their underlying regular expressions. When $l.get$ is applied to a string u matching r, it needs to split it into two strings $u = u_1 u_2$ and then pass u_1 to $l_1.get$ and u_2 to $l_2.get$. The two lenses l_1 and l_2 may do completely different things with these strings, for example, l_1 may be a *del* lens and l_2 a *store* lens. It is therefore imperative that there be no ambiguity in how u is split into two strings; otherwise, the way u is processed by l, and therefore the resulting tree, would depend on arcane implementation details of the split operation, and may change unexpectedly as code is changed.

The typechecker therefore checks every time a concatentation of two lenses is formed to ensure that the regular languages matched by them are *unambiguously concatenable*; in other words, that each string u matching r can be split in exactly one way in one part matching r_1 and one part matching r_2.

Similar checks are performed for iteration of lenses to ensure that a string matching an iterated lens l^* can be split in exactly one way into n pieces matching l. For the union $l_1|l_2$ of lenses, checks are performed to ensure that whether l_1 or l_2 is chosen is guaranteed to be unique.

The lens laws impose restrictions both on the *get* and the *put* direction of lenses, and both are enforced by the typechecker. The concatenation of two lenses is not only restricted by the requirement that any input string in the *get* direction can be split unambiguously, but also by the requirement that any tree in the *put* direction can be split unambiguously. Splitting (for concatenation and iteration) and choice (for union) of trees is performed solely on the labels of the immediate children of the node under consideration. This has the advantage that it is easy to implement, and can be easily reduced to checks similar to those for the *get* direction, but has the disadvantage that as far as the typechecker and the *put* direction of lenses are concerned, all tree nodes labeled `foo` are identical, no matter whether they are a leaf or whether they are the root of complicated subtrees. This simple approach to matching trees has not yet led to any significant problems in practice, but it is conceivable that a more sophisticated approach to trees and tree schemas is needed at some point in the not-too-distant future.

6 Future Work

The current implementation of AUGEAS is useful, but by no means complete. Improvements can be made in almost every area: first and foremost is the task of expanding the set of configuration files that AUGEAS can process "out-of-the-box."

Several limitations of the current implementation would be particularly interesting to remove. First, the public API lacks support for recursive matching. While path expressions can use the `*` wildcard operator, that operator does only match one level in the tree. An operator that matches multiple levels at once would be very useful, similar to the `**` extension to filename globbing in some programs, or the `//` operator in XPath expressions. Other changes to the public API, such as efficient iteration over large parts of the tree, are desirable.

The language currently misses the concept of permutations; for example, in some files entries take options, similar to shell commands. If individual options are processed by lenses $l_1, l_2, \ldots, l_n$, there is no convenient way in the language to construct a lens that matches a permutation of these n lenses; permutations either need to be written out manually or approximated with a construct like $(l_1|l_2|\ldots|l_n)^*$. Because of the combinatorial complexity involved, a straightforward implementation of permutations will be of limited use in practice. Instead, adding an operator like RelaxNG's `interleave` to the language seems more promising, but even that will require some special care to keep the runtime of the typechecker bearable.

AUGEAS can only handle file formats that can be described as a regular language. In particular, file formats that have constructs that can be nested arbitrarily deep can not be processed by AUGEAS. That is a fairly severe limitation in practice, as it precludes processing of the very popular `httpd.conf`: the Apache configuration allows some constructs, most notably `IfModule`, that can be nested to an arbitrary depths. While it is not terribly hard to expand the implementation to process such non-regular file formats, enhancing the typechecker to handle them is hard. The most promising approach is to expand the class of file formats that AUGEAS accepts only very slightly, e.g. by allowing *balanced languages*, but not all context-free languages, and basing typechecking such file formats off suitable regular approximations of the file format.

Another area of possible improvements are services built on top of AUGEAS: `system-config-boot`, one of the graphical configuration tools shipped with Fedora, contains some experimental code that separates the user interface from the logic changing `/etc/grub.conf` through DBus. The UI sends messages to a DBus activated service that checks the users credentials with PolicyKit and, if the users is authorized to make the change, uses Augeas to edit `/etc/grub.conf`. The backend does not need any specific knowledge about the file being edited, and it would be fairly easy to expand this code to add permissioning to configuration changes that distinguishes between different nodes in the Augeas tree, even if those nodes ultimately come from the same file.

In a similar vein, it would be interesting to investigate a remote-able configuration API built on top of Augeas, where a special daemon allows remote counterparts to modify a system's configuration.

Acknowledgments

We wish to thank Benjamin Pierce and Nathon Foster and their collaborators for their work on `Harmony` and `Boomerang`. Anders Moeller provided invaluable input, not the least through his `dk.brics.automaton` Java package, for the finite automata library used by the type checker.

References

[1] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: Resourceful lenses for string data. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Francisco, California*, January 2008.

[2] World Wide Web Consortium. XML Path Language (XPath) Version 1.0. `http://www.w3.org/TR/xpath`.

[3] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17, May 2007. Preliminary version presented at the *Workshop on Programming Language Technologies for XML (PLAN-X)*, 2004; extended abstract presented at *Principles of Programming Languages (POPL)*, 2005.

[4] Benjamin Pierce *et al.* Harmony. `http://alliance.seas.upenn.edu/~harmony`.

‘Real Time’ vs. ‘Real Fast’: How to Choose?

Paul E McKenney
IBM Linux Technology Center
paulmck@linux.vnet.ibm.com

Abstract

Although the oft-used aphorism “real-time is not real-fast” makes a nice sound bite, it does not provide much guidance to developers. This paper will provide the background needed to make a considered design choice between “real time” (getting started as quickly as possible) and “real fast” (getting done quickly once started). In many ways, “real fast” and “real time” are Aesop’s tortoise and hare, respectively. But in the real world of real time, sometimes the race goes to the tortoise and sometimes it goes to the hare, depending on the requirements as well as the details of the workload and the enclosing software environment.

1 Introduction

Linux™ has made much progress in the real-time arena over the past ten years, particularly with the advent of the -rt patchset [10], a significant fraction of which has now reached mainline. This naturally leads to the question of which workloads gain improved performance by running on real-time Linux. To help answer this question, we take a close look at the real-time vs. real-fast distinction in order to produce useful criteria for choosing between a real-time and non-real-time Linux.

Section 2 looks at a pair of example applications in order to make a clear distinction between real-time and real-fast, Section 3 examines some factors governing the choice between real-time and real-fast, and Section 4 gives an overview of the underlying causes of real-time Linux’s additional overhead. Section 5 lays out some simple criteria to help choose between real fast and real time, and finally, Section 6 presents concluding remarks.

2 Example Applications

This section considers a pair of diverse workloads, an embedded fuel-injection application and a Linux kernel build.

2.1 Fuel Injection

This rather fanciful fuel-injection scenario evaluates real-time Linux for controlling fuel injection for a mid-sized industrial engine with a maximum rotation rate of 1500 RPM. This is slower than an automotive engine; when all else is equal, larger mechanical artifacts move more slowly than do smaller ones. We will be ignoring complicating factors such as computing how much fuel is to be injected.

If we are required to inject the fuel within one degree of top dead center (the point in the combustion cycle where the piston is at the very top of the cylinder), what jitter can be tolerated in the injection timing? 1500 RPM is 25 RPS, which in turn is 9000 degrees per second. Therefore, a tolerance of one degree turns into a tolerance of one nine-thousandth of a second, or about 111 microseconds.

Such an engine would likely have a rotational position sensor that might generate an interrupt to a device driver, which might in turn awaken a real-time control process. This process could then calculate the time until top dead center for each cylinder, and then execute a sequence of `nanosleep()` system calls to control the timing. The code to actuate the fuel injector might be a short sequence of memory mapped I/O (MMIO) operations.

This is a classic real-time scenario. We need to do something before a deadline, and faster is most definitely *not* better. Injecting fuel too early is just as bad as injecting it too late. This situation calls for some benchmarking and validation of the `nanosleep()` system call, for example, with the code shown in Figure 1. On each pass through the loop, lines 2-5 record the start time, lines 6-9 execute the `nanosleep()` system call with the specified sleep duration, lines 10-13 record the end time, and lines 14-16 compute the jitter in microseconds and print it out. This jitter is negative if the `nanosleep()` call did not sleep long enough, and positive if it slept too long.

```
for (i = 0; i < iter; i++) {
  if (clock_gettime(CLOCK_MONOTONIC, &timestart) != 0) {
    perror("clock_gettime 1");
    exit(-1);
  }
  if (nanosleep(&timewait, NULL) != 0) {
    perror("nanosleep");
    exit(-1);
  }
  if (clock_gettime(CLOCK_MONOTONIC, &timeend) != 0) {
    perror("clock_gettime 2");
    exit(-1);
  }
  delta = (double)(timeend.tv_sec - timestart.tv_sec) * 1000000 +
    (double)(timeend.tv_nsec - timestart.tv_nsec) / 1000.;
  printf("iter %d delta %g\n", iter, delta - duration);
}
```

Figure 1: Loop to Validate nanosleep()

It is important to use `clock_gettime()` with the `CLOCK_MONOTONIC` argument. The more-intuitive `CLOCK_REALTIME` argument to `clock_gettime()` means “real” as in real-world wall-clock time, *not* as in real-time. System administrators and NTP can adjust real-world wall-clock time. If you incorrectly use `gettimeofday()` or `CLOCK_REALTIME` and the systems administrator sets the time back one minute, your program might fail to actuate the fuel injectors for a full minute, which will cause the engine to stop. You have been warned!

Before executing this validation code, it is first necessary to set a real-time scheduling priority, as shown in Figure 2. Line 2-5 invokes `sched_get_priority_max()` to obtain the highest possible real-time (`SCHED_FIFO`) priority (or print an error) and lines 6-9 set the current process’s priority. You must have appropriate privileges to switch to a real-time priority; either super-user or `CAP_SYS_NICE`. There is also a `sched_get_priority_min()` that gives the lowest priority for a given scheduler policy, so that `sched_get_priority_min(SCHED_FIFO)` returns the lowest real-time priority, allowing applications to allocate multiple priority levels in an implementation-independent manner, if desired.

However, real-time priority is not sufficient to obtain real-time behavior, because the program might still take page faults. The fix is to lock all of the pages into memory, as shown in Figure 3. The `mlockall()` system call will lock all of the process’s current memory down (`MCL_CURRENT`), and all future mappings as well (`MCL_FUTURE`).

Hardware irq handlers will preempt this code. However, the -rt Linux kernel has threaded irq handlers, which appear in the `ps` listing with names resembling `IRQ-16`. You can check their priority using the `sched_getscheduler()` system call, or by looking at the second-to-last field in `/proc/<PID>/stat`, where `<PID>` is replaced by the actual process ID of the irq thread of interest. It is possible to run your real-time application at a higher priority than that of the threaded irq handlers, but be warned that an infinite loop in such an application can lock out your irqs, which can cause your system to hang.

If you are running on a multi-core system, another way to get rid of hardware-irq latencies is to direct them to a specific CPU (also known as “hardware thread”). You can do this using `/proc/irq/<IRQ>/smp_affinity`, where `<IRQ>` is replaced by the irq number. You can then affinity your real-time program to some other CPU, thereby insulating your program from interrupt latency. It may be necessary to pin various kernel daemons to subsets of the CPUs as well, and the schedutils `taskset` command may be used for this purpose (though care is required, as some of the per-CPU kernel daemons really do need to run on the

```
sp.sched_priority = sched_get_priority_max(SCHED_FIFO);
if (sp.sched_priority == -1) {
  perror("sched_get_priority_max");
  exit(-1);
}
if (sched_setscheduler(0, SCHED_FIFO, &sp) != 0) {
  perror("sched_setscheduler");
  exit(-1);
}
```

Figure 2: Setting Real-Time Priority

```
if (mlockall(MCL_CURRENT | MCL_FUTURE) != 0) {
  perror("mlockall");
  exit(-1);
}
```

Figure 3: Preventing Page Faults

corresponding CPU). This has the downside of prohibiting your real-time program from using all of the CPUs, thereby limiting its performance. This technique is nonetheless useful in some cases.

Once we have shut down these sources of non-real-time behavior, we can run the program on both a real-time and a non-real-time Linux system. In both cases, we run on a four-CPU 2.2GHz x86 system running with low-latency firmware.

Even after taking all of these precautions, the non-real-time Linux fails miserably, missing the mark by up to 3 *milli*seconds. Non-real-time Linux systems are therefore completely inappropriate for this fuel-injection application.

As one might hope, real-time Linux does much better. Nanosleep always gets within 20 *micro*seconds of the requested value, and 99.999% of the time within 13 microseconds in a run of 10,000,000 trials. Please note that the results in this paper are from a lightly tuned system. More careful configuration (for example, using dedicated CPUs) might well produce better results.

If real-time Linux can so easily meet such an aggressive real-time response goal, it should do extremely well for more typical workloads, right? This question is taken up in the next section.

```
tar -xjf linux-2.6.24.tar.bz2
cd linux-2.6.24
make allyesconfig > /dev/null
time make -j8 > Make.out 2>&1
cd ..
rm -rf linux-2.6.24
```

Figure 4: Kernel Build Script

2.2 Kernel Build

Since the canonical kernel-hacking workload is a kernel build, this section runs a kernel build on both a real-time and a non-real-time Linux. The script used for this purpose is shown in Figure 4, featuring an 8-way parallel build of the 2.6.24 Linux kernel given an `allyesconfig` kernel configuration. The results (in decimal seconds) are shown on Table 1, and as you can see, real-time Linux is not helping this workload. The non-real-time Linux not only completed the build on average more than 15% faster than did the real-time Linux, but did so using less than half of the kernel-mode CPU time. Although there is much work in progress to narrow this gap, some of which will likely be complete before this paper is published, there is no getting around the fact that this is a large gap.

Clearly, there are jobs for which real-time Linux is not the right tool!

		Real Fast (s)	Real Time (s)
real	Raw Data	1350.4	1524.6
		1332.7	1574.2
		1314.5	1569.8
	Average	**1332.6**	**1556.2**
	Std. Dev.	14.6	22.4
user	Raw Data	3027.2	2940.9
		3013.1	2982.2
		2996.1	2971.2
	Average	**3012.2**	**2964.7**
	Std. Dev.	12.7	17.5
sys	Raw Data	314.7	644.3
		317.3	660.9
		317.9	665.9
	Average	**316.6**	**657.0**
	Std. Dev.	1.4	9.2

Table 1: Kernel Build Timings

2.3 Discussion

A key difference between these two applications is the duration of the computation. Fuel injection takes place in microseconds, while kernel builds take many seconds or minutes. In the fuel-injection scenario, we are therefore willing to sacrifice considerable performance in order to meet microsecond-scale deadlines. In contrast, even on a very fast and heavily tuned machine, handfuls of milliseconds are simply irrelevant on the kernel-build timescale.

The next section will look more closely at these issues.

3 Factors Governing Real Time and Real Fast

In the previous section, we saw that the duration of the work is a critical factor. Although there are a few exceptions, real-time response is usually only useful when performing very short units of work in response to a given real-time event. If the work unit is going to take three weeks to complete, then starting the work a few milliseconds late is unlikely to matter much. This relationship is displayed in Figure 5 for work-unit durations varying from one microsecond on the far left to 100 millisecond on the far right, where smaller latencies are better. The y-axis shows the total delay, including the scheduling latency and the time required to perform the unit of work. If the unit of work to be done is quite

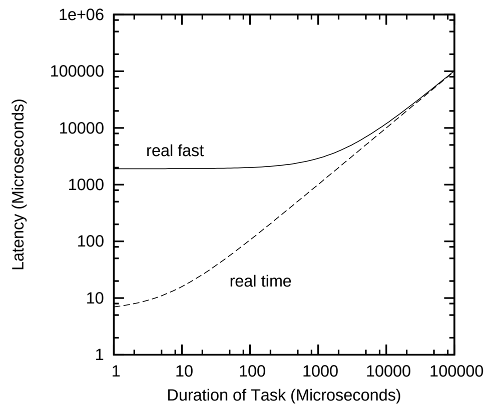


Figure 5: Real Time vs. Real Fast Against Work-Unit Duration for User-Mode Computation

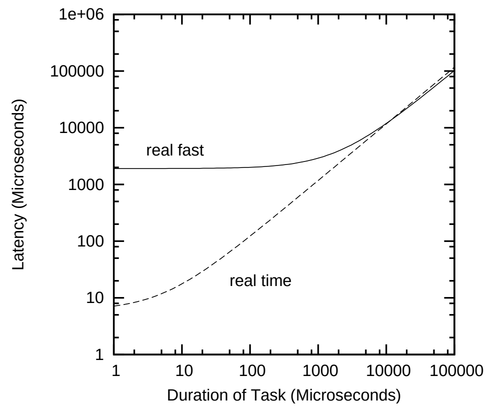


Figure 6: Real Time vs. Real Fast Against Work-Unit Duration for Kernel Build

small, a real-time system will out-perform a non-real-time system by orders of magnitude. However, when the duration of the unit of work exceeds a few tens of milliseconds, there is no discernable difference between the two.

Furthermore, Figure 5 favors the real-time system because it assumes that the real-time system processes the unit of work at the same rate as does the non-real-time system. However, in the kernel-build scenario discussed in Section 2.2, the non-real-time Linux built the kernel 16.78% faster than did the real-time Linux. If we factor in this real-time slowdown, the non-real-time kernel offers slightly *better* overall latency than does the real-time kernel for units of work requiring more than about ten milliseconds of processing, as shown in Figure 6. This breakeven would vary depending on the type of work. For example, floating-point processing speed

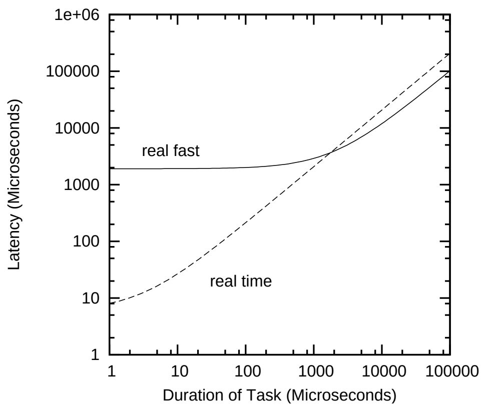


Figure 7: Real Time vs. Real Fast Against Work-Unit Duration for Heavy I/O

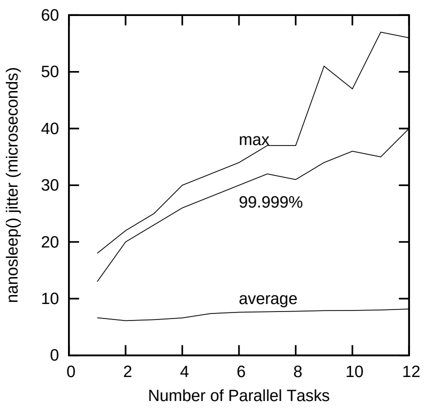


Figure 8: Nanosleep Jitter With Increased Load

would be largely independent of the type of kernel (and hence represented accurately by Figure 5), while heavy I/O workloads would likely be profoundly affected by the kernel type, as shown in Figure 7, which uses the 2-to-1 increase in kernel-build system time as an estimate of the slowdown. In this case, the crossover occurs at about one millisecond.

In addition, a concern with worst-case behavior should steer one towards real time, while a concern with throughput or efficiency should steer one towards real fast. In short, use real-time systems when the work to be done is both time-critical and of short duration. There are exceptions to this rule, but they are rare.

CPU utilization is another critical factor. To show this, we run a number of the `nanosleep()` test programs in parallel, with each program running 100,000 calls to `nanosleep` in a loop (code shown in Figure 1). Figure 8 shows the resulting average, 99.999 percentile delay, and maximum delay. The average jitter changes very little as we add tasks, which indicates that we are getting good scalability from a real-fast viewpoint. The 99.999 percentile and maximum delays tell a different story, as both increase by more than a factor of three as we go from a single task to 12 parallel tasks.

This is a key point: obtaining the best possible real-time response usually requires that the real-time system be run at low utilization. This is in direct conflict with the desire to conserve energy and reduce system footprint. In some cases, it is possible to get around this conflict by putting both real-time and non-realtime workload on the same system, but some care is still required. To illustrate this, run four parallel downloads of a kernel source tree onto the system, then unpack one of them and do a kernel build. When the `nanosleep` test program runs at maximum priority concurrently with this kernel-build workload, we see the 99.999% jitter at 59 microseconds with the worst case at 146 microseconds, which is worse than the parallel runs—but still much better than the multi-millisecond jitters from the non-real-time kernel.

Advancing technology can be expected to improve real-time Linux's ability to maintain real-time latencies in face of increasing CPU utilization, and careful choice of drivers and hardware might further improve the situation. Also, more-aggressive tuning might well produce better results. For example, this workload does not control the periodicity of the `nanosleep()` test programs, so that all 12 instances might well try to run simultaneously on a system that has but four CPUs. In real-world systems, mechanical constraints often limit the number of events that can occur simultaneously, in particular, engines are configured so that it is impossible for all cylinders to fire simultaneously. That said, sites requiring the best possible utilization will often need to sacrifice some real-time response.

Similarly, if you need to use virtualization to run multiple operating-system instances on a single server, you most likely need real fast as opposed to real time. Again, technology is advancing quite quickly in this area, especially in the embedded space, so we may soon see production-quality virtualization environments that can simultaneously support both real-time and real-fast operating systems. This is especially likely to work well if either: (1) CPUs and memory can be dedicated to a given operating instance or (2) the hypervisor (e.g., Linux with KVM) gives real-time response, but the guest operating systems need not do so. Longer term, it is quite possible that both the hypervisor and the guest OSes will offer real-time response.

4 Sources of Real-Time Overhead

The `nanosleep()` test program used the `mlockall()` system call to pin down memory in order to avoid page-fault latencies. This is great for this test program's latency, but has the side-effect of removing a chunk of memory from the VM system's control, which limits the system's ability to optimize memory usage. This can degrade throughput for some workloads.

Real-time Linux's more-aggressive preemption increases the overhead of locking and interrupts [2]. The reason for the increased locking overhead is that the corresponding critical sections may be preempted. Suppose that a given lock's critical section is preempted, and that each CPU subsequently attempts to acquire the lock. Non-real-time spinlocks would deadlock at this point. The CPUs would each spin until they acquired the lock, but the lock could not be released until the lock holder got a chance to run. Therefore, spinlock-acquisition primitives must block if they cannot immediately acquire the lock, resulting in increased overhead. The need to avoid priority inversion further increases locking overhead. This overhead results in particularly severe performance degradation for some disk-I/O benchmarks, however, real-time adaptive spinlocks may provide substantial improvements [4]. In addition, the performance of the user-level `pthread_mutex_lock()` primitives may be helped by private futexes [5].

Threaded interrupts permit long-running interrupt handlers to be preempted by high-priority real-time processes, greatly improving these processes' real-time latency. However, this adds a pair of context switches to each interrupt even in absence of preemption, one to awaken the handler thread and another when it goes back to sleep, and furthermore increases interrupt latency. Devices with very short interrupt handlers can specify `IRQF_NODELAY` in the `flags` field of their `struct irqaction` to retain the old hardirq behavior, but this is not acceptable for handlers that run for more than a small handful of microseconds.

Linux's $O(1)$ scheduler is extremely efficient on SMP systems, as a given CPU need only look at its own queue. This locality reduces cache thrashing, yielding extremely good performance and scalability, aside from infrequent load-balancing operations. However, real-time systems often impose the constraint that the N highest-priority runnable tasks be running at any given point in time, where N is the number of online CPUs. This constraint cannot be met without global scheduling, which re-introduces cache thrashing and lock contention, degrading performance, especially on workloads with large numbers of runnable real-time tasks. In the future, real-time Linux is likely to partition large SMP systems, so that this expensive global scheduling constraint will apply only within each partition rather than across the entire system.

Real-time Linux requires high-resolution timers with tens-of-microseconds accuracy and precision, resulting in higher-overhead timer management [3, 6]. However, these high-resolution timers are implemented on a per-CPU basis, so that it is unlikely that this overhead will be visible at the system level for most workloads. In addition, real-time Linux distinguishes between real-time "timers" and non-real-time "timeouts," and only the real-time timers use new and more-expensive high-resolution-timer infrastructure. Timeouts, for example, TCP/IP retransmission timeouts, continue to use the original high-efficiency timer-wheel implementation, further reducing the likelihood of problematic timer overheads.

Real-time Linux uses preemptible RCU, which has slightly higher read-side overhead than does Classic RCU [8]. However, the read-side difference is unlikely to be visible at the system level for most workloads. In contrast, preemptible RCU's update-side "grace-period" latency is significantly higher than that of RCU classic [7]. If this becomes a problem, it should be possible to expedite RCU grace period, albeit incurring additional overhead. It may then be possible to retire the Classic RCU implementation [9], but given that Classic RCU's read-side overhead is exactly zero, careful analysis will be required before such retirement can be appropriate.

In summary, the major contributors to the higher overhead of real-time Linux include increased overhead of locking, threaded interrupts, real-time task scheduling, and increased RCU grace-period latency. The next section gives some simple rules that help choose between the real fast non-real-time Linux kernel and the real-time Linux kernel.

5 How to Choose

The choice of real time vs. real fast is eased by considering the following principles:

1. Consider whether the goal is to get a lot of work done (real fast throughput), or to get a little bit of work done in a predictable and deterministic timeframe (real-time latency).

2. Consider whether the hardware and software can accommodate the heaviest possible peak load without missing deadlines (real time), or whether occasional peak loads will degrade response times (real fast). It is common real-time practice to reserve some fraction of resources, for example, to limit CPU utilization to 50%.

3. Consider memory utilization. If your workload oversubscribes memory, so that page faults will occur, you cannot expect real-time response.

4. If you use virtualization, you are unlikely to get real-time response—though this may be changing.

5. Consider the workload. A process that executes normal instructions in user mode will incur a smaller real-time average-overhead penalty than will a process that makes heavy use of kernel services.

6. Focus on work-item completion time instead of on start time. The longer the work item's execution time, the less helpful real-time Linux will be.

The need to focus on deterministic work-item completion cannot be stressed enough. Common practice in the real-time arena is to focus on when the work-item starts, in other words, on scheduling latency. This is understandable, given the historic separation of the real-time community into RTOS and real-time application developers, both working on proprietary products. It is hoped that the advent of open-source real-time operating systems will make it easier for developers to take the more global viewpoint, focusing on the time required for the application to both start *and* finish its work. Please note that it is important to focus on the proper level of detail, for example, event-driven systems should analyze deadlines on a per-event basis.

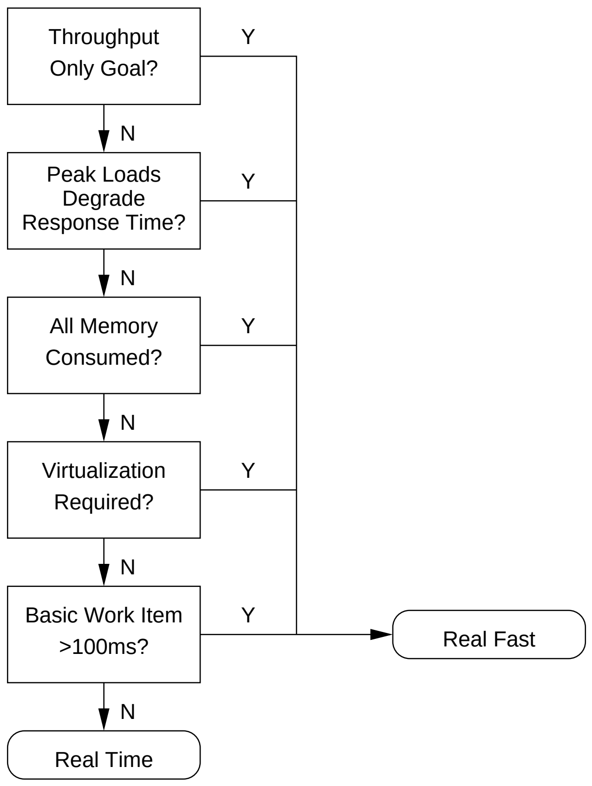


Figure 9: Real Time vs. Real Fast Decision Flow

A rough rule-of-thumb decision flow is shown in Figure 9. If you only care about throughput—the amount of work completed per unit time—then you want real fast. If cost, efficiency, or environmental concerns force you to run at high CPU utilization so that peak loads degrade response times, then you again want real fast—and as a rough rule of thumb, the more aggressive your real-time workload, the lower your CPU utilization must be. One exception to this occurs in some scientific barrier-based computations, where real-time Linux can reduce OS jitter, allowing the barrier computations to complete more quickly—and in this case, because floating point runs at full speed on real-time Linux, this is one of those rare cases where you get *both* real fast *and* real time simultaneously. If your workload will fill all of memory, then the `mlockall()` system call becomes infeasible, forcing you to either purchase more memory or allow the resulting page faults force you to go with real fast. Given the current state of the art, if you need virtualization, you are most likely in real-fast territory—though this may soon be changing, especially for carefully configured systems. Finally, if each basic item of work takes hundreds of milliseconds, any scheduling-latency benefit from real-time Linux is likely to be lost in the noise.

If you reach the real-time bubble in Figure 9, you may need some benchmarking to see which of real time or real fast works best for your workload. No benchmarking is needed to see that a workload requiring (say) 100 microseconds of processing with a 250-microsecond deadline will require real-time Linux, and there appears to be no shortage of applications of this type. In fact, it appears that real-time processing is becoming more mainstream. This is due to the fact that the availability of real-time Linux has made it easier to integrate real-time systems into enterprise workloads [1], which are starting to require increasing amounts of real-time behavior. Where traditional real-time systems were standalone systems, modern workloads increasingly require that the real-time systems be wired into the larger enterprise.

6 Concluding Remarks

If you remember only one thing from this paper, let it be this: "use the right tool for the job!!"

Ongoing work to reduce the overhead of real-time Linux will hopefully reduce the performance penalty imposed by the real-time kernel, which will in turn make real-time Linux the right tool for a greater variety of workloads.

Might real-time Linux's performance penalty eventually be reduced to the point that real-time Linux is used for all workloads? This outcome might seem quite impossible. On the other hand, any number of impossible things have come to pass in my lifetime, including space flight, computers beating humans at chess, my grandparents using computers on a daily basis, and a single operating-system-kernel source base scaling from cell phones to supercomputers. I have since learned to be exceedingly careful about labeling things *impossible*.

Impossible or not, here are some challenging but reasonable intermediate steps for the Linux kernel, some of which are already in progress:

1. Reduce the real-time performance penalty for multiple communications streams.
2. Reduce the real-time performance penalty for mass-storage I/O. (This becomes more urgent with the advent of solid-state disks.)
3. Reduce the preemptable RCU grace-period latency penalty.
4. Where feasible, adjust implementation so that performance penalties are incurred only when there are actually real-time tasks in the system.

It will also likely be possible to further optimize some of the real-time implementations. In any case, real-time Linux promises to remain an exciting and challenging area for some time to come.

Acknowledgements

No article mentioning the -rt patchset would be complete without a note of thanks to Ingo Molnar, Thomas Gleixner, Sven Deitrich, K.R. Foley, Gene Heskett, Bill Huey, Esben Neilsen, Nick Piggin, Steven Rostedt, Michal Schmidt, Daniel Walker, and Karsten Wiese. We all owe Ankita Garg, Vara Prasad, Ananth Mavinakayanahalli, Chirag Jog, and especially Darren Hart and Paul Giangarra a debt of gratitude for their help in making this paper human-readable. I am grateful to Paul Clarke for the use of his machines, and the ABAT team for providing easy access to these machines. Finally, I owe thanks to Daniel Frye and Kathy Bennett for their support of this effort.

Legal Statement

References

[1] BERRY, R. F., MCKENNEY, P. E., AND PARR, F. N. Responsive systems: An introduction. *IBM Systems Journal 47*, 2 (April 2008), 197–206.

[2] CORBET, J. Approaches to realtime Linux. Available: `http://lwn.net/Articles/106010/` [Viewed March 25, 2008], October 2004.

[3] CORBET, J. A new approach to kernel timers. Available: `http://lwn.net/Articles/152436/` [Viewed April 14, 2008], September 2005.

[4] CORBET, J. Realtime adaptive locks. Available: `http://lwn.net/Articles/271817/` [Viewed April 14, 2008], March 2008.

[5] DUMAZET, E. [PATCH] FUTEX : new PRIVATE futexes. Available: `http://lkml.org/lkml/2007/4/5/236` [Viewed April 18, 2008], April 2007.

[6] GLEIXNER, T., AND MOLNAR, I. [announce] ktimers subsystem. Available: `http://lwn.net/Articles/152363/` [Viewed April 14, 2008], September 2005.

[7] GUNIGUNTALA, D., MCKENNEY, P. E., TRIPLETT, J., AND WALPOLE, J. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux. *IBM Systems Journal 47*, 2 (May 2008), 221–236. Available: `http://www.research.ibm.com/journal/sj/472/guniguntala.pdf` [Viewed April 24, 2008].

[8] MCKENNEY, P. E. The design of preemptible read-copy-update. Available: `http://lwn.net/Articles/253651/` [Viewed October 25, 2007], October 2007.

[9] MCKENNEY, P. E., SARMA, D., MOLNAR, I., AND BHATTACHARYA, S. Extending RCU for realtime and embedded workloads. In *Ottawa Linux Symposium* (July 2006), pp. v2 123–138. Available: `http://www.linuxsymposium.org/2006/view_abstract.php?content_key=184` `http://www.rdrop.com/users/paulmck/RCU/OLSrtRCU.2006.08.11a.pdf` [Viewed January 1, 2007].

[10] MOLNAR, I. Index of /mingo/realtime-preempt. Available: `http://www.kernel.org/pub/linux/kernel/projects/rt/` [Viewed February 15, 2005], February 2005.

If I turn this knob. . . what happens?

Arnaldo Carvalho de Melo
Red Hat Inc.
acme@{redhat.com,ghostprotocols.net}

Abstract

Characterizing problems in systems with lots of configuration knobs while trying versions of components can be an error-prone task. System and application configuration details such as kernel boot options, SMP affinity, NIC and scheduler settings, /proc and /sys filesystem entries, lock_stat data, and other items can prove vital. Software to collect this information in a database, correlating to application performance numbers for automated and visual analysis, is needed to help in this process.

Work in this direction is presented in this paper, showing how changes in system tunings compare to previous results in the database. By automating the collection of performance numbers together with environment tunings, it helps in noticing trends in system behavior as system components evolve.

1 Introduction

Characterizing a performance or latency problem, benchmarking, testing new versions of system components or a new machine—all these require storing the results in a database or spreadsheet for comparisons. Creating graphics from the collected data also helps in this process.

The number of system (software and hardware) knobs keeps growing, and it is easy to overlook one setting and then have difficulty in reproducing it on another system with supposedly the same hardware and software components, as is common when trying to obtain help from fellow developers, a company help desk, or the support services of a vendor.

Automatically recreating a set of tunings after the update of one of the system hardware or software components and comparing the results of a series of benchmarks with previous results is important in the life cycle of any software.

A small variation in performance, latency, memory usage, and several other software metrics after the update of a component is usually acceptable. It is thus possible that continuous degradation of a metric is unnoticed over several development cycles as the base hardware used also gets upgraded.

This paper will describe efforts in providing software for reading and storing system settings in a database, independently from how these changes were performed, be it using basic system tools such as chrt or taskset, or using higher level tools, such as tuna, that will also be described.

Ways to do live analysis of changes on system components such as changing the scheduler policy, priority, and processor affinity of threads and interrupts will also be presented; as well as running benchmarks for post processing, storing results in a database, and then generating reports showing the sets of tunings that provided the best results; as well as graphics showing results from several sets of tunings.

2 Automated Testing

I started working in this area when trying to characterize performance degradations found when trying to run market data applications on the PREEMPT_RT Real Time enabled kernels.

Knobs such as enabling or disabling TSO (TCP Segmentation Offload), using private futexes by upgrading the system C library, disabling or enabling IRQ balancing, setting the affinity and priority of the hard IRQ threads, making sure that oprofile, systemtap or lock_stat[1] were not enabled, trying new patches by fellow developers, and many others quickly added up to make me crazy when trying to compare results.

Many times this leads to having to re-run tests already performed due to forgetting whether one of these many knobs had been set.

To help in comparing the results, I started working on a set of software components, mostly written in the Python language.

Some system interfaces lacked Python bindings, so one of the first tasks performed was to fill this gap.

Bindings for the interfaces exposed through the schedutils package were written, python-schedutils [3], allowing getting and setting the scheduler policy, real time priority, and processor affinity of threads.

Another Python binding, python-ethtool [4], allows getting and setting network interface drivers knobs such as TSO[1] and UFO;[2] these are hardware assists for common TCP and UDP operations that can greatly improve performance, but inherently can add delay as the network stack waits for more application buffers to coalesce into one big segment to send to the ethernet card in just one transfer. Disabling these features is one thing usually tried when characterizing a problem, as it involves software implementations both in the OS network stack and on the NIC firmware—two places where bugs can happen. Also, the interaction of these two software implementations can be a source of problems.

There is also a lot of information available in the `/proc` filesystem, that despite its initial goals of providing information about the processes in the system, has been overloaded with all sorts of system-wide information. A library that turns several areas of the information found there into Python dictionaries was also written.

Classes for turning `/proc/pid/stats` and status into dictionaries, for instance, are used by the other components to sample information such as the number of voluntary and involuntary context switches experienced by threads.

The sysctl information in `/proc/sys` is also turned into dictionaries and the ones that are changeable by the administrator (directly, using a simple `echo` shell command, or through higher level tools) are stored in a set of database tables, one per a selected set of `/proc/sys` subdirectories.

A tool that collects the state of this subset of sysctl settings and assigns a unique numeric identifier was also written.

[1]TCP Segmentation Offload

[2]UDP Fragmentation Offload

This tool initially was used to store more than just sysctl settings, with extra information such as if system analysis tools such as `lock_stat`, oprofile, or systemtap were in use, or the options passed through the kernel command line.

The tool is being rewritten so that it can be used just for sysctls, with another like-minded tool to be made available for other, non-sysctl knobs.

To better illustrate the use of such tool, here are some examples:

```
$ tuneit --show 1
tcp_congestion_control: bic
kcmd_idle: None
lock_stat: False
tso: lo=0,eth0=1
app_sched: SCHED_RR
app_affinity: ff
systemtap: False
vsyscall64: 1
kcmd_maxcpus: None
irqbalance: True
app_rtprio: 51
oprofile: False
$
```

This shows the set of tunings recorded in the database associated with the unique numeric identifier *1*.

To see what changed from the first set of tunings to the second one:

```
$ tuneit --diff 1,2
Tunings#: 1
  tso: lo=0,eth0=1
Tunings#: 2
  tso: lo=0,eth0=0
$
```

Only TSO was changed, being enabled on the first set of tunings for the `eth0` interface and disabled on the second set of tunings.

This tool can be used as well for identifying the sets of tunings where a knob had a particular value. For instance:

```
$ tuneit --query "vsyscall64=0"
71,111
```

Some work was done on allowing this tool to be used for replaying a specific set of tunings:

```
$ cat /proc/sys/kernel/vsyscall64
1
$ tuneit --replay 71
$ cat /proc/sys/kernel/vsyscall64
0
$
```

The settings that can not be replayed at run time, such as kernel command line options or the presence of kernel features such as lock_stat, will instead generate a warning, so that the user is aware when comparing the ensuing results.

Currently this suite also checks if lock_stat is enabled in the kernel, resetting it before running the benchmarks, and storing the contents of /proc/lock_stat immediately after its completion, so that later on they can be examined. The same procedure will be implemented for oprofile, when requested, so that one more of the best practices used by performance workers can be automated, avoiding cases where such valuable information gets lost, even having been collected.

Other tools in this suite are used to collect other relevant information such as details about the machine and the system components installed in it:

```
[root@doppio ait]# ./ait-get-sysinfo
arch: x86_64
cpu_model: Intel(R) Core(TM)2 CPU
         T7200  @ 2.00GHz
futex_performance_hack: None
irqbalance: False
kcmd_idle: None
kcmd_isolcpus: None
kcmd_maxcpus: None
kcmd_nohz: None
kernel_release: 2.6.25-rc8
lock_stat: False
nic_kthread_affinities: eth0=3
nic_kthread_rtprios:
nodename: doppio.ghostprotocols.net
nr_cpus: 2
oprofile: False
softirq_net_rx_prio:
softirq_net_tx_prio:
systemtap: False
tcp_congestion_control: bic
tcp_dsack: 1
tcp_sack: 1
tcp_window_scaling: 1
tso: lo=0,eth0=1,pan0=1
ufo: lo=0
vendor_id: GenuineIntel
vsyscall64: 0
[root@doppio ait]#
```

Existing tools such as sysreport will probably be used in the future, as they provide more information, although they take a considerably longer time to collect it.

Finally, to provide the information required for the reporting tools, the benchmark results are stored in the database, correlated with all the above information about the system hardware, software, and the respective settings of both.

One can report the best results in a textual form:

```
$ rit.py db perf4-1.lab.redhat.com \
            perf7-1.lab.redhat.com

server: perf4-1.lab.redhat.com

client: perf7-1.lab.redhat.com

latest report info:
  report id: 504
  kernel: 2.6.18-53.1.14.el5
  max rate: 25000

max rates per kernel release:

2.6.18-88.el5        : 100000
2.6.24-17.el5rt      : 97000
2.6.24-20.el5rt      : 100000
2.6.24-21.el5rt      : 94000
2.6.24-22.el5rt      : 64000
2.6.24.1-24.el5rt    : 100000
2.6.24.3-rt3.rwmult2: 100000
2.6.24.4-30nommapsem: 100000
2.6.24.4-41.el5rt    : 97000

rate: 1000
----------------------
Shared System tunings:

ufo: lo=0,eth0=0,eth1=0,eth2=0
softirq_net_tx_pri: 90,...,90
softirq_net_rx_pri: 90,...,90
app_rtprio: 0
irqbalance: False
app_affinity: ff
app_sched: SCHED_OTHER
kcmd_isolcpus: None
nic_kth_aff: eth1=ff;eth2=ff
nic_kth_rtpri: eth1=95;eth2=95
oprofile: False
systemtap: False
kcmd_maxcpus: None
futex_performance_hack: 0
kcmd_idle: poll
lock_stat: False
tcp_congestion_control: bic
client: perf7-1.lab.redhat.com
```

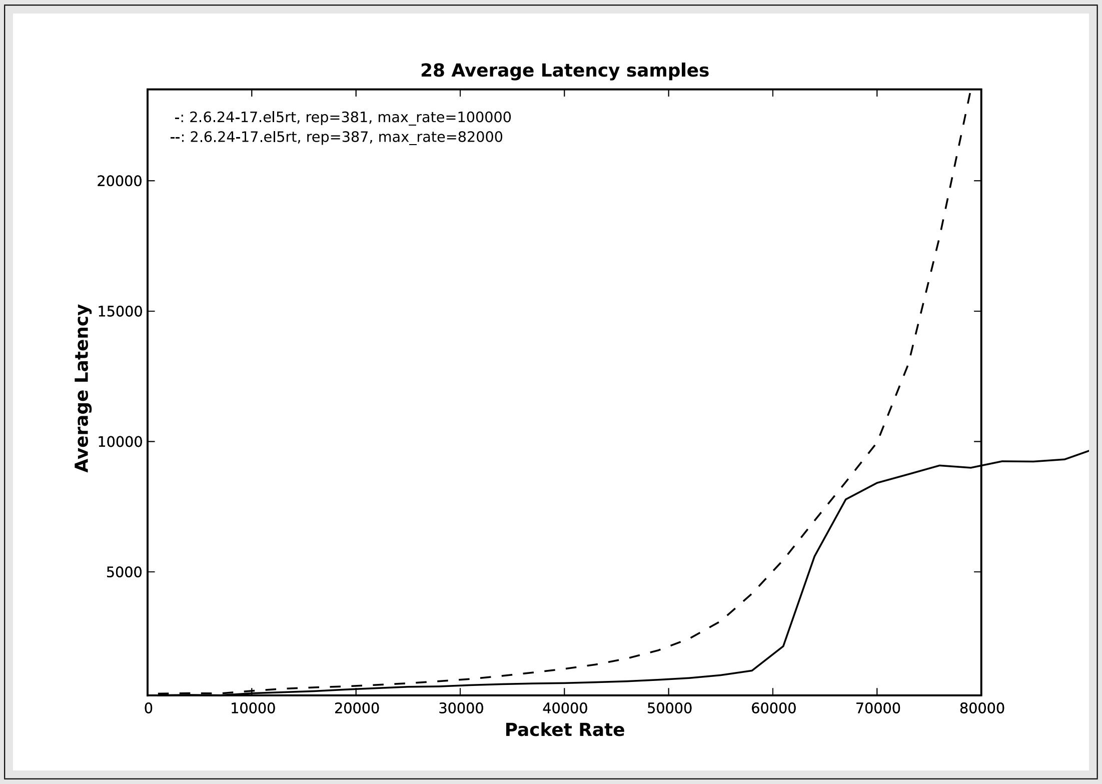


Figure 1: Solid line: libc-2.7.90 (uses private futexes), Dashed line: libc-2.5

```
Different system tunings:

env|tun|kernel    |avglat|   tso|vsc64|nhz
128|105| .24-22rt|249.38|eth2=1|   1 |  0
135|109| .24-20rt|252.41|eth2=1|   1 |off
127|104| .24-22rt|254.34|eth2=0|   1 |  0
136|109|24.1-24rt|258.53|eth2=1|   1 |off
134|108| .24-21rt|259.54|eth2=1|   1 |off
133|108|24.1-24rt|262.04|eth2=1|   1 |off
138|111|24.1-24rt|265.08|eth2=1|   0 |off
129|106| .24-22rt|266.51|eth2=1|   1 | on
130|106| .24-17rt|266.51|eth2=1|   1 | on
132|108| .24-17rt|267.97|eth2=1|   1 |off
```

The common set of tunings for all the 10 best test results is shown, followed by what really changed, and then the average latencies, the metric in this particular test.

Another result that can be generated is a set of graphs that show several benchmark results for visual comparison, as in Figure 1. This compares two versions of the system C library, one that uses private futexes and an older one that does not, on a system with 8 cores.

Another graphical report, this time with more than just two test results, is found in Figure 2, where the diamonds and squares lines are the ones with the old glibc, and all the tests have `lock_stat` on. The `html` file that includes these graphics has a link for the respective `/proc/lock_stat` data, where we can see that there is contention for `mmap_sem`, illustrated in Figure 3.

3 tuna

Also written was `tuna` [2], a tool to allow tweaking scheduler parameters, performing techniques such as CPU isolation.

It has three main boxes, one that shows the load for each CPU in the system, another with the interrupt sources, and the last one displaying the threads in the system.

Users can drag interrupt sources and threads into a CPU, setting the affinity of the dragged entities to that CPU.

Tuna also allows the user to right click on a CPU and isolate it, removing it from the CPU affinity masks of all threads and interrupt sources. It is also possible to do the opposite operation, including a CPU into the affinity masks of all interrupt sources and threads in the system.

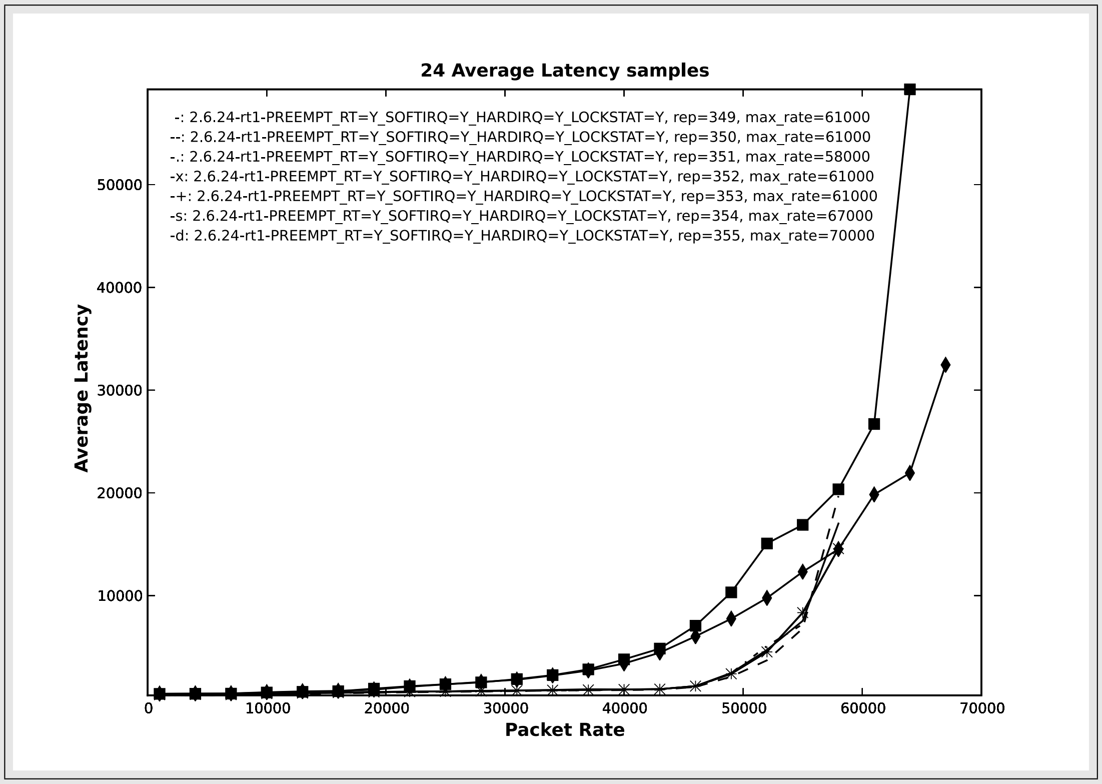


Figure 2: Diamonds and Squares: libc-2.5, Others: libc-2.7.90 (uses private futexes)

Usually the sequence is to isolate a CPU and then move some specific threads to that CPU so as to keep the isolated CPU cache hot, reducing accesses to main memory for a critical thread or set of threads.

More work is required to group cores per socket, for CPUs where caches are shared by the cores in a multi-core CPU socket. It then will be possible to move threads or interrupt sources to a socket and not just to a core.

The infrastructure put in place for multi-core CPUs will also allow creating groups that include not just cores in a socket, but any arbitrary grouping that is deemed useful for a particular purpose. That would permit arrangements such as dedicating two cores in a socket to a particular purpose, and the other two (assuming a quad-core CPU) for other purposes.

This will also help on big NUMA systems that have different costs for accessing memory that is one or more hops away from a particular core.

While it is understood that a general-purpose kernel tries hard to cope with all the underlying details on multi-core systems and NUMA topologies, it is generally considered useful to have such functionality for experimentation. As the complexity of such systems grows, a tool that exploits GUI facilities and provides higher level, simpler interfaces for performing these operations should be of help.

In trying to help a wider audience to understand more about the components in the kernel, `tuna` provides context-sensitive help, so far only for kernel threads. Right clicking on a kernel thread line in the threads box and clicking on the *What is This?* option opens up a window with information about it.

It is also possible to filter out kernel threads or user threads and sort by all the columns, and in the future it should be possible to add more columns from the fields found in the dictionaries built from the `/proc` files.

Most of the operations available in the GUI are also available on the command line, allowing its use in systems without graphical libraries.

```
lock_stat version 0.2
-------------------------------------------------------------------------------
    class name con-bounces contentions waittim-min  waittime-max waittime-total
-------------------------------------------------------------------------------

mm->mmap_sem-W     198626      265946         0.94      53724.80    17407373.03
mm->mmap_sem-R   21179643    49780404         0.74     299766.89  8739012694.54
--------------
  mm->mmap_sem   50039855 [<ffffffff802626ca>] rt_down_read+0xb/0xd
  mm->mmap_sem       4126 [<ffffffff80292c68>] sys_mprotect+0xce/0x22e
  mm->mmap_sem          0 [<ffffffff80211e7e>] sys_mmap+0xcf/0x119
  mm->mmap_sem          0 [<ffffffff80291014>] sys_munmap+0x32/0x59

...............................................................................

lock->wait_lock  39064428    40206507         0.94        316.86    56173839.20
---------------
lock->wait_lock   3584410 [<ffffffff804a1aed>] rt_spin_lock_slowunlock+0xf/0x5c
lock->wait_lock   8990657 [<ffffffff804a1bde>] rt_spin_lock_slowlock+0x21/0x19e
lock->wait_lock   3742935 [<ffffffff80262383>] rt_mutex_slowtrylock+0x18/0x79
lock->wait_lock    205398 [<ffffffff80262812>] rt_up_read+0x26/0x66

...............................................................................

dev->queue_lock#2 5918095     8546238         0.86      99690.79   506180455.19
-----------------
dev->queue_lock#2 1385877 [<ffffffff8043db2c>] __qdisc_run+0xa4/0x185
dev->queue_lock#2 7160361 [<ffffffff8042ddae>] dev_queue_xmit+0x12d/0x29f
dev->queue_lock#2       0 [<ffffffff8042d45d>] net_tx_action+0xbc/0xf3
```

Figure 3: Edited `lock_stat` data showing `mmap_sem` contention in report 354

4 Oscilloscope

The companion to `tuna` is an oscilloscope application. It should be fed with a stream of values that will be plotted on the screen together with a histogram.

The goal here is to be able to instantly see how a signal generator, such as `cyclictest`, `signaltest`, or even `ping`, reacts when (for instance) its scheduling policy or real time priority is changed, be it using `tuna` or plain `chrt`.

If the ftrace [5] feature is built into the running kernel, the oscilloscope classes will take snapshots of `/sys/kernel/debug/tracing/trace` and associate it with the position on the screen where the sample appears. So when the user clicks on the vicinity of such a sample, it will pop up a window with the ftrace-collected functions, usually what happened in the kernel while preemption and interrupts were disabled, causing the latency spike.

5 Future Directions

Integration with qpid [6] is planned, to get or set parameters on remote machines.

When this integration is complete, `tuna` will be just one of the interfaces to change knobs, another one being the AMQP Management Console, part of the qpid project [6].

Being able to use inventory systems for data about the systems used in the tests is also something to be considered.

The `tuneit` tool described earlier in this paper will be augmented to allow starting a tuning session. There it will look at all threads and interrupt sources in the system, recording the current scheduler policy, real time priority, and affinities. Then, after a `tuna`, plain `chrt` and `taskset`, or any other method, it will compare the new settings and record the changes in the database, allowing the settings to be replayed on the same machine or on another.

6 Conclusion

The activities that this paper describes should be familiar to many readers; ad-hoc ways to accomplish these goals probably have been performed by most.

I hope that by describing his efforts in this direction and talking about requirements for further usability encourages interested people to join forces with him in working on improving this infrastructure for wider use, saving work for people with similar needs.

Test results for more benchmarks (such as AMQP [6], netperf, and other open source benchmarks) will be performed and should be publicly available by July, in time for OLS 2008.

References

[1] lock stat documentation in the kernel sources `Documentation/lockstat.txt`

[2] tuna git repository `http://git.kernel.org/?p=linux/kernel/git/acme/tuna.git`

[3] python-schedutils git repository `http://git.kernel.org/?p=linux/kernel/git/acme/python-schedutils.git`

[4] python-ethtool git repository `http://git.kernel.org/?p=linux/kernel/git/acme/python-ethtool.git`

[5] ftrace tracing infrastructure `http://lwn.net/Articles/270971/`

[6] qpid project `http://cwiki.apache.org/qpid/`

Performance Inspector Tools with Instruction Tracing and Per-Thread / Function Profiling

Milena Milenkovic, Scott T. Jones, Frank Levine, Enio Pineda
International Business Machines Corporation
`{mmilenko, stjones, levinef, enio}@us.ibm.com`

Abstract

The open-source Performance Inspector™ project contains a suite of performance analysis tools for Linux® which can be used to help identify performance problems in Java™ and C/C++ applications, as well as help determine how applications interact with the Linux kernel. One such tool is the Per-Thread Time Facility (PTT); it consists of a kernel module and user-space components which maintain thread statistics for time (cycles) or any of a number of predefined metrics. JProf is a Java/C/C++ profiler which uses PTT to produce reports with per-method/function metrics. Another tool is a Tracing Facility, which may be used for tracing instructions, thread dispatches, and sampling events. In this paper we present the details of the most commonly used Performance Inspector tools, targeting the audience of developers interested in performance fine-tuning.

1 Introduction

"I suggest you count your bees, you may find that one of them is missing."

—Inspector Clouseau,
Pink Panther Strikes Again

With growing software complexity, a performance analyst job is becoming increasingly difficult. The Performance Inspector project (PI) consists of a set of tools that helps with analyzing application and system performance on Linux. It includes a kernel driver module (pitrace) and various user-space applications and libraries. The project is hosted on SourceForge (`http://perfinsp.sourceforge.net`). PI currently includes support for the Intel x86, Intel and AMD x86_64, and IBM PowerPC64 and s390 platforms.

The PI toolset enables analysts to identify the overall processor utilization, and application/thread hardware counter summary information. PI provides support for both application and kernel sample-based profiling or instruction tracing. Sample-based profiling without full context information may not give enough analysis information to tune large applications, so PI provides method (Java) or function/subroutine (C/C++) tracing at the application level, relying on built-in Java Virtual Machine (JVM) support for method entries and exits and gcc compile options to generate entry/exit notifications. The Per Thread Time facility, together with the metrics calibration, allows for accurate per-method counts of stable metrics such as *instruction completed*. Because of the repeatability of this metric, it has been used to assign instruction budgets to application components. Utilizing the PI programmatic control of tracing and the consistency of the measurements, changes in component instruction budgets can be identified.

PI encompasses the following user-space components:

- The *libperfutil* library includes a set of APIs for communication with the pitrace driver and other utilities. The JPerf.jar package includes support for the equivalent Java interfaces: for example, you can turn instruction tracing on and off directly from a Java application, thus enabling fine-grain control of the traced code.
- *JProf (libjprof)* is a Java profiling agent that responds to events from either the JVM Profiler Interface (JVMPI) or the JVM Tool Interface (JVMTI), based on the invocation options. Common usages of JProf are:
  - Capturing execution flow in the form of method call trees or a method trace.
  - Resolving Just-In-Time (JIT) compiled code

```
Type and Length | Major Code | Minor Code | TimeStamp | Variable Data

 16 bit         | 16 bit     | 32 bit     | 32 bit    | variable length
```

Figure 1: Trace record format

addresses to method names to support trace post-processing.

  - Capturing the state of the Java Heap that can later be processed by the *hdump* PI application to help locate memory leaks.
  - Capturing information about IBM™ JVM usage of locks via the Java Lock Monitor (JLM).

- The *rtdriver* application is a socket-based command interface that enables interactive control of JProf, such as when to start or stop of profiling, or when to dump the contents of the Java Heap.

- The *swtrace* application is used to control the Tracing facility. This application is also used to invoke the AboveIdle tool—a lightweight tool which reports processor utilization (busy, idle, and interrupt time).

- The *post* application is used to convert binary trace files to readable reports using the *liba2n* (A2N—address-to-name) library, which converts addresses to symbolic names. This library may be used to convert addresses to names for dynamically generated code, such as the code generated by Java via the Just-In-Time (JIT) support; along with time-stamped tracing, it provides accurate symbolic resolution even when addresses are reused.

- Other tools include *msr*, used to read and write model-specific registers (MSR); *mpevt*, used to manipulate hardware performance counter events; *ptt*, used to give summary per-thread metric counts; and *cpi*, used to measure cycles per instruction (CPI) for an application or a time interval.

The rest of the paper is organized as follows. Section 2 explains the Tracing facility and what kind of information can be traced with it. Section 3 explains the inner workings of the Per-Thread Time Facility which provides per-thread metric virtualization, and Section 4 explains how metrics are adjusted for instrumentation overhead in the JProf profiler. Section 5 briefly describes how users can visualize some of the PI reports, and the last section gives directions for future project development.

2 Tracing Facility

PI includes support for a software tracing mechanism—the Tracing Facility. Although there are already established Linux tools with somewhat similar functionality, our tracing mechanism accurately captures information necessary for address-to-name symbol resolution of dynamically generated code such as Java JITed code. The main issue with the JITed code is that it can be recompiled and moved around the address space. The tracing mechanism combines kernel knowledge about memory segments for a process with JProf jita2n (JITed code address-to-name) synchronizing records and the corresponding dynamically generated code.

We consider two main groups of trace records: one group consists of Module Table Entry (MTE) records and the other group consists of all the other record types, such as ITrace and tprof. Each group uses a per-cpu buffer, i.e., there is an *MTE buffer* and a non-MTE buffer (*trace buffer*) allocated for each CPU in the pitrace driver. These buffers are pinned (allocated in pitrace kernel module) and memory mapped, so that `libperfutil` can read them directly. All trace records have a similar format, shown in Figure 1. The *Type and Length* field specifies the record length and the type of the *Variable Data* field. The *Major Code* field specifies a trace record type, for example, MTE, tprof, or ITrace. The *Minor Code* field specifies a subtype within a type, e.g., a process name MTE record. The TimeStamp field has the lower 32-bits of the time stamp, and a special trace record is written to indicate a change in the higher 32 bits.

Figure 2 shows a block scheme of PI components and files involved when tracing a Java application.

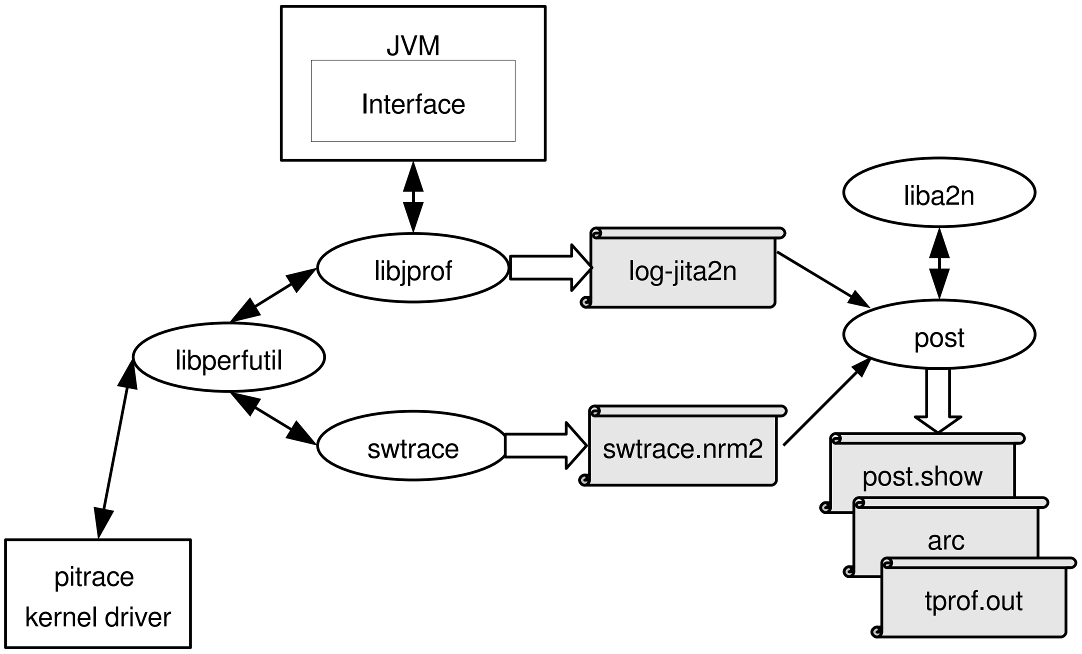


Figure 2: Block scheme for tprof/ITrace tracing of Java application

The `swtrace` application is the front-end which controls the tracing facility via `libperfutil` APIs. It is used to enable tracing of specific record types, specify the size of trace and MTE buffers, turn tracing on and off, write the content of trace buffers to a file, and select the Tracing facility mode. There are three possible modes: *normal*, *wrap-around*, and *continuous*. In the normal mode, tracing automatically stops when either an MTE buffer or a trace buffer becomes full. In the continuous mode, both MTE and trace buffer segments are written to a file when a segment size reaches a given threshold. In the wrap-around mode, meant to be used to analyze application crashes or the most recent application activity, MTE buffers are written continuously in a file, and other trace records wrap around the buffer. The default trace file name is `swtrace.nrm2`.

When initialized and turned on, the Tracing facility gets notifications about each task `exit` and `unmap`, using the existing kernel notification mechanism. When a task exits, we write its parent tree (if not already written), and if the task is not a clone, we also write a trace record for each of its mapped executable memory segments. Similarly, when a memory segment is unmapped, we write the parent tree and executable segment info for the corresponding task. When tracing is turned off, we write previously unwritten MTE data for all tasks still alive.

A trace record write can be initiated from the pitrace module, or from a user application, using the `libperfutil` `TraceHook()` function. The JProf profiler can use this function to trace start, stop, and name information for each Java thread; it then also writes the same information into a `log-jtnm` file. When a JITed method is loaded, JProf can write trace records with the method start address, current thread, and the current time stamp; it then also writes address to name translation info such as code address, method name, class name, time stamp, and possibly bytes of instructions, into a `log-jita2n` file. This information is used by post to resolve addresses of trace records to the correct Java method, class, and thread name. Post can create an ASCII version of a trace (`post.show`), a *tprof.out* report from `tprof` trace records, and an *arc* report from ITrace records. ITrace and `tprof` tracing mechanisms are explained in more details in the following subsections.

2.1 ITrace

To fully understand a complex performance issue, analysts sometimes need to see a full instruction trace. To get such a trace, we actually need to trace only taken branch instructions, using the underlying hardware support for trap on branch or taken branch. The only issue is that in some earlier 2.6 kernel distributions, trap

```
# arc Field Definition:
#   1: cpu no.
#   2: K(kernel) or U(user)
#   3: last instruction type
#     0=INTRPT, 1=CALL,  2=RETURN, 3=JUMP, 4=IRET, 5=OTHER, 6=UNDEFD, 7=ALLOC
#   4: no. of instructions
#   5: @|? = Call_Flow | pid_tid
#   6: offset (from symbol start)
#   7: symbol:module
#   8: pid_tid_pidname_[threadname]
#   9: last instruction type (string)
#  10: line number (if available)
...
  0 U 3      1 @  120 <plt>:/opt/ibm-java2-i386-50/jre/bin/libj9prt23.so   11c1_11c1_java_main JUMP 0
  0 U 3      2 @    0 __libc_write:/lib/libpthread-2.5.so   11c1_11c1_java_main JUMP 0
  0 U 1      1 @   2c __libc_write:/lib/libpthread-2.5.so   11c1_11c1_java_main CALL 0
  0 U 2     19 @    0 __pthread_enable_asynccancel:/lib/libpthread-2.5.so   11c1_11c1_java_main RETURN 0
  0 U 1      7 @   31 __libc_write:/lib/libpthread-2.5.so   11c1_11c1_java_main CALL 0
  0 K 5      1 @    6 no_singlestep:vmlinux   11c1_11c1_java_main OTHER 0
  0 K 5      0 @    0 syscall_trace_entry:vmlinux   11c1_11c1_java_main OTHER 0
  0 K 5      0 @    0 do_syscall_trace:vmlinux   11c1_11c1_java_main OTHER 0
  0 K 5      0 @   2e do_syscall_trace:vmlinux   11c1_11c1_java_main OTHER 0
...
  0 U 2      1 @   1a java/io/PrintStream.print(Ljava/lang/String;)V:JITCODE 11c1_11c1_java_main RETURN 0
  0 U 3      1 @  19f hellop.main([Ljava/lang/String;)V:JITCODE   11c1_11c1_java_main JUMP 0
  0 U 3      5 @  1af hellop.main([Ljava/lang/String;)V:JITCODE   11c1_11c1_java_main JUMP 0
  0 U 3     11 @   c4 hellop.main([Ljava/lang/String;)V:JITCODE   11c1_11c1_java_main JUMP 0
  0 U 3      7 @   d4 hellop.main([Ljava/lang/String;)V:JITCODE   11c1_11c1_java_main JUMP 0
  0 U 3      7 @   d4 hellop.main([Ljava/lang/String;)V:JITCODE   11c1_11c1_java_main JUMP 0
...
```

Figure 3: Excerpts from an arc file

on branch flags might not be correctly preserved across interrupts and system calls, so we need to dynamically patch such critical places or to use kprobes.

We call a branch trace *ITrace*. ITrace can include both user- and kernel-space trace records. One ITrace record has addresses of the branch and the branch target, and possibly the number of instructions executed from between the previous and the last branch execution. There are separate major codes for user and kernel addresses. On PowerPC, ITrace records can also include load and store addresses (with different major codes).

The post application can produce an *arc* report from an ITrace and the corresponding `log-jita2n` file. Figure 3 shows excerpts from an arc file obtained from the ITrace of a simple `hellop` application, where `main()` calls `myA()` in a loop and prints a value; `myA()` calls `myC()` which calculates that value. We can follow a write request from JITed code to a JVM library to a system library to the kernel and back. (One arc excerpt in the figure shows the entry to the kernel and the other one shows the exit from `PrintStream.print` to `hellop.main`.)

ITrace can be controlled using the provided `run.itrace` script, or `libperfutil` C or Java interfaces. `run.itrace` is normally used when we do not want to or cannot change the tracing target application; the script asks for the lowest pid to trace. A more controlled ITrace can be obtained by using `ITraceOn()` and `ITraceOff()` interfaces around the section of the code to be traced.

Currently PI does not include support for continuous ITrace. However, we are investigating an algorithm that might enable this feature in future releases.

2.2 Tprof

Tprof trace can be used for system performance analysis. It is based on a sampling technique which encompasses the following steps:

- Interrupt the system periodically if time-based, or when performance-monitoring hardware reaches a given threshold, if event-based.
- Determine the address of the interrupted code along with the process id (pid) and thread id (tid).

```
================================
 )) Process_Thread_Module_Symbol
================================

LAB    TKS   %%%     NAMES

PID   2372 51.25    java_103c
 TID   1704 36.82     tid_main_103c
  MOD    721 15.58      vmlinux
   SYM    123  2.66       _spin_unlock_irqrestore
   SYM     88  1.90       system_call
   SYM     48  1.04       write_chan
   SYM     42  0.91       __copy_to_user_ll
   ...
  MOD    338  7.30      JITCODE
   SYM     81  1.75       hellop.myC()V
   SYM     32  0.69       hellop.main([Ljava/lang/String;)V
   SYM     17  0.37       java/lang/String.indexOf(II)I
   SYM     17  0.37       java/io/PrintStream.write(Ljava/lang/String;)V
   SYM     16  0.35       java/lang/Long.getChars(JI[C)V
   SYM     15  0.32       java/io/FileOutputStream.write([BII)V
   SYM     13  0.28       java/lang/StringBuffer.append(Ljava/lang/String;)Ljava/lang/StringBuffer;
   SYM     12  0.26       sun/nio/cs/StreamEncoder.flushBuffer()V
  ...
```

Figure 4: A time-based tprof report excerpt

- Record tprof trace record in a trace buffer.
- Return to the interrupted code.

The detailed steps to obtain a `tprof` trace and a subsequent `tprof.out` report are encapsulated by the *run.tprof* script. This script interacts with the analyst to set up and run necessary steps. Similarly to ITrace, JProf is used to collect the necessary JIT address-to-name information.

The `Tprof.out` report shows percentages of tprof trace records for various granularity, such as for each process, module within a process, or a symbol within a module. Figure 4 shows an excerpt from a time-based `tprof` report for the `hellop` application, for symbols within a module within a thread. Such reports can be used to detect hot-spots in the application or to indicate the resource distribution.

2.3 Other Trace Types

The trace format can easily be used for various types of trace records. In addition to ITrace, tprof, MTE, system information, and time stamp change trace records, the tracing facility currently can produce traces of thread dispatches, and interrupt entries and exits.

3 Per-Thread Time Facility

To accurately determine performance metrics accumulated in an instrumented function, the user-space profiler needs operating system or device driver support for virtualized per-thread metrics (PTM). Such support needs to:

- *Keep separate metrics count for threads of interest.* The PTM code needs to get control when a new thread is about to be dispatched, and to read and save away values of hardware monitoring counters used for metrics.
- *Factor out time spent in external interrupts.* When applications are being monitored, there are some kernel operations that are being done as a direct result of the application code, such as those that require a kernel service. When those services are executed synchronously, it is usually best to include the overhead of the entire code path, including the kernel code path as part of the application because that is how the application will run normally, without instrumentation. For example, we do not want to remove the influence of page faults. However, some events, such as I/O interrupts, tend to occur randomly on a given thread. When trying to produce repeatable measurements, it is helpful to

factor out or separate out the metric counts related to asynchronous interrupts.

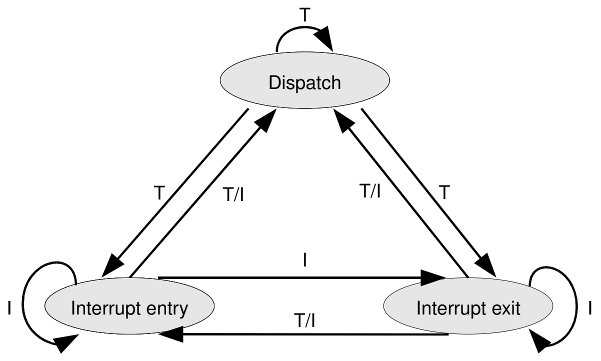
- *Make per-thread metric values available to the profiler.* One way to do it is to use a system call or an `ioctl`. However, we can avoid the overhead of system calls using a mapped data area with all necessary information.

The best solution would be to have all these features provided by the operating system. By having the OS monitor the selected threads, we avoid security issues, especially if an application is allowed to monitor only its own threads. The perfmon2 project is an excellent PTM candidate and we eagerly await its full inclusion into the mainstream Linux kernel [1]. In the mean time, we implement the necessary support in the Per-Thread Time Facility (PTT) in the PI driver module, `pitrace`. Note that a more correct name would be Per-Thread Metrics Facility; PTT is a legacy name from the time it supported only per-thread processor cycles. Today PTT can support virtualization of any physical metric provided by the performance monitoring counters.

The `pitrace` module hooks the scheduler close to its return, when the thread to be scheduled is already known. We either use kprobes or dynamically patch the kernel ourselves. To factor out external interrupts, we patch the interrupt entries and exits, so that the time spent in interrupts is accounted for in per-cpu interrupt buckets.

When a request to monitor a thread is made, the driver allocates and maps a thread work area which the application or profiler attached to the application may access directly. The profiler specifies the exact metrics to be monitored and the driver simply reads and accumulates the specified metrics at dispatch and interrupt entry/exit time. Figure 5 shows the PTM state machine. For example, when a previous state was the Dispatch and we are currently entering an interrupt, the metrics delta (difference from the metrics in the last state) should be added to the accumulated thread metrics.

The mapped thread work area contains the accumulated per-thread values and the per-cpu values in the last PTM state. The profiler reads the metrics, calculates the differences from the value of the counters at the time of the last PTM state, and adds those differences to the accumulated values. Since there is a chance that the thread could be dispatched out and back in during the



T – metrics applied to a thread
I – metrics applied to the interrupt bucket
T/I – applied to a thread or the interrupt bucket, depending whether there are pending interrupts

Figure 5: State machine for virtual per-thread metrics

calculations, there should be a simple way for the profiler to determine that this has occurred. One way to provide this feature is to also keep track of the number of dispatches and interrupts in the mapped thread work area. The profiler reads the count of dispatches and interrupts before reading the metrics and reads them again after performing the calculations. If the number of dispatches and interrupts does not change, then the calculated values can be used. If the thread was dispatched or interrupted while reading counters, then the calculations should be repeated until the number of dispatches and interrupts stays the same [2]. Our experiments indicate that this procedure needs to be repeated at most a couple of times.

3.1 PTT Interfaces and `ptt` Application

The `libperfutil` library provides interfaces to the PTT facility in the driver. There are APIs to initialize the PTT facility (`PttInit()`) and to terminate it (`PttTerminate()`). Instead of using a single function to get the current thread metric values, recent packages provide separate functions depending on whether the underlying platform is a uni- or multi-processor system, and on the number and combination of metrics (counters vs. cycles), so that the most frequently used cases of one or two metrics are optimized to reduce overhead. The required function is automatically selected by `libperfutil`, so that the profiler code only needs to set a pointer to it. The interested reader can get more details about available APIs from the package documentation. The maximum number of metrics collected concurrently is eight, regardless of the number of performance monitoring counters available.

One example of PTT facility usage is the `ptt` application, which can turn PTT on and off and dump information about threads for which PTT data is available.

4 JProf Callflow Tracing and Metrics Calibration

Identifying and reporting calling sequences, by receiving notifications on entries and exits to functions or Java methods, is an important methodology that has been shown to be very useful for performance analysis [3].

Based on the invocation options, JProf calls a `libperfutil` function, `PttInit()`, which in turn initializes the PTT facility. JProf receives notifications from the Java Virtual Machine (JVM) about Java method entries and exits, via the JVM Profiler Interface (JVMPI) or the JVM Tool Interface (JVMTI), and it can also query the JVM about the method type and other relevant information.

When an entry or exit event is received, JProf can get the virtualized metrics for the thread on which it is executing. However, the act of observing a metric in a running application almost always changes the behavior of that application in some way. For example, the instructions used to read a metric increase the execution path length of the application and the memory used to store what was read reduces the amount of memory available to the application. That is why the metrics need to be calibrated, that is, adjusted to compensate for the overhead caused by the instrumentation required to observe the metric.

All metrics can be calibrated, but the accuracy of the calibration depends on the stability of the metric being observed. For example, the number of processor cycles required to execute a method is not a stable metric, since it is influenced by a great number of factors such as memory latency, the size of the instruction or data cache, the amount of free memory, asynchronous interrupts, and even the size and complexity of the instructions used by the method. On the other hand, the number of instructions completed is a stable metric, because the number of instructions executed along any given path in the uninstrumented application is fixed. This is why the JProf calibration algorithm is optimized for instructions, although it can be applied to any metric.

The most obvious kind of calibration is performed by merely reading the values of the metrics at entry to and exit from JProf. By doing this, JProf can eliminate its own effects on the metrics between these two reads. We call this *internal calibration*. To maximize the accuracy of the internal calibration, we want to read the metrics as soon as possible after entry to JProf (Early Read) and again at the last possible moment before exit from JProf (Late Read). Another calibration component is *external calibration*—compensation for instrumentation overhead outside of JProf.

In an ideal world, there would be no instructions before the Early Read or after the Late Read, but this is never true. Even the instructions necessary to call the Early Read routine or setup the actual reading of the values are overhead that must be removed.

To achieve successful removal of the JProf portion of external overhead, there are no conditional branches before the Early Read and after the Late Read, to keep the instruction path constant.

4.1 The Calibration Algorithm

The basic assumption on which the calibration algorithm is based is that the overhead which must be removed can be computed from the minimum observed change in the metrics between calls to the profiler. Between any consecutive calls to the profiler, we can compute metric deltas which are the differences in the metric values between those acquired by the Early Read routine from the current call and those acquired by the Late Read routine from the previous call. Each delta includes both the external instrumentation overhead that we want to remove and the actual metric values that we want to keep.

However, the instrumentation overhead may vary depending on the type of the event (entry or exit), type of method (native, interpreted or JITed), and even the transition sequence between methods. The overhead associated with an entry event following an entry event may be different from the overhead associated with an entry event following an exit event, due to optimizations in the so-called *glue code*.

The solution is to maintain an array of minimum observed deltas for each sequence of method types and transition types. We have found that a sequence of three method types and the transitions between them is sufficient for all of the applications we have tested. For example, Interpreted-enters-JITed-enters-Interpreted is one sequence, while JITed-exits-to-JITed-enters-JITed is a different sequence.

Working with the JVM, we have found that there are really three different transition types: entry, exit, and exception-exit. We treat exception-exit (an exit from a routine as a result of an exception) as a unique type, in order to eliminate its influence on statistics gathered for normal exits.

There are also many different method types. We not only consider interpreted and JITed methods, but also native methods. We further distinguish between static and non-static methods for each of these types, since non-static routines require additional glue code to identify the object associated with the method. The last two method types we use are Compiling and Other. Just as we defined a special transition type to isolate the effects of exceptions, we define a special method type for the Java compiler to isolate its effects. Finally, we define an Other type to allow us to isolate the effects of methods whose type can not be accurately identified. This can occur when profiling is started in the middle of executing a method and we lack information about the context in which the method is executing. Thus, we use 8 different method types and 3 different transition types for a total of 8*3*8*3*8 = 4608 different sequences.

Although the categories are still relatively easy to manage, the sheer number of categories introduces other problems. As the number of categories increases, the number of events in each category decreases. This makes it harder to find the true minimum overhead for each category. It also makes it too costly to save counts of all of the different types of sequences with every method.

The solution to both of these problems is to train the profiler by saving the minimum observed values from other profiling runs. This is most effective if the training application generates events in as many valid categories as possible. Some categories will never occur, such as JITed-enters-Native-exits-to-Native, which is invalid because the native method must return to the JITed method which called it. We use a *trainer* Java test case, which is included in the PI package.

Transition	Num Instr
En-jitted-En-jitted	3
En-jitted-En-Jitted	6
En-jitted-En-native	28
En-jitted-En-Native	35
En-jitted-Ex-jitted	3
En-jitted-Ex-Jitted	3
En-Jitted-En-jitted	3
En-Jitted-En-Jitted	4
En-Jitted-En-native	28
En-Jitted-En-Native	29
En-Jitted-Ex-jitted	3
En-Jitted-Ex-Jitted	3
En-native-Ex-jitted	4
En-native-Ex-Jitted	4
En-Native-Ex-jitted	23
En-Native-Ex-Jitted	4
Ex-jitted-En-jitted	1
Ex-jitted-En-Jitted	4
Ex-jitted-En-native	38
Ex-jitted-En-Native	29
Ex-jitted-Ex-jitted	1
Ex-jitted-Ex-Jitted	1
Ex-Jitted-En-jitted	1
Ex-Jitted-En-Jitted	2
Ex-Jitted-En-native	38
Ex-Jitted-En-Native	39
Ex-Jitted-Ex-jitted	1
Ex-Jitted-Ex-Jitted	1

Table 1: Minimum number of instructions for the most frequently seen transitions. En–method entry, Ex–method exit, lower case–static methods, upper case–non-static methods.

4.2 Environmental Overhead

The calibration algorithm described so far still has one remaining flaw: not all glue code should be associated with instrumentation overhead. Some glue code will be executed even if the application is not being profiled. The calibration algorithm can accurately detect overhead, but it can not determine how much to remove and how much to keep. To do this, the profiler requires specific knowledge about the execution environment when executing applications that have not been instrumented.

The solution to this problem is to execute the trainer application while gathering an instruction trace. By carefully analyzing the results of the instruction trace, a set of minimum values after calibration can be determined. Table 1 shows an example of minimum calibration values for the most frequently seen transitions, for IBM

JVM 5.0 SR5 for 32-bit Linux on Intel platforms. For example, En-jitted-Ex-Jitted means that a method calls a static JITed method which then returns to a non-static JITed method.

Note that we are using only 4 of the types in the 5-type sequences we described. Due to the difficulty of generating every possible combination in the trainer code, we limit the number of the steps to 4. The sequences not covered by data Table 1 will assume a minimum overhead of 1. We do not try to determine values for interpreted methods, because all methods significantly contributing to the overall application profile will be JITed after the initial warm-up.

The calibration algorithm must still be used with these minimum calibration values. The amount of calibration needed can still vary based on the parameters specified during instrumentation, even though the environmental overhead represented by these minimum calibration values remains constant.

By applying the calibration algorithm with an appropriate set of minimum calibration values, profiling accuracy is nearly identical to that achieved by instruction tracing with a fraction of the impact on the execution speed of the application. We validated this approach by comparing a calibrated flow to the instruction trace, for several testcases.

Note that we assume that the instrumentation overhead is constant for a particular transition/type sequence. This is achieved in IBM Java 5. Another concern is that a Java compiler may in-line methods and then may or may not produce entry/exit events for such methods. Disabling in-lining may affect the general overhead of the application. One approach for Java is to simply let in-lining occur as normal and only get the entry/exit events for the methods that are not in-lined. Moreover, a compiler may optimize code and change it in various ways, such as partial in-lining or loop unrolling. These and other optimizations may cause the calibrated metrics to vary from what is expected by examining the code.

4.3 Profiling Exit/Entry Events in C/C++ Code

The same calibration concept can be extended to code written in other programming languages, such as C. JProf needs two additional event categories, for C code entries and exits, so that it can be used for profiling of both standalone C code and code called from Java using the Java Native Interface (JNI). We implemented a prototype profiler library *hookit* which sends entry/exit notifications to JProf. The code to be profiled needs to be compiled using the gcc compile option `-finstrument-functions` and to be statically linked with `libhookit`.

4.4 JProf Callflow Reports

JProf can produce two kinds of callflow reports, depending on the invocation options. More frequently used is a `log-rt` report, which represents methods organized into a call tree, with the number of callers and callees for each method. Figure 6 shows an excerpt of a `log-rt` file for `hellop`. The number of loop iterations was 1000, so both `myA` and `myC` are called 1000 times. The BASE column shows the accumulated number of instructions executed for all 1000 calls.

The other type of callflow report is a `log-gen` file, which has a full callflow trace, with one line for each method entry or exit event, together with the metric value(s) between two successive events. The records in a `log-gen` file are written immediately after a method entry/exit, so the calibration algorithm has to apply whatever is the current minimum delta.

5 Performance Inspector report visualization

Several types of reports produced by PI toolset can be visualized using Visual Performance Analyzer (VPA), which is an Eclipse-based visual performance toolkit [4].

With the help of VPA, users can visualize tprof reports with its Profile Analyzer component, and callflow reports with the Control Flow Analyzer.

6 Conclusion and Future Directions

Although some of PI tools have overlapping functionalities with other Linux utilities or kernel modules, we believe that the project significantly contributes to the always-demanding field of performance analysis, by providing some unique useful features. One such feature is per-thread metrics virtualization which, together

```
   LV CALLS CEE      BASE   DELTA  DS IN  NAME
    2  1000 1000     11888 2378000  1  6  J:hellop.myA()V
    3  1000    0   7004919 1225000  0 12  J:hellop.myC()V
    2  1000 1000      9200 2384000  0  7  J:java/lang/StringBuffer.<init>()V
    3  1000 1000     48529 2382000  0  8  J:java/lang/StringBuffer.<init>(I)V
    4  1000    0      3880 1221000  0  5  J:java/lang/Object.<init>()V
    2  1000 2000     37855 3543000  1  6  J:java/lang/StringBuffer.append(J)Ljava/lang/StringBuffer;
    3  1000 3000    114912 4709000  5 14  J:java/lang/Long.toString(J)Ljava/lang/String;
    4  1000    0    108379 1225000  1  5  J:java/lang/Long.stringSize(J)I
    4  1000    0    178282 1225000  0  4  J:java/lang/Long.getChars(JI[C)V
    4  1000 1000     10800 2384000  0  5  J:java/lang/String.<init>(II[C)V
    5  1000    0      3320 1221000  0  2  J:java/lang/Object.<init>()V

 Column Labels:
  : LV    = Level of nesting       (Call Depth)
  : CALLS = Calls to this method   (Callers)
  : CEE   = Calls from this method (Callees)
  : BASE  = Metrics observed
  : DELTA = BASE adjustment due to calibration
  : DS    = Dispatches observed
  : IN    = Interrupts observed
  : NAME  = Name of Method or Thread
```

Figure 6: A log-rt report excerpt with the instruction completed metric

with the metrics calibration mechanism, enables accurate profiling of Java methods or C functions. Also useful is the combination of the Tracing Facility and address-to-name resolution mechanism, which results in correct trace interpretation for dynamically generated code.

We constantly add support for new hardware platforms, and we will continue to do so in the future. Current tools support a limited set of hardware performance counter events, but this set can be easily extended by adding new events to per-platform event description files.

We also strive to support new Linux releases. The sensitivity to kernel changes would be significantly reduced if we could build on the top of mechanisms integrated with the mainline Linux kernel. In the future, we might be able to use perfmon2 for per-thread performance counter virtualization [1]. It would be nice to merge the Tracing facility with some of the on-going tracing efforts, such as the Driver Tracing Infrastructure [5].

We maintain a long wish list of useful additions to the PI project, such as the capability for continuous ITrace. As always, new ideas and contributions are welcome.

Legal Statement

References

[1] *perfmon2: the hardware-based performance monitoring interface for Linux*, `http://perfmon2.sourceforge.net/`

[2] R.F. Berry, et al., *Method and system for low-overhead measurement of per-thread performance information in a multithreaded environment*, US Patent No. 6658654, 2003.

[3] W.P. Alexander, R.F. Berry, F.E. Levine, R.J. Urquhart, *A unifying approach to performance analysis in the Java environment*, IBM Systems Journal, Vol. 39, Nov. 1, 2000, pp. 118–134.

[4] *Visual Performance Analyzer*, `http://www.alphaworks.ibm.com/tech/vpa`

[5] D. Wilder, *Unified Driver Tracing Infrastructure*, Linux Symposium, Ottawa, Canada, 2007.

Containers checkpointing and live migration

Andrey Mirkin
OpenVZ
major@openvz.org

Alexey Kuznetsov
OpenVZ
alexey@openvz.org

Kir Kolyshkin
OpenVZ
kir@openvz.org

Abstract

Container-type virtualization is an ability to run multiple isolated sets of processes, known as containers, under a single kernel instance. Having such an isolation opens the possibility to save the complete state of (in other words, to checkpoint) a container and later to restart it. Checkpointing itself is used for live migration, in particular for implementing high-availability solutions.

In this paper, we present the checkpointing and restart feature for containers as implemented in OpenVZ. The feature allows one to checkpoint the state of a running container and restart it later on the same or a different host, in a way transparent for running applications and network connections. Checkpointing and restart are implemented as loadable kernel modules plus a set of user-space utilities. Its efficiency is proven on various real-world applications. The overall design, implementation details and complexities, and possible future improvements are explained.

1 Introduction

OpenVZ is container-based virtualization for Linux. OpenVZ partitions a single physical server into multiple isolated *containers*. As opposed to other virtualization solutions, all containers are running on top of a single kernel instance. Each container acts exactly like a stand-alone server; a container can be rebooted independently and have root access, users, IP address(es), memory, processes, files, etc. From the kernel point of view, a container is the separate set of processes completely isolated from the other containers and the host system.

Having a container not tied to a particular piece of hardware makes it possible to migrate such a container between different physical servers. The trivial form of migration is known as *cold* (or *offline*) *migration*, which is performed as follows: stop a container, copy its file system to another server, start it. Cold migration is of limited use since it involves a downtime which usually requires prior planning.

Since a container is an isolated entity (meaning that all the inter-process relations, such as parent-child relationships and inter-process communications, are within the container boundaries), its complete state can be saved into a disk file—the procedure is known as *checkpointing*. A container can then be *restarted* back from that file.

The ability to checkpoint and restart a container has many applications, such as:

- Hardware upgrade or maintenance.
- Kernel upgrade or server reboot.

Checkpoint and restart also makes it possible to move a running container from one server to another without a reboot. This feature is known as *live migration*. Simplistically, the process consists of the following steps:

1. Container's file system transfer to another server.
2. Complete state of container (all the processes and their resources) is saved to a file on disk.
3. The file is copied to another server.
4. The container is restarted on another server from the file.

Live migration is useful for:

- High availability and fault tolerance.
- Dynamic load balancing between servers in a cluster of servers.

This paper is organized as follows. Section 2 discusses related work. Section 3 provides prerequisites and requirements for checkpointing. Section 4 presents overall design of checkpoint and restart system. Section 5 describes the algorithm for live migration of containers. Section 6 provides possible ways for live migration optimization. Finally, the paper is ended with a brief conclusion.

2 Related Work

There are many another projects which proposed checkpoint and restart mechanisms:

- CHPOX (Checkpoint for Linux) [1]
- EPCKPT (Eduardo Pinheiro Checkpoint Project) [2]
- TCPCP (TCP Connection Passing) [4]
- BLCR (Berkeley Lab Checkpoint/Restart) [7]
- CRAK (Checkpoint/Restart As a Kernel Module) [5]
- ZAP [6]
- Sprite [10]
- Xen [8]
- VMware [9]

Not all the systems are available as open source software, and the information about some of them is pretty scarce. All the systems which are available under an open source license lack one feature or another. First, except for some written-from-scratch process migration operating systems (such as Sprite [10]), they can not preserve established network connections. Second, general-purpose operating systems such as UNIX were not designed to support process migration, so checkpoint and restart systems built on top of existing OSs usually only support a limited set of applications. Third, no system guarantees processes restoration on the other side because of resource conflicts (e.g., there can be a process on a destination server with the same PID).

Hardware virtualization approaches like Xen [8] and VMware [9] allow checkpointing and restarting only an entire operating system environment, and they can not provide checkpointing and restarting of small sets of processes. That leads to higher checkpointing and restart overhead.

3 Prerequisites and Requirements for System Checkpointing and Restart

Checkpointing and restarting a system has some prerequisites which must be supplied by the OS which we use to implement it. First of all, a container infrastructure is required which gives:

1. *PID virtualization* – to make sure that during restart the same PID can be assigned to a process as it had before checkpointing.
2. *Process group isolation* – to make sure that parent-child process relationships will not lead to outside a container.
3. *Network isolation and virtualization* – to make sure that all the networking connections will be isolated from all the other containers and the host OS.
4. *Resources virtualization* – to be independent from hardware and be able to restart the container on a different server.

OpenVZ [11] container-type virtualization meets all these requirements. Other requirements which must be taken into account during the design phase are:

1. The system should be able to checkpoint and restart a container with the full set of each process' resources including register set, address space, allocated resources, network connections, and other per-process private data.
2. Dump file size should be minimized, and all actions happening between a freeze and a resume should be optimized to have the shortest possible *delay in service*.

4 Checkpointing and Restart

The checkpointing and restart procedure is initiated from the user-level, but it is mostly implemented at the kernel-level, thus providing full transparency of the checkpointing process. Also, a kernel-level implementation does not require any special interfaces for resources re-creation.

The checkpointing procedure consists of the following three stages:

1. *Freeze processes* – move processes to previously known state and disable network.

2. *Dump the container* – collect and save the complete state of all the container's processes and the container itself to a *dump file*.

3. *Stop the container* – kill all the processes and unmount container's file system.

The restart procedure is checkpointing, *vice versa*:

1. *Restart the container* – create a container with the same state as previously saved in a dump file.

2. *Restart processes* – create all the processes inside the container in the frozen state, and restore all of their resources from the dump file.

3. *Resume the container* – resume processes' execution and enable the network. After that, the container continues its normal execution.

The first step of the checkpointing procedure and also the last step of restart procedure before processes can resume their execution is process-freeze. The freeze is required to make sure that processes will not change their state and saved processes' data will be consistent. It is also easier to reconstruct frozen processes.

Process freeze is performed by setting the special flag `TIF_FREEZE` on all the processes' threads. In this case, the `PF_FREEZE` task flag can not be used, as atomic change is required. After `TIF_FREEZE` flag is set on all the threads, each process receives a fake signal. Sending the fake signal is for moving all the threads to a beforehand known state—in this case, it is `refrigerator()`. Using just a fake signal for freezing processes has the benefit that all the signals which are on the way to a process will be saved and delivered after the process restart.

Using such a mechanism for processes freeze has benefits for processes which are in the kernel context at the moment of freezing—they will handle the fake signal before returning to user mode, and will be frozen as all the other processes are. If a process is in an uninterruptible state (system call or interrupt handling), it will be frozen right after the kernel event is completed. If a process is in an interruptible system call, it will be interrupted and handle the fake signal. In most cases, such system calls will be automatically restarted; otherwise, the caller should be prepared for the appropriate error handling. Such a mechanism is simple, as it uses the already implemented "software suspend" kernel feature, and so does not require much change in the kernel source code.

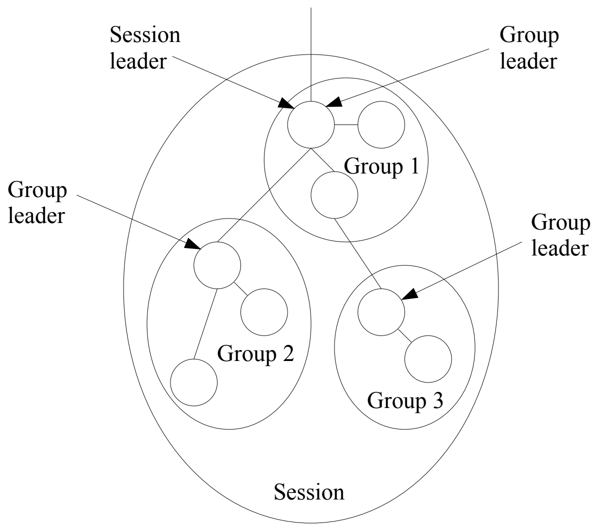


Figure 1: Process hierarchy

It is very important to save a consistent state of all the container's processes. All process dependencies should be saved and reconstructed during restart. Dependencies include the *process hierarchy* (see Figure 1), *identifiers* (PGID, SID, TGID, and other identifiers), and *shared resources* (open files, SystemV IPC objects, etc.). During the restart, all such resources and identifiers should be set correctly. Any incorrectly restored parameter can lead to a process termination, or even to a kernel oops.

Another big area of checkpointing and restart is networking. During checkpointing and restart, the network should be *disabled*—it is needed to preserve network connections. The simplest way to disable the network is to drop all incoming packets, as processes are frozen and can not process incoming packets. From the point of view of an outside, user it looks like a temporary link network problem, not something like "host unreachable" message. Such a behavior is acceptable since the TCP protocol has a mechanism to resend packets if no acknowledgment is received, and for the UDP protocol, packet loss is expected.

As most of the resources must be restored from the process context, a special function (called *"hook"*) is added on top of the stack for each process during the restart procedure. Thus, the first function which will be executed by a process will be that "hook," and the process itself will restore its resources. For the container's `init` process, this "hook" also restores the container state including mount points, networking (interfaces, route tables, iptables rules, and conntracks), and SystemV IPC objects; and it initiates process tree reconstruction.

5 Live Migration

Using the checkpointing and restart feature, it is easy to implement live migration. A simple algorithm is implemented which does not require any special hardware like SAN or iSCSI storage:

1. *Container's file system synchronization.* Transfer the container's file system to the destination server. This can be done using the `rsync` utility.

2. *Freeze the container.* Freeze all the processes and disable networking.

3. *Dump the container.* Collect all the resources and save them to a file on disk.

4. *Second container's file system synchronization.* During the first synchronization, a container is still running, so some files on the destination server can become outdated. That is why, after a container is frozen and its files are not being changed, the second synchronization is performed.

5. *Copy the dump file.* Transfer the dump file to the destination server.

6. *Restart the container on the destination server.* At this stage, we are creating a container on the destination server and creating processes inside it in the same state as saved in dump file. After this stage, the processes will be in the frozen state.

7. *Resume the container.* Resume the container's execution on the destination server.

8. *Stop the container on the source server.* Kill the container's processes and unmount its file system.

9. *Destroy the container on source server.* Remove the container's file system and config files on the source server.

If, during the restart, something goes wrong, the migration process can be rolled back to the source server, and the container will resume execution on the source server as if nothing happened.

In live migration for external clients which connected to the container via the network, the migration process will look like a temporary network problem (as live migration is not instantaneous). But after a delay, the container continues its execution normally, with the only difference being that it will already be on the destination server.

In the above migration scheme, Stages 3–6 are responsible for the most delay in service. Let us take a look at them again and dig in a little bit deeper:

1. *Dump time* – the time needed to traverse over all the processes and their resources and save this data to a file on disk.

2. *Second file system sync time* – time needed to perform the second file system synchronization.

3. *Dump file copying time* – time needed to copy the dump file over the network from the source server to the destination server.

4. *Undump time* – time needed to create a container and all its processes from a dump file.

6 Migration Optimizations

Experiments show that second file system sync time and dump file copying time are responsible for about 95% of all the delay in service. That is why optimization of these stages can make sense. The following options are possible:

1. Second file system sync optimization – decrease the number of files being compared during the second sync. This could be done with the help of *file system changes tracking* mechanism.

2. Decreasing the size of a dump file:

   (a) *Lazy migration* – migration of memory after actual migration of container, i.e., memory pages are transferred from the source server to the destination on demand.

   (b) *Iterative migration* – iterative migration of memory before actual migration of container.

These three optimizations are described below.

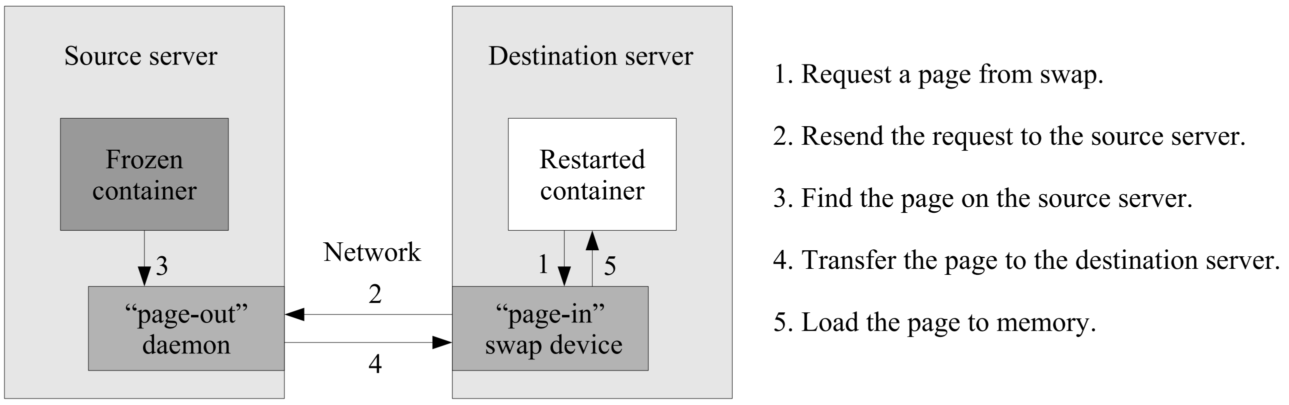

Source server
Frozen container
"page-out" daemon
Network
Destination server
Restarted container
"page-in" swap device
1
2
3
4
5
1. Request a page from swap.
2. Resend the request to the source server.
3. Find the page on the source server.
4. Transfer the page to the destination server.
5. Load the page to memory.

Figure 2: Lazy migration

6.1 File System Changes Tracking

The idea is that when this system is activated, it begins to collect the names of the files being changed and stores them in a list. The list of modified files is to be used during the second file system synchronization. It can dramatically decrease second file system synchronization time. Tracking file system changes can not be implemented as a separate loadable kernel module, as it requires core kernel changes.

6.2 Lazy Migration

During live migration, all processes' private data are saved to a dump file, which is then transferred to the destination server. In the case of large memory usage, the size of the dump file can be huge, resulting in an increase of dump file transfer time, and thus in an increased delay in service. To handle this case, another type of live migration can be used—*lazy migration*. The idea is the following—all the memory pages allocated by processes are marked with a special flag, which is cleared if a page is changed. After that, a container can be frozen and its state can be dumped, but in this case only pages without this flag are stored. That helps to reduce the size of a dump file.

The only problem which should be also solved here is how to transfer all the remaining memory pages from the source server to the destination. A special *page-in* swap device on the destination server and a *page-out* daemon on the source server are proposed to solve this problem.

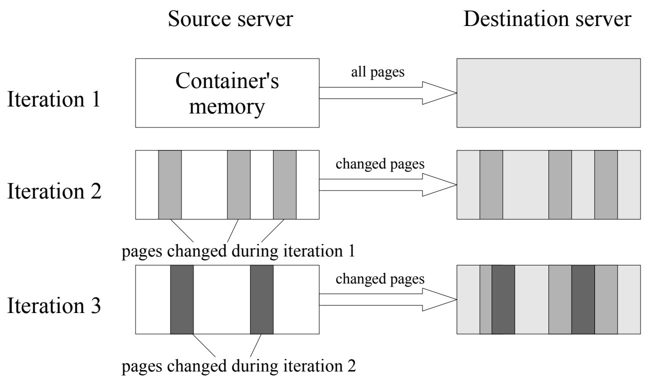


Figure 3: Iterative migration

During processes restart on the destination server, all the pages which are not saved to the dump file are marked as swapped to a page-in device. When a process resumed on the destination server accesses a page which is marked as swapped, a request to the swap device is generated. The page-in device resends this request to the page-out daemon on the source server. The page-out daemon sends the requested page to the destination server, and then this page is loaded into memory on the destination server. See Figure 2 for details. During the first few minutes pages, are transferred to the destination server on demand. After a while, the swap-out is forced, and all the pages are transferred from the source server to the destination.

6.3 Iterative Migration

Another way to decrease the size of the dump file is to transfer memory pages in advance. In this case, all the pages are transferred to the destination server before container freeze. But as processes continue their normal execution, pages can be changed and transferred pages can become outdated. That is why pages should be transferred *iteratively*. On the first step, all pages are marked with a *clean* flag and transferred to the destination server. Some pages can be changed during this process, and the *clean* flag will be removed in this case. On the second step, only the changed pages are transferred to the destination server. See Figure 3 for details. This iterative process stops if the number of the changed pages becomes zero, or the number of the changed pages becomes more than $\frac{N}{2^i}$, where N is the total number of pages and i is the iteration number.

All the transferred pages temporarily stored on the destination server are used during the restart process. All the pages changed during the last iteration are stored in a dump file and restored from it during the restart process.

7 Conclusion

The checkpointing and restart mechanism for containers has been designed and implemented in the OpenVZ Linux kernel. On top of this mechanism, the live migration feature has been implemented, allowing the movement of containers from one server to another without a reboot. The efficiency of the system has been proven on various real-world applications. Possible optimizations of the migration algorithm have been proposed to decrease the delay in service.

References

[1] O.O. Sudakov, Yu.V. Boyko, O.V. Tretyak, T.P. Korotkova, E.S. Meshcheryakov, *Process checkpointing and restart system for Linux*, Mathematical Machines and Systems, 2003.

[2] Eduardo Pinheiro, *Truly-Transparent Checkpointing of Parallel Applications*, Federal University of Rio de Janeiro UFRJ.

[3] Eduardo Pinheiro, Ricardo Bianchini, *Nomad*, COPPE Systems Engineering, Federal University of Rio de Janeiro, Rio de Janeiro, Brazil.

[4] Werner Almesberger, *TCP Connection Passing*, In *Proceedings of the Linux Symposium* (Ottawa, Ontario, Canada, July, 2004).

[5] Hua Zhong, Jason Nieh, *CRAK: Linux Checkpoint/Restart As a Kernel Module*, Department of Computer Science, Columbia University, Technical Report CUCS-014-01, November 2001.

[6] Steven Osman, Dinesh Subhraveti, Gong Su, Jason Nieh, *The Design and Implementation of Zap: A System for Migrating Computing Environments*. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, December 9–11, 2002.

[7] Jason Duell, *The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart*, Lawrence Berkeley National Laboratory

[8] Xen. `http://www.xen.org`

[9] VMware, Inc. `http://www.vmware.com`

[10] The Sprite Operating System. `http://www.eecs.berkeley.edu/Research/Projects/CS/sprite/sprite.html`

[11] OpenVZ. `http://openvz.org`

Building a Robust Linux kernel piggybacking The Linux Test Project

Subrata Modak
Linux Technology Centre, IBM, India
`subrata@linux.vnet.ibm.com`

Balbir Singh
Linux Technology Centre, IBM, India
`balbir@linux.vnet.ibm.com`

Abstract

The Linux™ kernel is growing at a rapid rate and runs across many architectures and platforms; ensuring that the kernel is reliable, robust, and stable is very critical. The Linux Test Project (LTP) was established to meet the very goals stated above. Testing is often ignored in major development, and we pay the cost through frequent updates, frequent crashes and unhappy users. LTP is now breathing a new life; we want to add more test cases, cover more code, test new features, update existing test cases, and improve the framework.

In this paper, we explore the effectiveness of testing via LTP, and look at coverage statistics and number of new test cases added. We look at where LTP development and kernel testing via LTP stands with regard to kernel development. This paper also demonstrates how to write a simple LTP test case, and enjoy the benefits of using it over and over again.

1 Introduction

The Linux Test Project, created by SGI[1], was one of the first to bring organized testing to Linux. No formal testing methodology was available to Linux developers prior to the arrival of LTP. Systematic integration testing was a distant dream, though most developers unit-tested their own enhancements and patches. LTP's primary goal continues to be to provide a test suite to the Linux community that helps to validate the reliability, robustness, and stability of the Linux kernel. It provides functional and regression testing with or without stress, utilizing its own execution harness to allow for test automation.

At the time when the 2.3 kernel was released, LTP had around 100 tests [6]. As Linux grew and matured through the 2.4, 2.5 & 2.6 kernels, the LTP test suite also grew and matured as well. Today, the Linux Test Project contains well over 3000 tests, and the number of tests is still growing. It has evolved over the years to become very comprehensive, capable of testing various features of the kernel including system calls, memory management, inter-process communication, device drivers, I/O, file systems, and networking. With 95% of the test code written in C, the Linux Test Project has become one of the de-facto verification suites used by developers, testers, and Linux distributors who contribute enhancements, bug fixes, and new tests back to the project.

2 Breathing a new life into LTP

LTP is now breathing a new life. New test cases have been added; many more test cases have been fixed between early 2007 and April, 2008. Table 1[2] provides details of test cases that have been added in the mentioned period, and as of the date of writing of this paper. Kdump test cases, by their very own nature, test the Kdump kernel; similarly, the Real Time Linux test cases are meant for the RT kernel. These additions show that LTP is a part of the daily testing activity of several people involved with kernel testing. It also shows the flexibility that LTP provides to test case writers and executors.

LTP [2] also saw a massive cleanup of the existing test cases. Around 350 patches were applied and 1000 new sources added, ending up modifying 1000 and removing 247 source files. The broken issues in LTP, one of the limitations preventing further adoption and expansion of the project, were also addressed effectively.

Some of the issues that were addressed as part of the LTP refresh are:

- The release pattern of file packages were revived to include results on various architectures. With that,

[1]Silicon Graphics Inc.

[2]Data as on 15th April 2008, from initial addition to subsequent patching

Name/Type	Total Sources	Avg. Code Size (bytes)
Kdump, Kdump for Network Partition dumps	26	2312
Uts, Sysvipc, & Pid Namespace	27	2614
Inotify	4	5894
Writev	7	7712
Swapon	4	8975
Numa	6	6986
Remap_file_pages	3	6465
Nfs Check Tests	1	1834
Posix_Fadvise & Fadvise64	5	4003
Madvise	4	6572
Sendfile64	7	5625
Arm Specific Test Cases	1	1091
Real Time Linux Test Cases	101	3400
Fallocate	5	7071
Filescaps	11	2579
Cpu Controllers	17	5134
Msgctl	12	7985
Ti-rpc	588	3218

Table 1: Specific List of Test Cases Added to LTP [4]

LTP achieved the release of 169 packages (totaling 265 MB of code), with 31458 packages downloaded overall, making an average of 65 downloads per day [1].

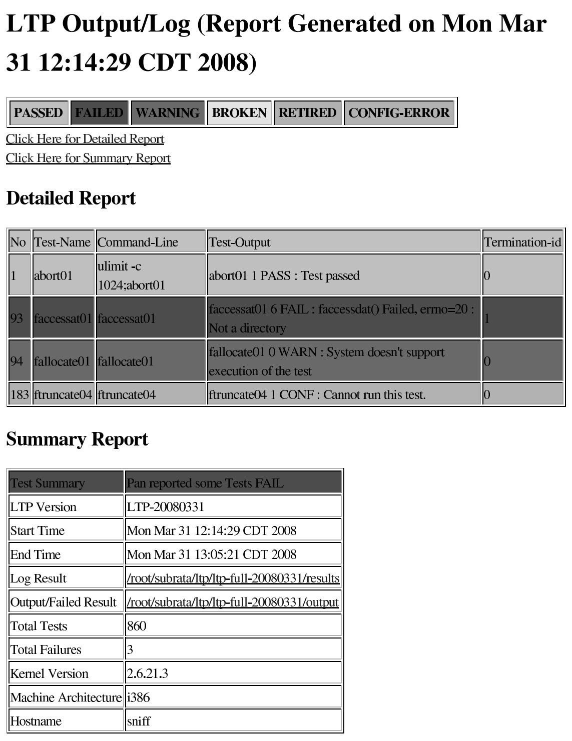
- Gcov-kernel patches for kernels 2.6.18, 2.6.19, 2.6.20, 2.6.21, 2.6.22, 2.6.23, 2.6.24 & 2.6.25 were also made available to the community through LTP.

- Addition of RHEL5 LSPP Test suite Release (EAL4 + Certification Test Suite).

- Addition of SGI Common Criteria EAL4 certification test suite for RHEL5.1 on SGI Altix 4700 (ia64) and Altix XE (x86_64) Systems.

While LTP worked hard to retain the confidence of the Linux community, it also saw a revamp of the testing infrastructure by providing:

- *Output logs in a more attractive/decipherable HTML format.* Figure 1 depicts the new HTML output format for LTP with clear distinction between FAILED, PASSED, WARNED, and BROKEN testcases. It is expected that HTML format can be used to show overall test status, and help to interactively explore failures. It is also envisioned that using XML in the future will allow the results to be validated and converted to attractive formats using style sheets; other such advantages of XML can be similarly exploited.



LTP Output/Log (Report Generated on Mon Mar 31 12:14:29 CDT 2008)

Detailed Report

No	Test-Name	Command-Line	Test-Output	Termination-id
1	abort01	ulimit -c 1024;abort01	abort01 1 PASS : Test passed	0
93	faccessat01	faccessat01	faccessat01 6 FAIL : faccessdat() Failed, errno=20 : Not a directory	1
94	fallocate01	fallocate01	fallocate01 0 WARN : System doesn't support execution of the test	0
183	ftruncate04	ftruncate04	ftruncate04 1 CONF : Cannot run this test.	0

Summary Report

Test Summary	Pan reported some Tests FAIL
LTP Version	LTP-20080331
Start Time	Mon Mar 31 12:14:29 CDT 2008
End Time	Mon Mar 31 13:05:21 CDT 2008
Log Result	/root/subrata/ltp/ltp-full-20080331/results
Output/Failed Result	/root/subrata/ltp/ltp-full-20080331/output
Total Tests	860
Total Failures	3
Kernel Version	2.6.21.3
Machine Architecture	i386
Hostname	sniff

Figure 1: A Sample new HTML format for LTP Output

- *Adding discrete sequential run capability.* LTP has an existing option to run the suite for definite period of time, say 24 hrs [7]. The drawback with this approach is that the test run can terminate midway without completing the last loop due to time pre-emption. This new feature allows the test to execute as many loops as specified by the user, irrespective of the time consumed. Particular test cases executed in multiple loops are properly tagged to distinguish outputs generated in multiple loops.

- *Auto Mail Back option of reports.* LTP now provides the option to collate all outputs and logs, `tar(1)` them and finally mail them back to a specified email address, after testing is complete. This can be handy in situations where the tests are run (in background) on remote servers for longer duration of time. On completion, the user gets the collated reports in his/her mailbox, a handy indication of the completion of the tests.

- *Generating default file for failed tests*. LTP now generates a file containing a list of exclusive test cases which *failed* during test run. This file is created in a format which then can be directly used to do a quick re-run of these failed tests. The user now can collate the output of only failed tests, and debug more efficiently.
- *Integrating better stress generation capability*. LTP employs a parallel infrastructure to create stress (I/O, memory, storage, network) on the system, to verify test case behavior under extreme condition(s). The full potential of this infrastructure was not exploited earlier. LTP now provides expanded options to utilize the existing features of stress generation.

In the recent past, there has been focus on running LTP tests concurrently [3]. Several fixes have been provided in this regard to allow tests to run concurrently.

The other area of focus has been to help developers write unit test cases without the need to download the original LTP-Suite. LTP development rpms for various architectures (i386, x86_64, ia64, ppc64, s390x, etc.) are now being regularly released to address this. This is to motivate developers to write unit test case(s) on their own, build them with the LTP development rpms, test them, and finally integrate them to mainstream LTP. Intermediate releases are now regular, which gives developers time to fix any build breaks before the final month-end release.

While we aim to increase the kernel code coverage, we also took a holistic look into the source code that we added to LTP suite during this transition period. The results showed that the LTP code has increased **42%**[3] starting 1st January 2007 till 15th April 2008. Though this is quite a small figure compared to what Linux kernel has grown, the most important thing to note is that the *same has been achieved by a very small group of LTP developers*.

[3]Data Generated from diffs of ltp-20061222 & ltp-20060415.

3 Kernel Code Coverage Statistics

One of the metrics to measure the effectiveness of testing is code coverage [10]. We've run coverage with the gcov patch (linux-2.6.24-gcov.patch[4]) on a x86 system and run different versions of LTP on the same kernel. However, during code coverage we have not considered Kdump tests, RT tests, DOTS, Open_Posix_Testsuite, Open_HPI_Testsuite, Pounder21 & SElinux testcases. Table 2 shows the code coverage for the top 10 items.

Directory	Coverage	
fs	49.8%	10135/20367 lines
include/asm	49.4%	595/1204 lines
include/linux	58.7%	2239/3812 lines
include/net	56.2%	990/1762 lines
ipc	52.8%	1442/2729 lines
kernel	38.2%	9880/25837 lines
lib	42.2%	2105/4992 lines
mm	51.5%	6899/13396 lines
net	65.4%	630/964 lines
security	51.9%	666/1283 lines

Table 2: 2.6.24 kernel code coverage using December 2006 LTP

Directory	Coverage	
fs	52.9%	10778/20367 lines
include/asm	50.9%	613/1204 lines
include/linux	60.0%	2283/3812 lines
include/net	57.6%	1015/1762 lines
ipc	56.4%	1539/2729 lines
kernel	39.1%	10097/25837 lines
lib	43.2%	2159/4992 lines
mm	52.7%	7066/13396 lines
net	65.7%	633/964 lines
security	51.9%	666/1283 lines

Table 3: 2.6.24 kernel code coverage using March 2008 LTP

The coverage was obtained by running December 2006 LTP against a gcov instrumented 2.6.24 release of the

[4]Available at `http://ltp.cvs.sourceforge.net/ltp/utils/analysis/gcov-kernel`

kernel. Table 3 also shows the code coverage for the top 10 items. This coverage was obtained by running March 2008 LTP against a gcov-instrumented 2.6.24 release of the kernel.

Comparing Tables 2 and 3, we make the following observations:

- Between the two runs, the coverage of the recent LTP is better. This is a good sign and is indicative of the progress that LTP has made. Table 4 shows the percentage increase in coverage between December, 2006 and March, 2008.

Subsystem	**% Increase**
Filesystems	3.1
include/asm	1.5
include/linux	1.3
include/net	1.4
ipc	3.6
kernel	0.9
lib	1.0
mm	1.2
net	0.3
security	0

Table 4: Increased coverage due to LTP enhancements between Dec 2006 & March 2008.

- Two subsystems, *fs* and *include/asm*, now have coverage greater than 50 percent.
- The data also points us to some interesting facts, such as:
  - LTP needs to do a better job of covering the error paths. Some of them need to be covered using the fault injection framework. One limitation of LTP is that the test cases cannot handle faults from the kernel. The test case exits on failure. We propose a new *LTP robust* subproject to allow LTP to work well with fault injection.
  - It is not possible for LTP to cover certain scenarios. With a wide set of permutable config and boot options, it is not possible to test every config/boot option and extract coverage. We've tested the most common and minimal configuration that works on our machine.
  - It is not possible for LTP to handle coverage of code that is not exposed to user space. For example, a machine may be configured with SPARSEMEM, FLATMEM or DISCONTIGMEM. Testing these options and obtaining coverage data is not possible.
  - There are several areas of code that have no coverage. We've taken up those areas as areas that need more test cases. Section 7 provides more details about our future plans.
- We intend to make code coverage data available to the LTP website,[5] so that developers can see how well their code is tested. This might even motivate them to contribute the test cases they've used for testing the feature to LTP.

4 Role of LTP in testing Linux

Software testing can be broadly categorized into

- Compilation Testing[6]
- Unit Testing
- Functional Testing
- System Testing
- Stress Testing
- Performance Testing

LTP helps with *Functional*, *System* and *Stress* testing. LTP cannot directly do *Compilation*, *Performance* or *Unit* testing.

There are several ways of testing the Linux kernel. Most developers run the latest kernel on their desktops and servers. The kernel gets tested via the applications that get executed. Any major performance regression is observed and reported.

LTP goes a step further by providing test cases that test user interfaces with several valid and invalid parameters. It tests various subsystems of the kernel such as

[5]`http://ltp.sourceforge.net/`

[6]Many textbooks on software testing, do not include build as a part of the test effort. Since Linux runs on several platforms and has several features that can be enabled/disabled at compile time, ensuring that the build works well across the platforms, architectures and features is an important aspect of testing Linux

the memory management code, the scheduler, system calls, file systems, real time features, POSIX semantics, networking, resource management, containers, IPC, security, timer subsystem and much more. LTP provides an infrastructure to stress test the system by:

- Providing test cases that stress the system.
- Allowing concurrent execution of test cases.
- Providing noise in the background (CPU, Memory, Storage, Network, etc.) while running tests.

LTP plays an important role in system testing. Several users of LTP use it to validate their entire system. Running LTP validates the "C" library and the user interface(s) provided by the kernel (to the extent test cases have been added).

LTP is also run by kernel testers for regression testing. Given the size and nature of the LTP test cases, it provides a good framework for executing desired tests, selecting a subset of those tests as basic acceptance test, and running them.

In the future, we intend to enhance LTP to provide facilities for performance testing[7] and more test cases that can test the functionality of features not yet in the mainline Linux kernel. This would help provide extensive testing of a feature before it gets into the mainline Linux kernel.

5 Early and Effective Testing

Up to a point it is better to let the snags [bugs] be there than to spend such time in design that there are none (how many decades would this course take?)
A M Turing, Proposals for ACE (1945)

The importance and significance of effective and early testing cannot be stressed enough. According to Barry Boehm's Software Engineering Economics [5], the time required to identify a defect in software after it has been deployed is 40 to 1000 times longer than if had been found in the requirements analysis. While testing cannot really catch bugs introduced in the requirements phase, it certainly can help catch them before the code is deployed.

[7]By providing a performance testing framework

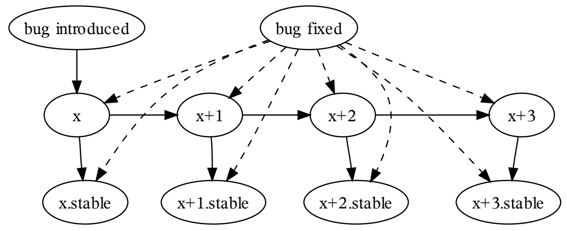


Figure 2: Sample bug fix flow for a bug introduced in version **x** and fixed in version **x+3**.

Consider a hypothetical example of a feature that introduces a bug into version *x* of the kernel. The bug is tested and detected in version *x+3*. Figure 2 shows in dotted lines the versions into which the bug needs to be fixed. If Linux distributions have spawned off kernels in between versions *x* and *x+3*, then more bug fixing, hot fixes and updates need to take place.

The example scenario above shows the advantage of early and effective bug fixing. Had the bug been detected in version *x* itself, the unnecessary overhead of bug-fixing, maintenance and additional testing could have been avoided.

This brings up an important question: To what extent do we test the kernel? We believe that all bugs that can be caught easily and with some effort and observation must be discovered and fixed. Turing's quote at the beginning of this section refers to a good trade off between bugs and time spent.

6 Simplest way to write a LTP test case

There have been papers written by LTP Maintainers/developers regarding ways/methodology to write a simple LTP test case. Notably amongst them are:

- Testing Linux with the Linux Test Project [9] , and
- Improving the Linux Test Project with Kernel Code Coverage Analysis [8]

All of these are easily available in archives, hence we skip the intricate details of writing a testcase. We instead focus on presenting a set of workflows to depict the overall mechanism to run the LTP suite, and individual test case execution.

6.1 LTP Suite Execution Framework

Figure 3 depicts the flow of how the entire LTP suite works. On invocation, ***runltp*** script parses all options. It proceeds to generate the list of test cases to be executed depending on user choice at command line. It exports all the identifiers necessary for test execution next. Optionally, it can also generate certain stress on the system. Next, it invokes the test driver PAN, which then takes care of executing each test case in the list. Once all test cases are executed, PAN reports PASS if all test cases have executed successfully. Else, it returns fail if at least one of them failed. Generation of HTML output and auto-mail-back is optional. Once that is over, the script does the necessary cleanups (releasing resources, clearing system stress, etc.) and exits.

6.2 Individual Test Execution Framework

Figure 4 shows the preamble, which starts with mentioning the copyright statement(s) followed by the GPLv2 declaration (which is mandatory). Following that, the test case name and algorithm are described. Modification history is maintained to identify sources of this code modification: the author, date and reason of modification.

Figure 5 shows the the main body of the test case. It starts by including headers that declare general and LTP-specific global and static identifiers. Once inside the *main()* block, the first thing is to check whether the feature under test is supported by this kernel version/architecture/FS type/glibc version. If any of these evaluate to false, the test is aborted, corresponding message written to logs/output, cleanups done and test exits with proper exit-value. If everything is supported, the main code of testing is executed.

If there are some BROKEN or WARNING messages generated, then the test takes appropriate action. Assuming everything goes well, test execution status is written to log/output, cleanups are done, and the test exits with a proper return value.

7 Future Plans

Several initiatives have been taken so far to improve LTP. Most of them have been successful. We plan to take up more such initiatives in future. As a part of

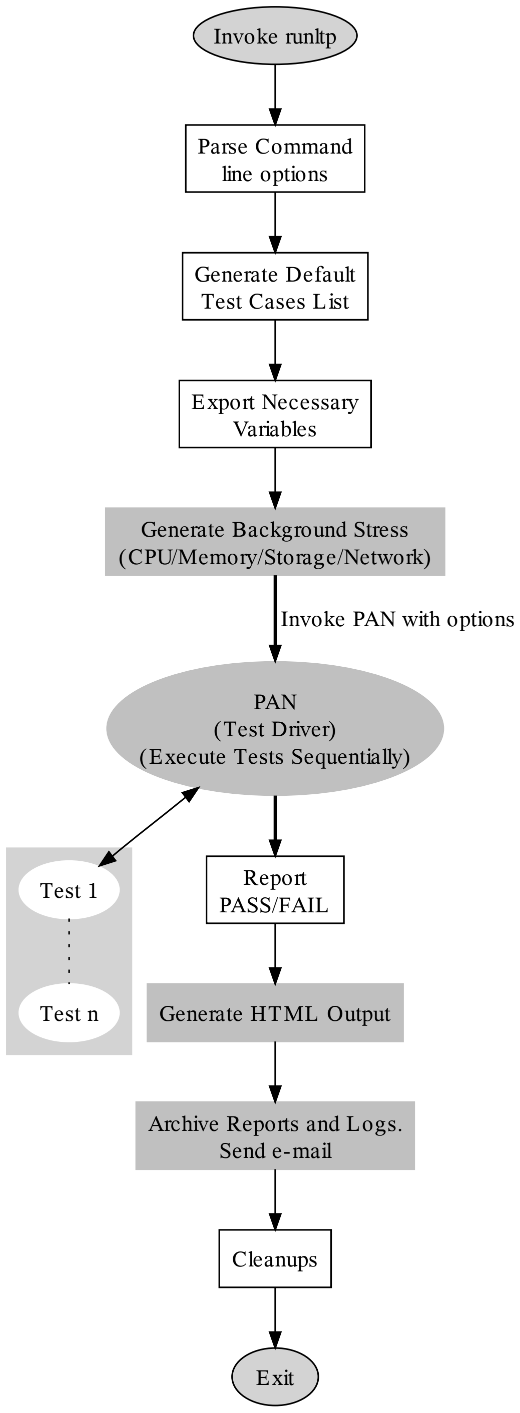

Invoke runltp
Parse Command line options
Generate Default Test Cases List
Export Necessary Variables
Generate Background Stress (CPU/Memory/Storage/Network)
Invoke PAN with options
PAN (Test Driver) (Execute Tests Sequentially)
Test 1
Test n
Report PASS/FAIL
Generate HTML Output
Archive Reports and Logs. Send e-mail
Cleanups
Exit

Figure 3: LTP Flow Diagram

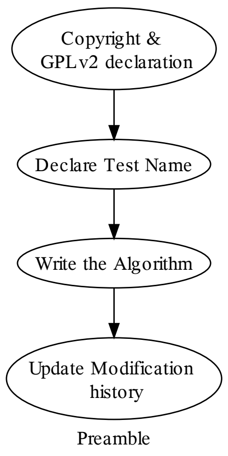


Figure 4: LTP Test Flow Preamble

LTP's initiative to involve kernel developers in particular, we have already started with the concept of LTP development packages. These are a combination of LTP-specific libraries, header files, executables and man-pages, easing developers' task of developing unit test cases, for the features that they plan to merge to the kernel.

Another initiative is starting the LTP-*mm* tree. Developers might not wait for their features to be part of mainline kernel and then open up the test cases. The same test cases can be contributed to the LTP-*mm*. Test cases can be contributed to the LTP-*mm* project as early as the corresponding feature hits any kernel tree (mm,rc,etc.), or planning to get into any tree. The test case(s) themselves can be modified multiple times in resemblance to the corresponding feature changes/modification in the kernel tree. Once the feature becomes part of mainline kernel, the corresponding test cases are moved from LTP-*mm* project to main LTP project.

While we will be happy to have those test cases use the LTP-specific logging libraries, it is not a mandatory requirement. If the test case(s) is/are written in C/Shell and returns 0/1 on PASS/FAIL, then it is a very good candidate for inclusion into the LTP. We encourage kernel developers to contribute their unit test cases in whatever form they have. The LTP community will help them in converting them to the required format across time. We would also urge test cases to find their way to the LTP in many unexplored areas, such as device drivers, and also in areas where the **kernel code coverage is very low.**

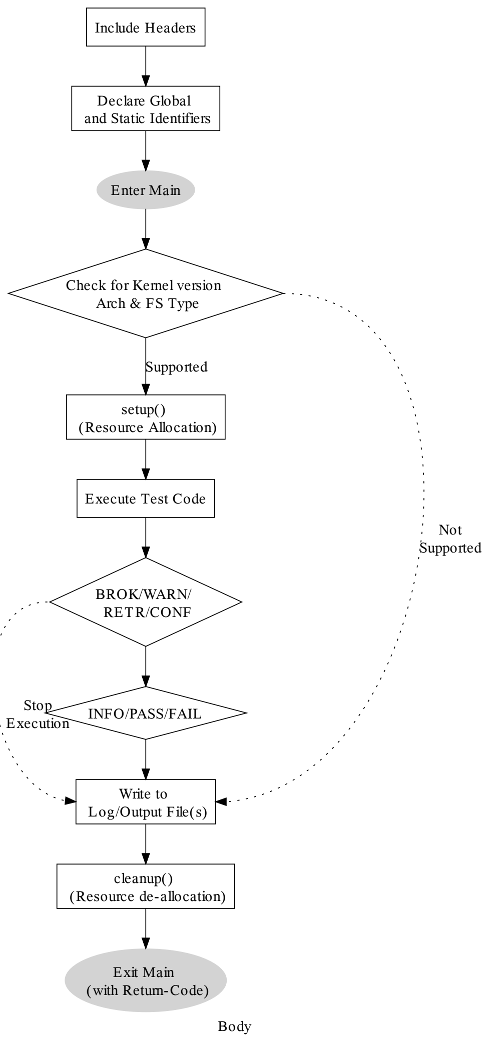

Include Headers
Declare Global and Static Identifiers
Enter Main
Check for Kernel version Arch & FS Type
Supported
Not Supported
setup() (Resource Allocation)
Execute Test Code
BROK/WARN/ RETR/CONF
Stop Execution
INFO/PASS/FAIL
Write to Log/Output File(s)
cleanup() (Resource de-allocation)
Exit Main (with Return-Code)
Body

Figure 5: Individual Test Flow Diagram

When we request the community to contribute to test case(s) development, we also want to convey to you that the LTP aims to include new test cases in the areas of:

- Power Management testing,
- Controllers and Containers testing,
- KDUMP (kdump on Xen hypervisor and guests),
- Union Mount,
- Sharedsubtree, etc.

SAMPLE LTP OUTPUT

```
<<<test_start>>>
tag=remap_file_pages01 stime=1208361993
cmdline="remap_file_pages01"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
remap_file_pages01
```
1 PASS : Non-Linear shm file OK

```
<<<execution_status>>>
duration=1 termination_type=exited
termination_id=0 corefile=no
cutime=7 cstime=2
<<<test_end>>>
<<<test_start>>>
tag=faccessat01 stime=1208362004
cmdline="faccessat01"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
faccessat01
```

6 FAIL : faccessdat() Failed, errno=20 : Not a directory

```
<<<execution_status>>>
duration=1 termination_type=exited
termination_id=0 corefile=no
cutime=8 cstime=2
<<<test_end>>>
<<<test_start>>>
tag=fallocate01 stime=1208363009
cmdline="fallocate01"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
fallocate03
```

0 WARN : System doesn't support execution of the test

```
<<<execution_status>>>
duration=1 termination_type=exited
termination_id=0 corefile=no
cutime=8 cstime=2
<<<test_end>>>
```

Figure 6: Sample LTP Output

in very near future. We will continue to fix and improve upon the existing testcases. Forthcoming enhancements in the area of LTP infrastructure include:

- *Development of XML logs/output*. We plan to generate the logs/output in XML format so that they can be parsed easily by any XML parser.

SAMPLE LTP LOG

```
Test Start Time: Wed Apr 16 21:47:41 2008
-----------------------------------------
Testcase               Result     Exit Value
--------               ------     ----------
remap_file_pages01     PASS       0
faccesat01             FAIL       1
fallocate03            WARN       1
-----------------------------------------
Total Tests: 2
Total Failures: 0
Kernel Version: 2.6.18-53.1.13.el5
Machine Architecture: i686
Hostname: <sniff>
```

Figure 7: Sample LTP Log

- *.config File based Execution*. Options to run LTP are growing. It may be difficult for users to remember and mention all options at command line. We plan to provide .config file, which will host all such options in *variable=value* format. Users will be required to just run *runltp*, which will automatically parse options from the *.config* file.

- *Network based installation, execution and report collection*. We plan to create an infrastructure where it will be possible to provision LTP on multiple machines from a central machine. The provisioning server will be capable of deploying LTP suite across multiple machines, build & install, run, and bring back all of the reports.

Recently, we identified that many LTP test cases fail while running concurrently. We plan to make the entire LTP suite **concurrency-safe** [3]. We also plan to test all kernel releases *(mm, rc, etc.)* and make those results available on the LTP website, along with the code coverage details against the latest stable kernel. In the near future, LTP will continue to focus on all possible means to improve code coverage. However, in the long term, we will also consider adding **benchmark** infrastructure to LTP.

8 Conclusion

As Linux testing evolved through the ages, other test projects/suites with better infrastructure/features came to the center-stage. LTP is considered to have the following defects: Lacking the facility to provide automatic kernel build/reboot and test, having low code coverage, non-parsible output logs, and broken test cases. While it is true that the LTP does not provide autobuild and test options, that was not the main focus area. It was designed to be a very handy regression test suite. LTP used in conjunction with other test suite(s) can complement each other's features to **implement the much broader goal of making Linux better**.

The *kernel code coverage cannot be drastically improved without the corresponding test cases for kernel features being made available to the LTP*. While this **responsibility cannot be enforced**, the impact of such can be brought to developers notice. Each section of logs/output are properly tagged, which in turn can be parsed by even a simple parser. **Figure 6** depicts a sample test output, and, **Figure 7** shows a sample log file contents. The HTML output depicted in Figure 1 is testimony to the fact that LTP log/output can be parsed very easily. Hence, a simple parser was able to generate this HTML from a normal text output. And this very trivial output format will also enable us to write the future XML parser.

Many test cases were found to be broken, as many kernel features have undergone changes, and, the same testcases were not cleaned. LTP clearly distinguishes the way test cases should report results, with keywords like INFO, BROK, CONF, RETR, PASS, FAIL, etc. well documented in the LTP manpages. Elaborate information about the test case behavior is also reported along with these keywords.

Everybody can put forth their views of how LTP should move forward, what it should address, and what it should avoid. The LTP community highly appreciates patches, the benefit of which goes directly to all. LTP has discovered opportunity in positive criticism, and has focused with more vigor on its primary goal of providing a functional and regression test suite. It will keep growing at any cost along with the growing kernel, while simultaneously addressing bottlenecks in other areas too. **However, we need more active contribution from the kernel developers to make LTP a very strong and useful test suite for the Linux kernel.**

Acknowledgments

We would like to thank Robert Williamson, IBM, for his input to and review of drafts of this paper. We also owe lot of thanks to Sudarshan Rao, Premalatha Nair, and our team mates for their active support and enthusiasm. And a special thanks to all those LTP developers whose immense contribution keeps this project growing.

Legal Statement

References

[1] Linux test project download page. `https://sourceforge.net/project/showfiles.php?group_id=3382`.

[2] Linux test project home page. `https://sourceforge.net/projects/ltp`.

[3] Linux test project mailing list. `https://sourceforge.net/mailarchive/forum.php?forum_name=ltp-list`.

[4] Linux test project source code repository. `http://ltp.cvs.sourceforge.net/ltp/`.

[5] B. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[6] N. Hinds. Kernel korner: The linux test project: Finding 500 bugs in 50 different kernel versions is the fruit of this thorough linux testing and code coverage project. *Linux Journal*, 2004. `http://www.linuxjournal.com/article/7445`.

[7] M. Iyer. Linux test project documentation howto. `http://ltp.sourceforge.net/documentation/how-to/ltp.php`.

[8] P. Larson. Improving the linux test project with kernel code coverage analysis. Linux Symposium 2003.

[9] P. Larson. Testing linux with linux test project. Linux Symposium 2002.

[10] P. Larson, R. Williamson, and M. Ridgeway. Linux test project technical papers. `http://ltp.sourceforge.net/documentation/technicalpapers`.

Have You Driven an SELinux Lately?

An Update on the Security Enhanced Linux Project

James Morris
Red Hat Asia Pacific Pte Ltd
jmorris@redhat.com

Abstract

Security Enhanced Linux (SELinux) [18] has evolved rapidly over the last few years, with many enhancements made to both its core technology and higher-level tools.

Following integration into several Linux distributions, SELinux has become the first widely used Mandatory Access Control (MAC) scheme. It has helped Linux to receive the highest security certification likely possible for a mainstream off the shelf operating system.

SELinux has also proven its worth for general purpose use in mitigating several serious security flaws.

While SELinux has a reputation for being difficult to use, recent developments have helped significantly in this area, and user adoption is advancing rapidly.

This paper provides an informal update on the project, discussing key developments and challenges, with the aim of helping people to better understand current SELinux and to make more effective use of it in a wide variety of situations.

1 Introduction

The scope of this paper is to cover significant advances in the SELinux project since its initial integration with mainstream Linux distributions. For context, a brief technical overview of SELinux is provided, followed by project background information.

1.1 Technical Overview

SELinux is a flexible MAC framework for Linux, derived from the Flask research project [30], and integrated into the Linux kernel via the Linux Security Modules (LSM) API [34].

All security-relevant accesses between subjects and objects are controlled according to a dynamically loaded mandatory security policy. Clean separation of mechanism and policy provides considerable flexibility in the implementation of security goals for the system, while fine granularity of control ensures complete mediation.

An arbitrary number of different security models may be composed (or "stacked") by SELinux, with their combined effect being fully analyzable under a unified policy scheme.

Currently, the default SELinux implementation composes the following security models: Type Enforcement (TE) [7], Role Based Access Control (RBAC) [12], Muilti-level Security (MLS) [29], and Identity Based Access Control (IBAC). These complement the standard Linux Discretionary Access Control (DAC) scheme.

With these models, SELinux provides comprehensive mandatory enforcement of least privilege, confidentiality, and integrity. It is able to flexibly meet a wide range of general security requirements:

- Strong isolation of applications.
- Information flow control.
- Ensuring critical processing flow.
- Protection of system integrity.
- Containment of security vulnerabilities.

SELinux is also able to meet the requirements of and interoperate with traditional MLS systems.

1.2 Project Background

Based on the growing need for stronger security [19], SELinux was released as an open source project by the

US National Security Agency in December 2000. A vibrant community formed around the project, with Debian developers leading efforts to integrate SELinux into a mainstream distribution. Subsequent early integration efforts were also largely community-led, with individual developers undertaking SELinux packaging work for the Gentoo, SuSE, and Red Hat distributions.

In March 2001, the NSA presented SELinux to senior kernel developers at the Linux Kernel Summit. Feedback from the summit led to the formation of the LSM project—the aim of which was to allow different security mechanisms to be plugged into the core kernel.

SELinux helped to drive LSM requirements, while SELinux developers participated as core contributors in LSM development. SELinux was ported to the resulting LSM API, and then merged into the mainline Linux kernel for the kernel's v2.6 release in December 2003.

Concurrently, SELinux was fully integrated into the Fedora Core distribution, version 2 of which was released in May 2004. This marked the first release of an operating system with official support for SELinux. The security policy distributed with Fedora Core 2 was largely derived from the initial NSA example policy, as well as from ad-hoc community contributions. It had not yet been well-tuned to many general Linux deployment scenarios and was a noted source of frustration in causing many applications to fail unexpectedly due to policy violations.

In some cases, SELinux was blocking previously undetected programming errors (such as the leaking of open file descriptors), although the initial security policy was also generally regarded as being too strict. A solution was implemented during the development of Fedora Core 3 (released November 2004), whereby only critical applications were confined by policy. This policy was designated as *targeted*, and was considered to be a major improvement in usability. It allowed SELinux to be enabled by default, leading to what is believed to be the first ever release of a mainstream operating system with mandatory security enabled in a standard configuration.

Red Hat Enterprise Linux 4, based on Fedora Core 3, shipped a few months later in February 2005 as the first commercial distribution with full SELinux support.

These initial major distribution releases, while shipping with relatively few confined services by default, provided useful mandatory security out of the box. They were key milestones in the the transition of SELinux from a research project into a production-deployable system.

With SELinux now being distributed to the wider community, the project began to evolve rapidly in terms of both function and usability.

2 Policy

Early versions of targeted policy confined only a few critical and typically network-facing applications. Ongoing contributions from the community, in conjunction with continued efforts by core project developers, led to a steady increase in the number of applications confined by policy. As of April 2008, there was policy support for several hundred applications in the upstream repository.

2.1 Booleans

The *Booleans* facility[1] was developed to allow certain policy features to be selectively enabled or disabled without having to reload policy. This was originally designed with smaller systems in mind, but has proven to be significantly useful in the general case.

Feedback from the community indicated that there were common patterns in the way applications would be customized. Booleans were able to provide a suitable high-level mechanism for administrators to effect limited yet useful customization of policy according to these patterns.

For example: policy for an FTP server might have a boolean which enables remote access to user home directories. Rather than having to modify or even know any SELinux policy, the administrator could instead control this behaviour by simply running a command to enable or disable the associated boolean.

2.2 Loadable Policy Modules

SELinux initially shipped with a monolithic policy scheme, requiring a full rebuild of policy from source for any modification. The Loadable Policy Module architecture [20] was introduced to provide a more robust and flexible approach to composing and deploying policy.

[1] also referred to as *Conditional Policy*

With this new architecture, binary policy modules could be loaded dynamically, eliminating the need for systems to carry a full policy source tree and associated build infrastructure in case customization was required.

The core operating system policy was separated into a base policy module, allowing it to be streamlined and further tailored for different use scenarios. Higher-level components could now be developed and managed independently, enabling the distribution of third-party policy, as well as the ability to incorporate policy modules with application packages.

Loadable Policy Modules were first shipped with Fedora Core 5 in March 2006.

2.3 Reference Policy

One of the most significant advances in the SELinux project has been the *Reference Policy* effort [28]. This was a fundamental reworking of the policy framework to provide a more structured basis for policy design and analysis. A critical goal was to improve the quality of policy and thus facilitate increased overall assurance.

Key elements of the Reference Policy effort included:

- Introducing the principles of layering and interfaces to the policy language to facilitate better abstraction and modularization.
- Leveraging the new Loadable Policy Module architecture discussed in Section 2.2.
- Building support for documentation into the infrastructure to encourage the practice of literate programming [17].
- Easier configuration of security models, so that features such as MLS may enabled without requiring separate policy source trees.
- Porting the NSA example policy to the new framework.

For policy developers, Reference Policy meant a greatly simplified view of policy. Much of the low-level complexity was abstracted away. Design could be performed in a modular fashion, utilizing well-defined and documented interfaces. This facilitated a greater focus on high-level security goals and better comprehensibility of policy.

Reference Policy was first shipped along with Loadable Policy Modules in Fedora Core 5.

2.4 Other Developments

The (somewhat experimental) MLS functionality supplied with the original SELinux release was significantly revised to meet certification requirements and to be more useful in the real world. This included policy enhancements and the ability to enable MLS dynamically. MLS was previously a kernel compile-time option and not enabled at all in mainstream distributions.

The MLS infrastructure was also adapted to a new security model, Multi-Category Security (MCS) [24], which utilized the *category* attribute of MLS labels and a simple policy to provide end users with a discretionary labeling scheme. Fedora Core 5 shipped with MCS enabled by default, allowing much of the MLS infrastructure to be exercised by general users. The future of MCS is currently unclear, as its wider adoption requires extension to applications, and it may be better to instead utilize TE for the same purpose.

The RBAC scheme was greatly improved, allowing roles to be defined and loaded as policy modules, rather than having to be managed as part of the previous monolithic policy scheme.

By the time of writing, targeted and strict versions of policy in the upstream repository had been merged into a single version. Targeted behavior is now selected by including the "unconfined" policy module. Strict behavior may also be re-selected incrementally by mapping users to confined roles.

3 Toolchain and Management

Major changes such as the Loadable Policy Modules and Reference Policy projects entailed reworking much of the low level SELinux infrastructure. This provided opportunities to improve the function and usability of the toolchain, and to implement a foundation for the development of high-level SELinux applications.

The following are highlights of recent advances made in SELinux toolchain and management technology.

3.1 libsemanage

The extensible *libsemanage* library was developed in conjunction with Reference Policy as the first programmatic API for applications which need to manipulate policy. Such applications range in scope from the core system management utilities and scripts through to graphical policy design and management tools.

3.2 semanage

The *semanage* command line tool was introduced as a means to unify low-level SELinux administrative tasks, many of which, prior to Reference Policy, involved editing disparate policy source files and rebuilding policy.

For the system administrator, *semanage* improved overall usability by providing a well documented, canonical utility for managing key aspects of an SELinux system. Examples of *semanage* use include configuring local file labeling rules and the labeling of system objects such as network ports.

Recently, much the core of *semanage* has been refactored into a Python module to enable re-use by other management tools such as *system-config-selinux*.

3.3 system-config-selinux

system-config-selinux is a GUI tool for comprehensive SELinux system management. It has been integrated into Fedora-based distributions, and serves as a graphical alternative to *semanage*.

A flexible underlying Python-based architecture should make it readily adaptable to the management schemes of other distributions and operating systems.

3.4 Loadable Policy Module Tools

- *checkmodule* is the policy module compiler. It verifies the correctness of a policy source module then converts it into a binary representation.
- *semodule_package* bundles a binary policy file as created by *checkmodule* with optional related data such as file labeling data into a format ready to be installed by *semodule*.
- *semodule* is the core policy module management tool. It is used for installing, upgrading, querying and deleting binary policy modules.

3.5 Boolean Management

setsebool and *getsebool* are the standard command-line utilities for managing the state of policy Booleans (see Section 2.1). Booleans may also be managed via *system-config-selinux*.

3.6 restorecond

A common early issue for administrators was that some files were particularly susceptible to being mislabeled, such as the */etc/resolv.conf* file being recreated by certain system tools.

The *restorecond* utility was developed to automate relabeling of such files, reducing administrative burden.

Files to be monitored are listed in a configuration file (typically */etc/selinux/restorecond.conf*). *restorecond* utilizes the Inotify subsystem to detect changes to any of these files, then performs relabeling if needed.

Note that file labeling is usually handled transparently by SELinux policy, or by enabling system tools to preserve security labels on files. As such, *restorecond* is an optional SELinux component aimed at improving usability on general purpose systems.

3.7 setroubleshoot

setroubleshoot is a GNOME facility which triggers a user alert upon SELinux policy violations. The alert notifies the user of the problem and assists in resolving the issue or filing a report with SELinux developers. *setroubleshoot* may also be configured to send email alerts to an administrator for centralized management.

setroubleshoot utilizes a pluggable rule database and attempts to heuristically determine problem causes.

Community feedback indicates this facility has significantly improved the SELinux user experience, by helping to resolve issues encountered, and by giving users a clearer understanding of what is happening on their system when SELinux prevents an access.

setroubleshoot first shipped with Fedora Core 6 in October 2006.

3.8 SELinux Policy Management Server

The *SELinux Policy Management Server* [21] is an ongoing project which addresses several requirements in the management of policy on an SELinux system. Its aim is to provide a platform for installing, updating and querying SELinux policy on production systems, with the ability to safely delegate administration of policy to separate users.

Currently in a prototype phase, a notable planned feature of the policy management server is support for remotely managing a single policy across multiple machines.

4 Policy Development

Since the introduction of Reference Policy, there have been several advances in the area of authoring SELinux policy.

4.1 Command Line Tools

While SELinux policy is ideally shipped with the system and managed with high-level tools, administrators may still wish to develop their own enhancements to policy. This task has been simplified somewhat with modular policy and new or enhanced command-line tools.

4.1.1 audit2allow

The *audit2allow* utility parses the audit log and converts access denial records into security policy. It has proven to be a valuable tool in resolving local policy issues.

For example, in the case of an application not having any SELinux policy (e.g., locally developed or provided by a third party), a policy module may be developed for it with a few simple commands [31]. The system is configured in permissive mode, so that access denials will be logged but not be enforced, and then the audit log is passed to *audit2allow* to generate a binary policy module. The new module may then be loaded into the system via *semodule*. This is a form of "learning mode." It is always recommended that the administrator review the resulting policy, and to request further review from the community if necessary.

4.1.2 audit2why

The *audit2why* tool was developed to help administrators better understand SELinux audit messages. It takes raw audit logs as input and analyzes them to determine which policy component triggered a particular audit message.

Often, the audit messages alone do not provide enough information to trace an access denial back to the associated policy component. Access denials have several possible causes: missing TE rules, missing RBAC rules, and policy constraints. *audit2why* is able to pinpoint precisely which.

Recently, *audit2why* was extended to determine which policy Boolean, if applicable, may be modified to resolve the issue.

4.2 SLIDE

The *SELinux Policy IDE (SLIDE)* is a sophisticated GUI policy development environment, implemented as an Eclipse plugin. It is aimed at making policy development easier, and includes many developer-oriented features such as syntax highlighting, project exploration, auto-completion, wizards, and refactoring support.

SLIDE also includes support for testing, deploying, and remotely managing policy. It was recently integrated into the Fedora distribution as an official package.

4.3 Policy Druid

A policy generation druid is included in *system-config-selinux*. This is a simple graphical wizard which presents the user with a series of questions about an application based on common security traits. A policy module is then automatically generated for the application, which may then be further managed via *system-config-selinux*.

No knowledge of the SELinux policy language is required. This tool is useful for rapidly generating policy with broad confinement properties.

4.4 SEEdit

The *SELinux Policy Editor (SEEdit)* [27] is a graphical tool which allows users to develop policy in a simplified policy language. Introduced by Hitachi Software in

2005, *SEEdit* utilizes pathname-based configuration and is targeted at developing policy for embedded systems.

4.5 SETools

A powerful set of policy analysis tools has been developed by Tresys and packaged into the *SETools* [6] suite. Included in the suite are utilities for analyzing SELinux policy, analyzing audit messages and creating audit reports, and verifying and examining policy.

4.6 CDS Framework Toolkit

The *CDS Framework Toolkit* [1] is a high-level tool for designing SELinux policy for "cross-domain solutions" (CDSs). CDSs are specialized guard systems for controlling information flow between different security domains.

While typically employed for sensitive government and military use, the underlying principles of have more general applications, such as in the case of a corporate Internet gateway which filters email and other user traffic. As SELinux is able to enforce critical processing flow,[2] security policy can be used to ensure, for example, that incoming email is always passed through a virus checker and a spam filter.

The *CDS Framework Toolkit* utilizes a GUI and abstract representation of the system, and does not require detailed knowledge of SELinux policy on the part of the user.

5 Networking

SELinux provides fine-grained controls over network accesses at several layers of the networking stack. At the socket layer, all socket system calls are mediated. Specialized controls are implemented for critical networking protocols (such as Unix domain sockets), the IP layer, and the interface layer.

Since the initial releases of SELinux, several aspects of networking support have been reworked or newly implemented.

[2]also referred to as an *assured pipeline*

5.1 Secmark

The first release of SELinux included rudimentary IP layer controls for packet flow. An effort called *Secmark* [25] was undertaken in 2006 to modernize the IP layer controls, leveraging the rich firewalling capabilities available in the Linux kernel.

With *Secmark*, *iptables* rules are used to label packets based on attributes which can be determined from the packet alone (e.g., destination port). Packet flow is then mediated using these labels according to SELinux policy.

This allows the utilization of all available iptables matches as selectors, as well as features such as connection tracking (or "stateful inspection"). Use of the latter leads to greatly simplified network packet policy. A single rule can be used, for example, to permit the flow of all traffic in a validated "established" or "related" state, eliminating the need to explicitly allow (or ignore) ephemeral port use, and to automatically handle multi-connection protocols such as FTP.

The code for this has been merged upstream, although distribution integration is not yet complete, as there is currently an outstanding issue of how best to perform the integration without clashing with current users of iptables.

5.2 Labeled Networking

While *Secmark* provides local labeling of network traffic, there is also a need for the ability to mediate traffic based upon remote characteristics such as the security context of the peer application. This is achieved by conveying security labels with the traffic as it transits the network.

SELinux takes two approaches to this:

- **Labeled IPSec**

  This essentially involves labeling IPSec security associations (SAs) to implicitly label traffic carried over those SAs. This was derived from earlier Flask research [9], implemented for SELinux by an IBM team [15], and further refined by Trusted Computer Solutions for better MLS support and improved usability.

- **NetLabel**

  Legacy trusted systems utilize IP options to convey security labels across the network. It is desirable for SELinux to interoperate with these systems. A flexible implementation based on the abandoned CIPSO [14] standard has been developed and integrated into the kernel by HP.

All of the above network labeling schemes were designed to be security framework agnostic, and are available for use by other security modules.

6 Memory Protection

Linux systems take a layered approach to security, to provide "defense in depth," ensuring that security measures are implemented at every possible level. No single measure can defeat all attacks, but many attacks can be defeated by a combination of measures.

SELinux utilizes kernel-based mechanisms to confine the behavior of userland applications. It is thus not designed to protect directly against kernel bugs, nor certain classes of application issues such as memory-based attacks, although it can help confine the damage done by such attacks.

Many distributions ship with mechanisms to protect against memory-based attacks, such as support for non-executable pages (NX) and glibc memory checks. In some cases, applications may wish to override memory checks (e.g., certain virtual machine interpreters), and SELinux has been extended to allow these overrides to be controlled by SELinux policy [11].

Enforcement of these memory checks via SELinux policy has led to the discovery of several applications which were inadvertently performing dangerous memory operations. While initially causing inconvenience to users, many of these applications were subsequently fixed, while developer awareness of the underlying issues was increased.

7 Security Evaluation and Accreditation

For some classes of government and military users, security certification is an important procurement requirement. The integration of SELinux made it possible to have Linux certified to the highest level currently achieved by mainstream operating systems.

In 2007, Red Hat Enterprise Linux version 5 in a server configuration was certified under the Common Criteria Evaluation and Validation Scheme (CCEVS) to Evaluation Assurance Level 4 Augmented (EAL4+), against the protection profiles:

- LSPP: Labeled Security Protection Profile
- RBACPP: Role Based Access Control Protection Profile
- CAPP: Controlled Access Protection Profile (Audit)

As CCEVS certifications include hardware, certification was performed for both HP and IBM platforms. A similar certification is currently underway for SGI hardware. EAL4+ is the highest assurance level likely achievable without a specially designed operating system, while LSPP is an updated equivalent of the earlier Trusted Computer System Evaluation Criteria (TCSEC or "Orange Book") B1 rating.

Several aspects of the certification are notable. It was the first time that Linux itself was known to be subjected to such a rigorous security validation process. Linux has now become directly competitive in the marketplace with existing "trusted" operating systems. An innovative approach was taken such that the Linux certifications were performed on the standard product line, rather than on separately maintained versions. Also innovative was the cooperative and open community certification effort, which led to a pooling of resources between several different companies and organizations.

The certification efforts also led to the development of several enhancements to SELinux and Linux itself, such as an overhaul of the Audit subsystem, and the introduction of polyinstantiated directories via Linux namespaces and the Pluggable Authentication Modules (PAM) subsystem. Many of these certification-related developments have proven generally useful, with a notable example being the introduction of *Kiosk Mode* (see Section 12).

SELinux-based systems have also been accredited on a per-system basis, independently of CCEVS. Details of such accreditations are often not public, although a case study was presented at the 2007 SELinux Symposium where an SELinux-based system was developed to allow US Coast Guard Intelligence to consolidate access to separate classified networks [16].

The Certifiable Linux Integration Platform (CLIP) [2] project consists of specialized SELinux policy and system configuration packages aimed at meeting rigorous security requirements. For example, CLIP has been used to help SELinux systems meet *Director of Central Intelligence Directive 6/3 at Protection Level 4*, a common requirement for government and military systems which handle Sensitive Compartmented Information (SCI).

8 Performance and Scalability

The initial SELinux code release was not tuned for performance. Increased SELinux adoption saw the contribution of several enhancements aimed at improving performance, scalability, and resource utilization.

Examples include:

- *Utilizing Read Copy Update (RCU) [23] to remove locking and atomic operations from critical performance paths.* In January 2005, patches were merged upstream which dramatically increased the scalability of the core SELinux kernel code. Benchmarks run on 4-node 16-way NUMA system indicated a 50% increase in memory bandwidth and near-linear scheduler scalability.

- *Calculating information on non-critical paths as needed rather than ahead of time.* Patches were developed to perform more kernel access decision calculations on the fly and also reduce the size of related kernel structures. These changes, merged in September 2005, resulted in savings of around 8 MB of kernel memory for targeted policy and 16MB for strict policy. Under Linux, kernel memory is not pagable and thus a particularly precious resource.

- *Better utilization of the Linux slab allocator [13].* The kernel objects which hold SELinux policy rules were originally allocated via a generically-sized slab class. In August 2004, a patch was merged which implemented a custom slab class for policy rules, leading to memory savings of 37% on 64-bit systems.

- *Caching information rather than calculating it in performance critical paths.* Recent patches from HP have introduced label caches for virtual entities such as network addresses and ports, to avoid consulting the kernel policy database for labels at each access.

- *Optimized revalidation.* Under SELinux, read and write permissions for an open file are revalidated on each access. This is to ensure correct mediation even if there have been labeling or policy changes since the initial access. In September 2007, patches were merged which ensured that such revalidation would only occur if it was known that labels or policy had actually changed. Benchmarking indicates that overhead for read and write was reduced by a factor of five on Pentium-based systems and a factor of ten on the SuperH platform.

Many of the performance and resource utilization enhancements were contributed by developers from the embedded community.

9 Mitigation

A significant goal of SELinux is to mitigate threats arising from flaws in applications. Programming flaws are relatively common and effectively impossible to prevent entirely. Some flaws, especially those present in network-facing services and in privileged applications, may be exploited by malicious attacks. Such attacks can lead to disclosure of private information, corruption or destruction of valuable data, abuse of resources (e.g., hijacking a system to send spam), and the staging of more sophisticated attacks.

SELinux policy may be used to confine an application to ensure that it is capable of performing only the accesses needed for normal operation. This is an application of the principle of least privilege, which ensures that if an application is compromised or even malfunctions, that its actions will be limited to those it was supposed to be performing.

For example, if a web server was vulnerable to remote attack, an exploit may attempt to publish sensitive information from user directories or to send spam. With an appropriate SELinux policy, such actions would not be permitted, and these threats would be mitigated.

Several cases of successful threat mitigation with SELinux have been documented, where systems running with SELinux enabled were protected against real vulnerabilities and exploits [22].

These cases have demonstrated the value of deploying SELinux for general use.

10 Extending SELinux

The SELinux architecture has been extended beyond the Linux kernel to other areas of the system, and to other operating systems. Such efforts benefit from the ability to re-use existing SELinux components, such as the policy framework, code, and tools.

10.1 Desktop

Extending SELinux to the desktop environment is a significant undertaking. The modern desktop is made up of many layers and components, all of which need to be analyzed and understood in terms of information flow. At each layer, an SELinux policy model needs to be developed, and mediation hooks inserted into the code. This area has seen steady progress in parallel with the integration of SELinux into the base operating system.

The X Access Control Extension (XACE) [32] is a pluggable mandatory security framework for the X server, developed by the NSA, and merged into the upstream X.org tree. A Flask/TE module for XACE called XSELinux was also developed and merged upstream.

Work has also begun on securing the GNOME desktop environment by extending the SELinux architecture to GConf (the GNOME configuration system) [8] and D-BUS (the freedesktop.org messaging system). The D-BUS work has been merged upstream, while the GConf extensions are expected to remain in a prototype stage until further core desktop security infrastructure is in place.

A proof of concept project called *Imsep* [33] demonstrated a promising approach to protecting desktop applications by separating image processing functions into a separate security domain.

10.2 Database

SE-PostgreSQL [5], an extension of the SELinux architecture to the PostgreSQL relational database system, was released in September 2007. It features security labeling of data at the row and column levels, with enforcement of mandatory access control for authorized clients.

This importantly allows security policy to be uniformly applied to data at the OS level and within the database, where previously, fine-grained mandatory control was lost once information entered the database.

10.3 Virtualization

Efforts have been made by the NSA to integrate a flexible MAC scheme into the Xen hypervisor. Currently, the Xen Security Modules (XSM) project [10] implements a pluggable hook framework within the hypervisor, allowing different security models to be selected. An existing MAC scheme for Xen called Access Control Module (ACM) has been ported to XSM, and a Flask/TE module has been developed based on SELinux principles.

XSM removes security model logic from the core Xen code, and provides security model configuration flexibility. Hooks are implemented to allow mediation of privileged hypercalls, inter-domain communication (e.g., event channels and grant tables), and access to system resources by domains. An aim of this work is to increase robustness and assurance by decomposing the highly privileged Dom0 into separate domains. Interactions between these domains may then be controlled with fine-grained mandatory policy.

XSM and the Flask/TE module were merged into version 3.2 of the mainline Xen release.

10.4 Storage

While many local filesystems support SELinux via extended attributes, support for remote filesystems is rudimentary. When a remote filesystem is mounted, a policy-defined default security label may be assigned to all files on the mount; or in the case of NFS, the label may be specified by the administrator as a mount option. This provides coarse protection, although fine-grained labeling is not available; remote labels if they exist are ignored; and there is no way to remotely set labels on files. There is also no awareness of whether the remote system is performing any SELinux enforcement.

The Labeled NFS project [3] was started in 2007 to address these issues. Thus far, detailed requirements based on previous similar projects have been gathered, while proof of concept code has been published and reviewed by Linux NFS developers. A key challenge of the project is to accommodate the needs of multiple upstream groups including the IETF, Linux NFS developers and core kernel maintainers.

Labeled NFS was presented by the NSA at the IETF 71 meeting in March 2008. RFC documents are being developed with the aim of establishing Labeled NFS as an Internet standard, while work is continuing on the prototype Linux code.

10.5 Beyond Linux

Several non-Linux operating systems have adopted, or are in the process of adopting, Flask/TE technology.

FreeBSD led the way in this area with the SE-BSD project, which ported the SELinux architecture to its TrustedBSD framework. This work was then ported to Apple's Darwin operating system, as SE-Darwin. These projects are not currently incorporated into their respective mainline trees.

Recently, the OpenSolaris project announced Flexible Mandatory Access Control (FMAC) [4], an effort to incorporate the Flask/TE architecture into their mainline operating system. This project is also expected to leverage the Flask/TE port to Xen, and to implement Labeled NFS and Labeled IPSec.

The FMAC project presents an exciting opportunity for Linux and OpenSolaris to offer compatible mandatory security to users. A significant potential benefit is increased overall adoption of mandatory security.

11 Community

As discussed in Section 1.2, the SELinux project emerged from academic and government research efforts. As SELinux has been integrated into Linux distributions and made more generally usable, the SELinux community has expanded in both size and scope.

11.1 SELinux Symposium

The SELinux Symposium (2005–2007) has been of profound importance to the project, bringing together core developers, security researchers and significant early adopters.

A small, invite-only developer summit, arising from earlier informal meetings, was typically held after the main symposium, driving much of the direction of development for each year. The developer summit is now planned to be a separate, open event aimed at engaging the wider community.

11.2 Online Resources

The nsa.gov site remains the primary point of focus for the project, hosting many critical documents, the main mailing list, and historical core code releases.

Other SELinux sites have been deployed in recent years:

- `selinuxnews.org` – project news and events, and an SELinux community blog aggregator.
- `oss.tresys.com` – hosts several open source SELinux projects developed by Tresys.
- `selinuxproject.org` – a wiki hosting miscellaneous developer and project resources.

Distributions with SELinux support typically also utilize mailing lists and web sites to provide resources to developers and users. Links to these may be found at the SELinux for Distributions site: `selinux.sourceforge.net`.

Many seminal SELinux papers and presentations are archived at the SELinux Symposium web site: `selinux-symposium.org`.

SELinux kernel development is now hosted in public repositories on `kernel.org`.

11.3 Distributions

While Fedora-based distributions have been at the forefront of SELinux development in recent years, SELinux integration efforts have continued in other distributions.

Gentoo, Debian, and Ubuntu all have security hardening efforts involving SELinux integration into their mainline distribution releases.

SELinux was incorporated into the mainline release of Debian 4.0 ("Etch"). Previously, SELinux packages for Debian were only available from separate repositories.

As of version 8.04, Ubuntu includes full support for SELinux in its server release.

11.4 Adoption

SELinux was always intended to be adaptable to a wide variety of usage scenarios, and to provide useful protection in the general case. Initially, serious adoption appeared to be focused around traditional higher-assurance users such as government and military organizations.

With the integration of SELinux into general purpose, mainstream distributions, wider and more general adoption is now being seen.

There has been growing interest from industry sectors with critical security needs, such as finance and healthcare.

Consumer electronics is significant area of adoption which was not perhaps originally anticipated. As consumer devices become more sophisticated and connected, the nature and scope of their security requirements is markedly increased. A flexible MAC scheme such as SELinux often proves useful in this area, as security policy can be tailored to the specific function of each product, leading to tighter security than is typically possible for a general purpose system. There is potential to increase the time to patch if a vulnerability is discovered which is mitigated by SELinux policy, perhaps delaying the update until another critical update is required. Updating software in fielded devices can be expensive and risky.

In terms of general purpose adoption, statistics gathered by the Fedora project since the release of Fedora 8 tentatively indicate that a significant majority of users have SELinux enabled. Recent advances in usability and increased awareness of the value of mandatory security are likely factors here.

12 Current Developments

At the time of writing, an area of active development is in confining users. Under targeted policy, general users on an SELinux system are unconfined, with a focus on protecting against external threats. With the release of Fedora 8, a facility called *Kiosk Mode* [26] (or *xguest*) was introduced. This allows an anonymous user to access a desktop session in limited but useful manner, such as to only allow web browsing. The security goal in this case is to protect the system from the user. *Kiosk Mode* may be useful for providing public access to desktop and Internet applications in varied settings, such as libraries, cafes, conferences, product demonstrations, and training sessions.

Another current development is *permissive domains*, a mechanism to allow permissive mode to be invoked only for specific applications. SELinux enforcement may be disabled for a selected application, say, to debug its policy, while maintaining SELinux enforcement for the rest of the system.

13 Future Work

Areas of ongoing and future work in the SELinux project include:

- Continued extension of SELinux architecture to the desktop infrastructure and major applications. The Imsep work mentioned in Section 10.1 looks to be a promising model for general separation of security domains within applications.
- Working with the IETF to standardize Labeled NFS, and with the Linux community to have it accepted into the mainline kernel.
- Ongoing performance improvement, and efforts to further reduce the memory footprint of SELinux.
- Further simplification of policy, perhaps through the development of a higher-level policy language with idioms more familiar to Linux administrators.
- Support for more virtualization models, including Linux as hypervisor (e.g., KVM) and containers.
- Improved support for third party distribution of policy modules, such as the case of cross-building RPMs on systems with a conflicting host policy.

- Continued usability improvements for end users, administrators and developers.
- Better documentation.

14 Conclusion

Following the transition of SELinux from a research project into a deployable, community-driven technology, there has been rapid and intense growth in its function, usability, and adoption.

The SELinux project has pioneered the practical application of mandatory security in general purpose computing, utilizing a flexible, open approach to both the technology and the execution of the project itself.

While still a work in progress, SELinux has matured to the point where it is able to provide useful mandatory protection to general users, while also meeting higher assurance goals in critical security environments.

Other operating systems have begun adopting concepts innovated by the SELinux project, such as incorporating mandatory security into mainline products and making it a standard feature enabled by default.

It is hoped that SELinux will continue to both provide and foster stronger computer security for users of the continually evolving globally networked environment.

15 Acknowledgements

Thanks to Stephen Smalley, Dan Walsh, Karl MacMillan, and Christopher PeBenito for providing valuable feedback during the preparation of this paper.

References

[1] CDS Framework Toolkit. `http://oss.tresys.com/projects/cdsframework`.

[2] Certifiable Linux Integration Platform. `http://oss.tresys.com/projects/clip`.

[3] Labeled NFS. `http://www.selinuxproject.org/page/Labeled_NFS`.

[4] OpenSolaris Project: Flexible Mandatory Access Control. `http://www.opensolaris.org/os/project/fmac/`.

[5] Security Enhanced PostgreSQL. `http://code.google.com/p/sepgsql/`.

[6] SETools. `http://oss.tresys.com/projects/setools`.

[7] W. E. Boebert and R. Y. Kain. A Practical Alternative to Hierarchical Integrity Policies. *Proceedings of the 8th National Computer Security Conference*, 1985.

[8] J. Carter. Using GConf as an Example of How to Create an Userspace Object Manager. *Proceedings of the 3rd Annual Security Enhanced Linux Symposium*, March 2007.

[9] Ajaya Chitturi. Implementing Mandatory Network Security in a Policy-flexible System. Master's thesis. `http://www.cs.utah.edu/flux/papers/ajay-thesis-abs.html`.

[10] G. Coker. Xen Security Modules (slides). `http://xen.org/files/xensummit_4/xsm-summit-041707_Coker.pdf`, April 2007.

[11] U. Drepper. SELinux Memory Protection Tests. `http://people.redhat.com/drepper/selinux-mem.html`, April 2006.

[12] D. Ferraiolo and R. Kuhn. Role-Based Access Controls. *Proceedings of the 15th National Computer Security Conference*, October 1992.

[13] Brad Fitzgibbons. The Linux Slab Allocator, October 2000.

[14] IETF CIPSO Working Group. Commercial IP Security Option (CIPSO 2.2), July 1992.

[15] Trent Jaeger, Kevin Butler, David H. King, Serge Hallyn, Joy Latten, and Xiaolan Zhang. Leveraging IPsec for Mandatory Access Control of Across Systems. *Proceedings of the 2nd International Conference on Security and Privacy in Communication Networks*, August 2006.

[16] G. Kamis. US Coast Guard / NetTop2 - Thin Client Implementation (slides). `http://selinux-symposium.org/2007/slides/08-tcs.pdf`, March 2007.

[17] Donald E. Knuth. *Literate programming*. Center for the Study of Language and Information, Stanford, CA, USA, 1992.

[18] P. Loscocco and S. Smalley. Meeting Critical Security Objectives with Security-Enhanced Linux. *Proceedings of the 2001 Ottawa Linux Symposium*, July 2001.

[19] P. Loscocco, S. Smalley, P. Muckelbauer, R. Taylor, S. Turner, and J. Farrell. The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments. *Proceedings of the 21st National Information Systems Security Conference*, October 1998. `http://www.cs.utah.edu/flux/papers/ajay-thesis-abs.html`.

[20] Karl MacMillan. Core Policy Management Infrastructure for SELinux, March 2005.

[21] Karl MacMillan, Joshua Brindle, Frank Mayer, Dave Caplan, and Jason Tang. Design and Implementation of the SELinux Policy Management Server. *Proceedings of the 2nd Annual Security Enhanced Linux Symposium*, March 2006.

[22] D. Marti. A seatbelt for server software: SELinux blocks real-world exploits, February 2008. `http://www.linuxworld.com/news/2008/022408-selinux.html`.

[23] Paul E. McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. Read-Copy Update. In *Ottawa Linux Symposium*, July 2001.

[24] J. Morris. A Brief Introduction to Multi-Category Security. `http://james-morris.livejournal.com/5583.html`, September 2005.

[25] J. Morris. New Secmark-based Controls for SELinux. `http://james-morris.livejournal.com/11010.html`, May 2006.

[26] J. Morris. Using SELinux Kiosk Mode in Fedora 8. `http://james-morris.livejournal.com/25640.html`, February 2008.

[27] Y. Nakamura. Simplifying Policy Management with SELinux Policy Editor. March 2005.

[28] C. PeBenito, F. Mayer, and K. MacMillan. Reference Policy for Security Enhanced Linux. *Proceedings of the 2nd Annual Security Enhanced Linux Symposium*, February 2006.

[29] R. Smith. Introduction to Multilevel Security. `http://www.cs.stthomas.edu/faculty/resmith/r/mls/index.html`, 2005.

[30] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. *Proceedings of the Eighth USENIX Security Symposium*, August 1999.

[31] D. Walsh. Creating Loadable Modules with Audit2Allow. `http://fedoraproject.org/wiki/SELinux/LoadableModules/Audit2allow`, February 2006.

[32] E. Walsh. Application of the Flask Architecture to the X Window System Server. *Proceedings of the 3rd Annual Security Enhanced Linux Symposium*, March 2007.

[33] C. Walters. Towards a Least Privilege Desktop (presentation slides), March 2005.

[34] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. *USENIX Security Conference*, August 2002.

Coding Eye-Candy for Portable Linux Devices

Bob Murphy
ACCESS Systems Americas, Inc.
`Bob.Murphy@access-company.com`

Abstract

Linux is increasingly being used in portable devices with unusual hardware display architectures. For instance, recent OMAP and XScale CPUs support multiple frame-buffer "overlays" whose contents can be combined to form what the user sees on the screen.

As part of the ACCESS Linux Platform, ALP, ACCESS has developed a set of kernel modifications, an X extension and driver, and additions to the GDK/GTK stack to take advantage of the XScale CPU's three-overlay architecture. This paper provides an overview of these modifications and how they have been used to achieve a variety of "eye-candy" features, such as GTK widgets that float translucently over an actively playing video.

1 Introduction

The evolution of display systems for computer graphics has included a wide variety of features and architectures. Some systems, like the vector displays used by Ivan Sutherland in 1963, have largely wound up in the dustbin of history. For the last twenty years, framebuffer-based raster displays have dominated the industry.

However, standardization on raster displays does not mean progress has stood still. Early framebuffer systems used simple video RAM that corresponded to the screen, and programs could modify what was displayed by poking into memory-mapped addresses. Nowadays, nobody would consider a desktop system that didn't include a framebuffer card with enough RAM to permit page flipping, and a floating point graphics processing unit to accelerate 3D rendering using systems like OpenGL.

In embedded devices such as cell phones, framebuffers and related graphics hardware are often built into the CPU or related chips. Cell phone users often don't care much about 3D first-person shooters, but they do want to see photos and videos, and they want eye candy.

Cell phone manufacturers have gone to great lengths to support eye candy, such as adding GPUs and using CPUs with vector integer capabilities. In fact, the iPhone is reputed to use a CPU[1] that has a vector floating point coprocessor, which would allow a straightforward port of the Quartz graphics system used in Mac OS X.

As a response, some embedded CPU vendors have begun to support **video overlays**. These are multiple framebuffers whose contents can be programmatically combined to create the image the user sees. Video overlays can allow developers to provide a wide variety of eye candy.

2 Video Overlays: Hardware Examples

2.1 TI OMAP

Texas Instruments' OMAP 3430 rev. 2 is an ARM CPU with three video overlay framebuffers, as shown in Table 1.[2]

Overlay	Pixel Formats
Graphics	Palette RGB, direct RGB, and RGB with alpha channel
Video 1	Direct RGB and YCbCr
Video 2	Direct RGB and YCbCr

Table 1: OMAP 3430 Overlays

Many common compressed raster formats, such as MPEG, QuickTime, and JPEG, store images or frames

[1]A chip based on the ARM1176JZF design.
[2]The OMAP display architecture has other options and modes not covered in this discussion.

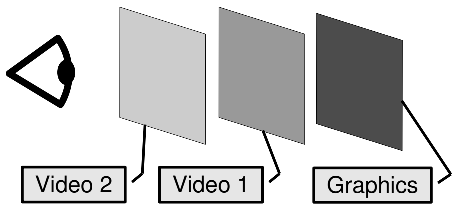


Figure 1: OMAP Normal Mode Overlays

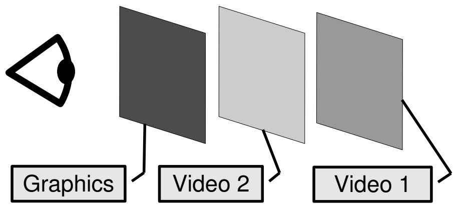


Figure 2: OMAP Alpha Mode Overlays

in a YCbCr color space. YCbCr encodes colors as a combination of *luminance* (brightness) and *chrominance* (hue) values. The human eye is more sensitive to luminance than chrominance, which makes YCbCr suitable for image compression. A codec can compress an image's chrominance and luminance separately, and apply a higher degree of compression to the chrominance values without much perceptual difference.

However, JPEG rendering and MPEG playback usually involve not only image decompression, but pixel color space conversion from YCbCr to RGB. The OMAP chip's direct hardware support for YCbCr formats in the Video 1 and Video 2 overlays allow high frame rates for image and video display with lower CPU effort.

The OMAP chip supports two overlay layering modes: *normal* and *alpha*.

In normal mode, the frame buffers are stacked as shown in Figure 1 . A simple use of this mode would be to treat the RGB-based graphics layer as a normal framebuffer, and map X11 into it. A developer could then display video or pictures "on top" of the X11 layer using one or both of the video overlays.

In alpha mode, the frame buffers are stacked as shown in Figure 2. This places the video layers "under" the graphics layer. Alpha mode is more interesting from an eye candy perspective because any alpha channel information in the graphics overlay is applied as color blending. When X11 is mapped onto the graphics layer, programs can not only "punch holes" in the X11 layer to display video from one of the underlying overlays, but they can have parts of the X11 layer "float translucently" on top of a playing video.

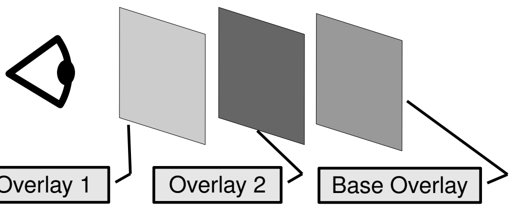


Figure 3: XScale Overlays

2.2 Marvell XScale

Beginning with the PXA270, the XScale ARM CPUs have offered a version of overlays that is generally similar to the OMAP alpha mode, but quite different in detail. These chips provide three overlay frame buffers, stacked as shown in Figure 3, and described in Table 2.[3]

Overlay	Pixel Formats
Overlay 1	Direct RGB, and RGB with T bit
Overlay 2	Direct RGB and YCbCr
Base Overlay	Palette RGB and direct RGB

Table 2: XScale PXA270 and PXA3XX Overlays

A simple use of this architecture would be to disable Overlays 1 and 2, set the base overlay to a direct RGB format, and map X11 to it. This would act like a very standard framebuffer. And as with OMAP, the XScale Overlay 2 is optimized for video and pictures due to its YCbCr pixel format options.

But the T bit in Overlay 1 is rather odd: it is an alpha channel of sorts, but not a normal one. A platform-wide flag in hardware activates or deactivates its effects. And

[3]The XScale display architecture also has other options and modes not covered in this discussion.

T Bit	Pixel RGB	Effect At That Pixel
0	Any	Overlay 1 is opaque: only the Overlay 1 color appears; no contribution comes from either of the other two overlays
1	Non-black	Overlay 1 is color-blended: the RGB color from Overlay 1 for that pixel is color-blended with the colors for that pixel from the topmost active underlying overlay, using a platform-wide alpha value that can be varied from 0.125 to 1.0 in 0.125 unit increments
1	Black	Overlay 1 is transparent: the pixel color from the topmost active underlying overlay is used

Table 3: T Bit Effect On A Pixel

if it is active, this one bit can have three effects on a per-pixel basis, as described in Table 3.

ALP maps X11 to Overlay 1, and displays video and pictures in Overlay 2. This permits GTK widgets that "float translucently" over pictures and playing video. ALP also can disable Overlay 2, and use the T bit in conjunction with the base overlay to create a variety of RGB-based eye candy effects.

3 Eye Candy Implementation in ALP

ACCESS engineers call alpha blending using the topmost hardware overlay **Overlay Transparency**, usually abbreviated **OT**, and distinguish three modes for each pixel, paralleling the XScale modes:

- **OT opaque** means the user sees only the topmost, X11 overlay's color.
- **OT transparent** means the topmost, X11 overlay contributes nothing. The user sees only the color from an underlying overlay, such as one playing video.
- **OT blended** means the user sees a color blended from the color in the X11 overlay and one of the underlying overlays.

Figure 4 summarizes the implementation of OT within the ALP platform. A typical OT-aware application will call into both GTK+ and the ALP OT API. The OT implementation, in turn, affects the behavior and/or implementation of other components in the graphics pipeline, including GTK+, GDK, X11, and the kernel video device drivers.

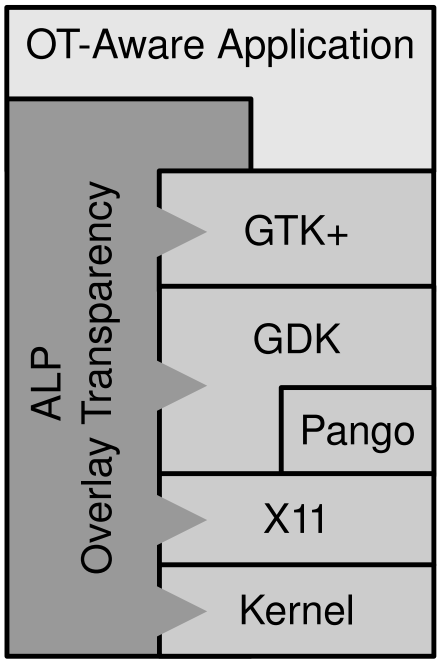


Figure 4: ALP Overlay Transparency Component Stack

3.1 Developer API

Developers can apply OT features using a GTK-based API built into the ALP platform. The API paradigm is that developers can set an 8-bit alpha value for various parts of widgets that is applied as OT, or specify that the alpha channel in a raster image be applied as OT. This makes the API portable while insulating developers from needing to know anything about the underlying hardware. Some hardware, such as OMAP, can support this paradigm directly. Other hardware, such as XScale, do not, but in such cases, the hardware-dependent implementation is written to "do the right thing" and come as close as possible to matching that paradigm.

The API provides several levels of calls:

- **Platform calls** let programs query the system to determine things like whether it supports OT at all.
- **Window calls** let programs enable or disable OT support for an entire GtkWindow and its child widgets. When OT support is disabled for a GtkWindow, it and all its child widgets are OT opaque, no matter what other OT settings have been applied to those widgets. There is also an option to indicate how the window manager should apply OT to the window frame.
- **Widget calls** let a program apply an 8-bit alpha value to an individual widget's background or text. They can also determine whether a widget containing a raster image with an alpha channel will have those alpha values applied to OT.
- **Container calls** parallel the widget calls, and let a program set default values for child widgets of a container. For instance, a program could apply one to a container to set a single OT background alpha value for all child widgets of a container. Programs can override these default values for individual widgets using the widget calls.

Figure 5 shows the widget portion of the API. Using an enum-based approach makes the API readily extensible, and provides backward compatibility since the system ignores enum values it does not recognize.

3.2 GTK/GDK and Platform Support

ACCESS has added OT support to ALP without any changes to GTK or GDK source code; instead, all support at the GDK level and above is written in platform-specific code. ALP applications must call a platform-specific initialization routine, which includes (among other things) a variety of OT-related run-time modifications to GTK and GDK behaviors.

When an application calls one of the API routines on a GTK widget, container, or window, the routine attaches OT-related state information to that GTK object using `g_object_set_data()`. Later, when a widget receives an expose event and draws itself, the OT code applies that state information, climbing the container hierarchy to determine default values if needed. Currently, ALP applies OT alpha values to portions of widgets as follows:

- **Background alpha** is applied to the widget's entire allocation.
- **Text alpha** is applied to the pixels actually rendered as text.
- **Foreground alpha** determines whether a raster image's alpha channel is applied for OT purposes.

3.3 Xtransp Extension

ALP includes a new X extension called **Xtransp**, which provides an interface to the OT features in the X server, as summarized in Table 4. Xtransp provides OT features on a per-Window basis, as well as calls that apply to the entire screen or system, and override any Window-based values to permit effects such as app transitions[4].

Inside the X server, Xtransp uses the devPrivates mechanism to add OT-related information to X Screens and Windows, including state information and an OT alpha array. It also overrides several Screen methods for window handling and drawing. Xtransp always keeps the screen consistent with OT changes to part of a Window or Screen, by adding that area to the Screen's damage region.

In particular, Xtransp head-patches the Screen's `PaintWindowBackground()` method, so that when a Window is painted, its OT alpha information is copied to the Screen.

3.4 X Server

3.4.1 Extensions

The OT code in the X server works in conjunction with several X extensions to provide features useful for cell phones and other portable devices:

[4]The routines that set blending values are useful with XScale hardware, which does not support different alpha blending values on different pixels, but not with OMAP, which does.

```
enum _AlpVidOvlWidgetFeatures
{
    // Widget alpha values; values range 0-255
    ALP_VIDOVL_FTR_WIDGET_BG_ALPHA = 0x05000000,   // Background
    ALP_VIDOVL_FTR_WIDGET_FG_ALPHA,                // Foreground
    ALP_VIDOVL_FTR_WIDGET_TEXT_ALPHA               // Text
};

/*
Get a feature value for a GtkWidget
[in] widget   A GtkWidget
[in] selector A feature code from _AlpVidOvlWidgetFeatures
[in] outValue The value to be retrieved
return        An error code, or ALP_STATUS_OK if no error

If the widget does not have an explicitly-set value, this will return
the corresponding default value from the widget's container stack.
*/
alp_status_t    alp_vidovl_widget_get_feature(GtkWidget *widget,
                    guint32 selector, guint32 *outValue);

/*
Set a feature value for a GtkWidget
[in] widget   A GtkWidget
[in] selector A feature code from _AlpVidOvlWidgetFeatures
[in] inValue  The value to be set for the feature
return        An error code, or ALP_STATUS_OK if no error
*/
alp_status_t    alp_vidovl_widget_set_feature(GtkWidget *widget,
                    guint32 selector, guint32 inValue);
```

Figure 5: Typical ALP Overlay Transparency API

- The **Shape** extension, which supports non-rectanglar Windows, also limits OT alpha value application to Windows. That permits features such as translucent dialogs and menus with rounded or bevelled corners.

- Using the **Shadow** extension simplifies the platform-dependent driver architecture.

- The OT-oriented hardware drivers support the **RandR** extension, so that portable device displays can be rotated. This is common in cell phones, where phone dialing is usually done in portrait mode, but camera features are used in landscape mode.

3.4.2 PXA3XX Video Driver

A new video driver for the PXA3XX series CPUs[5] provides the lowest level of support for OT in the X server. This is a kdrive driver that was developed from Keith Packard's fbdev driver. The changes to fbdev include things one would expect, such as setting up and maintaining hardware-specific states, and limiting pixel formats to those the hardware supports.

The heart of the driver changes are in the shadowbuffer-to-framebuffer blitter, which is where per-pixel OT alpha values in the 0-255 range are converted to the three states the XScale hardware supports. Its behavior is summarized in Table 5.[6]

[5]These are the PXA300, PXA310, and PXA320 XScale ARM CPUs from Marvell, code-named **Monahans**. The driver also sup-

Window Function	Effect
XOverlayTransparencySetWindowAlpha	Enable or disable OT support in a given X Window, and set its behavior toward child Windows.
XOverlayTransparencyUpdateWindowAlphaMap	Apply an array of 8-bit alpha values to a rectangle in a Window.
XOverlayTransparencySetBlending	Set a single set of RGB blending values to apply, platform-wide, to all Window-based operations.
Screen or System Function	**Effect**
XOverlayTransparencySetScreenAlpha	Apply a single alpha value to a rectangle on the screen, and disable any Window-based OT features.
XOverlayTransparencyDisableScreenAlpha	Disable the effect of XOverlayTransparency-SetScreenAlpha, and re-enable any Window-based OT features.
XOverlayTransparencySetScreenBlending	Set a single set of RGB blending values for screen-based operations.

Table 4: Xtransp API Summary

Alpha	Effect	32-bit TRGB Result
0	Transparent	0x01000000
1-254	Color-blended	0x01RRGGBB
255	Opaque	0x00RRGGBB

Table 5: Alpha Translation for XScale

3.4.3 Emulation in Xephyr

Embedded systems are notoriously difficult to develop for: programmers require working hardware, and then must use a cross-compiler, and flash or otherwise transfer executables to the device.

To speed development, the ALP Development Suite includes the *ALP Simulator*: an x86-native version of ALP that runs under User Mode Linux and mimics a device display via Xephyr. This lets in-house and third-party developers write and test code quickly on a Linux x86-based host computer. Then, when code works well on the host, they can cross-compile and transfer code for testing on an ARM device.

The ALP Simulator provides limited support for OT via changes to Xephyr, primarily replacing the `shadowUpdateRotatePacked()` blitter with one that is OT-aware. Since this is intended for initial development and testing, no attempt is made to simulate the XScale or OMAP video or base overlays. Instead, pixels that would be OT transparent are rendered as light aqua, and pixels that would be OT blended are rendered as a 50% average with light aqua. This lets developers quickly see whether their use of the GTK-based OT API is generally correct.

ports the PXA270 CPU, which shares the same overlay architecture.

[6]When color-blending, black is converted to 0x01000001 (very slightly blue) to avoid accidental transparency.

3.5 Kernel

The kernel sources require very few changes to support OT for XScale. They largely fall into three groups:

- A **new ioctl** lets the X server set the platform-wide color blending level that is used when a pixel has the T bit set and the pixel is not black.
- The video drivers include **pxafb_overlay.c**, an Intel-developed open-source driver for XScale overlays 1 and 2 (`/dev/fb1` and `/dev/fb2`).
- The open-source **pxafb video driver** for the base overlay (`/dev/fb0`) includes support for new pixel formats and other features supported by the XScale PXA270 and PXA3XX chips.

3.6 Video Playback

The current version of ALP includes a GStreamer-based media framework that plays video by activating XScale Overlay 2 as a YCbCr framebuffer, and blitting frames to it. Media playback occurs in a separate thread of execution, so while GTK-based programs can start and stop video playback, they otherwise proceed independently of the media framework.

4 Results

4.1 Transition Effects

One common form of eye candy is transition effects: items sliding on and off screen, zooming around, changing colors, and fading in and out.

The ALP platform achieves some of its transition effects by a combination of OT features in the X11 overlay and raster images in the base overlay. For instance, fade-in at app launch is done in several steps:

1. Fill the bottom (base) overlay with a solid color, typically a light gray.
2. Use the OT screen feature to make the entire X11 (top) overlay transparent. This makes the whole screen light gray.
3. When the app has finished displaying its user interface, gradually increase the opacity of the X11 overlay until it is fully opaque.

4.2 Floating Widgets

Another fun form of eye candy is translucency. ALP's OT implementation allows widgets to float transparently or translucently above playing video. Figures 6 through 11 show this in action. These are screen shots from an XScale development system, showing a sample application which plays a Hawaii travelogue video in the background on Overlay 2.

In Figure 6, the sample application is shown at entry. The main window and all its child widgets have been initialized with a variety of OT alpha values. However, OT support has not been activated on the main window, so it and its child widgets remain opaque. Although the

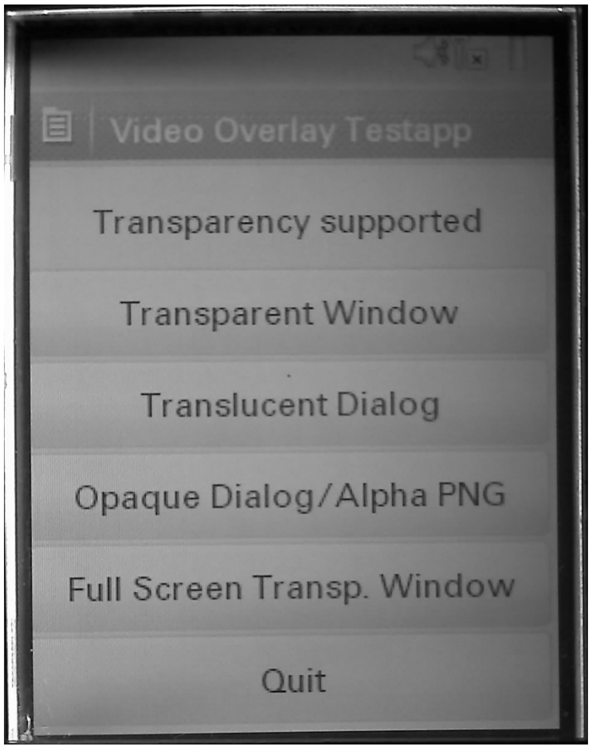


Figure 6: Application on entry

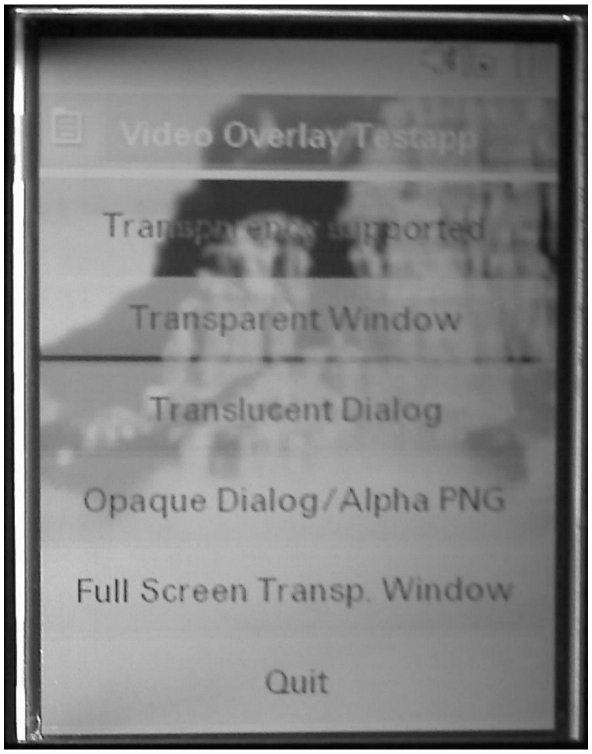


Figure 7: After pressing the *Transparent Window* button

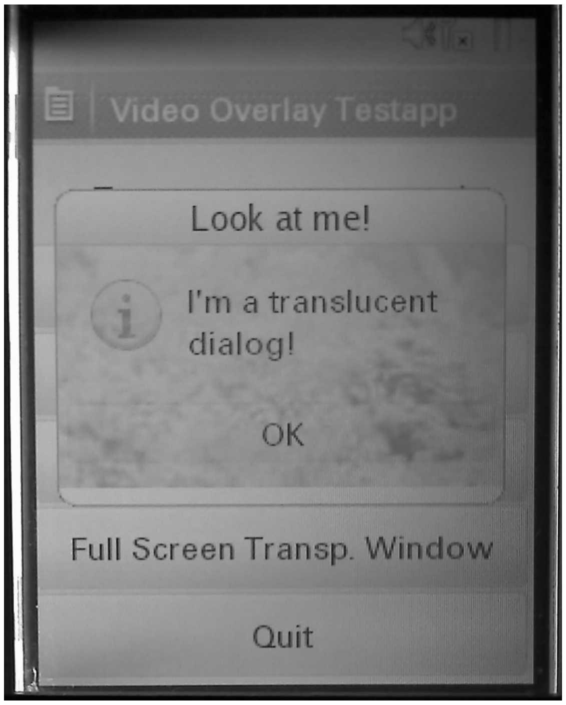


Figure 8: After pressing the *Translucent Dialog* button

video is playing in the background on Overlay 2, none of it is visible.

Pressing the *Transparent Window* button toggles OT support on the main window; Figure 7 shows the result. The playing video is visible through the window's transparent background and the translucent buttons; however, all the text is opaque. Above the main application window, the status bar remains opaque because it belongs to a different process.

Figure 8 shows what happens after toggling the main window's OT support off, and then pressing the *Translucent Dialog* button. The dialog contents, except for the opaque text, are translucent. However, the dialog frame has been set to opaque.

After dismissing the dialog and pressing the *Full Screen Transp. Window* button, Figure 9 shows a full-screen transparent window. In the center, there is a translucent button with opaque text.

Figure 10 shows the same *Transparent Window* result as Figure 7, with an opaque menu on top. The menu's rounded corners show the effect of the Shape extension on OT.[7]

[7]The screenshot shows a menu layout bug; the system was still under development when this paper was written.

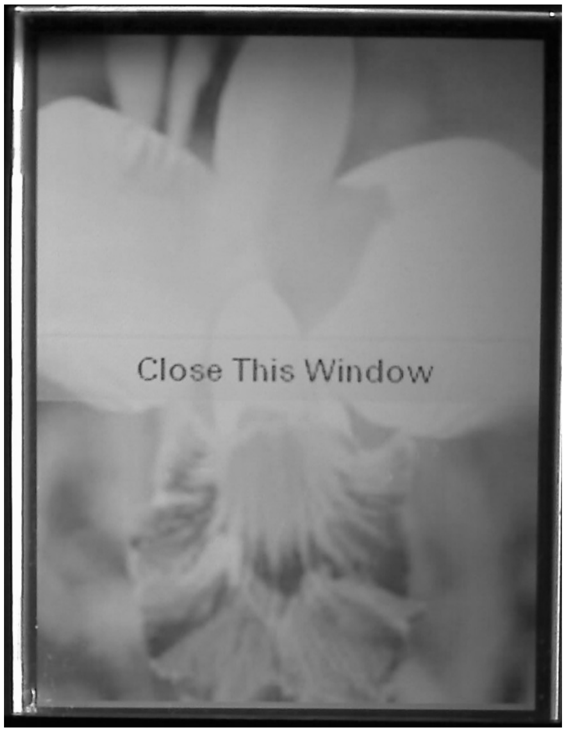


Figure 9: After pressing the *Full Screen Transp. Window* button

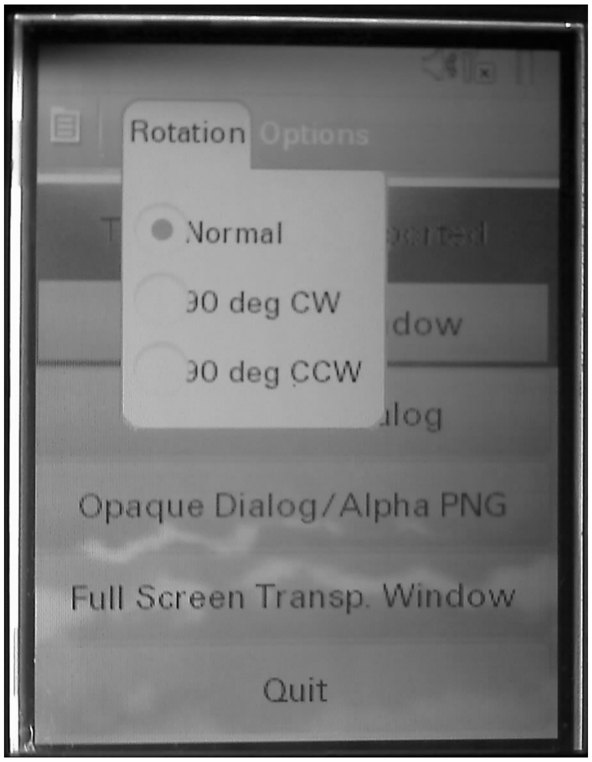


Figure 10: Opaque menu over transparent / translucent widgets

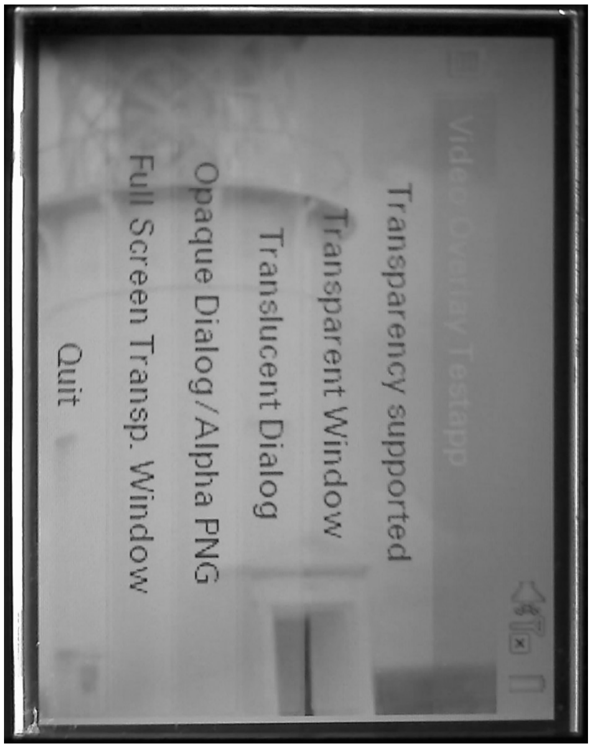


Figure 11: Landscape Mode

After selecting *90 deg CCW* from the menu in Figure 10, Figure 11 shows the result of rotating the entire X layer to landscape mode, while leaving the video playing in portrait mode.

5 References

Texas Instruments, *OMAP3430 Multimedia Device Silicon Revision 2.0 Technical Reference Manual*, 2007.

Marvell, *Monahans L Processor and Monahans LV Processor Developers Manual*, 2006.

SELinux for Consumer Electronics Devices

Yuichi Nakamura
Hitachi Software Engineering
ynakam@hitachisoft.jp

Yoshiki Sameshima
Hitachi Software Engineering
same@hitachisoft.jp

Abstract

As the number of network-connect Consumer Electronics (CE) devices has increased, the security of these devices has become important. SELinux is widely used for PC servers to prevent attacks from a network. However, there are problems in applying SELinux to CE devices. SELinux kernel, userland, and policy consume hardware resources unacceptably. This paper describes tuning SELinux for use in CE devices. The tuning has two features. The first is using our policy writing tool to reduce the policy size. It facilitates writing small policy by simplified policy syntax. The second is tuning the SELinux kernel and userland. We have tuned permission check, removed needless features for CE devices, and integrated userland to BusyBox. We have evaluated tuned SELinux on a SuperH (SH) processor based device, and found the consumption of hardware resources to be acceptable for CE devices.

1 Introduction

Linux is a leading OS for embedded systems [1], and has also been adopted in CE devices such as TVs, DVD recorders, set top boxes, mobile phones, and home gateways. Because CE devices are connected to the Internet, their security is now an important issue. Once vulnerabilities in CE devices are exploited by attackers, they can destroy the system, steal information, and attack others.

1.1 Requirements of security technologies for CE devices

To counter security problems, security technologies are necessary. However, security technologies for CE devices have to meet the following three requirements.

1. Effective without update
   The security technlogies have to be effective even without security updates, because the process of updating introduces several problems. Some CE devices do not have a network updater. To fix vulnerabilities for such devices, manufacturers have to recall devices, and re-write flash ROM. Even if devices do have a network updater, security patches tend to be delayed or not provided, because preparing updates is a heavy task for manufacturers. Updates for CE devices are provided from the manufacturers, not from OS distributors. The manufacturers have to track all vulnerabilities and develop security patches as soon as possible; as a result, manufacturers will likely give up providing updates.

2. Architecture-independent
   The security technologies have to be architecture-independent, since many CPU architecures—such as SuperH (SH), ARM, MIPS, PowerPC, and x86—are used in CE devices. Moreover, there are many variants within a CPU family. For example, SH has SH2, SH3, SH4, and SH64 variants. To port security technologies for such various CPUs, the security technologies need to be architecture-independent.

3. Small resource usage
   The security technologies have to work in resource-constrained environments. The architecture of a CPU is focused on power consumption rather than speed, thus the CPU clock is often slow such as 200Mhz. Main memory is often less than 64Mbyte, and the file storage area is often less than 32Mbyte to reduce hardware cost.

1.2 Porting security technologies to CE devices

Many security technlogies are already used in the PC environment. The most major ones are: buffer overrun protection, network updater, and anti-virus software.

However, they do not meet the previously stated requirements for CE devices. First, buffer overrun protections are architecture-dependent, because they depend on memory management of the CPU. For example, Exec Shield [3] is one of the most widely used buffer overrun protection technologies. It modifies codes under the `arch` directory in the kernel source tree. Second, network updaters such as yum [2] obviously do not meet the first requirement of being effective without update. Finally, anti-virus software does not meet the third requirement, because the pattern file consumes file storage area, sometimes more than 30Mbyte in a PC environment. In addition, the system becomes slow when a virus scan is running.

1.3 SELinux for CE devices

The Linux kernel includes Security-Enhanced Linux (SELinux) [4]. SELinux meets the first requirement, because it is effective even when a security update is not applied. SELinux provides label-based access control to Linux. Each process has a label called domain, and each resource has a label called type. Access rules between the domains and types are described in the security policy. Domains are configured to access only types that they need; thus, processes have only limited access rights. Assuming that a vulnerability exists in an application and it is not fixed, attackers can take control of the application. However, attack attempts usually fail due to lack of access rights; attackers obtain the domain of the application process that is allowed to access only limited types. SELinux has actually been widely used for PC servers and blocked attacks [5]. SELinux also meets the second requirement of being architecture-independent; there is no code modification under `arch` directory. However, SELinux does not meet the third requirement—small resource usage. SELinux was focused on PC usage, and many features were added. Consequently, its resource consumption has become unacceptable for CE devices.

The purpose of our work is to apply SELinux to CE devices. SELinux is tuned to meet resource requirements for that purpose. Our tuning has two features. The first is reducing policy size by using our policy writing tool. The tool facilitates writing small policy by simplifiying policy syntax. The second is tuning the SELinux kernel and userland. Permission checks are tuned, needless features for CE devices are removed, and userland commands are integrated to BusyBox [6]. Tuned SELinux is evaluted on a SH-based device. SH is a CPU family widely used for CE devices, including DVD recorders and home gateways. The evaluation results show the consumption of hardware resources is acceptable for CE devices.

2 Problems in applying SELinux to CE devices

To apply SELinux to CE devices, SELinux has to meet resource usage requirements. However, SELinux consumes CPU, file size, and memory unacceptably when used in CE devices. The detail is described in this section.

2.1 CPU usage

SELinux has overhead for system calls (syscalls) because of its security checks. In the PC environment, P. Loscocco et al. [4] measured the overhead and concluded that it is insignificant. However, the overhead is a problem when using SELinux on a CE device platform. The SELinux overhead measured on a CE device platform is shown in Table 1 and Table 2. We measured them by lmbench [7] and Unixbench [8]. Values in the tables are an average of 5 trials. The CE device platform is SH7751R (SH4 architecture, 240Mhz) processor, Linux 2.6.22. In particular, the read/write overhead is a problem, because they are executed frequently and the overhead is big. Overhead more than 100% is observed in null read/write; it is about 10% when measured in a Pentium 4 PC. Moreover, 16% overhead remains in reading a 4096-byte buffer. 4096 bytes are often used for I/O buffer because it is the page size in many CPUs for embedded systems, such as SH and ARM.

2.2 File size increase

The file size of the kernel and userland increases when SELinux is ported because of components listed in Table 3. The increase is about 2Mbyte if SELinux for PC (SELinux included in Fedora Core 6) is ported without tuning. However, the increase is not acceptable for CE devices, because flash ROM less than 32Mbyte is often used to store a file system. If SELinux consumes 2Mbyte, it is too much.

lmbench	Overhead(%)
Null read	130
Null write	147
Stat	97
Create	163
Unlink	86
Open/close	93
Pipe	67
UNIX socket	31
TCP	22
UDP	28

Table 1: System call overhead by SELinux on a CE device platform, measured by lmbench. To compute overhead, the time to execute syscall in SELinux disabled kernel is used as the baseline value.

Unixbench read/write	Overhead(%)
256 byte read	66.6
256 byte write	66.8
1024 byte read	40.5
1024 byte write	43.9
4096 byte read	16.2
4096 byte write	-3.1

Table 2: The SELinux overhead for read/write on a CE device platform, measured by Unixbench. To compute overhead, the throughput in SELinux disabled kernel is used as the baseline value.

2.3 Memory consumption

SELinux has data structures in the kernel to load the security policy. The memory consumption by the security policy used in PC is about 5Mbyte. However, it is also unacceptable for CE devices because the size of RAM is often less than 64Mbyte and swap is not prepared. If SELinux is used for CE devices, the possibility that memory can not be allocated increases. If memory can not be allocated, applications will not work correctly.

3 Tuning SELinux for CE devices

Tuning is needed because the resource consumption by SELinux is not acceptable for CE devices, as described above. Our tuning consists of two parts. The first is reducing policy size by utilizing our policy writing tool. The second part is tuning the kernel and userland.

Component	Additional features
Kernel	The SELinux access control feature, audit, and xattr support in filesystem.
Library	libselinux, libsepol, and libsemanage
Command	Commands to manage SELinux such as load_policy. Additional options for existing commands, such as -Z option for ls to view file label.
Policy file	The security policy

Table 3: Files related to SELinux

3.1 Reducing policy size by policy writing tool

Since the security policy consumes both file storage area and RAM, the security policy has to be small. Problems in preparing a small policy and our approach to resolving them are described.

3.1.1 Problems in preparing small policy

To prepare policy, refpolicy [9] is usually used and customized for the target system [10]. Refpolicy is a policy developed by the SELinux community, and is used in distributions such as Red Hat and Fedora by default. It is composed of sample configurations for many applications and a set of macros. Refpolicy works well on PC systems, but it is hard to use for CE devices. To prepare small policy based on refpolicy, one has to remove unnecessary configurations, then add necessary ones. However, there are three difficulties in the process.

1. Large amount of removal
   The amount of removal is large, because configurations for many distributions, applications, and use cases are included. For example, configurations for many Apache modules are included in refpolicy. To configure Apache that serves simple home page and CGI, configurations about unnecessary Apache modules have to be removed. We removed about 400 lines for that. For each application, such removal has to be done.

2. Many macros and labels
   Refpolicy contains many macros and labels. More

than 2,000 macros are defined and used. More than 1,000 labels are declared. Policy developers have to understand them in removing and adding configurations. Figure 1 is an example of configurations in refpolicy. It is hard for policy developers to understand so many macros and labels.

```
policy_module(apache,1.3.16)
type httpd_t;
type httpd_exec_t;
init_daemon_domain(httpd_t,httpd_exec_t)
role system_r types httpd_t;
....
ifdef(`targeted_policy',`
typealias httpd_sys_content_t
        alias httpd_user_content_t;
typealias httpd_sys_script_exec_t
        alias httpd_user_script_exec_t;
')
allow httpd_t httpd_sys_content_t:
dir r_dir_perms;
...
corenet_tcp_sendrecv_all_if(httpd_t)
...
```

Figure 1: Example of configurations used in PC. This is part of configuration of http server.

3. Dependency
   Two kinds of dependencies that appear in removing and adding configurations increase the cost of preparing policy.

   (a) Labels and declarations
   There are dependencies in labels (domain/type) and declaration. They make removing configurations difficult. The major part of SELinux configuration is allowing domain to access some type, like below.
   `allow httpd_t sendmail_exec_t:`
   `file execute;`
   This is part of configuration to allow web server to send mails. In SELinux policy syntax, all labels must be declared like below.
   `type httpd_t;`
   `type sendmail_exec_t;`
   Such text based policy configuratin is converted to binary representation to be loaded in the kernel. If the declaration is removed, error is outputted and conversion fails. This often happens in removing files. Refpolicy is composed of many files. For example, in file mta.te configurations related to sendmail is described and sendmail_exec_t is declared. If mta.te is removed, policy conversion fails in `allow httpd_t sendmail_exec_t: file execute;`. This line has to be removed to convert policy successfully.

   (b) Labeling change
   Dependencies also appear when labeling is changed. Assume an application `foo` running as `foo_t` domain is allowed to access `foo_file_t` type and under `/foo` directory are labeled as `foo_file_t`. Then `foo` can access under `/foo`. What happens an application `bar` running as `bar_t` domain needs configuration to access `/foo/bar`? One will define new type such as `bar_file_t` and labels `/foo/bar` as `bar_file_t` type, then allow `bar_t` to access `bar_file_t`. Problem is happenning here. `foo` can not access `/foo/bar`, because `foo_t` is not allowed to access `bar_t`. To resolve that, configuration that allows `foo_t` to access `bar_t` has to be described. In adding new configuration, such dependency have to be considered carefully.

3.1.2 Preparing policy by SELinux Policy Editor

Policy is prepared without using refpolicy to avoid above difficulties. SELinux Policy Editor (SEEdit) [11] is used for that. SEEdit was developed by the authors to facilitate policy writing. The main feature is Simplified Policy Description Language (SPDL). Fig 2 is an example of configuration written by SPDL. Type labels are hidden; in other words, file names and port numbers can be used to specify resources. SPDL is converted to usual SELinux policy expression by converter, and policy is applied.

```
domain httpd_t;
program /usr/sbin/httpd;
allow /var/www/** r;
allownet -protocol tcp -port 80 server;
```

Figure 2: Example of SPDL, part of configuration for http server

How difficulties described in Section 3.1.1 are resolved by SEEdit is shown below.

1. Large amount of removal
   Only configurations that are necessary for CE devices are described by SEEdit. Obviously, there is no need to remove unnecessary configurations. One has to create necessary configurations, but the number of lines to be described is small. For example, we wrote about 20 lines for web server that serves a simple homepage.

2. Macros and labels
   Labels are hidden, and macros are not used in SPDL. Syntax of SPDL appears instead of macros, but it is much simpler than macros.

3. Dependencies
   In SPDL, such dependencies are not included. Dependencies are resolved internally when SPDL is converted to the original SELinux configuration syntax.

3.2 Tuning the kernel and userland

SELinux was developed for PC usage, so there are unneeded features, functions, and data structures for CE devices. The SELinux overhead for syscalls, file size, and memory usage can be reduced by removing them. The removal strategy and implementation are described in this section.

3.2.1 Reducing overhead

The SELinux overhead is reduced by removing unneeded functions and redundant permission checks from the kernel.

1. Removal of function calls from SELinux access decision code
   Function calls can be removed from SELinux access decision function (`avc_has_perm`) by using inline functions and calling `avc_audit` only when it is necessary. `avc_has_perm` is called in all permission checks, thus the removal will reduce overhead.

2. Removal of duplicated permission checks in file open and read/write
   There are duplicated permission checks in the process of file open and read/write to the file descriptor. For example, when a process opens a file to read or write, read/write permission is checked. Read/write permission is checked again in every read/write system call to the file descriptor. The check at read/write time is duplicated, because read/write permission is already checked at open time. Therefore, the permission check at read/write time can be removed. There is one exception: When security policy is changed between file open and read/write time, permission has to be checked at read/write time to reflect the change.

3. Removal of permission checks related to network
   In the process of network communication, SELinux permissions are checked for NIC, IP address, and port number. Permission checks in NIC and IP address are removed, because they are rarely used. Note that if there is a domain that wants to communicate with only a specific IP address, they can not be removed.

3.2.2 Reducing file size

SELinux userland was intended for server usage, so many features are unnecessary for CE devices. File size can be reduced by choosing features that meet the following criteria.

- Access control feature of SELinux works.

- Security policy can be replaced.
  Most of the troubles related to SELinux are caused by a lack of policy configurations. To fix these issues, we need a feature to replace policy.

Features in SELinux userland can be classified like Table 4. Features 1 , 2, and 3 are chosen according to criteria above. Feature 1 is necessary to use access control; Features 2 and 3 are needed to replace policy.

Features 1 through 3 are chosen and implemented. In the implementation, commands are integrated to BusyBox, and libselinux is modified. By using BusyBox, the size can be reduced more. Libselinux was also tuned.

#	Feature	Description	Related packages
1	Load policy	Load security policy file to the kernel	libselinux
2	Change labels	View and change domains and types	libselinux policycoreutils coreutils procps
3	Switch mode	Switch permissive/enforcing mode	libselinux
4	User space AVC	Use the access control feature of SELinux from userland applications	libselinux
5	Analyze policy	Access data structure of policy	libsepol
6	Manage conditional policy	Change parameters of conditional policy feature	libselinux libsepol libsemanage
7	Manage policy module framework	Install and remove policy modules	libsemanage

Table 4: Features included in SELinux userland. We use 1,2, and 3 for CE devices.

Unnecessary features were removed and the dependency on libsepol, whose size is about 300Kbytes, was removed.

3.2.3 Reducing memory usage

To reduce memory usage, unnecessary data structures are removed from the kernel. The biggest data structure in SELinux is the hash table in `struct avtab`. Two avtabs are used in the kernel. Access control rules in the security policy are stored in hash tables of `struct avtab`. 32,768 hash slots are prepared for each hash table; they consume about 260Kbyte. Hash slots were shrunk to reduce the size of avtabs. In addition, hash slots are allocated dynamically based on number of access control rules in policy, i.e. the number of allocated hash slots is 1/4 of the number of rules. That change creates a concern about performance, because the hash chain length will increase. However, although the chain length becomes longer, regressions in performance were not observed.

4 Evaluation

To evaluate our tuning, we ported SELinux to an evaluation board and measured performance both before and after tuning.

4.1 Target device and software

The specification of the device and version of the software used in the evaluation are shown.

1. Target device
   The Renesas Technology R0P751RLC001RL (R2DPLUS) board was used as our target device. This board is often used to evaluate software for CE devices. The specification is shown below.

   - CPU: SH7751R(SH4) 240Mhz
   - RAM: 64Mbyte
   - Compact flash: 512Mbyte
   - Flash ROM: 64Mbyte (32Mbyte available for root file system)

   SELinux can be ported to both compact flash and flash ROM. We measured the benchmark on a compact flash system for convenience.

2. Software and policy
   The version of the software and policy used in the evaluation are listed below.

   - Kernel: Linux 2.6.22
   - SELinux userland: Obtained from SELinux svn tree (selinux.svn.sourceforge.net) as of Aug 1, 2007

- Security policy (before tuning): policy.21 and file_contexts file were taken from selinux-policy-targeted-2.4.6-80.fc6 (included in Fedora 6).
- Security policy (after tuning): Written by SELinux Policy Editor, including configurations for 10 applications. Not all applications are confined, similar to targeted policy [12].

4.2 Benchmark results

We ported SELinux to the target board and measured the benchmark before and after tuning. The benchmark results for syscall overhead, file size, and memory usage are shown.

4.2.1 Syscall overhead

Syscall overhead was measured by lmbench and unixbench. The result is shown in Table 5 and Table 6. The SELinux overhead for read/write was significant before tuning. Null read/write overhead is reduced to 1/10 of the previous overhead. The overhead in reading a 4096 buffer was especially problematic, but it is almost eliminated by our tuning.

lmbench	Overhead before tuning(%)	Overhead after tuning(%)
Null read	130	13
Null write	147	15
Stat	97	59
Create	163	146
Unlink	86	70
Open/close	93	62
Pipe	67	31
UNIX socket	31	6
TCP	22	11
UDP	28	12

Table 5: The SELinux overhead for system call on the evaluation board, measured by lmbench. Average of 5 trials.

4.2.2 File size

The file size related to SELinux is summarized in Table 7. As a result of tuning, the file size increase is re-

Unixbench read/write	Overhead before tuning(%)	Overhead after tuning(%)
256 byte read	66.6	16.2
256 byte write	66.8	26.8
1024 byte read	40.5	13.1
1024 byte write	43.9	19.0
4096 byte read	16.2	3.3
4096 byte write	-3.1	0

Table 6: The SELinux overhead for read/write on the evaluation board, measured by Unixbench. Average of 5 trials.

duced to 211Kbyte. In the evaluation board, flash ROM available for root file system is 32Mbyte. The size of SELinux is less than 1%, so it is acceptable for the evaluation board.

Component	File size before tuning(Kbyte)	File size after tuning(Kbyte)
Kernel(zimage) size increase	74	74
Library	482	66
Command	375	11
Policy file	1,356	60
Total	2,287	211

Table 7: File size related to SELinux. Userlands are built with -Os flag and stripped.

4.2.3 Memory usage

We measured memory usage by using the `free` command. The usage by SELinux was measured as follows.

A = The result of `free` when SELinux enabled kernel booted.
B = The result of `free` command when SELinux disabled kernel booted.
Memory usage by SELinux = A - B

The memory usage by SELinux was measured for both before tuning and after tuning. We also measured memory usage of a hash table in `struct avtab` to see the effect of tuning. We inserted code that shows the size of allocated tables for that purpose. The result is shown in Table 8. The memory consumption after tuning is 465Kbyte. In the evaluation board, memory size is 64Mbyte. The consumption by SELinux is less than 1%—small enough.

Component	Memory usage before tuning (Kbyte)	Memory usage after tuning (Kbyte)
Hash tables in struct avtab	252	1
SELinux program and policy	5,113	464
Total	5,365	465

Table 8: Memory usage by SELinux

5 Related works

R. Coker [13] ported SELinux to an ARM-based device, but SELinux was Linux 2.4 based. Since then, SELinux has changed a lot. Implementation has changed, many features were added, and policy development process has changed. KaiGai [14] ported xattr support to jffs2 and was merged to Linux 2.6.18. We are using his work when we run SELinux on a flash ROM system. The seBusyBox project in the Japan SELinux Users Group worked to port SELinux commands and options to BusyBox. We joined this project and did the porting together. Applets ported in this project were merged to BusyBox. In this project H. Shinji [15] also worked to assign different domains to applets, and his work was merged to BusyBox 1.8.0. H. Nahari [16] presented a design of a secure embedded system. He also mentioned SELinux for embedded devices, but the detail was not described.

6 Conclusion

There are problems in applying SELinux to CE devices. SELinux kernel, userland, and the security policy consume hardware resources unacceptably. We tuned SELinux to meet the resource requirements of CE devices. The tuning has two features. The first is using our policy writing tool to reduce policy size. It facilitates writing small policy by simplifying policy syntax. The second is tuning SELinux kernel and userland. Permission checking was tuned, needless features for CE devices were removed, and userland was integrated to BusyBox. Tuned SELinux was evaluated on a SH-based CE device evaluation board. The benchmark result shows that the SELinux overhead for read/write is almost negligible. File size is about 200 Kbyte, and memory usage is about 500Kbyte, about 1% of the flash ROM and RAM of the evaluation board. We conclude that SELinux can be applied to CE devices easier as the result of our work.

7 Future works

There are remaining issues to be done in the future.

1. Xattr for file systems on flash ROMs
   There are several files systems for flash ROMs, including jffs2, yaffs2, and logfs. Jffs2 supports xattr; yaffs2 and logfs do not support xattr. The porting of xattr is needed to use SELinux on such file systems.

2. Strict policy
   We adopted targeted policy in this paper. We have to write strict policy for more security. That is to say, domains for every program are prepared. The number of access control rules and policy size will be large. To write small strict policy is a remaining problem.

8 Availability

We have submitted related patches to Linux, BusyBox and SELinux community. The merged works are shown in Table 9. The links to patches can be seen on the Embedded Linux Wiki, `http://elinux.org/SELinux`. The SELinux Policy Editor is available on the SELinux Policy Editor Website [11]. 2.2.0 supports the writing of policy for CE devices.

Work	Merged version
Reducing read/write overhead	Linux 2.6.24
Reducing size of avtab	Linux 2.6.24
Reducing size of libselinux	libselinux 2.0.35
Integrating SELinux commands	BusyBox 1.9.0

Table 9: Availability of our work

9 Acknowledgements

We are helped by many people during the work. We would like to thank them. seBusyBox project community members and KaiGai Kohei (NEC) gave us useful comments. People in the SELinux community gave us feedback, Stephen Smalley gave us ideas for the implementation of tuning. People in the BusyBox community, especially Denys Vlasenko, reviewed patches and gave us feedback. Yusuke Goda (Renesas Solutions) provided a driver for flash ROM.

References

[1] Linux Devices.com: Linux to remain a leading embedded OS, says analyst (2007), `http://www.linuxdevices.com/news/NS2335393489.html`

[2] Yum: Yellow dog Update, Modified: `http://linux.duke.edu/projects/yum/`

[3] A. van de Ven: Limiting buffer overflows with ExecShield, Red Hat Magazine, July 2005 (2005), `http://www.redhat.com/magazine/009jul05/features/execshield/`

[4] P. Loscocco and S. Smalley: Integrating Flexible Support for Security Policies into the Linux Operating System: the Proceedings of the FREENIX Track of the 2001 USENIX Annual Technical Conference (2001)

[5] D. Marti: A seatbelt for server software: SELinux blocks real-world exploits, Linuxworld.com (2008), `http://www.linuxworld.com/news/2008/022408-selinux.html`

[6] BusyBox, `http://www.busybox.net/`

[7] L. McVoy and C. Staelin: lmbench: Portable tools for performance analysis: the Proceedings of USENIX 1996 Annual Technical Conference (1996)

[8] UNIX Bench, `http://www.tux.org/pub/tux/benchmarks/System/unixbench/`

[9] Refpolicy, `http://oss.tresys.com/projects/refpolicy`

[10] F. Mayer, K. MacMillan and D. Caplan: SELinux by Example, Prentice Hall (2006)

[11] SELinux Policy Editor, `http://seedit.sourceforge.net/`

[12] F. Coker and R. Coker: Taking advantage of SELinux in Red Hat® Enterprise Linux®, Red Hat Magazine, April 2005 (2005), `http://www.redhat.com/magazine/006apr05/features/selinux/`

[13] R. Coker: Porting NSA Security Enhanced Linux to Hand-held devices, Proceedings of the Linux Symposium, Ottawa, Ontario, Canada (2003)

[14] KaiGai: Migration of XATTR on JFFS2 and SELinux, CELF Jamboree 11(2006), `http://tree.celinuxforum.org/CelfPubWiki/JapanTechnicalJamboree11`

[15] H. Shinji: Domain assignment support for SELinux/AppArmor/LIDS, BusyBox mailing list, `http://www.busybox.net/lists/busybox/2007-August/028481.html`

[16] H. Narari: Trusted Secure Embedded Linux: From Hardware Root of Trust to Mandatory Access Control, Proceedings of the Linux Symposium, Ottawa, Ontario, Canada (2007)

PATting Linux

Venkatesh Pallipadi
Suresh Siddha
Intel Open Source Technology Center
`{venkatesh.pallipadi|suresh.b.siddha}@intel.com`

Abstract

The Page Attribute Table *(PAT)*, introduced in Pentium III, provides the x86 architecture with an option to assign memory types to physical memory, based on page table mappings. The Linux kernel, however, does not fully support this feature, though there were many earlier efforts to add support for this feature over the years.

The need for PAT in Linux is becoming critical with the latest platforms supporting huge memory; these cannot be supported by limited number of MTRRs, often leading to poor graphics and IO performance.

In this paper, we will provide insight into earlier attempts to enable PAT in the Linux kernel and details of our latest attempt, highlighting various issues that were encountered. We will describe the new APIs for userspace/drivers for specifying various memory attributes.

1 Introduction

Page Attribute Table *(PAT)* has been a hardware feature in x86 processors starting from the Pentium III generation of processors. The PAT extends the IA-32 architecture's page-table format to allow memory types to be assigned to regions of physical memory based on linear address mappings [1].

PAT is a complementary feature to Memory Type Range Registers *(MTRR)*. The MTRRs are used to map regions in physical address space with specific memory types.

PAT and MTRR together allow the processor to optimize operations for different types of memory such as RAM, ROM, frame-buffer memory, and memory-mapped I/O devices. The MTRRs are useful for statically describing memory types for physical ranges, and are typically set up by system firmware. The PAT allows dynamic assignment of memory types to pages in linear address space.

The Linux kernel (as in linux-2.6.25 [2]) does not fully support PAT. It only uses PAT for write-back and uncached mappings, and uses MTRR to dynamically assign write-combining memory type. This results in over-dependence on MTRR, causing issues like performance problems with the display driver (X), for example.

The Linux kernel also does not enforce the *no-aliasing* requirement while mapping memory-types, potentially resulting in various linear addresses from the same or different processes, mapping to the same physical address with different effective memory types. This aliasing can potentially cause inconsistent or undefined operations that can result in system failure [1].

In this paper, we present details about our recent effort towards adding PAT support for the x86 architecture in the Linux kernel [13]. We start with an architecture primer on PAT and MTRR in Section 2, followed by a description of the Linux kernel status (as in linux-2.6.25) in Section 3. Section 4 provides details about our proposed PAT implementation. APIs for drivers and applications to set memory types is discussed in Section 5. We conclude the paper with a look at the future in Section 6.

2 Architectural Background

In this section, we provide an architectural overview of PAT, MTRR, and interactions between PAT and MTRR. Refer to the chapter on `Memory Cache Control` in [1] for a complete reference.

2.1 PAT

The PAT extends the IA-32 architecture's page-table format to allow memory type to be assigned to pages based

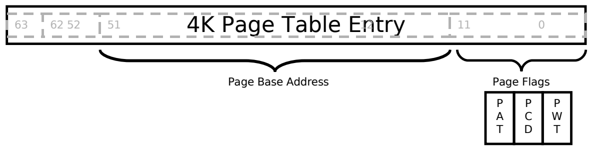


Figure 1: PAT flags in a 4K Page Table Entry

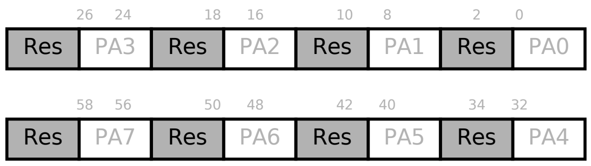

26 24 18 16 10 8 2 0
Res PA3 Res PA2 Res PA1 Res PA0
58 56 50 48 42 40 34 32
Res PA7 Res PA6 Res PA5 Res PA4

Figure 2: IA32_CR_PAT MSR

on linear address mappings. Figure 1 shows a regular 4K page table entry, with bits 12–51 forming the physical frame number. Bits 0–11 have various page flags associated with this mapping. Page flags PAT, PCD, and PWT together represent the PAT attribute of the page. These three bits index into 8-page attribute types in the `IA32_CR_PAT` MSR. This MSR's content is depicted in Figure 2, with each Page Attribute (PA0 through PA7) mapping to a particular memory type encoding.

The memory types that can be encoded by PAT are listed in Figure 3. PA0 through PA7 in the `IA32_CR_PAT` MSR can contain any encoding from Figure 3.

The `IA32_CR_PAT` MSR is set with a predefined default setting for each PAT entry (PA0 through PA7) upon power up or reset. System software can write different encodings to these entries with the `WRMSR` instruction. On a multi-processor system, the `IA32_CR_PAT` MSR of all processors must contain the same value.

The PAT allows any memory type to be specified in the page tables, and therefore it is possible to have a single physical region mapped to two or more different linear addresses, each with different memory types. These mappings may be the part of the same address space, or may be in the address spaces of different processes. Such a mapping of a single physical region with multiple linear address ranges with different memory types is referred to as *aliasing*. Architecturally, any such aliasing can lead to undefined operations that can result in a system failure. It is the operating system's responsibility to prevent such aliasing when PAT is being used.

Encoding	Memory Type
0	Uncacheable (UC)
1	Write Combining (WC)
2	Reserved
3	Reserved
4	Write Through (WT)
5	Write Protected (WP)
6	Write Back (WB)
7	Uncached (UC-)

Figure 3: Memory types that can be encoded with PAT

2.2 MTRR

The MTRR provides a mechanism for associating the memory types with physical address ranges in system memory. MTRR capability can be determined by the `IA32_MTRRCAP` MSR. The encodings supported by MTRR are listed in Figure 4. The MTRRs are defined as a combination of:

- **Fixed Range MTRRs:** A Fixed Range MTRR consists of predetermined regions of size 64K, 16K, and 4k in the 0–1MB physical memory range. This includes eight 64K ranges, sixteen 16K ranges, and sixty-four 4K ranges. Each such range can be defined as a particular memory type encoding.
- **Variable Range MTRRs:** Most x86 processors support up to eight Variable Range MTRRs. They are specified using a pair of MSRs: `IA32_MTRR_PHYSBASEn` defines the base address; and `IA32_MTRR_PHYSMASKn` contains a mask used to determine the range.
- **Default MTRR Type:** Any physical memory range not covered by a fixed or variable MTRR range takes the memory type attributes from the `IA32_MTRR_DEF_TYPE` MSR.

When MTRRs are enabled, MTRR range overlaps are not defined, except for:

- Any range overlap, with one of the ranges being of uncached type, will result in the effective memory type of uncached.
- Any range overlap, with one range being write-back and another range being write-through, will result in the effective memory type of write-through.

Encoding	Memory Type
0	Uncacheable (UC)
1	Write Combining (WC)
2	Reserved
3	Reserved
4	Write Through (WT)
5	Write Protected (WP)
6	Write Back (WB)
7–0xFF	Reserved

Figure 4: Memory types that can be encoded with MTRR

While MTRRs are enabled and being used, the operating system has to make sure that there are no overlapping MTRR regions with overlapping types not defined above.

2.3 MTRR and PAT overlap

PAT and MTRR may define different memory types for the same physical address. The effective memory type resulting from this overlap is architecturally defined as in the chapter on `Memory Cache Control` in reference [1]. Specifically,

- The PAT memory type of write-combine takes precedence over any memory type assigned to that range by MTRR.
- The PAT memory type of uncached-minus gives precedence to any MTRR write-combine setting for the same physical address. If there are no MTRR memory types or if the memory type in MTRR is write-back, the effective memory type for that region will be uncached.

3 Linux Kernel Background

All references to the Linux kernel in this section refers to version 2.6.25. The Linux kernel supports PAT in a very restrictive sense, with PAT memory types uncached and uncached-minus being used by kernel APIs like `ioremap_noncache()`, `set_memory_uc()`, `pgprot_noncached()`, etc. User-level APIs that use the PAT uncached memory type are the `/proc`, and `/sys` PCI resource interfaces and `mmap` of `/dev/mem`.

Linux also supports adding new MTRR ranges using the kernel API `mtrr_add()`. There is also a user-level API to set MTRR ranges by using `/proc/mtrr` writes.

The kernel does not do any aliasing checks while setting the PAT mappings. The kernel only makes sure that kernel identity mapping of physical memory is consistent while changing the PAT memory type with some of the APIs above. The kernel does check for any overlap with existing MTRR ranges, while adding a new MTRR.

Following is an example of MTRR usage on a typical server. Figure 5 shows the contents of `/proc/mtrr`. The effective memory type on this system will be as in Figure 6.

Below we will look at the memory-attribute-related problems that we have in the Linux kernel.

3.1 Limited number of MTRRs

As explained in Section 2, typically there are only eight variable-range MTRRs supported on x86 CPUs. With an increasing amount of physical memory in the platform, the platform firmware ends up using most of those MTRRs to statically define memory types for the system memory range. When a driver later wants to use MTRR to map some range as write-combining, for example, there may not be any free MTRRs available for use. This results in the driver not being able to set a memory type. This can appear to the end user as (for instance) the video driver running less optimially, as it ends up using an uncached memory type for frame-buffer instead of the desired write-combine memory type.

3.2 MTRR conflict with BIOS MTRRs

As described in Section 2, certain overlapping MTRR memory types like write-combine and uncached result in causing the effective type to be uncached. On some platforms, the BIOS sets up PCI ranges explicitly mapped as uncached and a potential overlapping write-back for RAM. Later, when some driver which wants to map the same range as a write-combine memory type using MTRRs, there will be a conflict which results in the effective memory type being uncached. This results in the driver not being able to get optimal performance.

This has been reported a few times by end users on `lkml` [10] [3] [11] on various platforms. Unfortunately,

```
# cat /proc/mtrr
  reg00: base=0xd0000000 (3328MB), size= 256MB: uncacheable, count=1
  reg01: base=0xe0000000 (3584MB), size= 512MB: uncacheable, count=1
  reg02: base=0x00000000 (   0MB), size=8192MB: write-back, count=1
  reg03: base=0x200000000 (8192MB), size= 512MB: write-back, count=1
  reg04: base=0x220000000 (8704MB), size= 256MB: write-back, count=1
  reg05: base=0xcff80000 (3327MB), size= 512KB: uncacheable, count=1
```

Figure 5: Variable MTRR example

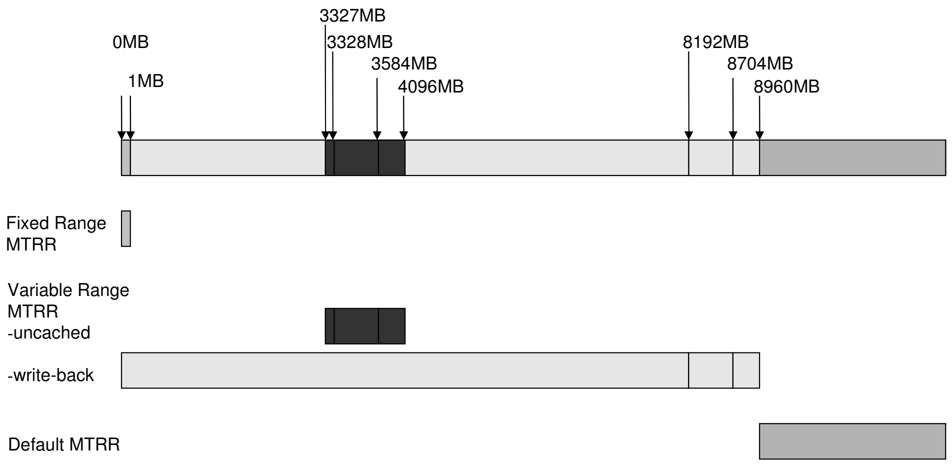


Figure 6: Effective memory types due to MTRR set by BIOS

there is no way to resolve this with existing PAT and MTRR support in the 2.6.25 kernel.

3.3 No well-defined APIs

Poorly defined APIs across PAT and MTRRs have led to various issues like:

- Drivers making assumptions about the underlying kernel implementation. For instance, the frame buffer driver is assuming that `ioremap()` will use PAT uncached-minus mapping and the driver follows the `ioremap()` call by an `mtrr_add()` call to set a write-combine memory type to the same range. Changing `ioremap()` from uncached-minus to uncached resulted in poor frame buffer performance in this specific case.

- Drivers setting the page table entries along with the PAT memory types natively, either by using `pgprot_noncached()` or by directly using `PAT`, `PCD`, and `PWT` bits. This results in flaky code, where the driver depends on kernel PAT usage.

4 Current PAT effort

4.1 Earlier PAT attempts

As was emphasised earlier, PAT as a hardware feature has been around for few years. During those years, there were quite a few attempts to enable PAT support in the Linux kernel.

One of the initial attempts was from Jeff Hartmann [8] in January, 2001. The patch proposed a `vmalloc()` kind of interface to support per 4K-page level write-combining memory type control. There were no responses on the mailing list archive, and so we conclude that the patch did not make its way into the Linux kernel for some unknown reason.

Terence Ripperda proposed PAT support with [9] in May, 2003, which supported adding write-combining mapping for AGP and framebuffers. There were concerns expressed about this patch in the mailing list, mainly related to keeping the memory type consistent for a physical address across different virtual mappings that may exist in the system.

```
mtrr: type mismatch for e0000000,8000000 old: write-back new: write-combining
mtrr: type mismatch for e0000000,4000000 old: write-back new: write-combining
```

Figure 7: common MTRR error message in kernel log

Terence followed it up with [4] in April, 2004, adding memory type tracking. That patch never made into the Linux kernel either, due to some concerns about various CPU errata on the mailing list.

Eric W. Biederman proposed PAT support with [6] in August, 2005, which started a fresh discussion on the mailing list about aliasing and PAT-related processor errata. Andi Kleen took up this patch and included it in his test tree [12]. However, the PAT support never got wide enough testing and did not get into the upstream Linux kernel. Also, none of the patches fully addressed attribute aliasing concerns.

4.2 Current PAT Proposal

Our initial proposals [7] [5] were based on Eric and Andi's patchset, with changes around identity mapping of reserved regions or holes and a few other cleanups and bug fixes. Those patches broke a lot of systems and provided us with a lot of feedback about what was not being done correctly.

Based on the feedback and breakage reports, we changed our patches to eliminate the issues around not having identity mapping for reserved regions and eliminated the changes for `early_ioremap`, simplifying our approach along the way. We also got benefits from other changes like `x86 change_page_attr()`.

The patches here [13] are version 3 of the patchset which was included in `linux-2.6.25-rc8-mm2`. The patchset also took a slightly different top-down approach, defining the PAT-related APIs and the eventual PAT bit setting for those APIs in different use cases, trying to ensure backward compatibility with the older versions of drivers and applications. All of the APIs related to PAT and memory type changes are described in detail in Section 5.

4.3 Preventing PAT aliasing

One of the big roadblocks for earlier PAT patches was aliasing related to PAT attributes. As per the processor specification [1], single physical address mapped to two or more different linear addresses should not have different memory types. Such aliasing can lead to undefined operations that can result in system failure.

The current PAT proposal handles this by using two internal functions, `reserve_memtype()` and `free_memtype()`. Any API that wants to change the memory type for a region first has to go through `reserve_memtype` to be sure that there are no aliases to the physical address, reserving the memory type for the region in the process. At the time of unmapping the memory type, API will free the range with a `free_memtype()` call. APIs will fail if `reserve` fails due to existing aliases.

Internally, the `reserve` function goes through a linked list which keeps track of physical address ranges with a specific memory type. The linked list is maintained in sorted order, based on the start address. The linked list may contain ranges with different sizes, and will detect partial or full overlaps with a single existing mapping or overlaps with multiple regions, with conflicting memory types.

If there is more than one user for a specific range with same memory type, the reference counting for such users are tracked by having multiple entries in the linked list.

To keep the implementation simple, this list is implemented as a simple doubly linked list. In the future, if there are any bottlenecks around this list, it can be optimized to have some cache pointers to previously reserved or freed regions, and changing the list into a more efficient data structure.

5 PAT APIs

One of the major challenges with PAT was to add the support in a clean manner, causing as few issues with existing applications and drivers as possible. As described in Section 3, the current memory type usage in Linux has some API-level confusion. That highlighted

API	RAM	ACPI, ...	Rsvd/ Holes
ioremap()	–	UC-	UC-
ioremap_cache()	–	WB	WB
ioremap_nocache()	–	UC	UC
ioremap_wc()	–	WC	WC
set_memory_uc() set_memory_wb()	UC	–	–
set_memory_wc() set_memory_wb()	WC	–	–
pci /sys resource file	–	–	UC-
pci /sys resource_wc file	–	–	WC
pci /proc/ device file	–	–	UC-
pci /proc/ device file ioctl PCIIOC_ WRITE_COMBINE	–	–	WC
/dev/mem read-write	WB	UC	UC
/dev/mem mmap with O_SYNC	–	UC	UC
/dev/mem mmap no O_SYNC	–	alias	alias
/dev/mem mmap no O_SYNC with no alias MTRR type WB	–	WB	WB
/dev/mem mmap no O_SYNC with no alias MTRR type not WB	–	–	UC-

Table 1: PAT related API cheat-sheet

the need to establish a clear API for everything related to memory type changes.

The API defined with the proposed PAT patches is described in detail below.

5.1 ioremap

`ioremap()`, `ioremap_cache()`, `ioremap_nocache()`, and `ioremap_wc()` are the interfaces that a driver can use to mark a physical address range with some memory type. The expected usage of these interfaces is over a physical address range that is either reserved/hole or a region used by ACPI, etc. `ioremap` and friends should never be used on a RAM region that is being used by the kernel.

`ioremap` interfaces, when they change the memory type, keep the memory type consistent across the virtual address where the address is being remapped into, and the kernel identity mappings.

`ioremap` interfaces may fail if there is an existing stricter memory type mapping. *Example:* If there is an existing write-back mapping to a physical range, any request for uncached and write-combine mappings will fail.

`ioremap` interfaces will succeed if there is an existing, more lenient mapping. *Example:* If there is an existing uncached mapping to a physical range, any request for write-back or write-combine mapping will succeed, but will eventually map the memory as uncached.

5.2 set_memory

`set_memory_uc`, `set_memory_wc`, and `set_memory_wb` are used to change the memory type of a RAM region. A driver can allocate a memory buffer and then use `set_memory` APIs to change the memory type for that region. It is the driver's responsibility to change the memory type back to write-back before freeing the page. A failure to do that can have nasty performance side effects as the page gets allocated for different usages later.

As with the ioremap interfaces, the kernel makes sure that the identity map aliases, if any, are kept consistent.

`set_memory` APIs can fail if there is a different memory type that is already in use for the same physical memory region.

5.3 `/proc` access to PCI resources

User-level drivers/applications can access a PCI resource region through the `/proc` interface. The resource file at `/proc/bus/pci/<dev>/` is `mmap`-able and an application can use that `mmap`ped address to access the resource.

By default, such an `mmap` will provide uncached access to the region. Applications can use `PCIIOC_WRITE_COMBINE` ioctl and get write-combine access to the resource, in cases where the region is *prefetchable*.

A request to uncached access can fail if there is already an existing write-combine mapping for that region. A request for write-combine access can succeed with uncached mapping instead, in the case of already existing uncached mapping for this region.

5.4 `/sys` access to PCI resource

Apart from the `/proc` interface described above, there is also a `/sys`-based interface that can be used to access PCI resources. It resides under `devices/pci<bus>/<dev>/`. This is again an `mmap`-able interface. The file `resource` is useful to get a UC access, and file `resource_wc`, to get write-combing access (in case the region is *prefetchable*).

The success and failure conditions of a `mmap` of the `/sys` PCI resource file are the same as in the `/proc` resource file description above.

5.5 read and write of `/dev/mem`

The existing API that allows read and write of memory through `/dev/mem` will internally use `ioremap()` and hence read and write using the uncached memory type. To read/write RAM, we use the identity mapped address, with existing WB mapping.

5.6 `mmap` of `/dev/mem`

`/dev/mem` `mmap` is an interface for applications to access any non-RAM regions. Applications have been using a `mmap` of `/dev/mem` for multiple uses. Changing `mmap` of `/dev/mem` behavior to go along well with PAT changes was one of the major challenges we had.

For example, X drivers use `mmap` to map a PCI resource first, followed by adding a new MTRR to make that physical address either uncached or write-combine. This will work, as the current `mmap` will just use write-back mapping in PAT, and MTRR uncached or write-combine takes higher precedence and changes the effective memory type for this region.

We will look at all the different usage scenarios of `/dev/mmap` below:

- *mmap with O_SYNC:* Applications can open `/dev/mem` with the `O_SYNC` flag and then do `mmap` on it. With that, applications will be accessing that address with an uncached memory type. `mmap` will succeed only if there is no other conflicting mappings to the same region.

- *mmap without O_SYNC and existing mapping for the same region:* Applications that do not use O_SYNC, when there is an existing mapping for the same region, will inherit the memory type from the existing mapping. This will be the case with applications mapping memory with a driver already having used `ioremap` to set the memory as write-back, write-combining, or uncached.

- *mmap without O_SYNC, no existing mapping, and write-back region:* The *ACPI data* region and a few other such regions are non-RAM, but can be accessed as write-back. There are applications like `acpidump` that `mmap` such regions. There is also a legacy BIOS region that is write-back that applications like `dmidecode` want to mmap. To be friendly to such usages, on an `mmap` of `/dev/mem` without `O_SYNC` and with no existing mappings, we look at the MTRR to figure out the actual type. If the MTRR says that region is write-back, then we use write-back mapping for the `/dev/mem` `mmap` as well.

- *mmap without O_SYNC, no existing mapping, and not a write-back region:* For an `mmap` that comes under this category, we use uncached-minus type mapping. In the absence of any MTRR for this region, the effective type will be uncached. But in cases where there is an MTRR, making this region write-combine, then the effective type will be write-combine. This behavior was added for a very special case, to handle existing X drivers without breakage. The X model of `mmap`ing graphics memory and then adding write-combining MTRR to it will work with this mapping.

Table 1 is a API cheat-sheet for application/driver interfaces to make any memory type changes.

6 Future Work

Given the success rate of earlier PAT patches, our first goal here is to ensure that the basic PAT patches make it to the upstream kernel with minimal disruptions, and also to provide APIs to applications and drivers that will remove long-standing MTRR limitations. As a result, our initial patchset does not address all the problems associated with PAT. Specifically, we have the following items in our immediate to-do list.

- Provide an API for drivers using `pgprot_noncached()` or handling the page table flags by hand to manipulate `PAT`, `PCD`, and `PWT` bits, so that they can do so ensuring no aliases. Currently, such drivers (mostly framebuffer and video drivers) do various things like mapping the pages as write-back in the page table and then calling MTRR to mark it write-combine, directly map the pages as uncached, etc.
- Ensure that PAT is not breaking any architectural guidelines while changing memory type attributes. Specifically, [1] mentions a sequence of steps that needs to be followed while changing the memory type attribute of a page from cacheable to write-combining. We need to make sure that the Linux kernel is not breaking any such rules.
- The future of the `/proc/mtrr` and `mtrr_add()` interfaces needs to be determined. The options are: deprecate those APIs and encourage the drivers and applications to switch to new PAT APIs, or emulate those APIs using PAT internally. This can be better addressed as current PAT patches makes into upstream and we get more feedback from the users about the PAT APIs and their usability.

7 Acknowledgements

Thanks to the developers and testers in the community who took time to comment on, report issues with, and contribute to PAT discussion on the mailing list. Special thanks to Asit Mallick and Arjan van de Ven for their continued support during our work on PAT pacthes; Ingo Molnar, Thomas Glexiner, H Peter Anvin for reviewing and fixing problems with PAT patches and also pushing the PAT patches through into x86 testing tree; Andi Kleen, Eric Biederman for their PAT patches which formed the basis for our first implementation; and everyone else who worked on PAT feature before or affected by PAT/MTRR issues on their platform and took time to report it on lkml mailing list.

References

[1] `Intel`$^{®}$ `64` and `IA-32` architectures software developer's manuals: `Volume 3A`. `http://developer.intel.com/products/processor/manuals`.

[2] Linux 2.6.25. `http://kernel.org/pub/linux/kernel/v2.6/linux-2.6.25.tar.bz2`.

[3] Mailing list archive. `MTRR` initialization. `http://lkml.org/lkml/2007/9/14/185`.

[4] Mailing list archive. `PAT` support. `http://www.ussg.iu.edu/hypermail/linux/kernel/0404.1/0686.html`.

[5] Mailing list archive. `[patch 00/11] PAT` x86: `PAT` support for x86. `http://www.uwsg.iu.edu/hypermail/linux/kernel/0801.1/1428.html`.

[6] Mailing list archive. `[PATCH]` i386, x86_64 initial `PAT` implementation. `http://www.ussg.iu.edu/hypermail/linux/kernel/0508.3/1321.html`.

[7] Mailing list archive. `[RFC PATCH 00/12] PAT` 64b: `PAT` support for x86_64. `http://www.uwsg.iu.edu/hypermail/linux/kernel/0712.1/2268.html`.

[8] Mailing list archive. `[RFC] [PATCH] PAT` implementation. `http://www.ussg.iu.edu/hypermail/linux/kernel/0101.3/0630.html`.

[9] Mailing list archive. pat support in the kernel. `http://www.ussg.iu.edu/hypermail/linux/kernel/0305.2/0896.html`.

[10] Mailing list archive. type mismatch for e0000000,8000000 old: write-back new: write-combining on kernel 2.6.12. `http://lkml.org/lkml/2005/6/18/52`.

[11] Mailing list archive. type mismatch for f0000000, 1000000 old: write-back new: write-combining. `http://lists.us.dell.com/pipermail/linux-poweredge/2006-March/025043.html`.

[12] Mailing list archive. What will be in the x86-64/x86 2.6.21 merge. `http://www.ussg.iu.edu/hypermail/linux/kernel/0702.1/0832.html`.

[13] Mailing list archive. x86: `PAT` support updated - v3. `http://lwn.net/Articles/274175/`.

Pathfinder—A new approach to Trust Management

Patrick Patterson
Carillon Information Security Inc.
ppatterson@carillon.ca

Dave Coombs
Carillon Information Security Inc.
dcoombs@carillon.ca

Abstract

PKI has long promised to solve the problem of scalable identity management for us. Until now, that promise has been rather empty, especially in the Free and Open-Source Software (FOSS) space. Generally speaking, the problem is that making the proper trust decisions when presented with a certificate involves such esoteric magic as checking CRLs and OCSP and validating trust and policy chains, all of which are expressed as arcane ASN.1 structures; usually this is not the application developer's primary focus. Coupled with the lack of central PKI management tools in the FOSS environment, the result is a whole lot of applications that sort of do PKI, but not in any truly useful fashion. That is, an administrator cannot really replace username/password systems with certificates, which is what PKI was supposed to let us do. Microsoft has finally built a decent certificate-handling framework into their products, and we believe that Pathfinder adds this level of support to FOSS products. This presentation will focus on what Pathfinder is, how it can be used to deploy a scalable trust management framework, and, most importantly, will demonstrate how easy it is to make your own application "Pathfinder aware."

1 Introduction

Before we discuss Pathfinder in detail, it is useful to take a look at Public Key Infrastructure in general, its history, and the current directions within this field. This provides a backdrop and common reference for understanding why we have chosen the architecture for trust management with Pathfinder that we have.

While PKI is, generally speaking, not new technology, it has been quite slow to find mainstream adoption and use. This is partly because of the difficulty in designing an implementation that satisfies everybody's requirements, and partly because of interoperability problems arising from a range of implementations that interpret the standards differently. Recent developments, however, show great promise in growing interoperable deployments.

Formerly it was thought that the "holy grail" of PKI was a single, global Certificate Authority, or small group of such CAs, that would issue all the certificates, and that everybody would trust these CAs. This would eventually prove to be impossible, and the biggest problem is a political one: who runs the global CA? Who is, therefore, the global trust authority, and why should one group have so much power? These questions could not be satisfactorily answered. Furthermore, many different groups and interests are represented in such a system, and it is probably impossible to create a meaningful global PKI policy framework that is consistent with the needs of these different groups and different industries and different legal regulations.

In the past several years a new model has emerged, represented by interoperable communities of trust. Groups of people or companies with similar requirements, be they industry requirements, legal requirements or otherwise, can define a community of trust and a set of policies and procedures that are suitable for that community, which then can be adopted and followed by all participants.

The current way forward for these communities of trust is to use Bridge Certificate Authorities. A Bridge CA is one that doesn't issue certificates to subscribers itself, but rather exists to facilitate the trust fabric within a community. If a bridge CA is set up inside a specific community, participating community members with their own CAs can cross-certify with the bridge, using policy mapping to create equivalence among assurance levels which can allow trust to flow through the community. There are two principal benefits of using a bridge. The first is that participant CAs do not have to cross-certify with every other CA in the community, and instead only manage a single audited trust relationship.

The second benefit is that a user can declare an identity to the community, and all the participants in the community can recognize that identity. The bridge CA can also then cross-certify with other bridges, allowing trust to flow to other communities as well.

Figure 1 shows the current status of certain interconnected communities of trust that already exist. Currently the US Federal Bridge CA is acting as a "super bridge," cross-certified with government departments, but also cross-certified with an aerospace and defence bridge, a pharmaceutical bridge, and a higher education bridge. It is to be noted that each of these communities is standalone, and each of the companies listed operates its own PKI. The arrows only represent policy mapping between each of the participants.

Pathfinder was conceived as a method to allow FOSS projects to seamlessly handle the complexities inherent in this cross-certified "community PKI" framework.

2 Technical Expressions of Trust

IETF RFC3280 describes a detailed, standard profile for expressing an identity in a PKI context, and methods for ascertaining the status and current validity of such an identity. X.509 certificates have a Subject field, which can hold some representation of "identity," and the RFC3280 standard Internet profile also includes Subject Alternative Names that can express email addresses, DNS names, any many other forms (some people think that too many things can be expressed here!) There is also the Authority Information Access extension, which can contain an LDAP or HTTP pointer to download the signing certificate of the CA that issued and signed a given certificate, as well as a pointer to the Online Certificate Status Protocol (OCSP) service that can give revocation status about this certificate. If the CA doesn't support OCSP, a relying party can fall back to checking the Certificate Revocation List (CRL), a pointer to which can be found in the certificate's CRL Distribution Points extension. We can check the policy under which a certificate was issued, as found in the Certificate Policies extension. In a bridged environment, we may encounter Policy Mapping, so we will have to check that extension as well. And we mustn't forget Name Constraints, which, within a bridged (or even a hierarchical) PKI allow delegation of authority over name spaces such as email, DNS, and X.500, to particular Certificate Authorities.

Given the above, we can identify several hurdles. First of all, performing certificate validation correctly pretty much requires one to be a PKI expert, as there is a high degree of complexity involved. We shouldn't necessarily expect application programmers, who are experts in building web servers, mail servers, RADIUS servers, or other applications where using certificates for authentication may be desirable, to stop and learn how to do this validation correctly. It probably isn't their primary concern. Secondly, we shouldn't necessarily expect libraries commonly used for functions like TLS to help us out of this, at least not fully. For instance, OpenSSL and libNSS are very good security libraries, but there are just too many details involved in building a trust path to expect these libraries to possess all of the required capability. As a specific example, we rather strongly doubt that the OpenSSL maintainers will ever want to integrate an HTTP and LDAP client in their library and provide all the hooks necessary to synchronously and/or asynchronously fetch certificates, CRLs, and OCSP information from Certificate Authorities.

2.1 So, where does this leave us?

The situation today is that most applications that implement any form of certificate support do a rather poor job of it. Most commonly, they implement the capability to act as the server portion of a TLS connection, and either stop there or offer very rudimentary (in our experience) client authentication support, often limited to checking whether the certificate presented is signed by a known and trusted CA and is in its validity period. The need to somehow check certificate status, for example by CRL or OCSP, is mostly ignored, as is the requirement in some communities to only accept certificates issued to a certain policy. There is certainly no thought given to how to deploy such an application in an environment with a complex trust fabric, such as the above cross-certified domains, where the aforementioned name constraints and policy mappings need to be evaluated. Furthermore, client side validation of server certificates is almost never properly implemented, which is understandable in that it is very difficult to communicate to an end user what to do when, for example, the server's certificate is revoked.

To solve these issues, it is unrealistic to expect each application developer to be a PKI expert and correctly implement all the complexity and nuance of the Path Dis-

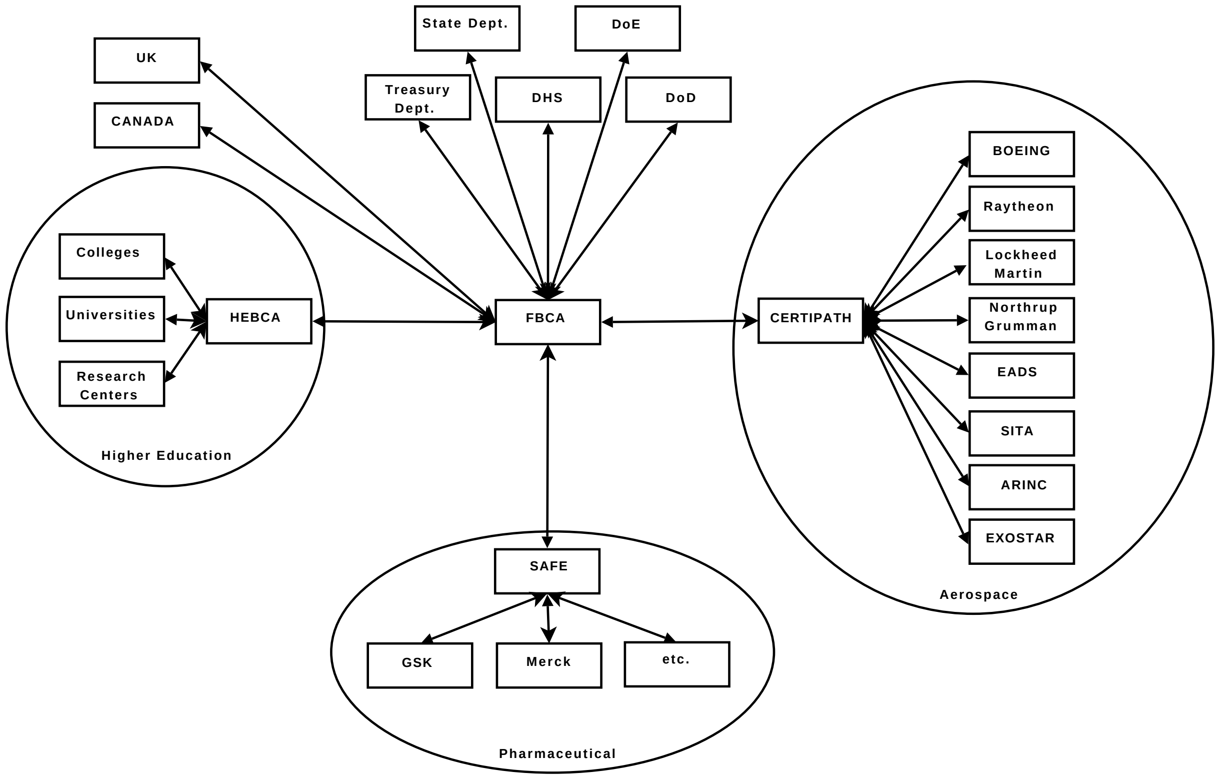


Figure 1: Interconnected Communities of trust, present and planned

covery and Validation algorithm described in RFC 3280. Therefore, another solution is needed.

Pathfinder solves the above problem by providing two components. The first is a series of client libraries that provide callbacks for certificate validation for all major security libraries (currently OpenSSL and libNSS, but hooks to the Java CryptoAPI and GNUTLS are also planned), and which imposes on applications only one additional dependency: the need to link with DBus, which the client uses to talk to the daemon

The second component is a system daemon that provides all of the certificate validation functions in a transparent and centrally manageable fashion. This approach allows an application to grow the capability of handling a complex trust environment simply, usually by only replacing a single line of code, and perhaps by adding an option for the policy to be set for that application by its configuration. All of the hard work is done in the daemon.

3 Advantages of Pathfinder

Pathfinder was written with the credo "Do the hard stuff once, let everyone benefit" in mind.

In a complex trust environment such as a bridged PKI, it is critical that any Policy Mapping and Name Constraints extensions are correctly handled. As illustrated by Figure 2, Policy Mapping enables a company to declare that its policy for issuing certificates is equivalent to another company's policy, often via a central, community-endorsed policy (that of the bridge.) Differentiation by policy, when evaluating a given certificate, is also useful as it is now fairly common not to include policy information in the Distinguished Name, including it instead only where it belongs, in the Certificate Policies extension. So, it would be possible to have two certificates with the same Distinguished Name (they refer to the same security principal after all), but issued according to a different policy. Without a way to tell an application only to accept certificates issued according to a certain policy, we have a security issue if the user presents a certificate issued according to a less stringent policy.

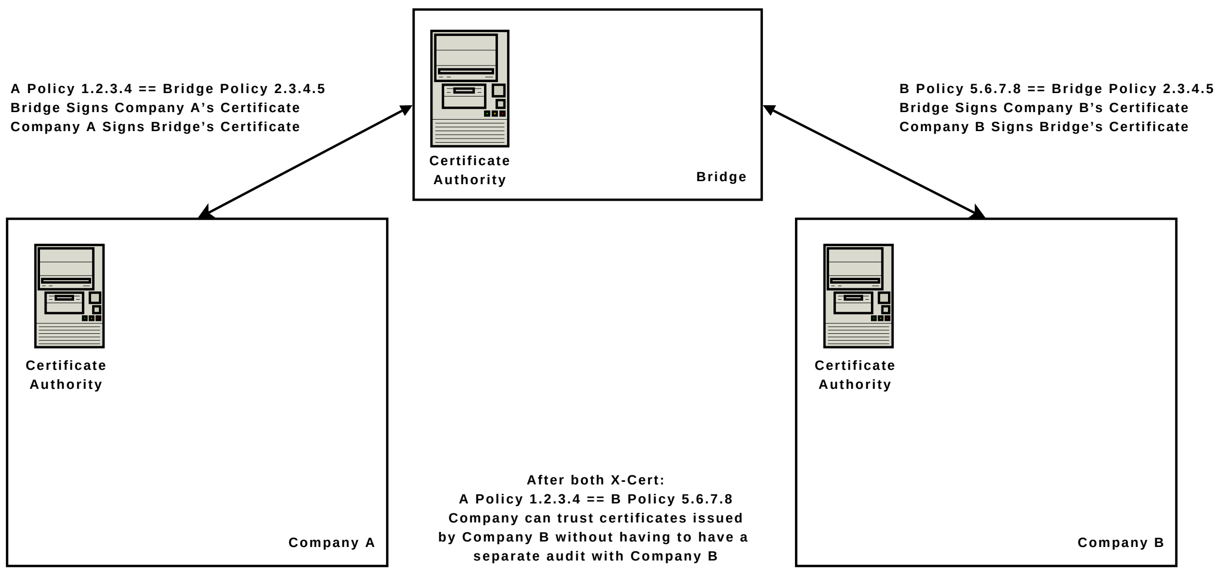

A Policy 1.2.3.4 == Bridge Policy 2.3.4.5
Bridge Signs Company A's Certificate
Company A Signs Bridge's Certificate
Certificate Authority
Bridge
B Policy 5.6.7.8 == Bridge Policy 2.3.4.5
Bridge Signs Company B's Certificate
Company B Signs Bridge's Certificate
Certificate Authority
Company A
After both X-Cert:
A Policy 1.2.3.4 == B Policy 5.6.7.8
Company can trust certificates issued
by Company B without having to have a
separate audit with Company B
Certificate Authority
Company B

Figure 2: Policy Mapping in a bridge environment

Name constraints are required to ensure that only a single organisation in the trust fabric can be authoritative for a particular instance of a security principal (in plain terms, you usually only want John Doe who works at Company A to have a certificate issued by Company A.) Another way to put this is that name constraints can be used to state that a given CA can only issue certificates to Subjects inside a given namespace (for instance, email domain), and can be used to state that no other CA, for example across the bridge, can issue certificates in that namespace.

The procedures for correctly handling policy mapping and name constraints run to several dozens of pages within various RFCs and specifications, and it is certainly a sufficiently tricky task that one would hope it is only done once on a given platform. As more and more applications are being deployed into the, rather large, communities mentioned in the introduction, it is becoming more and more desirable for applications to all handle these complex issues, and to handle them in a consistent manner.

It is equally important that the status of a certificate be validated, to ensure that the certificate has not been revoked. And, perhaps most interesting for those deploying an application in a complex trust environment, ?missing? certificates must be obtained, and a trust path between the client certificate and the configured trust anchor must be built. These transactions may involve round trips to external LDAP, HTTP, and OCSP servers, and thus, unless done very efficiently, may induce transaction delays and substantially impact server performance.

Which leads us to the question of providing a scalable caching layer.

When performing Path Discovery and Validation, there are several potential bottlenecks. Clearing these bottlenecks is highly desirable, since performing the certificate validation is a blocking function in the establishment of protocols such as IPsec and TLS. That is, there is no opportunity to send the request in an asynchronous manner. A Certificate is presented, and it must be validated there, on the spot, before proceeding with the rest of the establishment of the session. The functions where caching is most desirable are Authority Information Access (AIA) chasing, CRL downloading, and making an OCSP request. For all of these we have to worry about, from a performance point of view, the latency of performing the query, and the DNS or TCP timeouts if the host listed is not available. For CRLs, we have the additional problem that what we download is of an unknown, arbitrary size, and may be quite large (for instance, the US Department of Defence CRL was over 50MB at one point).

Optimization of the above functions is perhaps one of the greatest reasons for choosing a centralized daemon model for Pathfinder. By performing everything in the daemon, we can optimise these functions in a single lo-

```
Apache:

Before:
Incomplete support for complex trust environments:
No support for Policy Mapping
Must supply all intermediate certificates, which is undesirable in a
properly configured bridge environment.
No support for Name Constraints, as a matter of fact, if they are present
and critical, the certificate will not validate at all, which is correct
behaviour, but it means that Apache can't be deployed using Certificate
based authentication in much of the aerospace industry).
No support for OCSP
No support for certificate validation based on Certificate Policy
CRL update requires re-starting the server.
Per server configuration of trust anchors

After:
Support for full RFC3280 Path Discovery and Validation
Full support for CRL and OCSP.
Specification of policy against which a certificate would be validated.
Full support for policy mapping.
All servers within a farm may be configured at once, and trust anchors
updated to all.

Time to implement:
2 days

Biggest Challenge:
Dealing with the apache configure/makefile system.

Additional Dependencies:
libdbus
```

Figure 3: A concrete example

cation to minimise a very critical performance bottleneck, and we also have the greatest opportunity to re-use information already obtained. For instance, if application A requests the certificate chain W->X -> Y -> Z , and we need to fetch X and Y, then we can cache those certificates for the duration of their lifetimes, and can therefore avoid having to re-fetch the same certificates, thus saving time for application B that requests the same chain at some future date. Changes to the chain can be managed by having a maintenance thread which periodically checks the chain for any changes, such as refreshed certificates with different CRL or OCSP URIs, or different lifetimes. The same can be done with the CRL and OCSP responses, although, of course, the maintenance thread will have to refresh more frequently, due to the shorter lifetimes of these artifacts.

Performing these functions in one location also offers the option of a graceful "offline" mode, in case the validation is performed offline, such as a user validating an S/MIME signature while on a flight. Instead of each application needing built-in logic for permissible failure modes, Pathfinder can be centrally configured to perform the appropriate level of validation, thereby ensuring that each application handles offline validation in a consistent manner. An example of such a validation scheme is to have three settings, the first of which is to require full validation, which will fail to validate the signature because it can't find the revocation information when offline, and may not be able to chase AIA information. The second setting is to require a valid trust chain but not necessarily fresh revocation info, which may work, depending on how fresh the cache is, and the the third setting is to blindly accept all certificates if offline, which will always validate, assuming that the certificate hasn't expired.

This last feature is not yet implemented, but is definitely on the roadmap.

4 Conclusion

Proper handling of certain X.509 certificate extensions when performing certificate validation is essential in order to maintain a viable trust fabric in a bridged PKI framework serving a community of trust. In such a community, each participant need only be responsible for one trust relationship: its relationship with the bridge. The bridge, then, brokers trust among the community participants who have agreed to a common policy framework. However, in order for this to work, there is a substantial requirement on the various server software packages used in the community to correctly process the trust path and policy tree. Until now, due to the complexity of handling this processing, it has been difficult for application developers to deploy full PKI support in their applications. Pathfinder not only makes this simple, it provides a scalable and manageable way to deploy true PKI-enabled applications using only open source software.

Linux Data Integrity Extensions

Martin K. Petersen
Oracle
martin.petersen@oracle.com

Abstract

Many databases and filesystems feature checksums on their logical blocks, enabling detection of corrupted data. The scenario most people are familiar with involves bad sectors which develop while data is stored on disk. However, many corruptions are actually a result of errors that occurred when the data was originally written. While a database or filesystem can detect the corruption when data is eventually read back, the good data may have been lost forever.

A recent addition to SCSI allows extra protection information to be exchanged between controller and disk. We have extended this capability up into Linux, allowing filesystems (and eventually applications) to be able to attach integrity metadata to I/O requests. Controllers and disks can then verify the integrity of an I/O before committing it to stable storage. This paper will describe the changes needed to make Linux a data-integrity-aware OS.

1 Data Corruption

As witnessed by the prevalence of RAID deployment in the IT industry, there is a tendency to focus on data corruption caused by disk drive failures. While head misses and general bit corruptions on the platter *are* common problems, there are other possible corruption scenarios that occur frequently enough to warrant being remedied as well.

When corruption is experienced, hardware is often blamed. However, modern systems feature extensive protection on system buses, error checking and correcting memory, etc. In the Fibre Channel environments commonly used in enterprises, wire traffic is protected by a cyclic redundancy check. So in many ways the physical hardware level is becoming increasingly resilient against failures.

The software stack, however, is rapidly growing in complexity. This implies an increasing failure potential: Harddrive firmware, RAID controller firmware, host adapter firmware, operating system code, system libraries, and application errors. There are many things that can go wrong from the time data is generated in host memory until it is stored physically on disk.

Most storage devices feature extensive checking to prevent errors. However, these protective measures are almost exclusively being deployed internally to the device in a proprietary fashion. So far, there have been no means for collaboration between the layers in the I/O stack to ensure data integrity.

An extension to the SCSI family of protocols tries to remedy this by defining a way to check the integrity of an request as it traverses the I/O stack. This is done by appending extra information to the data. This extra information is known as *integrity metadata* or *protection information*.

The integrity metadata enables corrupted write requests to be detected and aborted, thereby preventing silent data corruption.

2 Data Integrity Field

A harddisk is generally divided into blocks of 512 bytes, called *sectors*. On the physical platter, the sector allocation is actually a bit bigger to allow for CRC and information internal to the drive. However, this extra information is not available outside of the disk drive firmware. Consumer-grade disks often trade capacity for reliability and use less space for the error checking information. Enterprise disks feature stronger protection schemes and as a result, expose less storage capacity to the user.

Unlike drives using parallel and serial ATA interfaces, SCSI[1] disks allow for sectors bigger than 512 bytes to be exposed to the operating system. Sizes of 520 or 528 bytes are common. It is important to note that these 'fat' sectors really are bigger, and that the extra bytes are orthogonal to space used for the drive's internal error checking.

Traditionally these extra few bytes of information have been used by RAID controllers to store their own internal checksums. The drives connected to the RAID head are formatted using 520-byte sectors. When talking to the host operating system, the RAID controller only exposes 512-byte blocks; the remaining 8 bytes are used in a way proprietary to the RAID controller firmware.

A few years ago, an extension to the SCSI Block Commands specification was approved by the T10 technical committee that governs the SCSI family of protocols. The extension, known as *Data Integrity Field*, or *DIF*, standardizes the contents of the extra 8 bytes of information per 520-byte sector.

This allows the integrity metadata to be visible outside of the domain of disk or RAID controller firmware. And as a result, this opens up the possibility of doing true end-to-end data integrity protection.

2.1 The DIF Format

Each 8-byte DIF tuple (see Figure 1) contains 3 tags:

- *Guard tag:* a 16-bit CRC of the sector data.
- *Application tag:* a 16-bit value that can be used by the operating system.
- *Reference tag:* a 32-bit number that is used to ensure the individual sectors are written in the right order, and in some cases, to the right physical sector.

A DIF-capable host adapter will generate the 8 bytes of integrity metadata on a write and append it to the 512-byte sectors received from the host operating system. For read commands, the controller will receive 520-byte sectors from the disk, verify that the integrity metadata matches the data, and return 512-byte sectors to the operating system.

[1]We will use the term 'SCSI' to refer to any device using the SCSI protocol, regardless of whether the physical transport is SPI, Fibre Channel, or SAS.

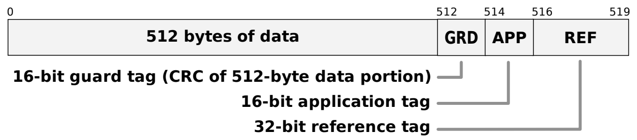


Figure 1: 520-byte sector containing 512 bytes of data followed by 8-byte DIF tuple

A DIF-capable disk drive can compare the data with the integrity metadata received from the host adapter and reject the I/O if there is a mismatch. How to interpret the DIF tuple content depends on the protection type the drive has been formatted with. The current specification allows three types, and they all mandate use of the guard tag to protect the contents of the 512-byte data portion of the sector.

- *DIF Type 1:* reference tag must match lower 32 bits of the target sector number.
- *DIF Type 2:* reference tag must match the seed value in the SCSI command + offset from beginning of I/O.
- *DIF Type 3:* this reference tag is undefined.

The drive must be low-level reformatted to switch between the three protection types or to turn off DIF and return to 512-byte sectors.

3 Data Integrity Extensions

The T10 standards committee only defines communication between SCSI controllers and storage devices. The DIF specification contains no means for sending/ receiving integrity metadata to/from host memory, and traditionally host adapter programming interfaces have been proprietary and highly vendor-specific.

Oracle approached several fibre channel adapter vendors putting forth a set of requirements for controllers to allow exchanging integrity metadata with the host operating system. This resulted in a specification [2] for what is now known as the *Data Integrity Extensions*, or *DIX*.

DIX defines a set of interfaces that host adapters must provide in order to send and receive I/O requests with integrity metadata attached. This in turn enables us to extend the exchange of protection information all the way up to the application.

If the controller is DIX-capable and the storage device is DIF-capable, we can create a protection envelope that covers the entire I/O path, thus providing true end-to-end data integrity (see Figure 2).

3.1 Performance Impact

The 16-bit CRC mandated by the DIF specification is somewhat expensive to calculate in software. Benchmarks showed that for some workloads, calculating the CRC16 in software had a detrimental impact on performance. One of Oracle's partners had hardware capable of using the IP checksum instead of CRC16. The IP checksum is much cheaper to calculate and offers weaker protection than the CRC, so there is a trade-off between data integrity and performance. Consequently, the IP checksum feature is optional and can be enabled at will.

If the IP feature is enabled, Linux will put IP checksums in the guard tag instead of CRC16. The controller will verify the checksums and convert them to the T10-mandated CRC before passing the data to the drive. On reads, the opposite conversion takes place.

From a performance perspective, the cost is very low. It has less impact on system performance than software RAID5.

4 SCSI Layer

We have implemented support for both DIF and DIX in the Linux kernel. The work has been done from the bottom up, starting with the SCSI layer. The following sections will describe the changes required.

4.1 Discovery

For the exchange of integrity metadata to happen, it would seem reasonable to require that controller and storage device are DIX- and DIF-capable, respectively. However, even in a setup where the disk does not support DIF, there is still value in having the host adapter verify the data integrity before sending the command on to the drive.

Similarly, some controllers may support DIF while talking to the drive, but may not have the capability to exchange integrity metadata with Linux. In that situation it is still desirable to have communications between host adapter and disk protected.

Consequently two orthogonal negotiations are taking place at discovery time: One for DIX between Linux and the SCSI controller, and one for DIF between controller and storage device.

The controller driver indicates its DIF and DIX capabilities when it registers itself with the SCSI layer. The DIF type is probed when a drive is scanned. If both DIX and DIF are supported, integrity metadata can be exchanged end-to-end.

4.2 Scatter-Gather List Separation

A buffer in host memory that needs to be transferred to or from a storage device is virtually contiguous. This means that the application sees it as one linear blob of data. In reality the buffer is likely to be physically discontiguous, made up of several scattered portions of physical memory. Consequently, a more complex construct is needed to describe what to transfer.

Network and storage controllers use a scatter-gather list for this purpose. The list consists of one or more `<page address, offset, length>` tuples, each identifying a region in memory to transfer as part of the request.

Linux performs all block I/O in multiples of 512 bytes and it would be highly inconvenient to support 520-byte sectors and buffers throughout the kernel.

On the wire between controller and disk, however, integrity metadata must be interleaved with the data sectors; therefore, the buffer sent to the disk must be a multiple of 520 bytes long.

As a result, DIX requires separating the data and integrity metadata in host memory. The data buffer remains unchanged, while the integrity metadata is stored in a separate buffer. The two buffers are then mapped into separate scatter-gather lists which are handed to the I/O controller.

When writing, the controller will transfer the memory described by the two scatterlists from the host, check them, and interleave data and integrity metadata before the request goes out on the wire as 520-byte sectors.

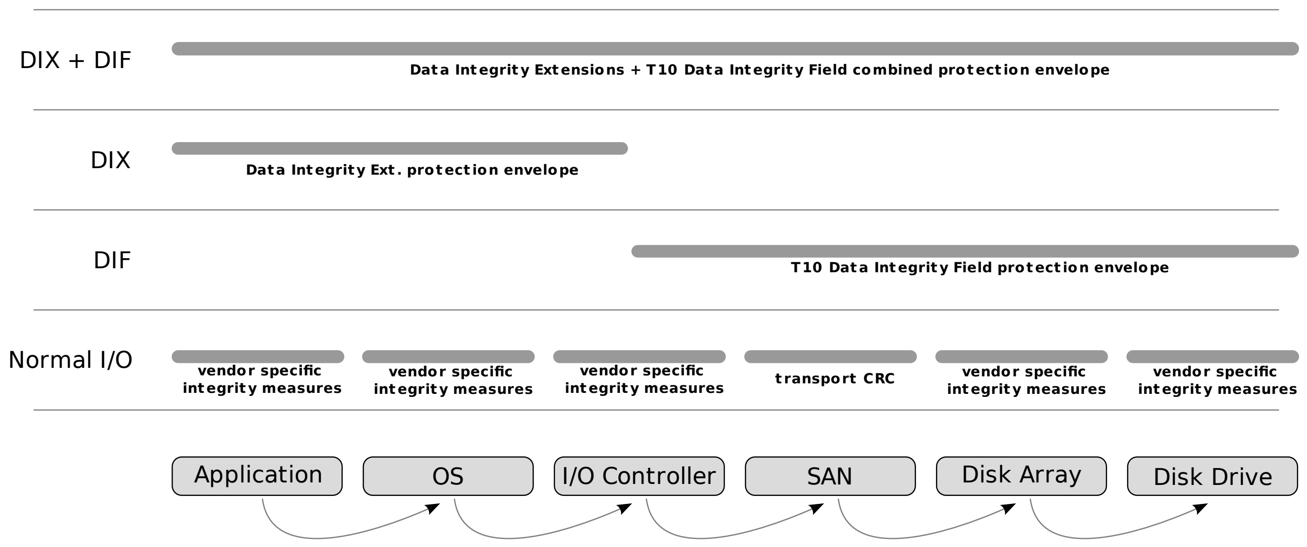

DIX + DIF
Data Integrity Extensions + T10 Data Integrity Field combined protection envelope
DIX
Data Integrity Ext. protection envelope
DIF
T10 Data Integrity Field protection envelope
Normal I/O
vendor specific integrity measures
vendor specific integrity measures
vendor specific integrity measures
transport CRC
vendor specific integrity measures
vendor specific integrity measures
Application
OS
I/O Controller
SAN
Disk Array
Disk Drive

Figure 2: Protection Envelopes: The 'Normal I/O' line shows disjoint protection domains in a normal setup. Above that, the 'DIF' line illustrates the area covered by the T10 DIF standard. 'DIX' displays the coverage of the Data Integrity Extensions, and at the top, 'DIX+DIF' combined yields a full end-to-end protection envelope.

On read, the 520-byte sectors sent by the drive are verified, split up, and transferred into the host memory described by the two scatter-gather lists provided by the kernel.

This separation of data and integrity metadata makes it much less intrusive to support DIF in the kernel.

The integrity buffer is described by an extra `scsi_data_buffer` in `struct scsi_cmnd`, which is the container for SCSI requests in the kernel.

4.3 Reference Tag Remapping

When a drive is formatted with Type 1 protection, the reference tag must contain a value corresponding to the physical sector the data is being written to for the I/O to complete successfully. Thanks to partitioning and stackable devices such as MD or the Device Mapper, the physical sector LBA is often very different from what the filesystem requests when submitting the I/O. The reference tag needs to be remapped accordingly.

One solution would be to postpone filling out the reference tag until the physical sector number is actually known. However, we would like to leverage the protection offered by the reference tag's ability to tie the individual sectors of an I/O together.

Another option would be for the filesystem to recursively query the underlying block devices requesting the start LBA. Unfortunately, this will not work, as an I/O may straddle physical devices. The solution is to have a virtual reference tag filled out when the I/O is submitted by the filesystem. That virtual tag is then remapped to the physical value at the bottom of the I/O stack when writing. Similarly, when data is read, the physical reference tags received from the drive are remapped to the virtual numbers expected by the filesystem.

This approach also avoids multiple remapping steps as the request traverses a layered I/O stack.

5 Block Layer

Conceptually, DIF and DIX constitute *a blatant layering violation*. Applications do not know or care whether they are accessing a SATA or a SCSI disk, or whether the data is mounted over the network. On the other hand, for the end-to-end protection to work, applications or filesystems need to know how to prepare integrity metadata in a format understood by the actual physical device.

Thankfully, the provider of the integrity metadata does not have to be aware of the intricate details of what is inside the integrity buffer, and consequently the block layer treats the integrity metadata in an opaque fashion.

It has no idea what is stored inside the extra structure attached to the bio.[2]

5.1 Block Integrity Payload

The integrity metadata is stored in the block integrity payload, or bip struct which attached to the bio. The bip is essentially a trimmed-down version of the I/O vector portions of a struct bio with a few extra fields for housekeeping, including the virtual sector number used for remapping.

A series of bio_integrity_* calls allows interaction with the protection information, and these have been designed to closely mirror the calls for allocating bio structures, adding pages to them, etc.

5.2 Integrity Properties, Splitting and Merging

There are only a few things the block layer really needs to be aware of with respect to the attached protection information:

- Because a bio can be split and merged, the block layer needs to know how much integrity metadata goes with each chunk of data.
- The layer needs to know whether the device is capable of storing extra information in the application tag.
- It must be capable of generating and verifying the integrity metadata.

All this information is communicated to the block layer when a storage device registers itself using blk_integrity_register(). In the DIF case, this is done just after the SCSI disk makes its presence known to the kernel.

The 16 bits of space in the DIF application tag may or may not be used internally by the storage device. A bit in the device's SCSI Control Mode Page indicates whether it is available. If it is, the SCSI disk driver will signal to the block layer that the space is available for use by the filesystem.

As part of that registration process, the SCSI disk driver also provides two callback functions to the block layer: one for generating integrity metadata, and one for verifying integrity metadata. This way, the block layer can call the functions to opaquely generate and check protection information without knowing the intricate details of SCSI, DIF, or how the drive has been formatted.

[2]struct bio is the fundamental block I/O container in the Linux kernel.

5.3 Stacked Devices

Servers often use software RAID (MD) and/or the Logical Volume Manager. These are implemented as virtual block devices inside the kernel. If all the disks that constitute an MD disk or a logical volume support the same type of protection, the virtual block device is tagged as being integrity-capable.

A similar approach is taken for virtual block devices exposed to virtualized guests, allowing the protection envelope to reach all the way from the application running on the guest through the hypervisor to the storage device.

5.4 Automatic Generation/Verification

Filesystems that allow integrity metadata to be transferred to/from userland are expected to interact directly with the bip calls. However, legacy filesystems like ext3 and ext4 are not integrity-aware. There are also other I/O code paths that either originate inside the kernel or map user pages directly. For those cases, the integrity infrastructure allows protection information to be automatically generated by the block layer (writes) or verified before the bio is returned to the submitter (reads).

Normally, I/O completion is run in interrupt context, as it usually only involves marking the pages referenced by the request as being up-to-date. However, calculating a checksum for the entire I/O is a time-consuming process. If the request needs to be verified, completion is postponed using a work_queue.

The automatic generation/verification of integrity metadata enables integrity protection of all I/O from the block layer to the disk without any changes to the filesystem code.

6 Filesystem Interface

6.1 Protection Information Passthrough

Filesystems that wish to allow transfer of integrity metadata to and from userland applications will need to manually attach it to the `bio`. This is done by attaching a `bip` to the `bio` and then adding the protection information pages using `bio_integrity_add_page()`.

6.2 Tagging

As mentioned above, the DIF tuple includes a 16-bit application tag that is stored by the block device as any other type of data; i.e., it is not used for integrity verification in any of the existing protection types.

These 16 bits can be used freely by the owner of the block device—in this case the filesystem—to tag the sectors. One possible use is to identify which inode a sector belongs to. This will significantly improve the `fsck` process' ability to recover a damaged filesystem.

Filesystems generally use blocks that are bigger than 512 bytes. Because two bytes per sector is a very limited space, the block integrity infrastructure allows tagging at the `bio` level instead. An opaque buffer containing the filesystem-internal information can be supplied at integrity-metadata-generation time. The data in the buffer is then interleaved between the application tags in the sectors targeted by the `bio`, enabling the filesystem to store 16 bytes of recovery information for each 4KB logical block.

The tag data can subsequently be read back by running `bio_integrity_get_tag()` upon completion of a read `bio`.

7 Future Work

Work is in progress to implement support for the data integrity extensions in btrfs [1], enabling the filesystem to use the application tag. The next step will be defining the interfaces that will allow applications to perform syscalls that include integrity metadata.

We are working on three different interfaces that expose integrity metadata to userspace applications:

1. *Transparent:* Integrity metadata is generated by the C library transparently to the application.

2. *Opaque:* This interface will allow the application to protect a buffer in memory prior to submitting the I/O to disk. Just like the block layer, the application will not know that the actual integrity metadata is in DIF format.

3. *Explicit:* Some applications will need direct access to the native integrity metadata, bypassing the filesystem. Examples are the `mkfs` and `fsck` programs that need to be able to read and write the application tag directly.

T13, the committee that governs the SATA specification, has proposed a feature called *External Path Protection* which is essentially the same as DIF. The Linux kernel data integrity infrastructure has been designed to accommodate DIF as well as EPP. A similar data integrity feature for SCSI tape drives is also in development.

Products supporting DIF and DIX are scheduled for general availability in 2008. The Linux Data Integrity Project can be found at `http://oss.oracle.com/projects/data-integrity/`.

Acknowledgements

Thanks to Randy Dunlap, Joel Becker, and Zach Brown for their feedback on this paper.

References

[1] Chris Mason. btrfs. `http://oss.oracle.com/projects/btrfs/`.

[2] Martin K. Petersen. I/O Controller Requirements for Data Integrity Aware Operating Systems. `http://oss.oracle.com/projects/data-integrity/dist/documentation/dif-dma.pdf`.

Red Hat Linux 5.1 vs. CentOS 5.1: ten years of change

D. Hugh Redelmeier
hugh@mimosa.com

Abstract

Red Hat Linux 5.1 was released in 1998. Almost ten years later, its direct descendant CentOS5.1 was released in 2007. How much has changed in the years since the first Ottawa Linux Symposium?

To investigate these changes, both systems were installed and used on the same hardware. What were the important changes? Did we use or abuse new resources as hardware developed along Moore's Law? Were the times as golden as some old-timers remember them to be? Can the youngsters still be taught a thing or two?

1 Introduction

In what ways has Linux changed? Most of us experience changes release by release. Taking a longer term view should yield a different set of insights.

Although recollection is a good tool, actual investigation seems worthwhile. To this end, I have installed Red Hat Linux 5.1 and CentOS5.1 on the same hardware. By using and examining these two platforms, I hope to investigate and compare them.

These platforms were chosen for several reasons. I have used each when they were current. Both were popular in their respective eras. One is a logical successor of the other (Red Hat Linux evolved to Red Hat Enterprise Linux, and CentOS is a clone of RHEL) so the codebases are strongly related. Finally, it is appealing that their version numbers happen to be identical.

2 Environment

Computer hardware made great capacity advances between the releases. Computers have become more pervasive in that same period. These environmental changes have affected the releases.

Date		1997 Oct	2007 June
price	$\div 5$	C$1965	C$400
brand		local shop	Acer Aspire E380
CPU	$\times 2$	AMD K6	AMD Athlon 64 x2
CPU clock	$\times 11$	200MHz	2200MHz
RAM	$\times 16$	64M	1024M
RAM type		PC66	PC2-5300
hard disk	$\times 39$	6.4G	250G
HD RPM	$\times 1.3$	5400	7200
optical		CD reader	DVD writer

This table illustrates the changes in hardware capacity. It sketches the dimensions of two computers that I bought to be Linux workstations. The first system's components were selected to be the most powerful I could get without leaving the mainstream whereas the second system was designed by Acer for normal home or office users.

The changes are large enough that they should drown out the effects of whether I selected a high-end or mainstream system at either time.

The changes in most dimensions are so large that one would expect them to be experienced as qualitative differences, not just quantitative. Think what it would be like if your house had forty times the floor space, the frequency of your piano's A key went up by a factor of eleven, or you desk had sixteen times the surface area.

The increased disk and RAM speeds are much less impressive. This suggests that algorithms, programs, and systems ought to be rebalanced to effectively use the new hardware.

In part, this paper was prompted by the question: how did Linux spend this increased capacity?

3 Installing RHL5.1 and CentOS5.1

To compare the distributions, I installed them both on the same computer.

The disparity in the hardware requirements made finding a computer that would support both a bit of a challenge. For example, only old video controllers are supported by RHL5.1; CentOS5.1 will only run on machines with perhaps 256M or more RAM (the graphical install requires 512M).

Just to see what would happen, I booted the RHL5.1 installation disk on my new HP Pavilion A6245n. It was quite confused by the 320G hard drive (fdisk, the kernel, and Disk Druid had varying wrong opinions of its size, based on various geometry lies) and about the 6G of RAM (it recognized only 64M). It saw one of the four CPU cores. Still, I expect RHL5.1 could have been installed.

For the actual installation, I chose a Compaq EN SFF box manufactured in 1999 April. I stuffed it with 320M of RAM and 120G of hard disk (it was probably originally shipped with 64M of RAM and a 6.4G hard drive). I expect that very few machines old enough to run RHL5.1 were initially assembled with enough RAM to install CentOS5.1.

The machine has no CD or DVD drive. Installation was through the network. In RHL5.1, the installation boot floppy can be told to find the installation tree via FTP, HTTP, or NFS. In CentOS5.1 the kernel has outgrown floppies so PXE netbooting was necessary for bootstrapping the installation.

RHL5.1 uses the LILO bootloader and this version does not use the extended int 13 features of modern BIOSes to access large disks. It could only access content on the first 1023 notional cylinders of the hard drive. So most of the drive was out of reach.

One approach to this problem is to create a separate `/boot` partition that is within the first 1023 cylinders. It appears as if RHL5.1 was not set up to support this. I did manage to accomplish this but there were a few odd failures that had to be dealt with. In the end, I used CentOS5.1's Grub to boot RHL5.1.

Even with the LILO problem dealt with, RHL5.1 seemed to only be able to use CHS mode to address the disk and thus was limited to the first 8.5G of the disk. After installation and updates, it seems to be able to use LBA addressing (thus supporting disks up to 137G). Making my way through a twisty maze of `fdisk` and `hdparm` seemed unrewarding so I did not resolve all of these mysteries.

RHL5.1 and CentOS5.1 cannot share swap partitions. RHL5.1 uses an older form of swapfile that is limited to 127M. From the standpoint of 2008, that limit is hard to believe.

To install CentOS5.1, I had to set up a PXE booting environment, something that I had never done before. This was made slightly more difficult by the fact that the documented technique for configuring CentOS5.1 as a boot server is to use the system-config-netboot package which turns out not to exist.

CentOS5.1 installed quite uneventfully, if slowly. The subsequent update process took an unreasonably long time. This seems to be a well-known problem even on current machines.

Lessons learned:

- It is possible to find hardware supported by distributions separated by a decade.
- Grub is a lot friendlier than LILO.
- The historical path of increasing disk size is littered with awkward limitations.

4 Experience with RHL5.1

In order to get current experience with RHL5.1, I attempted to use it to prepare this paper. This does not constitute a comprehensive survey but it was instructive.

Overall, I found using RHL5.1 was quite easy and effective. This depends on what the user is used to: someone habituated to current desktops would be much less comfortable. But even for me, the devil is in the details. What follows is a catalogue of issues.

RHL5.1 cannot be expected to support modern hardware. After all, the last changes to it were made in 1999 and they were just bug fixes. I used hardware from 1999 and found that worked.

The X desktop looks quite crude by current standards. It is based around FVWM. Looks don't matter very much. I didn't use the X desktop much, preferring to login from another desktop. That is mostly a reflection of the layout of my lab.

There is no SSH included in RHL5.1. I've grown very accustomed to its convenience and security so I missed it.

I tried to build a current version of OpenSSH on RHL5.1. I could have gone looking for a version of SSH's SSH (what I used in 1998) but I didn't really want to miss the years of bug fixes and other improvements.

I gave up on building OpenSSH because it demanded a newer version of Zlib and the addition of OpenSSL. It looked as if a cascade of backports would be required. This kind of barrier is probably typical when trying to backport current programs.

`rlogin(1)` worked. I hope that the security issues are not critical on my LAN. Unencrypted NFS is likely to be a juicier target.

JOVE is a text editor that I've used on UNIX-like systems for about 25 years. It has changed very little between the release of RHL5.1 and now. I built it on each system. CentOS5.1 was easy because the tarball includes a suitable `.spec` file for rpmbuild. For RHL5.1 a little work was required. The `.spec` file had comments that said how to change compile-time options to match RHL5.1 (mostly to do with POSIX conformance). One surprise was that RPM's macro processing seems to handle quoting differently—adjusting to that required an experimental approach.

The experience building JOVE would suggest that it isn't hard to make a program that can build in both environments. I don't think that this is accurate. JOVE had at least two advantages over most programs: it had been run on both systems before (albeit separated by many years), and its rate of change in that period has been very slow.

Building this paper using the OLS configuration did not work on RHL5.1. It failed with an unknown flag to `latex`: `-interaction=nonstopmode`. Even `xdvi` failed (missing fonts) on the `.dvi` file created by CentOS5.1. Being new to the LaTeX world, I decided not to attempt a work-around.[1]

The standard web browser is Netscape Communicator 4.08. Out of the box, the web pages I tried were blank or were missing a large part of their content (slashdot.org, google.ca). It turned out that turning off javascript helped considerably. The pages looked wrong or crude but the content was there. I had a look at some Gopher sites and they seemed fine. I would not like being limited to this browser these days.

In order to share files between the RHL5.1 and CentOS5.1 installations on the same machine, I tried to have each mount the others partition. CentOS5.1 could mount the RHL5.1 ext2 partition but RHL5.1 could not mount the CentOS5.1 ext3 partition, even though ext3, when properly unmounted, is supposed to be compatible with ext2. Mount's diagnostic was the infuriating "wrong fs type, bad option, bad superblock on `/dev/hda5` or too many mounted file systems." `dmesg(8)` showed the more specific `EXT2-FS:03:05: couldn't mount because of unsupported optional features`.

To solve the file sharing problem, I made a partition on another computer available via NFS. This worked well for both distributions.

5 Size of Programs

The two distributions share a lot of programs. How has their size changed?

I looked at all binary programs in `/bin`, `/usr/bin`, `/sbin`, and `/usr/sbin`. Symlinks were ignored but each hard link was counted. There were 1174 in RHL5.1 and 2413 in CentOS5.1. Of these, 655 were common to both (by name).

This attrition rate seems surprisingly high: 44% percent of the commands of RHL5.1 did not make it to CentOS5.1. A large number are probably explained by the fact that I did a "kitchen sink" installation of RHL5.1. Many of the programs that disappeared might have been short-lived marginal programs.

As reported by `size(1)`, the cumulative text space used by programs that were common to the two distributions has gone up by a factor of 2.7. Similarly, the size of data went up by 1.6 and BSS by 2.2.

Perhaps the programs found in `/bin` are in some sense more fundamental. Did they grow at a different rate? For programs found on both distributions and in `/bin` in either one of them, I find similar figures: a factor of 2.6 for text, 3.0 for data, and 1.4 for BSS.

I was surprised to find that for programs found in `/sbin` in either distribution, the growth was much

[1] *Ed. Note*: Workarounds would have failed due to requirements on a newer `geometry.sty` and other packages. *—Formatting Team*

higher: a factor of 5.3 for text, 2.6 for data, and 5.3 for BSS.

`bash(1)` is an important program, so it is worth looking at by itself. Text has grown by a factor of 2.1, data by 1.04, and BSS by 3.08. These figures are consistent with our cumulative ones.

I installed JOVE on both distributions. The text grew by a factor of 1.13; data and bss changed insignificantly. This was true whether the CentOS5.1 installation exploited the new POSIX capabilities or was configured identically to the RHL5.1 version.

This table shows programs whose text size shrank or grew by a factor larger than 10. `rmt` is included twice because RHL5.1 has two different versions.

Program	RHL	CentOS	Factor
gs	646943	3928	0.00607163
python	267795	2024	0.00755802
perl	419638	10186	0.0242733
symlinks	88660	6199	0.0699188
chroot	1377	14039	10.1954
tac	7802	82939	10.6305
repquota	5392	61268	11.3628
smbd	323322	4126046	12.7614
automount	14108	204301	14.4812
usleep	1495	22652	15.1518
mailstats	4113	63414	15.4179
warnquota	4347	70036	16.1113
smbpasswd	131262	2256936	17.1941
quotaon	3506	64400	18.3685
restore	53855	1040290	19.3165
praliases	2804	78458	27.9807
dump	33775	1101649	32.6173
rmt	4841	465708	96.2008
rmt	4296	465708	108.405
makedb	6136	815640	132.927

Each program that shrank did so because code moved to dynamic libraries and hence was not counted. In the case of `symlinks(8)`, the RHL5.1 version was statically linked for some reason.

Excluding these programs made only a modest change to the ratios: the text factor became 2.4, the data factor 1.7, and the BSS factor 2.1.

Almost every program uses libc. It has grown by a factor of two:

```
   text    data     bss
 591554   25728   48964
1282529   10072   11352
```

It seems as if there is a real expansion in the size of binaries but it is quite modest compared with the concurrent growth in hardware capacity.

As a point of comparison, I applied the same scripts to compare binary commands on CentOS5.1 i386 and x86_64. Of course there were many more commands in common. The cumulative text size went up by a factor of 1.24, the data size went up by a factor of 1.62, and the bss size went up by 1.07. I was surprised that the text of `/usb/bin/mbchk` was 139 times larger on x86_64. On the other hand `gedit` shrank by a factor of .40. In both cases the package versions were the same.

The RHL5.1 CD contains 528 packages taking up 298568 blocks. The CentOS5.1 DVD contains 2401 packages taking up 3570804 blocks. That is 4.5 times as many packages and 12 times as many blocks.

6 Functionality: the Qualitative Difference

Not only has hardware capability increased over the ten years, but open source developers have been working hard to exploit it. Here's a subjective list of important additions:

- Desktop integration, primarily GNOME and KDE. Or choose your own.
- Open Office
- support for a large portion of the proliferation of I/O devices and ways of connecting them (USB, FireWire, SATA, ...).
- scalable support for multiple processors
- scalable support for large memories and disk drives
- support for new architectures (although processor diversity on the desktop has gone down)
- complex and powerful tools for building internet services
- support for various media such as video (seriously constrained by patents). Official MP3 support has been dropped.

- Asterisk for telephony (not part of CentOS5.1, but available)
- MythTV for PVR replacement and much more (not part of CentOS5.1 but available)
- significant shift to higher level but less efficient languages such as Perl, Python, and Ruby.
- improved support for UNICODE (which itself has improved). But mention of support for Klingon has been dropped from the `unicode(7)` manpage.

7 Security or Don't try this at home

Best practices for security have changed quite a bit since RHL5.1 was released. ssh has replaced rlogin. Firewalls can be configured during installation. Servers are generally not installed listening to the internet. A system has evolved to publish security holes and patches for them in a timely fashion.

RHL5.1 did have `ipfwadm(8)` for implementing a firewall, but no canned configuration or easy-to-use configuration tool. Building a firewall out of this involved fairly arcane knowledge.

`rlogin(1)` was "kerberized" so its authentication was reasonable. It doesn't seem to pay attention to `~/.rhosts` by default. But the traffic is still passed in the clear.

Wietse Venema's TCP Wrappers is included and used.

There have been almost ten years of work discovering holes in the software without any patches for RHL5.1 (of course this isn't negligent: the intended fix is to move to a newer release). Maybe the holes are so obsolete that current attackers don't know of them or think of using them. But I would not count on that.

8 Bit Rot

My RHL5.1 disks were commercially pressed by Red Hat, Inc. They still work well. But when I went back in my archives to find the errata for RHL5.1, I found that a number of my burned CDRs had become damaged.

I stored the CDRs in paper envelopes with plastic windows. This was much more compact than storing them in jewel cases. Unfortunately the plastic widows deteriorated and began to stick to the label side of the disks. When I tried to remove the debris from a disk, it seemed as if the disk partially delaminated. I've put off attempted recovery until another day.

On the other hand, I have not been able to find `.iso` images for RHL5.1 on the internet. The errata are still available, but have been relocated. This is explained in a note on `http://www.redhat.com/security/updates/eol/`, at least for now.

It is unlikely that a store would stock a box of RHL5.1. Software does not seem to be like books in this regard. In fact, the legal regime for most commercial software makes used software stores legally suspect.

All this may seem inconsequential. But the problem is only going to get worse as time passes and yet there might be more interest in RHL5.1 in some years. This seems to be how antiques become so valuable: the objects must pass through a valley of interest during which most are lost, broken, or discarded. Mundane objects are the most subject to this attrition.

I touched on other types of bit-rot earlier: the difficulty in porting code back, dealing with the cumulative gradual (and not so gradual) changes to libraries and other requisites; the difficulty in running old operating systems on new hardware. At some point, genetic drift is sufficient for the systems to be considered different species.

The cure for bit rot is constant maintenance. It isn't clear who would find it worthwhile to perform this maintenance on RHL5.1.

9 The Structure of Growth

The subject of how systems grow is deep and interesting. You will have to look elsewhere for a thorough treatment. I commend Stewart Brand's book [3] on how buildings change as one place to look.

The skeleton of Linux is traditional UNIX [4], as elaborated by POSIX standards [2] [1]. Like a skeleton, these parts don't change very quickly. Most critically, little is removed from their interfaces since they are the bedrock on which other parts of the system are built.

This means that most programs from the RHL5.1 era should be easily moved to CentOS5.1.

One exception is that several programming language implementations have become more restrictive about what a proper program is. For example, many C programs need to be cleaned up to compile on current systems.

Some of the recent additions to Linux may turn out to be as important and (one hopes) long-lived and stable. HAL is an example. Various object models seem as important but not as convincingly right.

I expect that the vast majority of new packages in CentOS5.1 are not skeletal, and that is a good thing. That means they may well come and go without seriously disrupting other packages. This hypothesis should be investigated.

10 Observations

Since the code base for RHL5.1 has suffered serious bit rot, it seems unlikely that there are remaining practical applications for it.

The fact that RHL5.1 is strongly related to CentOS5.1, and yet so much smaller, suggests that it might be possible to subset the CentOS5.1 codebase to produce a modern lightweight system. A key advantage would be that the burden of maintaining the packages would be widely shared.

I would imagine that this approach would fit better with a project like Debian because it is directed by a diverse community of developers and not one coherent corporate strategy.

Understanding the trajectory from RHL5.1 to CentOS5.1 may help us prepare for the trajectory of the next ten years.

One thing that I have not observed is significant removal of complexity. It would be wonderful if that were possible, but it seems to violate some law of software thermodynamics. It seems to require starting over, and that seems too expensive for most situations.

References

[1] American National Standards Institute. *IEEE standard for information technology: Portable Operating System Interface (POSIX) : part 2, shell and utilities.* IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, September 1993.

[2] American National Standards Institute. *IEEE standard for information technology: Portable Operating Sytem Interface (POSIX). Part 1, system application program interface (API) — amendment 1 — realtime extension [C language].* IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1994.

[3] Stewart Brand. *How Buildings Learn: What Happens After They're Built.* Penguin, 1995.

[4] Dennis W. Ritchie and Ken Thompson. The UNIX time-sharing system. *Communications of the Association for Computing Machinery*, 17(7):365–375, July 1974.

Measuring DCCP for Linux against TCP and UDP With Wireless Mobile Devices

Leandro Melo de Sales, Hyggo Oliveira, Angelo Perkusich
Embedded Systems and Pervasive Computing Lab
`{leandro,hyggo,perkusic}@embedded.ufcg.edu.br`

Arnaldo Carvalho de Melo
Red Hat, Inc.
`acme@redhat.com`

Abstract

Multimedia applications are very popular in the Internet. The use of UDP in most of them may result in network collapse due to the lack of congestion control. To solve this problem, a promising protocol is DCCP. DCCP[1] is a new protocol to deliver multimedia congestion-controlled unreliable datagrams.

This paper presents experimental results for DCCP in the Linux kernel while competing with TCP and UDP. DCCP behaves better than UDP, while it is fair with respect to TCP. The goal in this work is to help developers choose the proper protocol to use, as well as disseminate the DCCP Linux project. It was used with four Nokia N800s, three WLAN access points, and one router to emulate congestion. Some parameters were evaluated: throughput, loss/delay, and effects of handoffs performed by mobile hosts.

1 Introduction

With the rapid growth in popularity of wireless data services and the increasing demand for wireless connectivity, Wireless Local Area Networks (WLANs) have become more widespread and are making their way into commercial and public areas. They are available in almost everywhere including business, office and home deployments. WLANs based on the IEEE 802.11 standards enjoy high popularity due to setup simplicity, increased deployment flexibility, unlicensed frequency band, low cost and connectivity with minimal infrastructure changes. Lately, the need for Real Time (RT) multimedia services over WLANs have been dramatically increased, including Voice over IP (VoIP), audio/video (AV) streaming, Internet video conference, IPTV, entertainment and gaming, and so forth.

[1]Datagram Congestion Control Protocol

In this scenario, companies are adopting this technology to easily connect devices and offer new mobile services. The main reasons for this growth are:

1. the improvements on the quality of wireless transmissions;
2. the provision of security mechanisms to safely transmit application data, thus increasing the number of available services;
3. users can walk and still have their devices connected—this can contribute to the new era of mobile services;
4. efficient and seamless connection to a wireless network, reducing the time for network setup and the necessity of any kind of cable;
5. the increasing number of low-cost mobile devices—allowing home users to have access to the world of wireless Internet access; and
6. everytime/everywhere computing, enabling Internet access in public spaces.

Based on this visible growth, multimedia applications receive special attention due to the popularization of high-speed residential Internet access and wireless connections, considering also new standards such as IEEE 802.16 (WiMax). This enables network applications that transmit and receive multimedia contents through the Internet to become feasible once developers and industry invest money and software development efforts in this area. They are developing specialized multimedia applications based on technologies such as Voice over IP (e.g., Skype, GoogleTalk, Gizmo), Internet Radio (e.g., SHOUTcast, Rhapsody), online games (e.g., Half Life, World of Warcraft), video conferencing, and others; these have also become popular in the context of mobile computing. These applications offer sophisticated solutions that can approximate a face-to-face dialog for people, although they can be physically separated by hundreds or thousands of kilometers in distance.

There are at least three reasons for the growth of the popularity on the usage of mobile multimedia applications. First, the availability of new development libraries for mobile multimedia applications focusing on data processing optimizations. Second, the availability of smaller mobile devices with higher processing power and data storage capacities. And third, the necessity of people to communicate considering cost and benefits. For instance, VoIP applications can, at least, halve the original costs of a voice call when compared to traditional means.

For these types of applications, non-functional requirements such as end-to-end delay (latency) and the variation of the delay (jitter) must be taken into account. Usually multimedia applications use TCP and UDP as their transport protocol, but they may present many drawbacks regarding these non-functional requirements, and hence decrease the quality of the multimedia content being transmitted. In order to deal with those types of requirements, IETF standardized the Datagram Congestion Control Protocol (DCCP) [4], which appears as an alternative to transport congestion controlled flows of multimedia data, mainly for those applications focusing on the Internet.

In this article we present the results of an experimental evaluation using TCP, UDP, and DCCP to transmit multimedia data over a test bed 802.11g wireless network, considering wireless mobile scenarios. In these scenarios, several parameters were evaluated, such as throughput, packet loss, jitter, and the execution of hand-off. Hand-off is a process of transferring a wireless connection in progress from one access point to another without interrupting the data transmission.

The remainder of this article is organized as follows: in Section 2, an overview of some characteristics available in the DCCP protocol is presented. In Section 3, the methods used to evaluate the experiments are explained. Results of the experiments are discussed in Section 4. Finally, we present conclusions and future works in Section 5.

2 Overview and Background

DCCP [4] was first introduced by Kohler *et al.* in July, 2001, at the IETF transport group. It provides specific features designed to fulfill the gap between TCP and UDP protocols for multimedia application requirements. It provides a connection-oriented transport layer for congestion-controlled but unreliable data transmission. In addition, DCCP provides a framework that enables addition of a new congestion control mechanism, which may be used and specified during the connection handshake, or even negotiated in an already established connection. DCCP also provides a mechanism to get connection statistics, which contain useful information about packet loss, a congestion control mechanism with Explicit Congestion Notification (ECN) support, and Path Maximum Transmission Unit (PMTU) discovery.

From TCP, DCCP implements the connection-oriented and congestion-controlled features, and from UDP, DCCP provides an unreliable data transmission. The main reasons to specify a connection-oriented protocol is to facilitate the implementation of congestion control algorithms and enable firewall traversal, a UDP limitation that motivated network researchers to specify the STUN [10] (Simple Traversal of UDP through NATs (Network Address Translation)). STUN is a mechanism that helps UDP applications to work over firewalled networks. An important feature of DCCP is the modular congestion control framework. The congestion control framework was designed to allow extending the congestion control mechanism, as well as to load and unload new congestion control algorithms based on the application requirements. All of these operations can be performed before the connection setup or during an already-established connection through the feature negotiation mechanism [4]. Each congestion control algorithm has an identifier called Congestion Control Identifier (CCID).

Considering motivations to design a new protocol, one of them is the way in which TCP provides congestion control and reliable data transfer. When loss of packets occurs, TCP decreases its transmission rate and increases the transmission rate again when it successfully sends data packets. To implement a reliable data transfer, when TCP losses packets, it retransmits them. In this case, new data generated by the application is queued until all lost packets have been sent. Because of this way of implementing reliable data transfer, using TCP may lead to a high level of flow delay. As a consequence, the user may experience interruptions in the multimedia content being transmitted. In addition, the TCP congestion control mechanism limits the transmission rate for a given connection. This means that TCP is fair with respect to other TCP flows and can be fair with other congestion controlled flows, such as those transmitted by DCCP. These characteristics of TCP make it proper for those applications that require reliable data transfers, such as web browsers, instant messengers, e-mail, file sharing, and so forth.

On the other hand, UDP is a very simple protocol working on top of the best-effort IP protocol, implementing minimal functions to transport data from one computer to another. It provides a connectionless service and it does not care about data packets' delivery, nor about network congestion control. In addition, it does not provide packet reordering on the receiver end, if taking into account the original ordering of packets transmitted by the sender. Due to the lack of any type of congestion control, UDP may lead to a network congestion collapse, where TCP-based applications may also become unusable. Hence, a UDP application can send data as much as it can, but much of that data may be lost or discarded by the routers due to network congestion. Some examples of UDP applications are VoIP applications, videoconferencing, and Internet radio.

When developing multimedia applications using TCP as the transport protocol, end users may experience high streaming delays due to high packet retransmission rates caused by network congestion. On the other hand, the use of UDP may lead to a network collapse or bad streaming quality, since UDP does not provide any kind of congestion control. The new option is DCCP, which combines the good features of each protocol to provide better quality for multimedia data streaming, as well as to share network bandwidth with TCP.

2.1 DCCP Congestion Control Identifiers

Nowadays, DCCP provides two CCIDs already standardized: the TCP-Like Congestion Control (or CCID-2) [5] and the TCP-Friendly Rate Control (or CCID-3) [6]. The goal behind this feature is to provide a way to control the flow of packets according to the type of data being transmitted. A CCID may be used at any time of a DCCP connection, and it is possible to have one CCID running in one direction, and other in the opposite direction. The flexibility on the CCID usage is important because the transmitted multimedia flow may present different characteristics. For example, a VoIP flow is characterized by a burst of small packets—when one interlocutor says something—between periods of silence—when this interlocutor stops talking and waits the other peer to talk. Another example is the Video-on-Demand traffic characteristic, which is smoothly and generally based on a Constant Bit Rate (CBR).

Thus, considering different types of multimedia applications, DCCP designers defined the congestion control framework for supporting the addition of new congestion control algorithms, as well as the deletion of them regardless of the core of the protocol. In addition to the initial standardized CCIDs, the DCCP IETF is specifying the CCID-4 [7], which is a new congestion control algorithm for DCCP to be used by applications that transmit small packets of data in a short period, such as VoIP applications.

TCP-Like Congestion Control

CCID-2 [5] is based on window flow control and resembles TCP congestion control. When a host receives a DCCP packet, it sends an ACK back to the sender. After receiving that packet, the sender adjusts the window size and the expiration time. The CCID-2 algorithm is based on the AIMD [1] algorithm for window-based flow control. Similarly TCP, the window size used in the algorithm is given as the congestion window size (`cwsize`), which is equal to the maximum number of in-transit packets allowed in the network at any time.

The sending host itself adjusts `cwsize` through congestion estimation according to the sequence of the ACK packet received. In this way, the `cwsize` is increased by one packet in the following cases:

1. every acknowledged packet arrives in a slow-start phase, and
2. every window of data is acknowledged without lost packets in a congestion-avoidance phase.

On the other hand, the `cwsize` is halved when the sender can infer that loss of packets occurs due to duplicate acknowledgments, which is equivalent to TCP.

If an ACK packet does not arrive at the sender before the timeout timer expires (i.e., when an entire window of packets is lost), the sender sets `cwsize` to one. The CCID-2 is proper for applications that want to use as much bandwidth as possible and are able to adapt to sudden changes in the available bandwidth [2, 8].

TCP-Friendly Rate Control TFRC

The CCID-3 [5] implements a receiver-based congestion control algorithm where the sender is rate-limited by packets sent by the receiver with information such as receive rate, loss intervals, and the time packets are kept in queues before being acknowledged. This CCID is intended for applications that smoothly support rate changes. Since the changes are not abrupt, it responds more slowly than TCP or TCP-like congestion controls.

The transmission rate is changed by varying the number of packets sent and is not suitable for applications that prefer variation in the sending rate by changing the packet size. In the CCID-3 implementation, the sending rate is computed by analyzing the loss event rate based on a throughput equation named TFRC Equation [5]. It supports Explicit Congestion Notification (ECN) and, to verify whether the receiver reported an accurate loss event, it also reports the ECN Nonce Sum [5] for all packets reported as received.

2.2 Summary of TCP, UDP and DCCP features

According to Table 1, which shows a comparison between the features of TCP, UDP, and DCCP, one may observe that DCCP is different from TCP in four points that are highlighted in bold. The first of them is the size of the header of each packet, which varies depending on the value of the X field presented in the header. The X field represents the Extended Sequence Number. If it is equals 0, the length of the packet is 12 bytes; if X is equal to 1, the length of the packet is 16 bytes. The second item is conceptual: while TCP sends segments, DCCP sends datagrams. The third difference between TCP and DCCP is that DCCP does not guarantee the delivery of data transmitted, except when the data transmitted is related to a feature negotiation provided by DCCP. The last difference is that DCCP does not guarantee packet reordering, even though it uses a sequence number in the packet header.

3 Methods and Experiments

In this section we describe two scenarios used to perform the experiments using the DCCP protocol, presenting the parameters and methods adopted to obtain the data for each metric collected during the experiments, such as instantaneous throughput and latency. We use a statistical method based on the probability theory [3] to calculate how many times it is necessary to repeat a given experiment to obtain an acceptable confidence level for each collected metric. In this work, it was considered 95% for the confidence level. By using this mechanism, it is possible to compare each protocol in terms of its respective performance while competing with each other.

The network topology used to execute the experiments was an 802.11g wireless network composed of both computers and Internet Tablets, in this case, Nokia N800. The experiments also examined the execution of hand-offs, where the internet tablets performed hand-offs at the link level of the 802.11g wireless network during data transmission. After explaining the general considerations adopted in the experiments, the network topology is presented.

3.1 General Considerations

The DCCP implementation used to run the experiments is available in the Linux kernel version 2.6.25, which can be obtained from the DCCP development `git` tree. Because the version of the Linux kernel for the Internet Tablets was 2.6.21, we backported the DCCP implementation from Linux kernel version 2.6.25 to version 2.6.21. Therefore, all the devices used in the experiments had the same DCCP implementation.

To generate TCP, UDP, and DCCP data flows, IPerf was used; it provides statistical reports about the connection during data transmissions. This includes statistics about packets lost and received, jitter, and throughput for a given instant. We defined the scenarios of experiments, which mean specifying values for IPerf parameters and the devices to be used in each experiment considering confronts between two given protocols (TCP $\times$ UDP, TCP $\times$ DCCP, and UDP $\times$ DCCP). Experiments with packet sizes of 512 bytes and 1424 bytes were performed. In this case, the idea was to verify whether varying the packet size would produce any impact on the protocol performance, since varying the packet size

Table 1: Comparison of TCP, UDP and DCCP features

Feature	UDP	TCP	DCCP
Packet size	8 bytes	20 bytes	**12 or 16 bytes**
Transport layer packet entity	Datagram	Segment	**Datagram**
Port numbering	Yes	Yes	Yes
Error detection	Optional	Yes	Yes
Reliability: Error recovery by ARQ	No	Yes	**No**
Sequence numbering and reordering	No	Yes	Yes/**No**
Flow control	No	Yes	Yes
Congestion Control	No	Yes	Yes
ECN support	No	Yes	Yes

during the transmission may lead to fragmentation of packets in the IP layer. If this is the case, it may affect the multimedia data quality being transmitted. Varying the packet size during the multimedia data streaming is one of the well-known techniques adopted by multimedia applications to adapt the quality of the flow in response to network congestion.

Regarding congestion control algorithms for TCP and DCCP, Reno, Cubic, and Veno were used for TCP; and for DCCP, CCID-2 and CCID-3 were used. The device used in the experiments was the Nokia N800, with an ARM 330 *MHz* processor, 128 *MB* and a Texas Instruments wireless network interface.

3.2 Network Topology

The goal was to study the performance of TCP, UDP, and DCCP when running on resource-limited devices, considering processor and memory capacities. An important point was to analyze the behavior of the protocols studied when the user application performs hand-offs between two consecutive 802.11g wireless access points. In this case, two Internet Tablets working as clients performed hand-offs, while the other two worked as servers and did not perform hand-offs. The execution time for each of the experiments was 300 *s*, where the hand-offs were performed in 100 *s* and 200 *s*.

The network topology defined for this scenario is shown in Figure 1. In this case we used three Internet Tablets to transmit two UDP or DCCP flows—each flow in one Internet Tablet—and the third one was used to transmit one TCP flow. In practice, this means two multimedia flows using UDP or DCCP (audio and video) against a data-oriented application such as HTTP.

3.3 Parameters for the Experiments

Four parameters were considered for the experiments: protocol confront; packet size; congestion control algorithm; and the existence of hand-offs. As discussed before, to transmit data in any scenario of a given experiment, the protocols were combined between them two-by-two. In Table 2, the quantity of flows transmitted during the experiments for each protocol is shown, according to each confront. The goal to define these confronts is to analyze the fairness among the three protocols in terms of network bandwidth usage.

For each scenario, the packet size was varied. For each confront of the protocols, the experiments were performed with different packet sizes, either 512 *bytes* or 1424 *bytes*. The goal for variation is to analyze whether the transmitted packet size impacts the performance of the studied protocols. As discussed before, some applications perform adaptation in the quality of the flow being transmitted as a response to the network congestion. To perform this task, codecs that support such a feature are called VBR, which varies the bit rate for each generated packet—or for a small set of continuous packets—according to the multimedia content being transmitted, which dynamically changes the packet size during data transmission.

Varying the congestion control algorithm allows the performance analysis of each congestion control algorithm during data transmission. Besides, fairness with respect to the network bandwidth usage can be evaluated. Also, by varying this parameter, is possible to study the behavior of each protocol when network congestion occurs.

The last parameter taken into account is the existence or

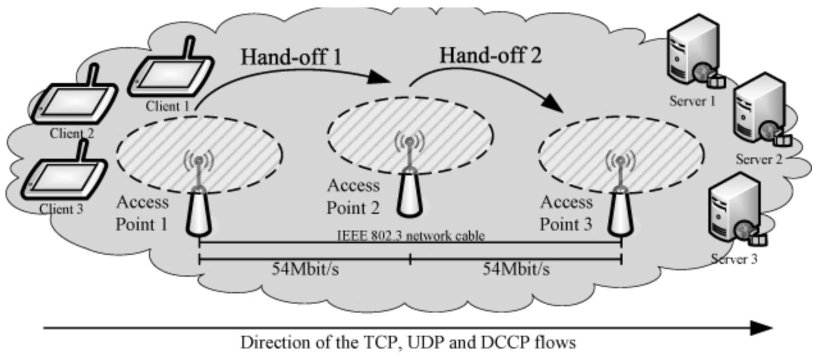


Figure 1: Network topology for the experiments.

#	Confronts	TCP flow	UDP flows	DCCP flows
1	**TCP × UDP**	1	2	0
2	**TCP × DCCP**	1	0	2
3	**UDP × DCCP**	0	2	1

Table 2: Number of flows used in each protocols confronts

not of hand-offs. Two of the four Internet Tablets performed hand-offs. It is known that during hand-off there are packet losses, but some congestion control algorithms mistake these losses as congestion, such as TCP Reno and the DCCP CCID-2. When a packet is lost, these algorithms assume congestion on the network and wrongly react by decreasing the allowed sending rate of the connection, but theses losses are temporary—they only occur during the hand-off. The goal in this case is to study the behavior of the congestion control algorithms in the existence of hand-off.

3.4 Collected Metrics and Derived Metrics

For all executed experiments, a TCP flow was first transmitted, and after 20 *s*, the other flows were started. By executing the experiments in this way, it was possible to evaluate how fair the protocols are with each other when new flows are introduced in the network. This enables analyzing whether any of them can impact the performance of the other—mainly whether DCCP and UDP impact the performance of TCP. In addition, the idea is to look for the most suitable TCP and DCCP congestion control algorithms to transmit multimedia data over the network.

To reach all of these goals, a set of metric values was collected for the flows transmitted during the experiments. The metrics were the throughput, packet loss, and latency. Considering these metrics, it is possible to obtain other two metrics: jitter and the rate of how many packets reached the receiver, which can be obtained from the quantity of transmitted packets. Through latency, it is possible to calculate jitter for a given instant; from throughput and the quantity of lost packets, it is possible to obtain the effective amount of data transmitted—how much data effectively reached the receiver.

3.5 Obtaining Throughput, Jitter and the Amount of Data Lost and Transmitted

The mean throughput and the amount of data transmitted for TCP was obtained through the average of the means in each repetition ***r*** of a given experiment. This is shown in Equations 1 and 2, where ***n*** is the total of repetitions.

$$\mu_{thoughput\_tcp} = \frac{\sum_{r=1}^{n} throughput\_mean_r}{n} \tag{1}$$

$$\mu_{load\_tcp} = \frac{\sum_{r=1}^{n} load\_mean_r}{n} \tag{2}$$

However, to obtain the means for the UDP and DCCP throughput and amount of data transmitted, the procedure was different. Considering that two UDP/DCCP flows have been transmitted—taken regardless—against a TCP flow, and considering also that the UDP/DCCP flows started only 20 s after the TCP flow started, it was necessary to define a mechanism that does not penalize both protocols in a confront. In this case the calculation of the means would not be an arithmetic average of the sum of the throughput and the amount of data transmitted by the two UDP/DCCP flows. Instead, it should be the mean throughput and the mean amount of data transmitted of each flow. This observation is represented in Equation 3, where each *thoughput_mean$_r$* of this equation can be obtained from Equation 4.

$$\mu_{partial\_throughput(udp/dccp)} = \frac{\sum_{r=1}^{n} thoughput\_mean_r}{n} \quad (3)$$

$$thoughput\_mean_r = \frac{\sum_{k=1}^{F} thoughput\_mean\_flow_k}{F} \quad (4)$$

Based on the same assumptions presented before, in Equation 4, the term *thoughput_mean_flow$_k$* is obtained through the arithmetic means of the throughput in each instant (per second) of the experiment. Therefore, the final value for the throughput for a given transmitted flow (connection) of UDP and DCCP can be obtained from Equation 5.

$$\begin{aligned}\mu_{final\_throughput(udp/dccp)} = &\mu_{partial\_throughput(udp/dccp)} + \\ &S \times \left(\frac{\mu_{partial\_throughput(udp/dccp)}}{T}\right)\end{aligned} \quad (5)$$

where F is the number of flows, $F_{UDP} = F_{DCCP} = 2$ for TCP $\times$ UDP/DCCP and $F_{DCCP} = 1$ for UDP $\times$ DCCP; S, is the await time to start the UDP or DCCP flows ($S = 20s$); and T, is the total time of the experiments ($T = 100\,s$ without hand-offs or $T = 300\,s$ with hand-off).

The means were normalized according to Equation 5, to avoid penalizing the protocols in the terms discussed before.

In a similar way the latency and effective amount of data transmitted can be obtained. Note that for UDP $\times$ DCCP confronts, the term F_{DCCP} is equal to 1. In these cases the throughput and amount of data transmitted are obtained through Equations 1 and 2, respectively.

Jitter

The calculation to obtain the mean *jitter* for a transmitted flow is very similar to the calculation of the mean throughput. The value for the jitter can be obtained through Equation 8 and it can be obtained as follows:

$$\mu_{partial\_jitter(udp/dccp)} = \frac{\sum_{r=1}^{n} jitter\_mean_r}{n} \quad (6)$$

and,

$$jitter\_mean_r = \frac{\sum_{k=1}^{F}\left(\frac{\sum_{k=1}^{QI} VA_k}{QI}\right)}{F} \quad (7)$$

then,

$$\begin{aligned}\mu_{final\_jitter(udp/dccp)} = &\mu_{partial\_jitter(udp/dccp)} + \\ &S \times \left(\frac{\mu_{partial\_jitter(udp/dccp)}}{T}\right)\end{aligned} \quad (8)$$

where: F is the number of flows used in the experiments, $F_{UDP} = F_{DCCP} = 2$ for TCP $\times$ UDP/DCCP and $F_{DCCP} = 1$ for UDP $\times$ DCCP, QI, is the quantity of intervals ($QI = T - 1$) for two consecutives read of collected data, VA, is the variation of the delay between packets of the same flow, for instance $time_1 = 10\,ms$ and $time_2 = 11\,ms$, $VA = 1\,ms$, T, is the total time of the experiments ($T = 100\,s$ without hand-off or $T = 300\,s$ with hand-off).

3.6 Statistic Methodology for the Final Calculation of the Collected Metrics

The results presented in this work—for instance, to determine what protocol performed better than the other in terms of bandwidth usage—were based on samples of data collected while performing the experiments. The methodology adopted was based on the concepts of confidence interval [3], considering $\rho = 95\,\%$ (confidence level) and therefore $\alpha = 5\,\%$ (significance level, or error).

Determining the Confidence Interval for $\rho = 95\%$

The principle for the confidence interval is based on the fact that it is impossible to determine a perfect mean μ for a infinite population of N samples, considering a finite number n of samples $\{x_1, ..., x_n\}$. However, it is possible to determine in a probabilistic way an interval where μ will belong to this interval, with probability equals to ρ, and that will be not in this interval with probability of α.

To determine the minimum value c_1 and the maximum value c_2 for this interval, called a confidence interval, it is considered the probability $1 - \alpha$, where the μ value will belong to this interval, for n repetitions of a certain executed experiment. The Equation 9 summarizes this consideration.

$$Probability\{c_1 \leq \mu \leq c_2\} = 1 - \alpha \tag{9}$$

where (c_1, c_2) is the confidence interval; α is the significance level, expressed by a fraction and typically close to zero, for instance, 0.05 or 0.1; $(1 - \alpha)$ coefficient of confidence; and ρ = 100 * $(1 - \alpha)$, is the confidence level, traditionally expressed in percent and closer to 100 %; this work uses 95 %.

From the Central Limit Theorem[2] [3], if a set of samples $\{x_1, ..., x_n\}$ is independent, has a mean $\bar{x}$, and belongs to the same population N, with mean μ and standard deviation σ, then the average of the samples is in a normal distribution with $\bar{x} = \mu$ and standard deviation $\sigma/\sqrt{n}$,$\bar{x} \simeq N(\mu, \frac{\sigma}{\sqrt{n}})$.

Considering Relation 9 and the Central Limit Theorem, the confidence interval (c_1, c_2) for $\rho = 95\%$ and $\alpha = 0.05$ can be obtained as shown in Equation 10.

$$(\mu - z_{1-\alpha/2} \times \frac{s}{\sqrt{n}}, \mu + z_{1-\alpha/2} \times \frac{s}{\sqrt{n}}) \tag{10}$$

where μ is the average for n repetition; $z_{1-\alpha/2}$ is equal to 1.96, this value determines 95 % of confidence level; n is equal to the number of repetitions; and s is the standard deviation of the means for n repetitions.

[2]Central Limit Theorem: the sum of a large number of independent and identically-distributed random variables will be approximately normally distributed if the random variables have a finite variance.

Regarding the value for $z_{1-\alpha/2}$, also named quantile, is based on the Central Limit Theorem and since it is frequently used, it can be found in a table named *Quantile Unit of the Normal Distribution*. This table can be found in the reference [3], Table A.2 of Appendix A. Using the Relation 11, next it is explained how to determine the value 1.96 for the term $z_{1-\alpha/2}$.

$$z_{1-\alpha/2} = (1 - 0.05)/2 = 0.975 \tag{11}$$

According to the table *Quantile Unit of the Normal Distribution* available in reference [3], the corresponding value for the result of the Equation 11 is 1.96, which is the value to be used as the variable z of the the Equation 10.

Therefore, based on the confidence interval of each average for each metric collected during the experiments (see Section 3.5), it is possible to perform comparisons with these values for the defined scenarios of experiments for 95 % of confidence with 5 % of error.

Determining the Value for n to obtain $\rho = 95\%$

The confidence level depends on the quantity of samples n collected for a certain metric of a given experiment. Thus, the higher the value of n is, the more precise the confidence level will be. However, to obtain big samples requires more effort and time. Therefore, it is important to define a value for n and avoid repeating a specific experiment unnecessarily, but maintaining the desired confidence level $\rho = 95\%$.

To start the process of the experiment performed in this work, each experiment was repeated 3 times ($n_{base} = 3$). For example, the initial throughput mean of a given trasmitted flow was obtained from the means obtained by running the experiment 3 times. This means that firstly we obtain a high value for the variance, which is used to determine the real value for n to obtain 95 % of confidence level.

Based on Equation 10, the confidence interval for a given value of n samples is defined by Equation 12.

$$\mu \pm z \times \frac{s}{\sqrt{n}} \tag{12}$$

Thus, for the confidence level of $\rho = 95\,\%$ and $\alpha = 0.05$, the confidence interval is determined by Equation 13.

$$(\mu(1-0.05),\mu(1+0.05)) \tag{13}$$

Then, equating the confidence interval specified in Expression 13 with the confidence interval specified in Expression 12 (general), Equation 14 is obtained.

$$\mu \pm z \times \frac{s}{\sqrt{n}} = \mu(1 \pm 0.05) \tag{14}$$

Therefore, organizing the expression by isolating the variable n, each experiment was repeated n times determined in Equation 15, considering a confidence level $\rho = 95\,\%$, which implies in $z = 1.96$ (from Equation 11), and the 3 initial times of experiment repetition (n_{base}). For example, if the value for n is 12 for a given experiment, it was repeated $n = n - n_{base}$, which is equal to 9, and the three first means is also considered for the value of the final average of a given metric.

$$n = (\frac{1.96 \times s}{0.05 \times \mu})^2 \tag{15}$$

4 Results

Using the definitions and methods presented in Section 3, in this section the results and discussions about the experiments are presented according to methods discussed in Section 3.

The results are organized in two tables considering the packet size used in the transmission. The results for the evaluated metrics for transmissions using packets of size 512 bytes are presented in Table 3. The results for transmissions using packets of size 1424 bytes are presented in Table 4.

The values presented in these tables have a 95% confidence level with a 5% margin of error. The confidence interval is presented immediately below the value for the corresponding metric. For the UDP and DCCP protocols the confidence interval for the metric Transmitted / Lost corresponds to the effective load of transmitted data, that is, the subtraction of Transmitted−Lost. Also, consider that the values presented in the two tables correspond to the execution of the experiments following the process described in Section 3.1, considering that: the execution time is $300\,s$, the instants that the hand-off was performed were at $100\,s$ and $200\,s$, the confront among protocols were TCP × UDP, TCP × DCCP, and UDP × DCCP. Also, the congestion control algorithms for TCP were: Reno, Cubic and Veno, and for DCCP: CCID-2 and CCID-3. The metrics analyzed were throughput, the amount of transmitted and lost data, and latency/jitter.

4.1 Discussions about the Experiments

The major point considered in the experiments is related to:

1. the impact of changing the data packet size on the performance of the protocol during transmissions;
2. the impact caused in terms of throughput and the amount of data transmitted and lost when performing hand-off during data transmission; and
3. the behavior of TCP, UDP, and DCCP in terms of fairness.

For the first item, there were no considerable changes in the behavior of the transmitted flow for all protocols taken regardless, mainly related to the metrics throughput and jitter. But it is possible to observe changes in the performance of TCP Reno, Cubic, and Veno algorithms for TCP × UDP and TCP × DCCP.

For the TCP × UDP test, the amount of transmitted data using the algorithms TCP Reno, Cubic, and Veno were not satisfactory if the result of the experiments is divided into two groups: one with experiments using a packet size of 512 bytes (Table 3) and another with experiments using packet size of 1424 bytes (Table 4). If the throughput of the TCP and UDP protocols is almost the same for the two groups of experiments (see lines 1, 2, and 3), we expect to observe that the bigger packet size is, bigger the amount of transmitted data should be, considering that there is no packet fragmentation in the network layer, since the MTU for the 802.11g connection is 1500 bytes.

On the other hand, there are no changes for TCP × DCCP, regardless of the algorithm used. Comparing the throughput values for the TCP × UDP and TCP × DCCP confronts presented in Tables 3 and 4, a balance among these values can be observed. For instance, in Tables 3 and 4 the mean throughput of TCP Reno

#	Confronts	Throughput (Kbits/s)	Transmitted / Lost (KBytes)	*Jitter* (ms)	n
1	**TCP Reno × UDP**	3359,12 (3202,15 – 3516,09)	123006,72 (117258,69 – 128754,75)	2,11 (1,72 – 2,21)	11
		2956,09 (2935,81 – 2976,36)	394825,25/293967,67 (100164,37 – 101550,79)	1,60 (1,57 – 1,64)	
2	**TCP Cubic × UDP**	3364,12 (3232,15 – 3496,09)	121855,28 (118258,44 – 125452,12)	7,51 (6,32 – 8,7)	5
		2919,76 (2893,96 – 2945,57)	419803,17/320187 (98735,37 – 100496,97)	1,71 (1,69 – 1,74)	
3	**TCP Veno × UDP**	3121,69 (3042,26 – 3201,13)	114322,77 (111412,94 – 117232,60)	4,2 (3,4 – 5,0)	7
		3002,75 (2994,15 – 3011,36)	378833,83/276384,92 (102150,85 – 102746,97)	1,5 (1,50 – 1,54)	
4	**TCP Reno × DCCP-2**	3042,90 (2951,57 – 3134,23)	111433,62 (108090,21 – 114777,03)	4,8 (4,36 – 5,24)	9
		2162,51 (2138,56 – 2186,47)	73688,67/287,51 (73401,16 – 74234,14)	5,4 (5,02 – 5,75)	
5	**TCP Cubic × DCCP-2**	3862,58 (3775,82 – 3949,34)	104830,49 (101653,65 – 108007,32)	3,32 (3,11 – 3,53)	9
		2119,10 (2109,87 – 2128,33)	72256,17/367,92 (71888,25 – 72215,14)	4,2 (4,02 – 4,48)	
6	**TCP Veno × DCCP-2**	2395,97 (2289,27 – 2502,67)	87744,27 (83836,15 – 91652,39)	7,81 (7,33 – 8,29)	20
		2899,25 (2810,27 – 2988,24)	64653,08/314,42 (61289,36 – 67387,96)	6,1 (5,50 – 6,64)	
7	**TCP Reno × DCCP-3**	3291,67 (3265,40 – 3317,94)	120549,10 (119587,17 – 121511,03)	3,6 (3,42 – 3,78)	6
		2851,59 (2841,67 – 2861,52)	97234/1181,92 (95725,77 – 96378,39)	0,85 (0,83 – 0,87)	
8	**TCP Cubic × DCCP-3**	3598,81 (3496,79 – 3700,84)	131790,21 (128052,30 – 135528,13)	4,13 (3,77 – 4,49)	8
		2665,55 (2571,62 – 2759,48)	90895/1533,67 (86127,94 – 92594,72)	1,18 (0,95 – 1,41)	
9	**TCP Veno × DCCP-3**	3734,55 (3634,92 – 3834,18)	136765,27 (133115,41 – 140415,13)	2,31 (2,15 – 2,47)	11
		2824,84 (2815,48 – 2834,20)	96381,08/1472,58 (92753,94 – 97063,06)	0,89 (0,85 – 0,93)	
10	**DCCP-2 × UDP**	1792,20 (1749,55 – 1834,86)	65194,33/303,33 (62404,79 – 64891,03)	4,91 (4,75 – 5,08)	11
		2552,41 (2475,76 – 2629,06)	562465,08/475381 (84469,96 – 89698,20)	2,02 (1,87 – 2,18)	
11	**DCCP-3 × UDP**	2519,84 (2461,98 – 2577,70)	91559,83/1696,83 (85041,99 – 94684,01)	1,01 (0,91 – 1,12)	13
		2898,34 (2841,75 – 2954,93)	427356,58/328470,17 (96902,29 – 100870,53)	1,82 (1,73 – 1,92)	

Table 3: Summary for the results of phase 1 for $\rho = 95\,\%$. Packet of size $512\,bytes$, execution of hand-off and considering the confronts between two protocols among the protocols TCP, UDP and DCCP.

was 3359.12 Kbits/s and 3162.41 Kbits/s, respectively. Moreover, if the throughput is almost the same, the bigger the packet size is, more data the flow should transmit. This happened only for the TCP × DCCP. In this case, TCP transmitted more data when the packet size was 1424 bytes for all the congestion control algorithms used. This was expected because DCCP also implements congestion control and it allows TCP flow to transmit more data when increasing the packet size, considering that the maximum size for a packet is the MTU value minus the space occupied by the headers of protocols in the network, transport, and application layers.

Therefore, it is possible to conclude that in transmissions where the TCP and UDP protocols share the same communication channel, to increase the packet size

#	Confronts	Throughput (Kbits/s)	Transmitted / Lost (KBytes)	*Jitter* (ms)	n
1	**TCP Reno** × **UDP**	3162,41 (2968,61 – 3156,21)	112156,65 (108719,62 – 115593,67)	3,21 (3,09 – 3,33)	4
		5773,26 (5493,16 – 6053,36)	730078,67/659256,67 (67701,48 – 73942,52)	2,37 (2,28 – 2,47)	
2	**TCP Cubic** × **UDP**	3201,33 (3063,57 – 3339,09)	80614,73 (75572,17 – 85657,29)	4,51 (4,42 – 4,60)	6
		2575,34 (2544,25 – 2606,43)	1213656,83/1182062,33 (27053,38 – 36135,62)	3,17 (3,05 – 3,29)	
3	**TCP Veno** × **UDP**	3221,97 (3072,96 – 3370,99)	117998,49 (112542,56 – 123454,43)	5,12 (5,04 – 5,2)	8
		2301,01 (2288,97 – 2313,06)	682801,67/605503,92 (73433,54 – 81161,96)	2,27 (2,20 – 2,35)	
4	**TCP Reno** × **DCCP-2**	3776,63 (3717,94 – 3835,31)	166004,14 (163776,65 – 168231,63)	6,3 (6,21 – 6,34)	4
		3372,64 (3217,38 – 3527,90)	141343,33/375,92 (139722,68 – 142963,98)	6,68 (5,40 – 7,97)	
5	**TCP Cubic** × **DCCP-2**	3969,39 (3901,75 – 4037,04)	172124,00 (169648,29 – 174599,72)	4,12 (4,03 – 4,21)	3
		3611,59 (3578,69 – 3644,49)	187001,58/1303,25 (182247,13 – 189149,53)	5,16 (4,90 – 5,43)	
6	**TCP Veno** × **DCCP-2**	4561,88 (4218,12 – 4905,64)	180149,85 (178025,55 – 182274,15)	6,51 (5,81 – 7,21)	3
		2685,21 (2442,01 – 2928,41)	13600,533/1150 (133295,41 – 136415,25)	6,25 (5,47 – 7,02)	
7	**TCP Reno** × **DCCP-3**	2735,82 (2576,05 – 2895,60)	100189,53 (94338,37 – 106040,69)	3,71 (3,28 – 4,14)	7
		3469,30 (3299,87 – 3638,73)	118268,52/5685,50 (111483,27 – 113682,77)	2,80 (2,56 – 3,05)	
8	**TCP Cubic** × **DCCP-3**	2980,47 (2950,09 – 3010,85)	109147,54 (108034,15 – 110260,92)	4,1 (3,49 – 4,71)	5
		3482,36 (3319,60 – 3645,13)	118835,16/1549,83 (112236,44 – 122334,22)	2,63 (2,34 – 2,92)	
9	**TCP Veno** × **DCCP-3**	2998,39 (2867,97 – 3128,80)	109806,54 (105029,84 – 114583,24)	2,33 (2,21 – 2,45)	6
		4831,47 (4576,48 – 5086,45)	184765,16/4835,08 (175018,62 – 180930,08)	1,69 (1,65 – 1,73)	
10	**DCCP-2** × **UDP**	3452,93 (3315,24 – 3590,62)	125611,37/482,50 (122601,76 – 127655,98)	7,81 (7,55 – 8,07)	11
		5236,62 (4892,45 – 5580,79)	806485,67/742278,17 (62959,4 – 65455,6)	4,30 (3,70 – 4,90)	
11	**DCCP-3** × **UDP**	4053,65 (3904,76 – 4202,54)	147460,73/3415 (142043,27 – 146048,19)	1,84 (1,63 – 2,04)	13
		5935,79 (5884,42 – 5987,15)	699595,58/626779,58 (71211 – 74421)	2,46 (2,42 – 2,51)	

Table 4: Summary for the results of phase 1 for $\rho = 95\,\%$. Packet of size 1424 *bytes*, execution of hand-off and considering the confronts between two protocols among the protocols TCP, UDP and DCCP.

from 512 bytes to 1424 bytes does not lead to performance improvement, even when considering the Veno congestion control algorithm. This suggests that TCP lost more data when packets with size 1424 bytes (without considering the packet retransmission mechanism implemented by TCP to provide reliability) were used. In this case, a future work can evaluate what the best packet size should be to minimize the amount of TCP packets lost in this scenario. A discussion in this context is presented in [12].

Regarding the hand-off executions, in Figure 2 the progression of the transmission for TCP Cubic × UDP is depicted, which corresponds to line 2 of Table 3. In Figure 2(a), the mean throughput for the protocols TCP and UDP are presented. In Figure 2(b), the relation between the amount of transmitted and lost data for UDP in this transmission is shown. The values for each point plotted in the graph shown in Figure 2 were calculated as an average for the values of each point for all repetitions of the experiment, in this case $n = 5$ (last column of the line 2 of Table 3).

It is important to observe that in Figure 2(a) the throughput for the TCP connection decreased due to hand-off, where packet loss occurred; that is reflected in the congestion control algorithms of TCP and DCCP. In the case of UDP, the throughput remained constant during all the transmission time. In Figure 2(b), it is possible to observe a high level of data loss when using UDP, mainly during hand-off. Around second 35, it is possible to observe another declining point for the TCP throughput. This fact can be explained by the introduction of the two UDP flows, where packet loss also happened.

In Figure 3, the transmission for TCP-Cubic $\times$ DCCP is presented, which corresponds to line 5 of Table 4. The values for each point were calculated in a similar way of previous ones, with $n = 9$. This procedure was also used for the other graphs in this section.

As it can be seen in Figure 3(a), the TCP throughput also decreased due to the hand-off, for the same reasons as in TCP $\times$ UDP. In the case of the DCCP protocol, there was a small drop in the throughput during the hand-off. Figure 3(b) shows the evolution for TCP $\times$ DCCP, and it is possible to observe that DCCP lost a small amount of data when compared to UDP in the confronts against to TCP. In addition, it can be seen that during the transmission, the DCCP and TCP protocols shared the channel in a fair way.

Regarding the fairness of the protocols in terms of network bandwidth usage, a congestion in the network caused by UDP was expected, but this did not happen. Therefore, it is possible to conclude that there is data contention in the source, in this case the Internet Tablets (N800 devices). As the processing power of such devices is limited, there is a throughput limitation of data processing and transmission, considering that the process (at the operating system level) of the IPerf application used less CPU clocks compared to a desktop computer, for instance. In Section 5 a discussion on this subject is presented, where a different behavior is observed: the wireless network presented a high level of congestion caused by the UDP protocol and in some cases avoiding TCP and DCCP protocols to transmit data.

It is also important to comment on two additional facts observed in the experiments.

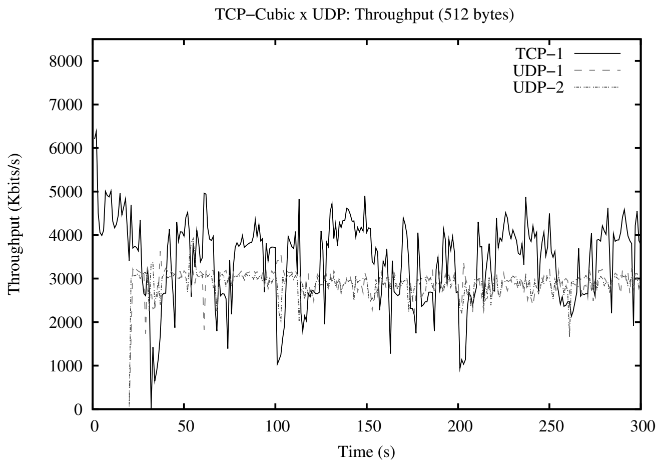
- Sudden wireless disconnections of the Internet Tablet were observed. This is probably associated with the processing and management capacity of the applications executing in this device, particularly the wireless interface driver running in the device. In order to have a more elaborate explanation of this, a deeper study is suggested; the focus should be to analyze situations where the processor is overloaded, leading to a malfunction of the wireless interface driver. In addition, the study should examine whether the disconnections were caused by hand-offs;

- In addition to the weak performance of the UDP protocol for wireless data transmission in terms of packet loss, in all transmissions using the UDP protocol, it was observed that the protocol delivered out-of-order packets. The out-of-order data delivery also occurred with the DCCP protocol, but in smaller proportion compared to UDP. This proportion is equivalent to the packet loss with the DCCP protocol in the TCP $\times$ DCCP confronts—Tables 3 and 4, field Transmitted/Lost, lines 4 to 9 (inclusive).

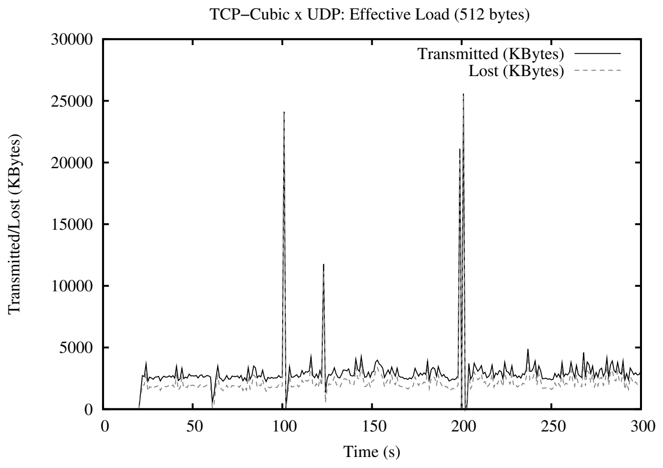
5 Conclusion

In this article, an experimental evaluation of the DCCP protocol over a 802.11g testbed wireless network is presented. We presented an overview of DCCP, an explanation of how the experiments were performed, and explored the methods adopted to calculate each metric studied in this work. An important issue in this case is the use of a statistical method based on probability theory to achieve a 95% confidence level in all the values of the studied metrics. The results obtained by executing the experiments are presented. The experiments used only resource-limited devices.

Considering the results discussed in Section 4, some conclusions can be presented. First, the UDP protocol when used in resource limited devices is not capable of generating high network traffic resulting in congestion. This is due to data generation contention on the Internet Tablets.

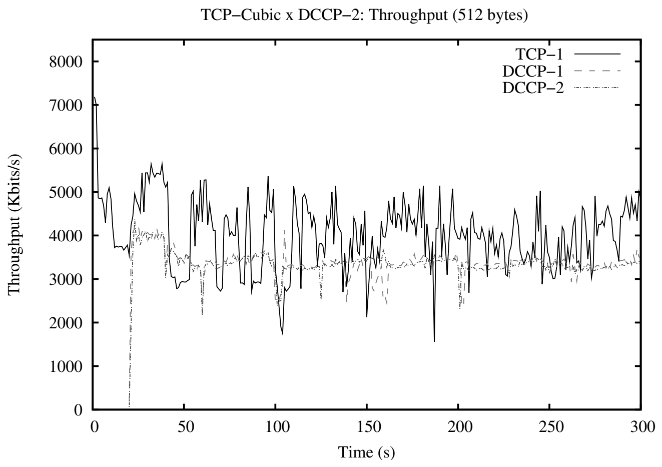


(a) Throughput achieved by TCP Cubic and UDP protocols.

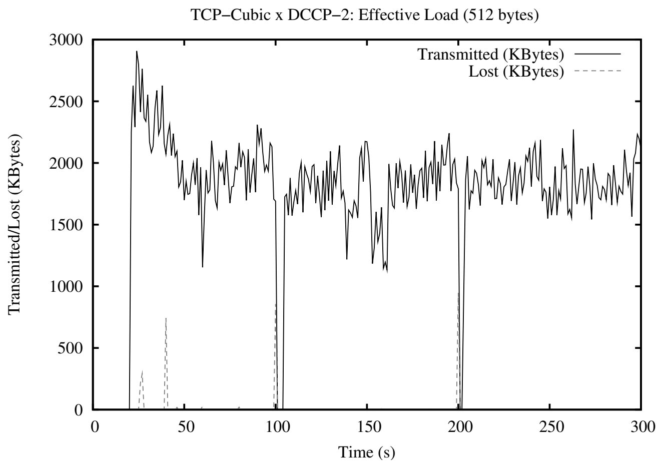


(b) Relation between the amount of data transmitted and lost by the UDP protocol.

Figure 2: TCP × UDP: thoughput and effective amount of data for TCP-Cubic × UDP with 300 *s* of transmission, with packet size of 512 *bytes* and execution of *hand-offs* in 100 *s* and in 200 *s*.



(a) Throughput achieved by TCP Cubic and DCCP CCID-2 protocols.



(b) Relation between the amount of data transmitted and lost by the DCCP protocol with CCID-2.

Figure 3: TCP × UDP: thoughput and effective amount of data of the TCP-Cubic × DCCP CCID-2 with 300 *s* of transmission, with packet size of 512 *bytes* and execution of *hand-offs* in 100 *s* and in 200 *s*.

It is important to point out that even though the UDP protocol was unable to cause network congestion, its use to transmit multimedia flows is not recommended, at least in the network topology used in this work. This recommendation is based on the observations that UDP lost a lot of packets when compared to DCCP, mostly in a network congestion period. This directly reflects in the multimedia quality being transmitted. In addition to the high level of packet loss, UDP interferes with the performance of other protocols that implement network congestion control, such as TCP and DCCP.

Unlike the discussion in [11], the hand-off execution during data transmission did not affect either the TCP or DCCP congestion control algorithms. In the results presented in this previous work, laptops were used, rather than resource-limited devices. There are two hypotheses to explain the non-effect of hand-offs using resource-limited devices: first, by using devices such as the N800, the hand-off occurs very fast and hence few packets are lost, if compared with hand-offs performed using computers (such as laptops) and considering that they are not manufactured with this type of service in mind, unlike the mobile devices. Thus, since the amount of the packet loss is small and considering that resource-limited devices are not capable of generating a big set of data in a short period (due to the short slice of time allocated to each application by the operating system), the small amount of data lost does not affect the congestion control algorithms. The second hypothesis completes the first one. Since N800-like devices are manufactured to work in wireless networks, the network driver is optimized for hand-off executions, unlike those available for the network interface of the laptops. For this point, it is necessary to conduct a more specific study to provide a more accurate conclusion.

Another important conclusion is that when the packet size for TCP was varied in data transmission against UDP flows, the results were not satisfactory. As observed in the results presented in Section 4, there is not a significant improvement in the amount of data transmitted when using 1424-byte (rather than 512-byte) packets. But for TCP $\times$ DCCP confronts, one may conclude that if the packet size is increased from 512 *bytes* to 1424 *bytes*, it is possible to improve the performance of both TCP and DCCP. Therefore, this procedure is encouraged. Although it was possible to observe this, it is necessary to run more experiments with packet size other than 1424 *bytes* and 512 *bytes*.

The current congestion control algorithms for DCCP performed worse than TCP when used to compete against UDP flows (except in the TCP Reno $\times$ UDP, where DCCP performed better than TCP in the DCCP CCID-3 $\times$ UDP), although DCCP seems to reach one of its goals: to be fair in respect to TCP. For this case DCCP performed very well, properly sharing the network bandwidth with TCP.

Supposing that the other part of the wireless and Internet traffic is TCP Veno, or TCP Cubic, the default congestion control algorithm for Linux, the UDP protocol must be fair in respect to TCP, since TCP Cubic and Veno performed very well in terms of the available network bandwidth. Until the end of this work, no references were found that explored possible congestion control algorithms for UDP, nor official comparative studies between TCP Cubic/Veno against UDP, since our work focused in the DCCP point of view. According to the results presented in this work, it is not recommended to use TCP Reno for data transmission mainly over wireless links and in the Internet. Moreover, based on the results presented in this work, for TCP transmissions it is recommended to use of TCP Cubic than TCP Veno, even though the official documentation for TCP Veno indicates that its main focus is on wireless networks. It is necessary to analyze the congestion control algorithms for DCCP in order to optimize them or provide new congestion control algorithms for it, equivalent to TCP Cubic and TCP Veno, preferentially.

The current work we are developing is a mVoIP application based on DCCP for mobile devices, focusing on the maemo$^{\text{TM}}$platform [9].

6 Additional Authors and Acknowledgments

The Additional authors are Heverton Stuart and Vinícius Nóbrega from Embedded Systems and Pervasive Computing Lab. Thanks to CAPES Brazil, for the scholarship and Nokia Institute of Technology Brazil, for the research and sponsor supports.

References

[1] J. Gao and N. S. V. Rao. TCP AIMD Dynamics over Internet Connections. In *IEEE Communication Latter*, pages 4–6, 1 2005.

[2] X. Gu, P. Di, and L. Wolf. Performance Evaluation of DCCP: A Focus on Smoothness and TCP-friendliness. In *Annals of Telecommunications Journal, Special Issue on Transport Protocols for Next Generation Networks*, volume 1, pages 191–206, 1 2005.

[3] Raj Jan. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, Inc, 1 edition, 3 1991.

[4] Eddie Kohler, Mark Handley, and Sally Floyd. Datagram Congestion Control Protocol (DCCP), 3 2006. http://www.ietf.org/rfc/rfc4340.txt. Last access on June 2008.

[5] Eddie Kohler, Mark Handley, and Sally Floyd. Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 2: TCP-like Congestion Control, 3 2006. http://www.ietf.org/rfc/rfc4341.txt. Last access on June 2008.

[6] Eddie Kohler, Mark Handley, and Sally Floyd. Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 3: TCP-Friendly Rate Control (TFRC), 3 2006. http://www.ietf.org/rfc/rfc4342.txt. Last access on June 2008.

[7] Eddie Kohler, Mark Handley, and Sally Floyd. Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 4: TCP-Friendly Rate Control for Small Packets, 6 2007. http://tools.ietf.org/wg/dccp/draft-ietf-dccp-ccid4. Last access on June 2008.

[8] P. Navaratnam, N. Akhtar, and R. Tafazolli. On the Performance of DCCP in Wireless Mesh Networks. In *Proceedings of the international workshop on Mobility management and wireless access*, volume 1, pages 144–147, 3 2006.

[9] Nokia Corporation. Maemo Platform, 2 2008. `http://www.maemo.org/`. Last access on June 2008.

[10] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy. STUN - Simple Traversal of User Datagram Protocol (UDP) through Network Address Translators (NATs), 3 2003. http://www.ietf.org/rfc/rfc3489.txt. Last access on June 2008.

[11] Leandro M. Sales, Hyggo O. Almeida, Angelo Perkusich, and Marcello Sales Jr. On the Performance of TCP, UDP and DCCP over 802.11g Networks. In *In Proceedings of the SAC 2008 23rd ACM Symposium on Applied Computing Fortaleza, CE*, pages 2074–2080, 1 2008.

[12] D. Wu, Song Ci, H. Sharif, and Yang Yang. Packet Size Optimization for Goodput Enhancement of Multi-Rate Wireless Networks. In *Consumer Communications and Networking Conference Proceedings*, pages 3575–3580, 3 2007.

Smack in Embedded Computing

Casey Schaufler
The Smack Project
casey@schaufler-ca.com

Abstract

Embedded computing devices are often called upon to provide multiple functions using special purpose software supplied by unrelated and sometimes mutually hostile parties. These devices are then put into the least well protected physical environment possible, your pocket, and connected to an unprotected wireless network.

This paper explores use of the Smack Linux Security Module (LSM) as a tool for improving the security of embedded devices with rich feature sets. The internet enabled cell phone is used to identify application interaction issues and describe how they can be addressed using Smack. The paper compares Smack-based solutions to what would be required to approach the problems using other technologies.

1 Mandatory Access Control

Mandatory Access Control (MAC) refers to any mechanism for restricting how a process is allowed to view or manipulate storage objects that does not allow unprivileged processes to change either their own access control state or the access control state of storage objects. This differs from Discretionary Access Control (DAC) in that a DAC mechanism, such as the traditional file permission bits or POSIX access control lists, may allow unprivileged processes to change their own access control state or that of storage objects. MAC is the mechanism best suited to providing strong separation of sensitive information while allowing controlled data sharing and communications between processes that deal with controlled data.

1.1 Alternatives To MAC

Isolation is easy. Sharing is hard.

Virtualization is currently getting the most attention of all the mechanisms available for providing strong separation. It is also the most expensive scheme, short of multiple instances of hardware, requiring additional processor speed, memory, and storage to provide multiple copies of the operating system. While sharing can be done using virtual network interfaces and authenticating application and system level protocols like NFS, it offers no improvement over having those processes on the same real machine. Further, there is no way to share IPC objects such as memory segments and message queues.

Chroot jails also provide limited isolation. While the filesystem name space can be broken up, the socket and IPC name spaces remain shared. Data sharing can also be achieved using a variety of mount options.

Mandatory Access Controls can isolate the IPC and networking name spaces as well as the filesystem name space while still allowing for appropriate sharing.

1.2 Bell and LaPadula

Prior to the current era of enlightened MAC, the only scheme available was the Bell and LaPadula sensitivity model. This model is a digital approximation of the United States Department of Defense paper document sensitivity policy. This model is fine for its intended purpose, but scales neither upward for more sophisticated polices nor downward to simpler ones. While it is possible to implement interesting protections for embedded systems using this scheme, [1] the combination of rigid access rules, the size of the implementations, and the sometimes excessive price of the products offering it prevented this model from ever gaining traction in the embedded space.

[1]HP actually sold a B&L based email appliance for some time.

1.3 Security Enhanced Linux - SELinux

Security Enhanced Linux, or SELinux for short, is a security infrastructure that provides type enforcement, role based access control, Bell & LaPadula sensitivity, and a mechanism to extend into future realms of security management including, but not limited, to control over the privilege scheme. SELinux associates a label with each executable that identifies the security characteristics of a process that invokes that program. The label applied to the process is influenced by the label of the program, but the previous label of the process has an impact as well. The access control decisions made by SELinux are based on a *policy*, which is a description of security transitions.

For an embedded system, SELinux has some drawbacks. Because the label attached to a program file impacts the security characteristics of the process, programs like busybox that perform multiple functions depending on their invocation have to be given all the rights any of its functions may require. The policy must be programmed to take into account the behavior of the applications, making it difficult to incorporate third party programs. The policy can be large, in excess of 800,000 lines for the Fedora distribution, with a significant filesystem data footprint as well as substantial kernel memory impact. If the policy changes, for example to accommodate a program being added to the system, it may require that the entire filesystem be relabeled and the policy be reloaded into the kernel. Finally, SELinux requires that the filesystem support extended attributes, a feature that can add cost to the system.

2 Smack

The Simplified Mandatory Access Control Kernel (Smack, as a name not an acronym) implements a general MAC scheme based on labels that are attached to tasks and storage objects. The labels are NULL terminated character strings, limited somewhat arbitrarily to 23 characters. The only operation that is carried out on these labels is comparison for equality.

Unless an explicit exception has been made, a task can access an object if and only if their labels match. There is a small set of predefined system labels for which explicit exceptions have already been defined. A system can be configured to allow other exceptions to suit any number of scenarios.

Unlike SELinux, which bases the label that a task runs with on the label of the program being run, Smack takes an approach more in line with that of the multilevel secure systems of the late twentieth century and allows only the explicit use of privilege as a mechanism for changing the label on a task. This means that security is a attribute of the task, not an attribute of the program. This is an especially important distinction in an environment that includes third party programs, programs written in scripting languages, and environments where a single program is used in very different ways, as is the case with busybox.

The label given a new storage object will be the label of the task that creates it, and only a privileged task can change the label of an object. This is another behavior that is consistent with multilevel secure systems and different from SELinux, which labels files based on a number of attributes that include the label of the task, but also the label on the containing directory.

2.1 Access Rules

The Smack system defines a small set of labels that are used for specific purposes and that have predefined access rules. The rules are applied in this order:

- * Pronounced *star*. The star label is given to a limited set of objects that require universal access but do not provide for information sharing, such as `/dev/null`. A process with the star label is denied access to all objects including those with the star label. A process with any other label is allowed access to an object with the star label.
- _ Pronounced *floor*. The floor label is the default label for system processes and system files. Processes with any label have read access to objects with the floor label.
- ^ Pronounced *hat*. The hat label is given to processes that need to read any object on the system. Processes with the hat label are allowed read access to all objects on the system.
- **matching labels** A process has access to an object if the labels match.
- **unmatched labels** If there is an explicit access defined for that combination of process and object labels and it includes the access requested, access is

```
cardfs /card cardfs smackfsroot=ESPN,smackfsdefault=ESPN 0 0
```

Table 1: Mount Options Example

permitted. If there is an explicit access defined for that combination of process and object labels and it does not include the access requested or there is no explicit definition, the access is denied.

2.2 Defining Access Rules

A Smack access rule consists of a subject label, an object label, and the access mode desired. This triple is written to `/smack/load`, which installs the rule in the kernel.

2.3 Unlabeled Filesystems

As previously mentioned, not all of the filesystems popular in embedded systems support the extended attributes required to label each file individually. In some cases, such as that of removable media, it is unreasonable to trust the labels that would be on the filesystem if it did support them. A reasonably common situation involves an embedded system with two filesystems, one that contains all the system data and a second that is devoted to user data and which may be removable. Even if neither filesystem supports extended attributes this is easily supported by Smack via filesystem mount options. The mount options supported by Smack are:

- **smackfsroot=***label* Specifies the label to be used for the root of the filesystem.
- **smackfsdefault=***label* Specifies the label to be used for files that do not have labels stored in extended attributes. For filesystems that do not support extended attributes this will be all files on the filesystem.

An easy way to isolate the system from applications that use external data then is to run the applications with a label other than the floor label and to mount the external data at that label. An entry in `/etc/fstab` for this might resemble Table 1.

The application running with the **ESPN** label can read the system data and modify anything on `/card`. Should the application run a program found on `/card` the process will continue running with the **ESPN** label and will have the same access.

2.4 Networking

Network based interprocess communications are far and away the dominant mechanism for passing information between processes. Smack imposes the same restrictions on writing information to another process as it does writing information to a storage object. The general rule is that the sending process and the receiving process must have the same label. If an explicit rule allows a process with the sender's label to write to an object with the receiver's label then a message can be sent. For UDP packets the sender need only have write access to the receiver. For TCP connections both ends must have write access to the other, but neither is required to have read access.

2.5 Network Labeling

Network labeling is accomplished by adding a CIPSO IP option that represents the sender's label to the packet header. With the label of the sender in hand an access decision can be made at the time of delivery, when the label of the receiver is known.

One label is designated the ambient label. All packets that have no CIPSO tag are given the ambient label. Symmetrically, packets created by processes running with the ambient label are not given CIPSO tags.

2.6 Sockets

Sockets are not themselves elements in the Smack security model. Sockets are data structures associated with processes, and can sometimes be shared. Socket attributes can be set by privileged processes to associate a particular label with outgoing packets and to change the label used on incoming checks. The labels attached to TCP connections and to individual UDP packets can be fetched by server processes.

3 Secure Embedded Systems

There are probably as many notions of what defines an embedded system as there are of what defines system security. For the purposes of embedded systems security, there is a specific set of characteristics that are interesting.

An embedded system will usually be resource constrained. Processor power, storage size, and system memory are only some of the things that can add cost and that are subject to scrutiny and reduction where possible. Security solutions that require significant additional resources introduce cost and may even push a product out of viability.

Embedded systems often do not have multiple users. Google's Android project assumes this to be universal and co-opts the userid mechanism for program isolation. Systems are designed around data flows or application sets rather than providing general purpose user environments. They may also assume that programs have restricted use patterns and limit security concerns to variations from those patterns.

Systems deployed in embedded environments are expected to function for extended periods of time without modification or with as few as possible. It is important to get the software and its configuration correct prior to release as it may never have the opportunity to be repaired. Even those systems that can be repaired in the field will usually require that changes be as few and as small as is absolutely possible. A security scheme based on regular updates to threat profiles would be inappropriate to a flight data recorder.

3.1 Filesystems

Embedded systems can have particularly sensitive filesystem requirements. The devices that they use are often slow and may have media with limits on the number of times it can be updated. They may also be limited in the amount of data they can store. These characteristics in particular encourage the use of filesystems that do a minimum of physical accesses and that are optimized for size in favor of functionality. Support for extended attributes is often eschewed because the additional media space required, the increase in code size, and the consequences of maintaining extended attributes on the media make them unappealing for the environment.

3.2 Networking

Networking is important to embedded systems for interprocess communications and external access.

Because IP protocols do not normally carry any sort of security identification information, application level protocols are often required to provide identification and authentication on their own. This requirement can add significantly to the application size and complexity, the number of libraries required, and the time required to perform communications, especially simple ones.

Some embedded systems communicate with the world at large and for many, including mobile phones and more sophisticated devices, this is their primary function. In many cases it is quite important that information be kept segregated based on its role on the device, and that the information be delivered only to appropriate applications which may themselves have come in over the airwaves. Clearly a mechanism needs to be available for distributing this information safely.

4 The Mobile Phone

It is probably impossible to identify a typical embedded system, but the mobile phone will serve for purposes of discussion because the mobile phone is familiar, some are known to run Linux, and they have obvious security concerns. Those who are unfamiliar with these devices are encouraged to have a look at Google's Android system [2] for a working example of how the software for one of these devices can be assembled.

Mobile phone service providers do not make money by selling phones. They make money by selling services that use the information network with which the mobile phone communicates. It is thus very important to the service company that the system software and user account data stored on the phone be protected from any user of the device. It is also important that access to features of the phone that the user is expected to pay extra for is tightly controlled. It is further a significant concern that only the applications the service provider gets paid for wind up on the phone, otherwise the consumer may not have to buy the provider's offerings to get the functionality desired.

[2] `http://code.google.com/android`

An important but often overlooked aspect of the mobile phone is that one of the most critical design criteria for the software it runs is time to market. It can be the case that a particular date, usually early in the holiday shopping season, is a hard deadline and software development must be completed sufficiently in advance of that date to allow volume manufacturing. Any architecture with a long time to market or that may interfere with the ability to deploy third party applications in a timely fashion will come into question even if it is adopted for reasons of security.

4.1 A Simple Application Example

Let us now consider a mobile phone that incorporates third party applications to provide its differentiation. The third party applications will have been delivered slightly late and will have little or nothing to say about any interactions they might have with their operating environment. The applications will certainly not have gone through any sort of security analysis. If only to prevent the applications from accidently interfering with each other it is prudent to isolate them.

For the sake of simplicity, our phone implements a display manager, a keypad manager, and a radio manager. These managers communicate with applications and each other using UDP datagrams. Each of these managers runs as a separate process, started when the phone is turned on, and all with the Smack label **phone**. Because they all have the same label they can share information freely, but because they are not running with the floor label they cannot modify the system files. All the device files that these managers access are also labeled **phone**.

The first application might be a news update service that queries a database at ABC, presenting a ticker tape of interesting headlines on the display. It is run with the Smack label **ABC**. To achieve this, the application will send messages to the radio manager asking it to call out for updates, and sending text to the display manager to put on the ticker tape. The radio manager needs to send responses to the ABC application. To allow this communication two Smack access rules are required.

- `phone ABC w`
- `ABC phone w`

Notice that read access is not provided in either case. Processes with either label can send datagrams to processes with the other, but neither can read their peer's data.

The second application is from a sports network and offers animated recreations of football highlights. It is run with the Smack label **ESPN**. The application will send messages to the radio manager asking it to call out for updates, and send animation frames to the display manager. The radio manager needs to send responses to the ESPN application and the keypad manager needs to send keystrokes for control purposes. As before two Smack access rules are required.

- `phone ESPN w`
- `ESPN phone w`

Notice that even though both ABC and ESPN processes can communicate with the manager processes they cannot communicate directly with each other.

It turns out that in the example here both applications provide service based on information from a common source, that being the shared parent company of the news service and the sports network. If the sports animation application understands the data stored by the news application and has read access to that information it could pre-load sport event information that appears on the ticker tape, improving the user experience. A single Smack rule makes this possible.

- `ESPN ABC r`

Now the sports animation application can read the news application's data and take whatever actions it deems appropriate. Notice that it cannot execute the news application or search directories because it does not have execute permissions.

If at some point in the future the parent company sells the sports network, access can be revoked without relabeling any files by changing the access rule.

- `ESPN ABC -`

Now access is explicitly denied.

4.2 Software Update

A common problem on embedded devices is live, controlled application software update. While getting new software onto the device may be straightforward, making sure that the transition occurs and that the new software is used while retaining the old in case of unforeseen issues can be tricky. One popular solution to this problem is to provide multiple filesystems, each of which is loaded with a different version of the complete set of software. One filesystems is mounted in the active path while the others are mounted to the side and the transition is made by unmounting the active path and mounting an alternative in its stead. Updates are performed on the out-of-path filesystems.

The Smack solution is to include all possible paths, but to label each set differently, and to determine which gets used by a particular process by access rules. The start-up script running with an appropriate label, in this case **ESPN**, sets its path

```
export PATH=/slot-a:/slot-b
```

then invokes the desired program

```
spiffyapp -color -football
```

which will of course use the version in `/slot-a` if it is accessible, and the version in `/slot-b` if it is not. The installer labels `/slot-a` and all the files therein with the same label, for simplicity **Slot-A** and similarly the contents of `/slot-b` with **Slot-B**. To allow access to either version the rules would be

- `ESPN Slot-A rx`
- `ESPN Slot-B rx`

When it comes time to update `/slot-a` setting the access rule

- `ESPN Slot-A -`
- `ESPN Slot-B rx`

ensures that the version in `/slot-b` gets used. Once `/slot-a` is updated setting the access rules

- `ESPN Slot-A rx`
- `ESPN Slot-B -`

ensures that the new version is used. Note that the files in each of the slot directories do not get relabeled as part of this process, they retain the label that they are given by the installer. The only change required is in the access rules.

5 Comparisons and Conclusions

From the examples provided it should be clear that many of the security concerns that are typical of an embedded system can be addressed readily by Smack. It is not enough to provide the security facilities, it is also necessary to provide them in a way that is appropriate to the problem at hand. Other schemes, including virtualization and SELinux, can be used to address specific security concerns, but Smack is better suited to the resource-constrained embedded environment.

5.1 Distributions

The purpose of a distribution is to provide a set of configuration files, documentation, programs, libraries, scripts, and various other digital components with which a complete system can be composed. Most distributions available today are full-featured, offering as complete a set of utilities as possible, often even including multiple alternatives for email services, web servers, and window systems environments. Distributions targeted for the embedded space will offer a slightly different set of content and configuration, but are not fundamentally different from their desktop or enterprise peers.

A MAC scheme based on the behavior of applications will have to be customized to each distribution on which it is available. The Red Hat distributions include customized SELinux policies that match the programs they contain. The SuSE distributions include configurations for AppArmor. Other distributions claim support for SELinux as well.

The embedded systems developer is typically not looking for the advantages of integration that a distribution provides. The embedded systems developer will be carefully choosing the components that go onto the box and while it will be convenient if they all come from the same place it is perfectly reasonable for a legacy version of certain applications to be chosen for size, compatibility, or performance. This is a major problem for a system like SELinux that depends on specific versions of specific applications for the policy to be correct. A system like Smack that is strictly based on processes, rather than programs, in its security view has a serious advantage.

5.2 User Space Impact

The user space component of a security mechanism ought not to be a major concern for an embedded system. Because Smack rules are trivial, the program that loads them into the kernel need only ensure that they are formatted correctly and can hence be kept very small. Because labels are text strings there is no need for functions that compose or format them. The current Smack user space library provides only two functions.

- `smackaccess` Takes a process label, an object label, and an access string as arguments and returns an access approval or denial based on the access rules currently loaded in the kernel. Using this function an application can make the same decisions that the kernel would. Because the kernel table is readable, any program can use this function to determine what the answer is to a specific access question.

- `smackrecvmsg` This is a wrapper around `recvmsg` that does control message processing associated with `SCM_SECURITY`. It is typically used by label-cognizant server programs that may change their behavior based on the label of a connection. These programs will require privilege to allow connections at multiple labels and will hence be required to be treated as trusted components of the system.

One reason that there are so few library functions is the direct scheme that Smack uses for labeling. Because labels are text strings that require no interpretation, their manipulation is limited to setting and fetching. The existing extended attribute interfaces are sufficient for manipulating labels on files. Process labels are dealt with through the `/proc/self/attr/current` virtual files. Socket labeling is manipulated using `fsetxattr` to set outbound labels and set inbound labeling, but only by privileged processes.

5.3 Configuration Issues

Embedded systems are usually designed to be as simple as possible. Sophisticated configuration requirements go against this design principle the same way that excesses in scripting would.

One problem with a virtualization solution is having multiple operating system configurations to maintain. Another is the hypervisor configuration. Finally, there is the configuration required for the virtual machines to share or communicate.

SELinux is notoriously difficult to administer. Because the security model labels programs based on their behavior, any change, even a simple version update, may require a change to the system security policy configuration. A policy that does not take the entire set of applications on the system into account does not provide the controls necessary for accurate containment. This is true regardless of how much of the full utility of SELinux is actually required to achieve the security goals.

Simplicity is a design goal of Smack. The coarser granularity of access control provided by a process-oriented scheme requires much less detail in the configuration than does a fine-grained scheme such as SELinux. Because it is an access control mechanism that can be configured, it is much easier to use than the multiple configurations required in a virtualized scheme.

5.4 Summation

Embedded systems are not general purpose computers. Smack is intended to address clearly identifiable and specific access control issues without requiring extensive theoretical understanding of security lore. It does not require the intervention of a highly trained security professional. The low impact and strong control of Smack make it ideal for solving the controlled access problems of applications in embedded systems. Freedom from dependence on a distribution makes it attractive to developers inclined to "roll their own" system software. With process oriented access control emphasis can be placed on the pragmatic security issues that matter in the embedded space.

Energy-aware task and interrupt management in Linux

Vaidyanathan Srinivasan, Gautham R Shenoy,
Srivatsa Vaddagiri, Dipankar Sarma
IBM Linux Technology Center
{svaidy, vatsa}@linux.vnet.ibm.com, {ego, dipankar}@in.ibm.com

Venkatesh Pallipadi
Intel Open Source Technology Center
venkatesh.pallipadi@intel.com

Abstract

As multi-core and SMP systems become more generally available, energy management needs in Linux™ have also become more complex. Energy management in Linux was primarily designed for interactive systems where relatively simple inactivity based strategies worked effectively for most cases. Modern enterprise class hardware needs a more complex power management strategy to save energy with the least impact on the performance of enterprise workloads. Traditionally, the Linux kernel for servers has been optimized for throughput and not power efficiency.

This paper discusses the behaviour of the current task management subsystem (scheduler and loadbalancer) on a multi-core SMP system and its effectiveness in saving energy consumption under several situations (idle, moderate load). It then describes several techniques such as timer migration, task wakeup biasing and related heuristics for reducing energy consumption. The paper also looks at possible methods to mitigate interrupts for energy savings during different workloads and concludes by discussing some results of these new strategies.

1 Introduction

Traditionally, operating systems designers have focussed on optimizing for performance. The key design goals have been to make maximum usage of resources to get the most out of the underlying systems. On multiprocessor and multi-core systems, this approach led to using all CPU resources in parallel as much as possible. This approach to system design had to be re-evaluated when battery operated low-power devices became important. Various system technologies like DVFS (Dynamic voltage and frequency scaling) and exploitation of them in operating systems led to significant improvement in power consumption [1][4]. Various operating system techniques were adopted to manage tasks with a goal of reducing power consumption [17][6][18]. With the advent of multiprocessor systems, additional techniques have been used to do CPU power management [15]. With the cost of energy going up in recent years, the need for energy efficiency has been acutely felt across the entire spectrum of systems—small handheld computers to large multi-processor servers in datacenters. Due to increased computation density of modern enterprise servers, the thermal limits of the design is beginning to constrain the integration and performance. Power management in enterprise servers primarily help data centers improve their computation density by getting more computation done without increasing power consumption. The objective of power management in laptops and other battery powered devices has been primarily to extend the battery life, while on enterprise server and datacenters, power management forms the building blocks to provide higher level services like power trending and power capping. Thermal management, which is an interesting side effect of power management, and power capping are of great interest to enterprise customers. Fundamentally, enterprise customer would like to control parameters that have been previously considered passive and hence ignored.

This paper investigates power management in two areas to simplify the discussion, namely

[1]With additional contributions from **Suresh B. Siddha**, *Intel Open Source Technology Center*, suresh.b.siddha@intel.com

1. Idle system power management
2. Power management in under-utilized (or non-idle) systems

An idle system is one where no useful work is done by the system with respect to its workload and applications. Such a system could be waiting for inputs from user or requests on the network. On this kind of system, it is usually the system house keeping jobs that are active.

On a non-idle system, the system is actively running the workload or application but the overall system capacity is under-utilised. This provides scope to perform several run-time power management strategies such as frequency and voltage scaling. In the Linux kernel, the ondemand governor [11] does the job of selecting the correct CPU capacity or frequency that would match the current workload.

The new SPECPower benchmark [2] tries to characterise performance-per-watt under various system loads.

Avoiding the periodic scheduler tick in an idle system with the tickless kernel feature significantly helps to save power in an idle system, while process scheduler tweaks are needed to save power in an non-idle system. The next section explains the problem space and existing solutions in detail.

2 Scheduler Overview

In a multi-processor system, an important goal for a power-aware operating system is to consolidate *all* activity (like execution of tasks, interrupt handling etc) on fewer CPUs so that remaining CPUs can become idle and enter low-power states. That implies a constant tug between providing good throughput for applications and providing good energy savings.

We now provide a brief overview of Linux CPU scheduler, how it is currently meeting the needs of a power-aware operating system and some potential enhancements to make it more energy-concious.

- **CFS scheduler**
  From the primitive v2.4 scheduler, to the scalable O(1) scheduler in v2.5, to the more recent scalable and responsive CFS scheduler, the Linux cpu scheduler has significantly changed several times. The most recent rewrite, termed CFS (Completely Fair Scheduler), was authored by Ingo Molnar [8] and has been adopted since v2.6.23 (July '07). It was mainly written to address several interactivity woes that the Linux community complained about. Its salient highlights are:

  – More modular scheduler core by introducing scheduler classes
  – Time indexed rb-tree as runqueue for SCHED_OTHER tasks
  – Excellent interactivity for desktop users by doing away with concept of fixed timeslice
  – Scheduler tunables
  – Group scheduler – divide bandwidth fairly between task-groups first and then between tasks in the group

- **Power awaress existing in current Linux scheduler**

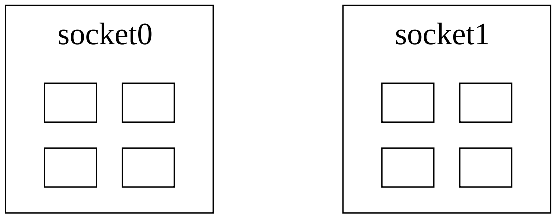


Figure 1: Two socket quad-core system

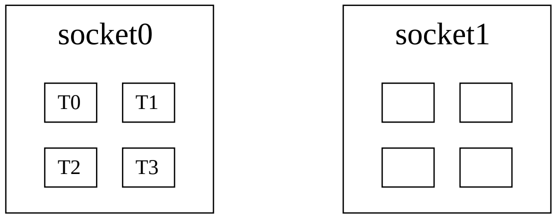


Figure 2: Good task distribution from power perspective

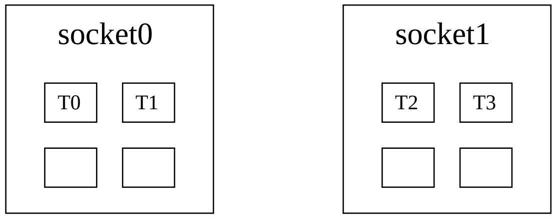


Figure 3: Bad task distribution from power perspective

Linux CPU scheduler has options, both compile time and runtime [10], through which it can be directed to consolidate tasks across fewer CPUs rather than spreading them apart on all available CPUs for improved throughput. This lets more CPUs to become idle and thus enter low-power states when system is not heavily loaded. Additionally, the Linux scheduler is aware of the underlying multi-core and NUMA topology. This makes it possible for the scheduler to further optimize power-savings. For example, consider a multi-core system as shown in Figure 1. The system has two sockets, each of which can accommodate a quad-core chip. Typically in such systems, the granularity at which frequency/voltage can be varied is at each chip or package level [5]. In other words, the frequency/voltage cannot be different for different cores resident in the same chip. In such a scenario, the CPU scheduler is required to recognize such *power* domains and work towards not only consolidating tasks on just fewer cpus but also on fewer *power* domains (in this case, the chips). As an example consider that the system in Figure 1 had just four tasks. Then it is better for a power-aware CPU scheduler to consolidate these 4 tasks on the 4 cpus in same chip (as in Figure 2) rather than on *any* 4 arbitrary cpus (as in Figure 3). Linux CPU scheduler has the capability to do this chip-wise consolidation of tasks when required.

- **Areas for improving power-awareness in scheduler**

  Consolidating tasks on fewer cpus and chips relies on accurate cpu load (number of tasks on a cpu) calculation. Since cpu load is sampled periodically, it is possible that short lived tasks (ex: daemon that run periodically for short intervals of time) don't show up as cpu load, which can result in failure to consolidate on fewer cpus/chips. This is discussed in detail in the Section 3.5. Typically the total CPU time utilised by the daemons in an idle system will be less than 1% but the distribution of this jobs across all CPUs influence the CPU's low power sleep time thereby affecting the power consumption at idle. Section 3 describes the idle system in detail.

  In addition, CPUs that are in their low-power states can be interrupted prematurely by task wakeup code, which attempts to schedule waking tasks on the same cpu where they last slept.

3 Idle system power management

Apart from the applications or the workload, there are a host of system daemons, device drivers, and interrupt processing that happen in an idle system. If the operating system and hardware can be optimised to significantly reduce these house keeping tasks, then an idle system can sleep for longer duration leading to power savings. There are significant activities even on a tickless idle [12][7] system that reduce the duration of a CPU's low power sleep time leading to an increase in the energy consumption at idle.

The objective of idle system power management in an enterprise server is to consolidate daemon tasks and interrupts to fewer CPUs or packages in an idle system. Typically an enterprise server would have more than one CPU in SMP configuration. Multi-core processors have helped increase the number of cores in an enterprise system. Dual socket server can have dual core processor modules thus forming a 4-way SMP configuration. Optionally processor threading feature can be enabled which would further increase the number of logical CPUs to eight. In this sample configuration, (assume no threading) we have 4 logical CPUs in two physical packages.

Under low system utilisation or idle, if all the housekeeping work can be consolidated to one package, then the other package can continue to be in low power sleep state, thus saving power. In general the SMP scheduler will try to spread the workload across different package for better throughput. This is the main design point in power savings and performance trade off.

If the number of tasks to run is less than the number of cores, spreading the tasks to one core on each physical package will provide better throughput (assuming the tasks do not share data), while consolidating them on one physical package is better for power savings.

In the following subsections we shall discuss the challenges involved in consolidating daemon jobs in an idle system to one physical package in a dual package system.

3.1 Process, timers, and interrupts at idle

The daemon process that are running in an idle system can be easily identified using *ps* or *top* commands.

Other than the processes that use CPU time, there could be interrupts from IO devices like ethernet and harddisk that wakeup CPUs and consume power. Timers programmed by applications and device drivers are actually interrupts that wakeup CPUs when the timer expires.

CPU utilisation and interrupt rate can give a good idea of the idleness of the system. All these events are required for normal system operation, however in an idle system, these events contribute to reduced sleep time of the CPU.

In a typical distro installation,[1] CPU utilisation from *top* and process wakeup rate from *powertop* at idle are shown in Table 3.1.

The number of interrupts observed during the 15-second duration is listed in Table 1.

IRQ	CPU0	CPU1	CPU2	CPU3	Description
17	21				ata_piix
214		61			eth0
LOC	66	55	95	71	Local timer interrupts
TLB		1		1	TLB Shootdowns

Table 1: /proc/interrupts diff for 15 seconds

In order to improve CPU sleep time, idle polling activities should be reduced and moved to asynchronous notification. USB inherently needs to have time based polling loops. USB auto suspend will work as long as there are no devices connected, but if the USB port is being connected even to an idle keyboard, the polling loops are needed.

Since the introduction of powertop utility, the behaviour of user space applications and drivers have significantly improved and moved away from unnecessary polling. On an enterprise hardware with multiple CPU packages, the timers and interrupts that cannot be reduced can be consolidated to one CPU package. This provides new opportunity for power savings by allowing parts of the system to be more idle. The objective of idle system power management is to significantly increase the idle time for some of the CPU packages in the system.

For our discussion lets assume *idle time* is the duration over which a CPU is in tickless idle state. During this time, no task is scheduled on the CPU. Thus, the CPU can potentially go to low power sleep state and save power. However in this state, the CPU can receive interrupts. The interrupts can be from an IO device or a programmed timer. Hence the *idle time* duration can further be fragmented by interrupts and timers. Lets call the time interval between such interrupts where the CPU can really sleep in low power mode as *sleep time*. Tickless kernels that turn off periodic timer interrupts have a significantly long *idle time*. However only the uninterrupted *sleep time* contributes to power savings. Hence, to characterise various scenarios, we extract two parameters, namely the CPU *idle time* and *sleep time*. *Sleep time* can be obtained by *idle time* divided by the number of interrupts and timers received during the interval.

[1]Fedora 9 beta was used in this experiment.

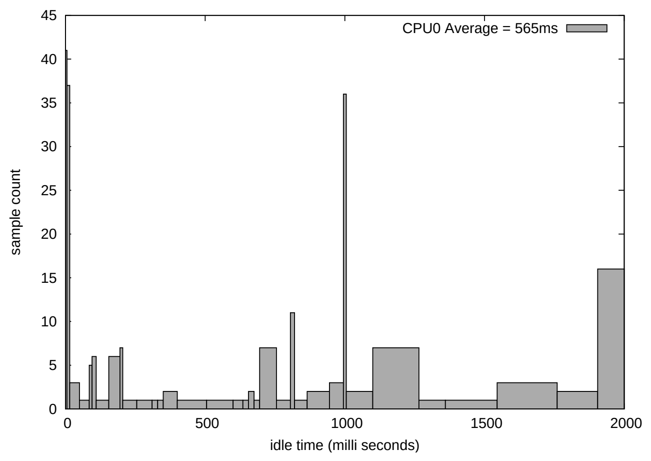


Figure 4: Fedora 9 distro on 4 CPU system

Idle time distribution can be obtained by instrumenting `tick_nohz_stop_sched_tick()` and `tick_nohz_restart_sched_tick()` code [13]. Figure 4 plots the histogram of idle time obtained on one of the CPU in a typical distro. Basically the idle time values for 120 seconds in an idle system has been converted to a histogram for easy visualisation. The x-axis is the idle time and y-axis is the sample count observed during 120 seconds. This gives an idea of expected sleep time for a given CPU in a multi-cpu system. Actually this experiment was done on a two socket dual core system and such histograms are available for each of the four CPUs. The distribution is similar in other CPUs and thus we will not discuss the histogram for all four CPUs. As observed in the histogram, the maximum idle time was 2 seconds while most of the samples are concentrated at less than 10ms. There is a pattern of 1s idle time as well.

Figure 5 is a histogram of sleep time. Sleep time is much smaller than the idle time and inversely proportional to

Utilisation from top:

```
Cpu(s):  0.0%us,  0.1%sy,  0.0%ni, 99.9%id,  0.0%wa,  0.0%hi,  0.0%si, 0.0%st
```

Output of powertop -d:

```
PowerTOP 1.8    (C) 2007 Intel Corporation

Collecting data for 15 seconds
< Detailed C-state information is only available on Mobile CPUs (laptops) >
P-states (frequencies)
  2.40 Ghz     0.0%
  2.13 Ghz     0.0%
  1.87 Ghz     0.0%
  1.60 Ghz   100.0%
Wakeups-from-idle per second : 10.8     interval: 15.0s
Top causes for wakeups:
  28.7% (  4.0)   <kernel module> : usb_hcd_poll_rh_status (rh_timer_func)
  27.3% (  3.8)       <interrupt> : eth0
  10.0% (  1.4)       <interrupt> : ata_piix
   7.2% (  1.0)                ip : bnx2_open (bnx2_timer)
```

the interrupt rate. The maximum sleep time was 400ms while the typical sleep time was less than 10ms.

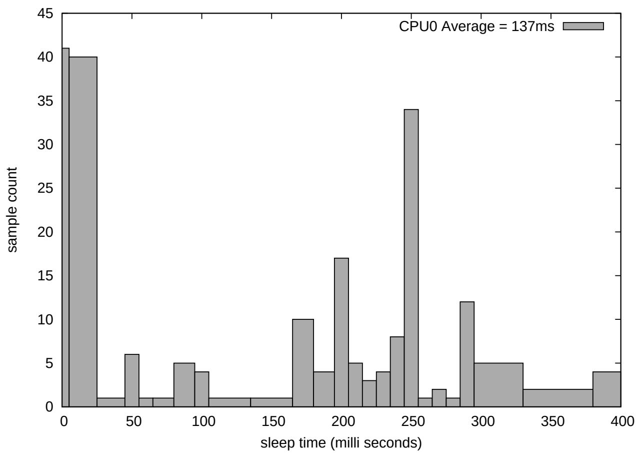


Figure 5: Fedora 9 distro on 4 CPU system

In order to analyse the effect of various kernel tunables and scheduler changes we need to derive a metric for comparison. The obvious and simplest metric is the average idle time and average sleep time on each CPU. Basically the weighted average of samples obtained from the histogram for each CPU in the system gives the average idle and sleep time values. The approximation in assuming average value per CPU over long duration is that even marginal change in average value could significantly affect the power savings. The longer the CPU is in sleep state, the more power is saved. Small number of long sleep intervals is better than large number of small sleep intervals. Hence to improve the accuracy of the evaluation model and its correlation with real power consumption, a weight factor may be needed for each sleep duration corresponding to the processor's deep sleep state transition latency. Based on typical processor sleep state transition latencies and power values, we can perhaps assume that a sleep duration of more than 100ms is good enough for the CPU to transition into deep sleep state. We have omitted the power penalty for transition into various sleep states as well. The average value has been marked in the histogram.

3.2 Multi core scheduler heuristics

One of the first tunables in the kernel to tweak in an multi core, multi socket system is `/sys/devices/system/cpu/sched_mc_power_savings`. When the multi-core power saving mode [3] is enabled, the scheduler's load balancer is biased to keep workload on single physical package. This has significant impact when the number of tasks running in the system is less than the number of cores. Figure 6 plots the total idle time for each CPU for a 120 second observation interval. The system topology was a two socket dual core, with CPU0 and CPU1 sharing a package and CPU2 and CPU3 sharing the other package. Power savings can be improved if CPU0-1 are idle or CPU2-3 are idle allowing the other package to go to low power sleep state. Ebizzy is a simple cpu intensive benchmarking tool. It was simple to use and demonstrate the effect of `sched_mc_power_savings`. Figure 7 shows that the idle time on first package has improved by consolidating the workload on CPU2 and CPU3. Ebizzy[2] was run with two threads for a duration of 120 seconds. There is a power savings of 5.4% by enabling `sched_mc_power_savings` for such cpu intensive tasks where the number of threads are less than the total number of cores available in the system.

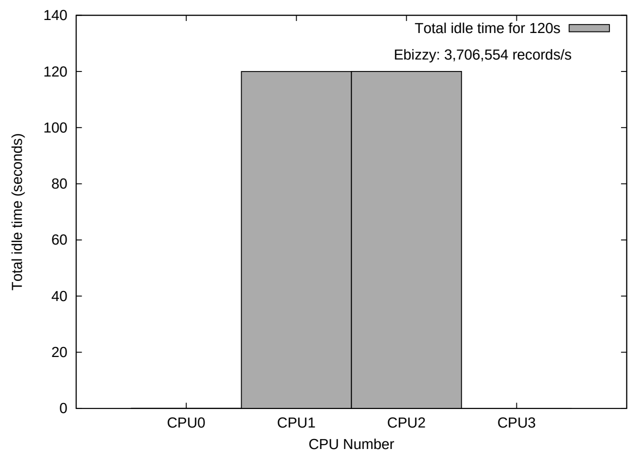


Figure 6: ebizzy with sched_mc_power_savings=0

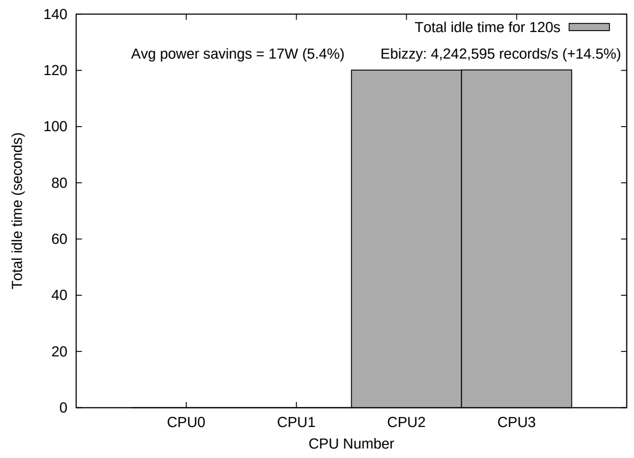


Figure 7: ebizzy with sched_mc_power_savings=1

However, the tunable will bias the scheduler loadbalancer and will not explicitly move tasks to different package. The impact is that short running daemon jobs that wakeup on various CPUs in the system will finish execution before the loadbalancer is invoked or a CPU load imbalance is detected. Hence they will continue to wakeup idle CPUs in the system.

The ineffectiveness of `sched_mc_power_savings` for short running jobs can be observed in case of kernel compilation (kernbench) workload. Figure 8 shows the idle time for `make-j2` on the same box used in the ebizzy experiment. When `sched_mc_power_savings` is enabled as shown in Figure 9, there is not much variation in the total idle time across all CPUs and hence there is no influence in the power value. The effectiveness of the heuristics is workload dependent. Kernel compilation consist of large number of short running jobs and high rate of process creation and exit as compared to pure CPU burn type workload. The variation in characteristics is mainly due to the mix of IO operations.

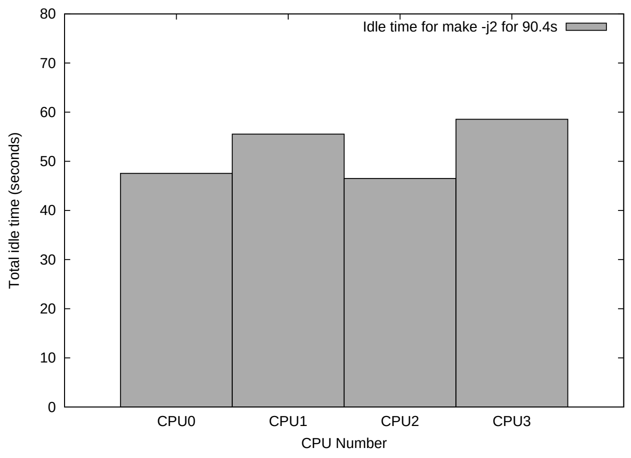


Figure 8: make -j2 with sched_mc_power_savings=0

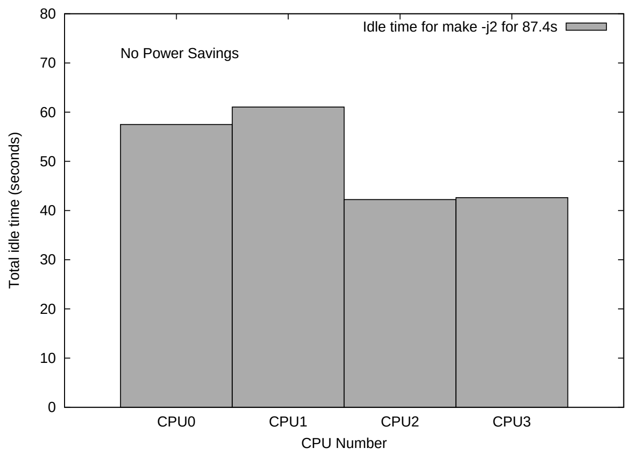


Figure 9: make -j2 with sched_mc_power_savings=1

3.3 Interrupt Migration

Interrupts in the system can be routed to one or many CPU cores in an SMP system. Default kernel routing of interrupts is to broadcast to all available CPU if that is supported by the hardware or else the interrupts stay

[2]ebizzy -t 2 -s 4096 -S 120

with the boot-up CPU or logical CPU0. There are in-kernel interrupt balancing thread and user space solutions available. The user space solution to manage interrupt routing in an SMP system is the *irqbalance daemon* [16] which is already included by most distros. The latest version 0.55 of the daemon includes power management feature where interrupts will be consolidated to CPU0 at very low interrupt rate. Once the system activity increases, the interrupt rate will increase and then the daemon will re-calculate the load and distribute the interrupts among different CPUs appropriately. The irqbalance daemon takes into account the system topology, threads, cores and packages while making the interrupt routing decision. The class of interrupt is also considered since cost of migrating ethernet interrupts is higher than that of storage interrupts.

Why does interrupt routing matter for power savings? Any CPU in its low power sleep state can be woken-up by an interrupt. Interrupts are critical for system operation and they cannot be avoided. If the interrupts are distributed to all CPUs, then even at low interrupt rate (idle system), more CPUs in the system need to wake-up for a short duration and process the interrupt. Effectively the low power sleep time of the CPU is reduced. In the case of SMP system where many CPUs are available, the interrupts can be routed to one of the CPU package or core. This leads to only one CPU package in the system to paying the power penalty for wake-up while rest of the CPU packages in the system can continue to be in low power sleep state for longer time.

Interrupt routing can be modified by writing the bitmask corresponding to the destination CPU to `/proc/irq/<irq_nr>/smp_affinity`.

`echo2>/proc/irq/214/smp_affinity` would route eth0 interrupt to CPU1 in the experimental setup described in Section 3.1, Table 1.

The user space irqbalance daemon controlls interrupt routing by writing appropriate bitmask to `/proc/irq/<irq_nr>/smp_affinity`. More than one bit can be set in the bitmask which enables the hardware to distribute the interrupt to the subset of CPUs if such interrupt broadcasting is supported by the platform and chipset.

Interrupt migration helps to improve CPU sleep time on some of the CPUs in an SMP system, but timers queued by the device drivers and applications are not affected since timers are triggered when the CPU receives an interrupt from the per-cpu tick-device. Timers queued on various CPUs in an SMP system significantly contribute to CPU wake-up from deep sleep state.

3.4 Timer Migration

3.4.1 Timers—an Introduction

Device drivers and other subsystems keep a sense of time in the kernel by means of timers. The kernel provides APIs such as `add_timer()`, `mod_timer()`, `add_timer_on()` that allow the subsystems to add or modify a timer to expire sometime in the future. With the introduction of High Resolution timer infrastructure, users can now opt for timers with finer granularity should they need it. The APIs present in the Linux kernel for the High-Resolution timers are `hrtimer_start()` and `hrtimer_forward()`.

When a timer expires, the timer subsystem will call the associated handler function which will perform the required task.

At the time this paper was written, the Non-High-Resolution timers in the Linux kernel can be classified into two types:

1. **Non-Deferrable Timers**: These are normal timers which expire when a specified amount of time elapses

2. **Deferrable Timers**: These are timers, which on a busy system behave the same way as a normal timer. But on an idle system they can be ignored while determining the next timer event. Thus they will expire when the next non-deferrable timer on the idle CPU expires. [9]

Most of these timers are initialized and queued for the first time from the task context. However, the handler function gets called from the softirq context. The re-queuing of a timer can occur from the softirq context or the task context. It is easy to observe that currently, the timers which get requeued from the softirq context will be pinned to the CPU where they had been first initialized. However, those timers which are queued from the task context can migrate as the tasks queueing them get migrated. Thus the timer distribution on an SMP system is currently dependent on which CPU did the timer initialization happen and the load balancing.

3.4.2 Effect of timers on Idle CPU

On an idle CPU which is in a NO_HZ state, we program the timer hardware to interrupt the idle cpu to coincide with the nearest non-deferrable timer expiry time. Thus if there are device drivers which had initialized timers on a CPU which is now mostly idle, we would nevertheless have to wake up the idle cpu to service this timer.

This highlights the importance of consolidation of timers onto a fewer number of CPUs. Migration of these timers in a idle system is possible. As of now, the timers are migrated during CPU *offline* operation.

However CPU hotplug for the sake of idle system power management is too heavy. We will need to implement light weight timer migration framework that can be invoked in a idle system for power management purposes.

Typical distribution of timers in a distro[3] at idle in a 120 second duration is detailed in Tables 2 and 3. These results were obtained by instrumenting the timer code `__next_timer_interrupt` in `kernel/timer.c` and post processing the trace data [14].

As mentioned earlier, timers can re-queue themself in task context or softirq context. Table 2 details the list of timers that were queued in softirq context. They are generally stuck to the same CPU until forcefully moved or the application or device driver removes the timer. Table 3 details the timers that were queued in task context. These timers will generally be queued on the CPU where the corresponding task has run. The idea behind this data is to assess the percentage of timers that can be consolidated by just moving or biasing the tasks. From the data it can be observed that almost half of the timers are from task context and they can be moved by moving the task which is much easier than migrating the timer. Migrating the timer may need notification and opportunity for the task to cancel the timer all together.

3.5 Workload Migration and Consolidation

As mentioned in Section 2, the load consolidation algorithm in the current scheduler relies on accurate cpu load, i.e., the number of tasks on a cpu as an input parameter. This value is updated by sampling periodically. Since, every task running on a system need not be CPU intensive, it is possible that techniques such as `sched_mc_power_savings` fail to capture the characteristics of such tasks when it comes to workload consolidation.

[3]Fedora 9 beta

To prove this, consider an experiment where we have a *cpuset A*, which has a bash shell as a member, that runs `make-j2` of a Linux kernel. The experiment is run on a 2 socket dual core machine. The logical CPUs 0 and 1 are core siblings in the first socket and logical CPUs 2 and 3 are the core siblings in the other socket. We vary the number of CPUs allocated to the `cpuset` by writing different values to `cpuset.cpus` file. The time taken to complete the `make`, the avg power consumed (normalised value) during this interval, and the utilization of the individual CPUs in the system are recorded. During the experiment, `sched_mc_power_savings` is set to 1 and the `cpufreq` governor is set to `ondemand`. Table 4 details the result of this experiment.

Ideally, one would expect that running the job with only two cpus would yield the same results as running the job with all the four cpus with `sched_mc_power_savings` enabled. However, from the experiment, we observe that `sched_mc_power_savings` does not seem to have much effect when we run with all the four CPUs. There are only two active tasks in the system but they get distributed across all the CPUs.

In the experiment, the energy consumed for the case with only two CPUs is:

$$\begin{aligned} E_2 &= 71.751 \times 1.045x \\ &= 74.980xJ \end{aligned} \tag{1}$$

Energy consumed for the case with all the four CPUs is:

$$\begin{aligned} E_4 &= 85.185 \times 0.941x \\ &= 80.159xJ \end{aligned} \tag{2}$$

Thus, additional amount of energy spent would be:

$$\begin{aligned} E_{extra} &= E_4 - E_2 \\ &= 80.159x - 74.980x \\ &= 5.179xJ \end{aligned} \tag{3}$$

$$\begin{aligned} E_{extra}\% &= \frac{E_{extra}}{E_2} \times 100 \\ &= 6.91\% \end{aligned} \tag{4}$$

From Table 4 we can also observe that the time taken to complete the job is higher when all the 4 CPUs were

Function_Name	CPU0	CPU1	CPU2	CPU3	Total
rh_timer_func			483		483
delayed_work_timer_fn	62	59	60	71	252
bnx2_timer		119			119
neigh_periodic_timer	30		60		90
dev_watchdog		48			48
process_timeout	32		1	5	38
wb_timer_fn	24				24
peer_check_expire	4				4
neigh_timer_handler			4		4
hangcheck_fire	3				3
commit_timeout	1			2	3
addrconf_verify			3		3
Total	156	226	611	78	1071

Table 2: Timer in SOFTIRQ context at idle for 120s

Function_Name	CPU0	CPU1	CPU2	CPU3	Total
hrtick	159	132	107	81	479
delayed_work_timer_fn	62	60	60	72	254
ide_timer_expiry	34	30	33	29	126
scsi_times_out	40	1	3	1	45
process_timeout	31		2	6	39
wb_timer_fn	24				24
blk_unplug_timeout	19	1	3	1	24
hrtimer_wakeup	2				2
commit_timeout	2				2
tcp_write_timer		1			1
it_real_fn	1				1
Total	374	225	208	190	997

Table 3: Timer in task context at idle for 120s

Experiment 'make -j2' of linux-2.6.25-rc7			
CPUs allocated	**Time taken**	**Power Consumed**	**% Utilization of the CPUs**
0	120.678 s	1.000x W	99, 02, 00, 00
0-1	71.751 s	1.045x W	83, 89, 01, 01
0-3	85.185 s	0.941x W	34, 33, 59, 57

Table 4: *'make -j2'* with varying number of cpus

used, when compared to the case where only 2 CPUs were used. Since the ondemand governor changes the processor frequency based on the processor utilization, when 2 threads were bouncing across all 4 processors, the system utilization was not high enough to increase the frequency to the maximum, and hence it took longer time. However, in the case of allocating just 2 processors, we note that the utilization is sufficiently high for the ondemand governor to run the job at the maximum frequency, thus finishing it faster.

Thus we observe that by allocating more processors than what is required, we're not only degrading the power savings, but also the performance in this case. Hence there is scope for power savings by improving the power aware task/workload consolidation in an under utilised system.

One of the possible solutions could be to consider the following parameters during scheduler load balancing or consolidation decision apart from just counting the number of waiting tasks:

- The nature of each task, whether it is CPU intensive or IO bound
- The overall utilization of the system.
- Any hints from the tasks themselves

Using some of these parameters, it is also possible to bias the wake up of a task onto a non-idle CPU, thereby avoiding waking up an idle CPU when the number of tasks on a particular runqueue is nonzero.

4 Sleep states

Coming to the core of the issue, why do we want the CPUs to be idle for long duration. Modern processors supports multiple idle states that vary from high power low latency idle states to low power high latency idle states.

With CPUs being idle for extended period, they can be put into low power high latency idle state, conserving significant power in the process. On the other end, frequently waking up CPUs cannot use deepest idle state, as if they do, they end up paying significant overhead due to higher transition latency in and out of the deepest idle state.

Other factors to keep note of with regard to idle CPUs are:

- Most of the current generation CPUs control the CPU voltage at the socket level. This means, if some cores in a socket are idle and other cores are busy, idle cores may not be at optimal power state due to the higher voltage on the socket leading to higher leakage power.
- Most of the current generation multi-core CPUs have some shared resources across all the cores, like last level cache. This shared resource will not be able to go to low power state unless all cores in the socket are idle.

This means it is important to keep as many cores and as many sockets in idle state as long as possible. That helps the CPUs to be at the most optimal power state.

The current Linux kernel has cpuidle governor that takes care of entering the right idle state based on the individual CPU activity and requirements. The scheduler power savings tunable takes care of keeping the entire socket idle in case of long running tasks. Newer versions of irqbalance take care of routing interrupts to one CPU while the system is relatively idle.

The things that are missing include:

- power-aware scheduling for short-running tasks
- smart routing of timers interrupts in the idle scenarios, and
- making the CPU latency requirements per CPU (instead of system-wide) so that processes and interrupts with critical latency requirements continue to have good response time in partially idle case, with other idle cores being in deepest idle state.

5 Conclusion

Every watt saved is a watt that doesn't have to be generated! Doing more computation with less power helps the environment and every power management feature makes the world greener. System power management helps to improve compute density and manage power as a resource by matching the power consumption to workload just as in an automobile.

The Linux kernel already takes advantage of various power management features available in the platform, however there is still scope for improvement as new platform features will become available in future.

6 Acknowledgments

We owe thanks to Ingo Molnar, Arjan van de Ven and members of linux-pm and lesswatts-discuss for their inputs on Linux kernel issues. We are also indebted to Patricia Gaughen, Vani S. Kulkarni, Sudarshan Rao, and Premalatha M. Nair for their support of this effort.

7 Legal Statement

References

[1] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen. Low-power cmos digital design. *JSSC*, 27:473–484, 1992.

[2] Standard Performance Evaluation Corporation. `SPECpower_ssj2008`. `http://www.spec.org/power_ssj2008/`.

[3] Suresh B Siddha et al. Chip multi processing aware linux kernel scheduler. `http://www.linuxsymposium.org/2005/linuxsymposium_procv2.pdf`.

[4] Kinshuk Govil, Edwin Chan, and Hal Wasserman. Comparing algorithm for dynamic speed-setting of a low-power cpu. In *MobiCom '95: Proceedings of the 1st annual international conference on Mobile computing and networking*, pages 13–25, New York, NY, USA, 1995. ACM.

[5] Intel Technology Journal. Power and thermal management in the intel core duo processor. `http://download.intel.com/technology/itj/2006/volume10issue02/vol10_art03.pdf`.

[6] Jacob R. Lorch and Alan Jay Smith. Reducing processor power consumption by improving processor time management in a single-user operating system. In *MobiCom '96: Proceedings of the 2nd annual international conference on Mobile computing and networking*, pages 143–154, New York, NY, USA, 1996. ACM.

[7] LWN. Tickless, acpi and thermal management. `http://lwn.net/Articles/240253/`.

[8] Ingo Molnar. `CFS` scheduler. `http://kerneltrap.org/node/8059`.

[9] Venkatesh Pallipadi. Deferrable timers. `http://lwn.net/Articles/228143/`.

[10] Venkatesh Pallipadi and Suresh B Siddha. Processor power management fatures and process scheduler: Do we need to tie them together. `http://www.linuxconf.eu/2007/papers/Pallipadi.pdf`.

[11] Venkatesh Pallipadi and Alexiy Starikovsky. The ondemand governor - past, present and future. In *Proceedings of the Linux Symposium*, volume 2, Ottawa, ON, Canada, 2006. Linux Symposium.

[12] Suresh Siddha. Getting maximum mileage out of tickless. `http://ols.108.redhat.com/2007/Reprints/siddha-Reprint.pdf`.

[13] Vaidyanathan Srinivasan. Instrumenting idle time. `http://lkml.org/lkml/2008/1/8/243`.

[14] Vaidyanathan Srinivasan. Instrumenting timers at idle. `http://lkml.org/lkml/2008/3/2/109`.

[15] Jinwoo Suh, Dong-In Kang, and Stephen P. Crago. Dynamic power management of multiprocessor systems. *ipdps*, 2:0097b, 2002.

[16] Arjan van de Ven. Irqbalance - handholding your interrupts for power and performance. `http://irqbalance.org`.

[17] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced cpu energy. In *OSDI '94: Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, page 2, Berkeley, CA, USA, 1994. USENIX Association.

[18] Fan Zhang and Samuel T. Chanson. Power-aware processor scheduling under average delay constraints. In *RTAS '05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium*, pages 202–212, Washington, DC, USA, 2005. IEEE Computer Society.

Choosing an application framework for your Linux mobile device

Shreyas Srinivasan and Phaneendra Kumar
Geodesic Information Systems
shreyas@geodesic.com

Abstract

Application Frameworks define the user experience. For consumer mobile devices, choosing a feature-rich and high-performance application framework becomes a premium. Creators of Linux mobile devices have a range of application frameworks (gtk/qt/efl) to choose from, but this choice also makes it hard to pick a framework which suits a specific set of requirements.

This paper evaluates various open source application frameworks and their underlying technologies. It also explains performance benchmarks of the frameworks on different types of hardware and the capability of the framework to use specific hardware features to improve performance.

The Application frameworks which will be evaluated are Gtk/Gnome, QT/KDE, Clutter/Tidy, and EFL/E. The talk will present performance benchmarks of these frameworks on Omap, Freescale, and Intel mobile processors.

1 Introduction

User interfaces have become an important factor to decide the popularity and success of a consumer device. The launch of the IPhone has showed the importance of a well designed and intuitive user interface, and has heightened expectations of users all over the world. Increasingly, most processors have built-in floating-point processors and support open standardes like Open GL ES, so the ability to support fluid interfaces and animations exists.

Processors for embedded devices have largely been dominated by ARM-based processors, but with X86 processors improving rapidly, it is important to understand the current features and capabilities of each of these architectures.

ARM

The ARM architecture has always dominated the embedded device market due to its low power consumption. There have been various versions of ARM over time, but with performance improvements plateauing, most ARM processors have chosen to add co-processors to provide specialized performance to applications. Some of the widely used embedded processors based on ARM are:

- **OMAP**
  The Omap series of processors are based on ARM Cortext A8. Different families can have co-processors like PowerVR SGX 530 2D/3D and a DSP Video accelerator. The OMAP 2430 processor range is used by Nokia for their internet tablets.
- **Freescale I.MX**
  The I.MX range of processors by Freescale is based on an ARM 1136JF-S core. Different families have options of different co-processors like the VFP11 numeric processor, IPU, H.263/MPEG4 encoding accelerator, and ARM MBX R-S graphics accelerator. The I.Mx31 processor is used in the Microsoft Zune.

X86

The X86 architecture has always been seen as the one valid for desktops but unsuitable for embedded devices, mainly due to power consumption. With the Menlow family of processors, X86 processors can finally compete with ARM ones, even in power consumption.

- **Pentium Mobile**
  The Pentium mobile range of processors are X86-based processors with 1 GHz processor, a built-in FPU, and power consumption of 14.44 Watts.

These are mainly used for laptops, but some devices like the Founder also use this chip because of its very high performance.

- **Menlow**
  The Menlow (Atom) series of processors are the first of Intel's chips to foray into low power for embedded hardware. The major difference is removal of predictive instruction execution; this reduces power consumption. The Menlow range of processors range from 800 MHz to 1.6 GHz, have a built-in floating point unit, and consume just 5W of power.

The trends of hardware means different requirements for application platforms depending on the underlying architecture.

2 Characteristics of a good Application Framework

Timeline and Animation Support. Creating intuitive and fluid interfaces requires a state-aware canvas which can move different objects over a sequence of coordinates with regards to time.

Hardware support. The ability to use specific hardware features to increase performance is of premium importance. Embedded hardware-based rendering has been standardized around OpenGLES. This is particularly important on the ARM architecture, where graphics performance can be considerably improved by using the Graphical Processing Unit and conserving the relatively low computational power of the CPU.

Multi Language Bindings. Multi Language bindings make the application framework viable to a wide variety of programmers.

Email Libraries. Email is one of the core applications. Email libraries which support a range of protocols like POP, IMAP, and Exchange are of great importance.

Browser Support. Support for rendering and embedding web pages is a powerful feature which enables applications to enrich the user experience by supporting local and cloud-based applications.

Multimedia. Capability of rendering video and audio.

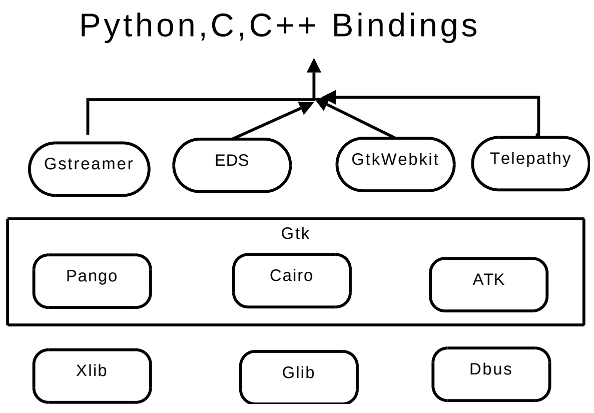


Figure 1: Architecture of Gnome Mobile Application Framework

Inter Process Communication. Communication between various applications helps build a good user experience.

This paper evaluates multiple application frameworks and their performance in some of the aforementioned categories.

3 Application Frameworks

Gnome Gtk

The GNOME Mobile Platform is a subset of the proven, widely used GNOME Platform. The platform definition represents components that are currently shipping in production devices.

- **Components**
  1. **Cairo**
     Cairo is a 2D graphics library with support for multiple output devices. Currently supported output targets include the X Window System, Quartz, Win32, image buffers, PostScript, PDF, and SVG file output. Experimental backends include OpenGL (through glitz), XCB, BeOS, OS/2, and DirectFB.
  2. **EDS**
     Evolution Data Server is a PIM server which manages access to calendar, addressbooks, and tasks. All these items are served over Dbus.

3. **GtkWebKit**
   WebKit/GTK+ is the new GTK+ port of the WebKit, an open-source web content engine that powers numerous applications such as web browsers, email clients, feed readers, and web and text editors.
4. **GStreamer**
   GStreamer is a library that allows the construction of graphs of media-handling components, ranging from simple Ogg/Vorbis playback to complex audio (mixing) and video (non-linear editing) processing. Applications can take advantage of advances in codec and filter technology transparently. Developers can add new codecs and filters by writing a simple plugin with a clean, generic interface.

- **Advantages**
  1. Existing precedent of devices which ship with this platform.
  2. Well defined roadmap and enthusiastic developer community.
  3. High profile industry support.
  4. Focus on minimal footprint.
  5. Non-free codecs can be licensed on top of gstreamer and shipped legally.

- **Disadvantages**
  1. No current support for offscreen rendering.
  2. Cairo OpenGL backend is extremely unstable.
  3. Gobject API has a steep learning curve.

EFL E

- **Components**
  1. **Evas**
     Evas is a hardware-accelerated canvas API for the X Window System that can draw anti-aliased text, smooth super and sub-sampled images, alpha-blend, as well as drop down to using normal X11 primitives such as pixmaps, lines, and rectangles for speed if your CPU or graphics hardware is too slow.

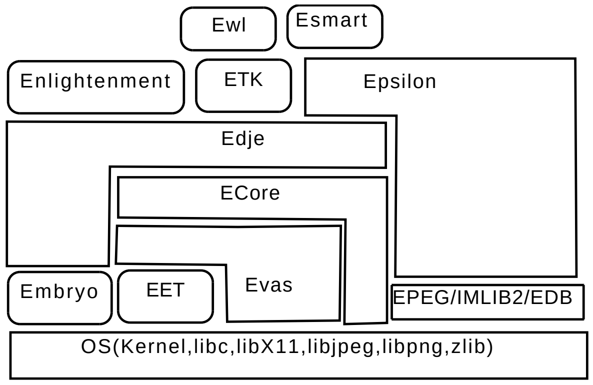


Figure 2: Architecture of Enlightenment Foundation libraries

  2. **Ecore**
     Ecore is the core event abstraction layer and X abstraction layer that makes doing selections, Xdnd, general X stuff, and event loops, timeouts, and idle handlers fast, optimized, and convenient. It's a separate library so anyone can make use of the work put into Ecore to make this job easy for applications.
  3. **Edje**
     Edje is a graphical design and layout library based on Evas that provides an abstraction layer between the application code and the interface, while allowing extremely flexible dynamic layouts and animations.
  4. **EWL**
     The Enlightened Widget Library (EWL) is a high-level toolkit providing all of the widgets you'll need to create your application. The expansive object-oriented-style API provides tools to easily expand widgets and containers for new situations.
  5. **Emotion**
     Emotion is a library providing video-playing capabilities through the use of smart objects. Emotion is based on libxine, a well established video playing library, and so supports all of the video formats that libxine supports, including Ogg Theora, DiVX, MPEG2, etc.

- **Advantages**
  1. Small Memory footprint
  2. Evas is a state-aware canvas which supports timeline-based animations.

3. Evas has an OpenGL backend and hence can be hardware accelerated.

- **Disadvantages**

  1. Long release cycles.
  2. Rapidly changing mainline; makes it hard to keep up.

Clutter

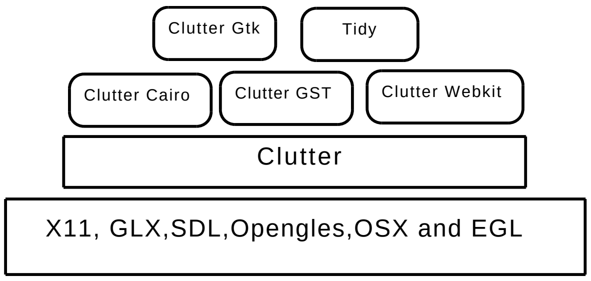


Figure 3: Architecture of Clutter

- **Components**

  1. **Clutter**
     Clutter uses OpenGL (and optionally OpenGL ES for use on Mobile and embedded platforms) for rendering, but with an API which hides the underlying GL complexity from the developer.
  2. **Clutter-GST**
     Clutter-GStreamer (clutter-gst) is an integration library for using GStreamer with Clutter. GStreamer is a streaming media framework, based on graphs of filters which operate on media data. Applications using this library can do anything from real-time sound processing to playing videos, and just about anything else media-related.
  3. **Clutter-Webkit**
     Clutter Webkit is an integration library which allows HTML rendering on GL textures. This could also provide hardware acceleration for video rendering through the browser.

- **Advantages**

  1. Clutter has a OpenGLES backend, which makes it suitable for ARM-based devices with a GLES-based GPU.
  2. Most hardware provides gstreamer-based libraries for hardware codec support.

- **Disadvantages**

  1. Clutter is not production-ready.
  2. Tidy, the toolkit built on top of clutter, is still nascent and does not have a comprehensive set of widgets.

4 Benchmarks and suitability

Frame Rate

Frame rate, or frame frequency, is the measurement of the frequency (rate) at which an imaging device produces unique consecutive images called frames. The term applies equally well to computer graphics, video cameras, film cameras, and motion capture systems. Frame rate is most often expressed in frames per second (FPS) and in monitors as Hertz (Hz). To create a fluid interface, the underlying framework should at least output between 25–30 frames per second. This section benchmarks the frame rate across various hardware.

1. Intel Mobile Processor
   The Intel Mobile Processor range consists of high-performance chips which are mainly used in ultra-mobile PCs.

   - **Hardware Specifications**
   - Processor: Intel Mobile 1 Ghz
   - Memory: 1 GB
   - Power Consumption: 14.44 Watts

2. Freescale IMX.31
   The Freescale chip consists of an ARM 1136JF-S core with an onboard GPU which supports OpenGLES.

   - **Hardware Specifications**
   - Processor: 533 MHz
   - Memory: 128 MB
   - Power Consumption: 6.5 Watts

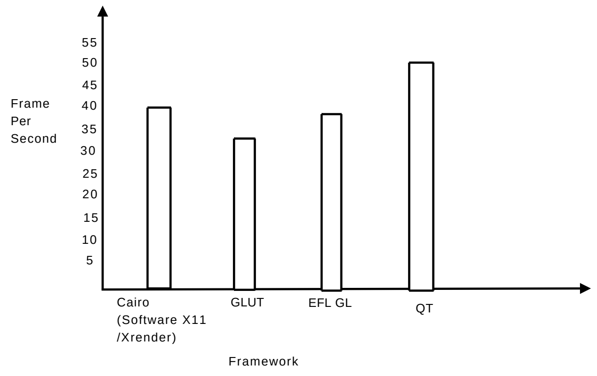


Figure 4: Frame Rate on Intel Mobile Processors

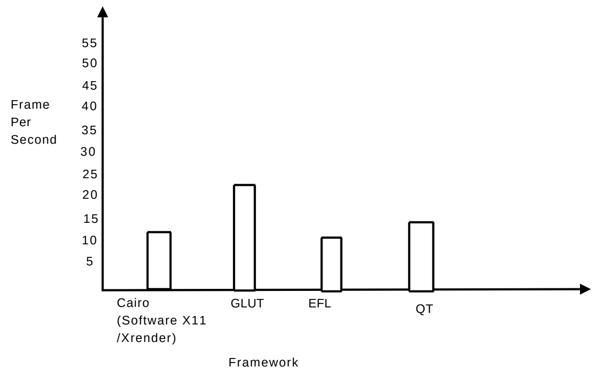


Figure 5: Frame Rate on Freescale I.MX 31

Quality vs. performance

Pixel-perfect drawing is necessary for accurate event processing and coherent visual representation. Current hardware access abstractions like OpenGL/GLES don't provide pixel-exact hardware aliasing. This may result in substantial pain when trying to deal with constant user input and interaction.

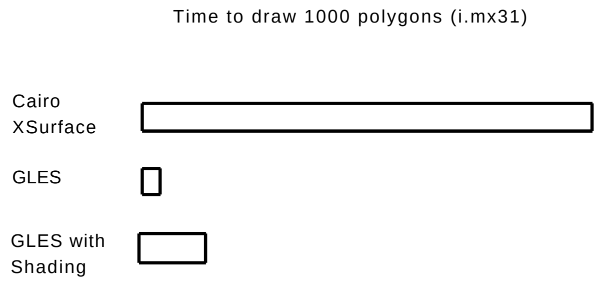


Figure 6: Time taken to draw 1000 polygons

To understand the tradeoffs involved in rendering versus quality, this test draws shaded polygons using OpenGL and compares the time taken to draw 1000 such polygons against a software-only API provided by Cairo. We show the output of both tests to understand quality.

5 Future

As we compete with application frameworks like Cocoa, FOSS application frameworks need to accomplish the following:

1. **Look nicer and feel intuitive**
   Implement aspects of physics and 3D to create a new intuitive paradigm which feels natural and exciting.

2. **Easier to develop, extend, and deploy**
   Learning lessons from web development are extremely necessary, a bridge needs to be built between application developers and graphic artists.

3. **Flexible design to handle multiple interaction modes**
   Increasingly new methods to interact with the computer are gaining popularity. An open design which can easily handle multiple interaction modes makes it easier to build quick support for new input devices.

4. **Established method for hardware acceleration**
   The FOSS application frameworks continue to have a varied approach to hardware acceleration. An accepted approach preferably using X (DRI) would help in making it easy to differentiate one framework from other on a functional basis while also allowing them to work well in unison.

6 Conclusion

This paper analyzes and benchmarks various application development frameworks. There are a lot of application development frameworks which one can use right now, but the ability to build cutting-edge, hardware-accelerated interfaces is still in its infancy. We are still seeing a multitude of approaches, all of which are works in progress. Issues such as hardware acceleration, animation support, and multi-input handling all need more work to challenge current market leaders. The benchmarks presented cover most of the important usage scenarios and various directions currently being pursued.

Making a decision on choosing an application platform continues to be a subjective decision over a purely objective one. The current bout of approaches need to stabilize for us to make a complete set of benchmark tests which can help you decide one way or the other, for sure!

References

[Keith] Keith Packard, *Getting X Off The Hardware*, `http://keithp.com/~keithp/talks/xserver_ols2004/xserver-ols2004-html/`

[Zack] Zack Rusin, *Benchmarking tesselation*, `http://zrusin.blogspot.com/2006/10/benchmarks.html`

[Michael Dominic] Michael Dominic, *OpenGL Shaders and Cairo*, `http://www.mdk.org.pl/2007/8/6/vector-drawing-opengl-shaders-and-cairo`

[OpenGLES] OpenGLES, *OpenGLES 1.1 and 2.0 specification*, `http://www.khronos.org/registry/gles/specs/1.1/es_full_spec.1.1.12.pdf`

[Tim] Tim Janik, *OpenGL for Gdk/Gtk+*, `http://blogs.gnome.org/timj/2007/07/17/17072007-opengl-for-gdkgtk/`

[QT and OpenGL] Qt and OpenGL, *QT and OpenGL*, `http://doc.trolltech.com/3.3/opengl.html`

[GTK+3.0] GTK+3.0, *Imendio's GTK+ 3.0 vision*, `http://developer.imendio.com/sites/developer.imendio.com/files/gtk-hackfest-berlin2008.pdf`

[Carl] Carl Worth, *EXA*, `http://cworth.org/tag/exa/`

SCSI Fault Injection Test

Kenichi Tanaka
NEC Corporation
k-tanaka@ce.jp.nec.com

Masayuki Hamaguchi
NEC Software Tohoku, Ltd.
m-hamaguchi@ys.jp.nec.com

Takatoshi Sato
NEC Software Tohoku, Ltd.
t-sato@wm.jp.nec.com

Kosuke Tatsukawa
NEC Corporation
tatsu@ab.jp.nec.com

Abstract

It has been widely recognized that the testing of Linux kernel is important. However, error handling code is one of the places where testing is difficult. In this paper, a new block I/O test tool is introduced. This makes testing of error handling codes easy. The new test tool has driver level fault injectors which have flexible and fully controllable interface for user level programs to simulate real device errors. This paper describes the detailed design of the new test tool and a fault injector implementation for SCSI. Also, the usefulness of the new test tool is shown by actual evaluation of Linux software RAID drivers.

1 Introduction

There is an increasing opportunity to use Linux in enterprise systems, where the users expect very high reliability. Storage is one of the areas for which the highest reliability is required because its failure may cause system downtime and data loss. For the case of hardware failures, the operating system must provide high quality error handling. That means thorough testing of the error handling code is inevitable.

However, error handling code is one of the places where testing is difficult. There are two reasons why evaluation of error handling code is difficult;

- Error handling code is rarely executed. It can not be tested just by running the system under normal operation.
- Fault patterns vary. They can occur during various timings, and it is difficult to thoroughly test each combination.

Fault injection is a generally used technique to overcome the difficulty by controlling the fault occurrence and forcing the execution of error handling code. Several fault injection methods are already available for Linux but all of them lack either variety of fault patterns or flexibility to inject faults as intended, which are needed for systematic evaluation of error handling code. For example, with the existing fault injection methods, it is difficult to make a test program which tests error handling code of a hard disk drive (HDD) access timeout while the software RAID recovery is in progress.

In this paper, a new test tool with SCSI fault injection is introduced. The test tool is capable of injecting SCSI faults with realistic fault patterns and includes a set of test programs to cover various combinations of fault conditions. Since SCSI is the most widely used storage subsystem in Linux, this test tool enables systematic evaluation of error handlings in Linux block device drivers.

In section 2, design overview of the test tool and comparison with other existing fault injection tools are described. Design and implementation details are explained in section 3. Section 4 shows an example of evaluation using the test tool for Linux software RAID drivers, that was the original motivation of developing this test tool, and the result of the evaluation. We conclude in section 5 and explain possible future works.

2 Testing Error Handler Using SCSI Fault Injection

In order to systematically test error handling code using fault injection for a target kernel component which is being tested, a set of test programs is necessary where each individual test program checks whether a certain type of fault occurring when the target kernel component is in a certain state is handled correctly. It is necessary for the test program to prepare the target kernel module to be in the desired state, and then inject the desired fault on the desired access which the test program will trigger, and test if the result is correct.

In order to achieve this goal, the fault injector provides an interface to specify the type of fault which will be injected, and on which access will cause the injection.

2.1 Specifying the Type of Fault

The SCSI HDD fault patterns will be categorized to determine the fault pattern which the fault injector has to generate.

SCSI fault can be classified in two patterns. One is "The SCSI device respond with an error" pattern, which is the case when the drive explicitly returns an error condition to the OS. The other is "The device does not respond" pattern, which is the case when the drive does not return any status to the OS resulting in a timeout. For example, the former can be caused by media error and the latter can be caused by SCSI cable fault.

Alternatively, HDD hardware fault can be divided into temporary faults and permanent faults. A temporary fault can be caused by an accidental and recoverable HDD fault. A permanent fault can be caused by a severe HDD fault.

Based on the type of access which will cause the fault in each of the above four areas, we have categorized the HDD faults into the following eight categories.

Temporary faults with error return can be classified into the following two cases, based on the type of access.

1. Temporary read error – This type of fault is accidentally caused by read access, which occurs just once.

2. Temporary write error – This type of fault is accidentally caused by write access, which occurs just once.

Permanent error with error return can be classified into the following three cases.

3. Read error correctable by write – This type of fault is a medium error which can be corrected by writing data to the failed sector. After writing to the sector, subsequent reads and writes will both succeed.

4. Permanent read error – This type of fault is a permanent medium error on a particular sector. Any read access to the sector fails, but write will succeed, because many disks can not detect errors while writing data to the medium.

5. Permanent read/write error – This type of fault is a severe error. Both read and write fail permanently.

Temporary timeout errors can be classified into the following two cases, based on the type of request.

6. Temporary no response on read access – This type of fault can be caused by congestion, resulting in SCSI command timeout on a read access. After the congestion disappears, both read and write accesses will succeed.

7. Temporary no response on write access – This type of fault can be caused by congestion, resulting in SCSI command timeout on a write access. After the congestion disappears, both read and write accesses will succeed.

Practically, permanent timeout errors occur regardless of the type of request, read or write. So we have only one class for this type of error.

8. Permanent no response on either read or write access – This type of fault is a device failure resulting in SCSI command timeout. Both read and write requests fail permanently.

2.2 Specifying the Access to Trigger the Fault

Fault location of a SCSI HDD can be identified by the disk device and the failed sector within the disk. In Linux, the disk device can be specified by the major number and minor number of the block device.

The failed sector can be specified by the sector number. However it is difficult for a user-level test program to be aware of the sector number. So, the fault injector also accepts either the file system block number or the inode number to specify the fault location. Those numbers will be automatically converted to the corresponding sector number by the fault injector.

2.3 Design Comparison with Existing Methods

Several methods have already been proposed for injecting faults into block I/O processing, which can be used for evaluation of block I/O error handling code;

- *Linux `scsi_debug` driver* – This is a SCSI low level driver used for debugging. This driver creates a simulated SCSI device using a ramdisk and is capable of injecting various SCSI faults when accessed. However, the condition for injecting faults is limited. For example medium error can only be injected by accessing sector 0x1234 [6].
- *Linux Fault Injection Framework* – This kernel feature, which was merged into 2.6.20 kernel, is used to inject various types of errors into Linux kernel. The framework also supports I/O fault injection, but the fault pattern it can simulate is very limited. For example, it can not inject faults to simulate device timeouts.
- *Using special hardware* – This method uses special hardware for fault injection such as failed HDD. The most precise evaluation result can be obtained since actual hardware is used to inject faults. However, the availability of such hardware is very limited and they are typically expensive.

These existing fault injection methods do not have enough flexibility to inject various faults into the system as intended, which is needed for systematic evaluation of error handling code. The proposed SCSI fault injector described in this paper has a following benefits compared with existing methods.

- *A flexible fault injection trigger* – The SCSI fault injector can trigger a fault on accessing the user specified location of any SCSI device, which is missing in the `scsi_debug` driver.
- *A realistic fault simulation* – The SCSI fault injector can simulate a realistic fault condition by inserting a fault generation code in the SCSI driver, which is missing in the *Linux Fault Injection Framework*.
- *No needs for external hardware* – The SCSI fault injector provides a realistic fault simulation without any external hardware. Also it does not require software modification including Linux kernel.

3 SCSI Fault Injector

In this section, the design and implementation of the SCSI fault injector is explained in detail.

The SCSI fault injector is implemented as a set of SystemTap scripts. SystemTap is used to track information when an I/O request is passed between various layers within the kernel, and to add a hook to inject a fault.

SystemTap provides infrastructure to embed a hook in the kernel dynamically, and to change the value of variables or function return values. Also, SystemTap makes it possible to keep these values in SystemTap variables. By using SystemTap, a fake response from a SCSI device can be created as if a SCSI device had reported an error to the OS [7].

The SCSI fault injector takes a fault pattern and a trigger condition as arguments from a test program. Once started, the injector tracks I/O requests and injects a fault if the condition is met.

The fault injection works in 2 steps.

1. Identify the target SCSI command matching the user-specified condition
2. Inject a fault in the processing of the target SCSI command

3.1 Identifying the target SCSI command

The flow of block I/O processing from a user space test program to the SCSI middle layer is described. Also it is explained how the SCSI fault injector tracks the request to identify the target SCSI command in the flow.

A test program can initiate an I/O request with a read or write system call to the Linux kernel (see Figure 1).

The system call is sent to filesystem layer and eventually translated into a `struct bio`, which is passed to the block I/O layer by the `submit_bio()` function.

The `bio` contains the necessary information for performing I/O. Especially, a `bio` has `bi_sector` and `bi_size`, which represent the logical I/O destination of the target block device and access length respectively at the beginning of `submit_bio()`.

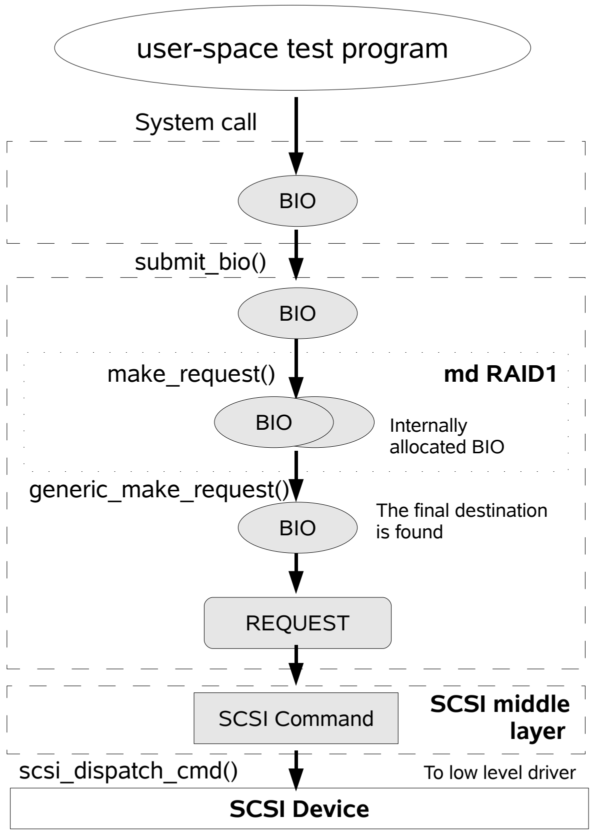


Figure 1: I/O flow from userspace to SCSI device

The software RAID driver resides in the middle of the block I/O layer. If software RAID is used, a `bio` is also generated by the software RAID driver, and the physical I/O destination and access length are stored in the `bio` accordingly.

Then, the `bio` is converted to a `struct request`. At that time, the `bi_sector` and `bi_size` of `bio` are stored in `sector` and `nr_sector` of `request`. `request` also includes a `rq_disk` member which links to `struct gendisk` representing the associated disk. The `gendisk` includes major and minor number of the disk.

Next, the `request` is sent to the SCSI middle layer from the I/O scheduler in a form of `struct scsi_cmnd`, which represents a SCSI command and it is linked to an associated `request`.

The physical I/O destination can be retrieved from `scsi_cmnd` through associated `request`'s `sector` member. A physical I/O access length can be retrieved from `scsi_cmnd`'s `request_bufflen` member. Also, the target device of the SCSI command is found in the associated `request`'s `rq_disk` member.

The SCSI command is issued to SCSI devices through SCSI lower level drivers. The command result is sent to SCSI middle layer and handled accordingly. When the command completed, the result is sent back to the block I/O layer.

The target SCSI command corresponding to the I/O request needs to be identified to inject a SCSI fault triggered by the I/O request represented by a `bio`. The SCSI fault injector will find the `bio` which corresponds with the trigger I/O request from the test program, find the `bio` sent to the SCSI middle layer, and finally find the SCSI command which corresponds to the `bio` requested from the block I/O layer. The target SCSI command is found by tracking the `bio` in the I/O flow described above, and comparing its members with a `scsi_cmnd` (see Figure 2).

Linux block I/O is classified into two types; cached I/O and direct I/O. Both types use `submit_bio()` function and `struct bio` to send a request to the block layer. The inode number of the file corresponding to the I/O request can be identified from `struct bio` for cached I/O or from `struct dio` for direct I/O.

At `submit_bio()`, which is the entry of block layer, the target `struct bio` can be distinguished by comparing block number, access length, inode number, and access direction taken from `bio` or `dio`, with those given by a test program. By tracking the `bio` in the I/O flow, the `bio` which contains the physical I/O destination, can be identified.

Before issuing a SCSI command to SCSI devices, the SCSI fault injector identifies the target SCSI command by comparing information taken from `scsi_cmnd` with that information given by the test program and taken from target `bio` found in the previous step. The compared information includes physical I/O destination, access length, access direction, and device major/minor number.

The `struct scsi_cmnd` representing the target SCSI command identified in this process is saved in a SystemTap variable for later use.

3.2 Injecting SCSI fault

First, the method to inject a fault for a single disk access is explained. Next, we show how each fault pattern de-

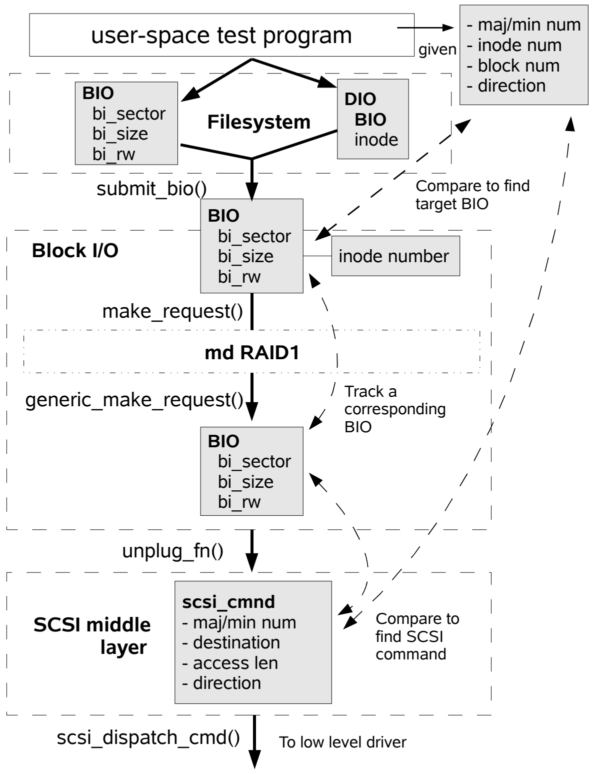


Figure 2: Block I/O tracking from BIO to SCSI command

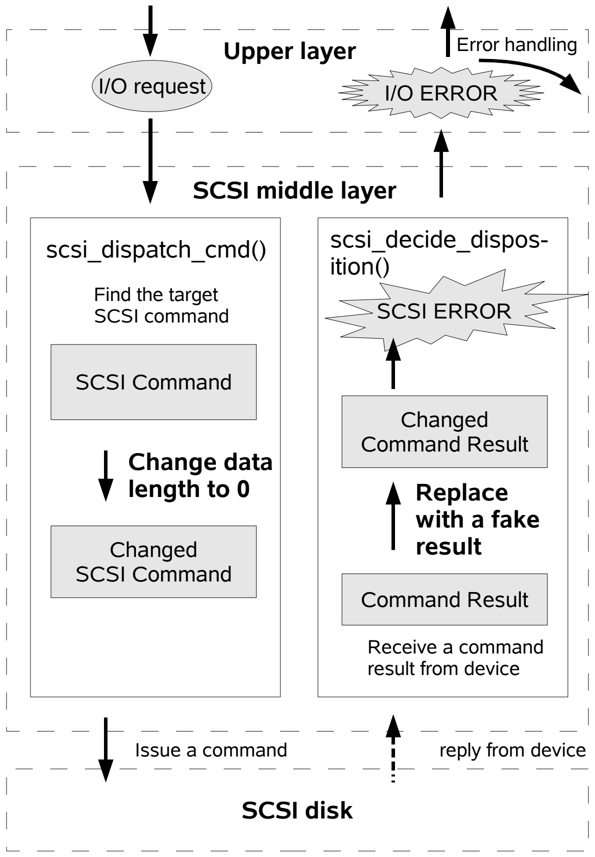


Figure 3: Simulating a fault with error response

scribed in Section 2.1 can be generated by changing the behavior in sequence.

3.2.1 Fault Injection Method

Once the target SCSI command is found, the SCSI fault injector modifies the target SCSI command both before issuing it to the lower layer driver and after returning the result, to simulate a SCSI fault.

The implementation details of "The SCSI device respond with an error" pattern and "The device does not respond" pattern described in Section 2.1 are as follows.

Error Response Case

To simulate device error response, modification of the result of the target SCSI command is needed to fake an error response to the upper layer. Also, actual data transfer generated by the SCSI command should not complete because when a real SCSI fault occurs, the DMA buffer may contain incomplete data (see Figure 3).

For simulating incomplete data transfer to test whether the poisoned data is not sent to the upper layers, the data transfer length of the SCSI command is modified before issuing the SCSI command. When entering the `scsi_dispatch_cmd()` function before issuing a target SCSI command to lower layer driver, the data transfer length in `scsi_cmnd->cmnd` is overwritten to be zero. By this modification, the actual data transfer will not happen as expected.

To simulate error response, the result of target SCSI command needs to be modified before it is sent back to the upper layer. The SCSI command result is analyzed to be sent back to the upper layer in the `scsi_decide_disposition()` function. At the beginning of the function, to identify the target SCSI command, a `scsi_cmnd` which is given as an argument of the function is compared with the `scsi_cmnd` previously saved in a SystemTap variable. If it is the target command, the result stored in `scsi_cmnd` is modified using SystemTap so that the OS detects a medium error which is the most common HDD

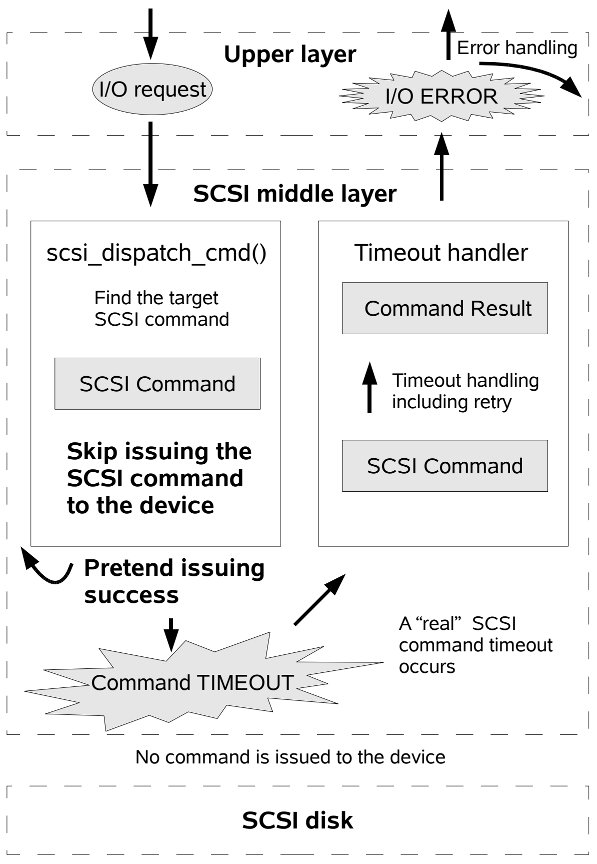


Figure 4: Simulating a fault with no response

error. More precisely, the following error values are stored in `scsi_cmnd` respectively; `(scsi_cmnd.result, scsi_cmnd.sense_buffer[2], scsi_cmnd.sense_buffer[12], scsi_cmnd.sense_buffer[13]) = (2, 3, 11, 4)`. This means "medium error, unrecovered read error, auto reallocated fail" which is one of the medium errors. Then, the changed status will be sent back to the upper layer.

No Response (Timeout) Case

The target SCSI command issued to the low level driver is skipped to simulate no device response (see Figure 4).

All SCSI commands are sent to low level drivers by the `queuecommand` operation in the Linux SCSI middle layer. If the `queuecommand` operation is skipped, the upper layer thinks that SCSI command is issued successfully. But actually it is not issued, consequently the SCSI command results in a timeout.

When the target `scsi_cmnd` is given as an argument of the `scsi_dispatch_cmd()` function, the `queuecommand` operation is skipped by using SystemTap.

3.2.2 Fault Patterns

A single error caused by a SCSI fault (medium error or timeout) can be injected as previously explained. The target SCSI command may be detected several times at `scsi_dispatch_cmd()` after accessing the faulty disk from the test program once, because the error handling code of the upper layer may retry the failed I/O request by an error handling code. The fault patterns described in Section 2.1 can be created by changing the SCSI command manipulation behavior at `scsi_dispatch_cmd()` in sequence as follows.

1. *Temporary read error*

2. *Temporary write error* – When a target SCSI command is detected at `scsi_dispatch_cmd()`, inject a fault just once. If the target SCSI command is detected at `scsi_dispatch_cmd()` again, the fault will not be injected any more.

3. *Read error correctable by write* – In this case, a fault is injected for read access to the target sector until error handling code tries to write data to the sector by tracking `scsi_dispatch_cmd()`. After the error handling code writes to the target sector, no fault will be injected because the error sector is assumed to be corrected.

4. *Permanent read error*

5. *Permanent read/write error* – When a target SCSI command is detected at `scsi_dispatch_cmd()` the fault is injected every time.

6. *Temporary no response on read access*

7. *Temporary no response on write access* – If a target SCSI command is detected at `scsi_dispatch_cmd()`, inject a fault by using "No Response (Timeout) Case" method. If the target SCSI command is detected at `scsi_dispatch_cmd()` again, the fault will not be injected again.

8. *Permanent no response on both read and write access* – A "No Response (Timeout) Case" fault will be injected every time a target SCSI command is detected at `scsi_dispatch_cmd()`.

Thus, all HDD fault patterns can be simulated.

4 Linux Software RAID Evaluation Using SCSI Fault Injection Test Tool

This section describes an example of evaluation using the proposed test tool by applying it to test error handling in the software RAID drivers.

The software RAID drivers were evaluated by injecting various SCSI faults to various RAID drivers and checking if SCSI middle layer and software RAID driver error handling code work properly.

First, the expected behavior of the fault handling code is explained for each fault pattern described in Section 2.1. Next, the test environment and procedure are explained. Finally, the test results and detected bugs are shown.

4.1 Expected Error Handling of Software RAID

The following are the expected error handling behavior of normal RAID array for each of the HDD fault patterns defined in Section 2.1. The test program will check if the system will behave this way when the fault is injected.

1. *Temporary read error* – The SCSI layer detects a read error and the error handler in the RAID driver retries the failed sector. The I/O completes successfully.

2. *Temporary write error* – The SCSI layer detects a write error and the RAID driver's error handler will detach the failed disk immediately. The I/O completes successfully because the write access is issued to both failed disk and redundant disk. The error is recorded in syslog.

3. *Read error correctable by write* – After detecting a read error, the RAID driver retries once, which also fails, the RAID driver may try to write data to the failed sector and re-read from the failed sector. The failed sector will be corrected and the subsequent reads will succeed. The write fix behavior is recorded in syslog.

4. *Permanent read error* – This is the case that a read access fails even after sector correction attempts. As a result, the RAID driver will detach the failed device, read access is issued to another mirror disk, and the I/O completes successfully. The error is recorded in syslog.

5. *Permanent read /write error* – When a fault is triggered by read access, the error handling behavior is same as "Permanent read error." When a fault is caused by write access, it is the same as "Temporary write error."

6. *Temporary no response on a read access* – The SCSI layer detects timeout on target read access and the error handler of the SCSI layer retries the SCSI command. After the SCSI layer gives up, the read error is sent to the RAID driver. After that, the behavior is the same as "Temporary read error" and the I/O completes successfully.

7. *Temporary no response on a write access* – In this case, timeout detection and error handling by SCSI layer is same as "Temporary no response on a read access." After the write access error is sent to the RAID driver, the behavior is the same as "Temporary write error" and the failed disk is detached and the I/O completes successfully.

8. *Permanent no response on both read and write access* – This case is the same as "Temporary no response on a read access" for timeout detection and error handling by SCSI layer, and the read/write request error is sent to the RAID driver. After that the behavior is the same as "Permanent read/write error" and the failed disk is detached and the I/O completes successfully.

4.2 Test Environment

The following test environment was used for the evaluation.

- A server with a single Xeon CPU, 4GB of RAM, and six 36GB SCSI HDDs.
- Fedora 7(i386) running Linux kernel 2.6.22.6
- The tested software RAID drivers were md RAID1, md RAID10, md RAID5, md RAID6, and dm-mirror in the following array conditions.

The hardware fault can occur on any of the disks constituting a software RAID volume. Evaluation was done for each case where the fault was injected when accessing each of the following disks in the software RAID volume.

1. *Active disk of redundant (normally working) array with spare disks* – In this case the failure occurs in one of the disks constructing a RAID array, which has redundancy with spare disks. If a disk is detached from this array as a result of the fault, the RAID array will start recovery using a spare disk.
2. *Active disk of redundant array without spare disks* – In this case, failure occurs in one of the disks constructing a RAID array, which has redundancy, but no spare disk. If a disk is detached from this RAID array as a result of the fault, it will lose its redundancy and become a degraded array.
3. *Active disk of degraded array* – In this case, failure occurs in one of the disks constructing a RAID array, which has no redundancy. If a disk is detached from this array as a result of the fault, the RAID array will collapse because this array has no redundancy.
4. *Active disk of recovering array* – In this case, failure occurs in one of the disks constructing a RAID array, on which the recovery process of the degraded array is running.
5. *Resyncing disk of recovering array* – In this case, failure occurs in a disk which is currently resyncing in the recovery process of the degraded array.

4.3 Test Procedure

The following procedure was performed in the evaluation.

- Install the OS on one of the SCSI disks. Using the rest of the SCSI disks, each of which has a single 8GB ext3 partition, construct a software RAID array.
- Inject various patterns of HDD fault defined in Section 2.1 to various conditions of software RAID array defined in Section 4.2. The fault injections are triggered by accessing a file located in the tested RAID device.
- Check if the target I/O request results in a SCSI error and inspect if the SCSI error is treated properly by error handler of the SCSI layer and the software RAID driver.

Since a set of operations needs to be repeated for all of the many test patterns, the evaluation used the test program to automatically perform the following works.

- Configure one of the five software RAID types (md RAID1, md RAID10, md RAID5, md RAID6, and dm mirror.)
- Set one of the five status of a RAID condition described in Section 4.2.
- Invoke one of eight SystemTap scripts corresponding to one of the HDD fault patterns defined in Section 2.1.
- Generate a SCSI I/O to inject a fault and log the results.
- Loop through all combinations to cover all patterns automatically.

The test program was implemented as a set of shell scripts.

4.4 Bugs Detected in the Software RAID Drivers Evaluation

All combinations of RAID volume types, fault patterns, and RAID volume conditions were tested. The evaluation revealed the following bugs related to error handling of software RAID.

md RAID1 issue is as follows:

- The kernel thread of md RAID1 could cause a deadlock when the error handler of md RAID1 contends with the write access to the md RAID1 array [2].

md RAID10 issues are as follows:

- The kernel thread of md RAID10 could cause a deadlock when the error handler of md RAID10 contends with the write access to the md RAID10 array [2].

- When a SCSI command timeout occurs during RAID10 recovery, the kernel threads of md RAID10 could cause a md RAID10 array deadlock [3].

- When a SCSI error results in disabling a disk during RAID10 recovery, the resync threads of md RAID10 could stall [4].

dm-mirror issue is as follows:

- dm-mirror's redundancy doesn't work. A read error detected on a disk constructing the array will be directly passed to the upper layer, without reading from the other mirror. It turns out that this was a known issue, but the patch was not merged [1].

All these bugs have already been reported to the community and a fix will be incorporated into future kernels. Many bugs found in our evaluation were caused by race conditions between the normal I/O operation and threads in the RAID driver. Probably such bugs were hard to detect. However the proposed test tool using SCSI fault injection was able to find such issues.

5 Conclusion and Future Works

The evaluation result proves that the proposed test tool, which is a combination of the SCSI fault injector and test programs, has the powerful functionality to inject various patterns of HDD fault on various configurations of a software RAID volume to be used for error handler testing. Especially, the flexible user interface of the proposed SCSI fault injector, which existing test methods do not have, realizes a user-controllable fault injection. Also, by applying the proposed test tool, some delicate timing issues in Linux software RAID drivers are found, which are difficult to detect without thorough testing. Therefore, it can be concluded that the proposed test tool using SCSI fault injection is useful for systematic SCSI block I/O test.

The authors are planning to propose the SCSI fault injector to SystemTap community so that the injector becomes available as a SystemTap-embedded tool. Contribution of the injectors for other drivers are welcome as the wider set of fault injectors can form a more generalized block I/O test framework.

Acknowledgments

This project is supported by the Information Technology Promotion Agency (IPA), Japan.

References

[1] Announcement of SCSI fault injection test framework (mail archive). `http://marc.info/?l=linux-raid&m=120036612032066&w=2`.

[2] Bug report of md RAID1 deadlock problem (mail archive). `http://marc.info/?l=linux-raid&m=120036652032432&w=2`.

[3] Bug report of md RAID10 kernel thread deadlock (mail archive). `http://marc.info/?l=linux-raid&m=120289135430654&w=2`.

[4] Bug report of md RAID10 resync thread deadlock (mail archive). `http://marc.info/?l=linux-raid&m=120416727002584&w=2`.

[5] Fault Injection Test project site on SourceForge. `https://sourceforge.net/projects/scsifaultinjtst/`.

[6] `scsi_debug` adapter driver. `http://sg.torque.net/sg/sdebug26.html`.

[7] SystemTap project site. `http://sourceware.org/systemtap/index.html`.

A Survey of Virtualization Workloads

Andrew Theurer
IBM Linux Technology Center
habanero@us.ibm.com

Karl Rister
IBM Linux Technology Center
kmr@us.ibm.com

Steve Dobbelstein
IBM Linux Technology Center
steved@us.ibm.com

Abstract

We survey several virtualization benchmarks, including benchmarks from different hardware and software vendors, comparing their strengths and weaknesses. We also cover the development (in progress) of a new virtualization benchmark by a well known performance evaluation group. We evaluate all the benchmarks' ease of use, accuracy, and methods to quantify virtualization performance. For each benchmark, we also detail the areas of a virtualization solution they stress. In this study, we use Linux where applicable, but also use other operating systems when necessary.

1 Introduction

Although the concept of virtualization is not new [1], there is a recent surge of interest in exploiting it. Virtualization can help with several challenges in computing today, from host and guest management, energy consumption reduction, reliability, and serviceability. There are now several virtualization offerings, such as VMware® ESX [2], IBM PowerVM™, Xen [3][4] technology from Citrix, Virtual Iron, RedHat, and SUSE, and Microsoft® Windows Server® 2008 Hyper-V [5]. As the competition heats up, we are observing a growth of performance competitiveness across these vendors, yielding "marketing collateral" in the form of benchmark publications.

1.1 Why are Virtualization Benchmarks Different?

A key difference in benchmarking a virtualization-based solution is that a hypervisor is included. The hypervisor is responsible for sharing the hardware resources for one or more guests in a safe way. The use of a hypervisor and the sharing of hardware can introduce overhead. One of the goals of benchmarking virtualization is to quantify this overhead and ideally show that virtualization solution X has lower overhead than virtualization solution Y. Another difference in benchmarking virtualization is that the benchmark scenarios can be very different than one without virtualization. Server consolidation is such a scenario. Server consolidation may not typically be benchmarked without the use of a hypervisor (but not out of the realm of possibility; for example, containers may be used). Server consolidation benchmarks strive to show how effective a virtualization solution can host many guests. Since many guests can be involved in this scenario, it may require the use of several benchmarks running concurrently. This concept is not common on traditional benchmarks.

2 Recently Published Benchmarks

The following are virtualization benchmarks with published specifications and run rules that users can replicate in their own environment. These benchmarks strive to set a standard for virtualization benchmarking. In this section, we discuss the strengths and weaknesses of these benchmarks.

2.1 vConsolidate

The vConsolidate benchmark [6] was developed by Intel® to measure the performance of a system running consolidated workloads. As one of the earlier proposals for a virtualization benchmark, vConsolidate was written to prompt the industry to discuss how the performance of a system running with virtualization should be measured.

vConsolidate runs a benchmark for a web server, a mail server, a database server, and a Java™ server, each in a separate guest. There is also a guest that runs no benchmark, which is meant to simulate an idle server. These five guests make up a *consolidation stack unit*, or CSU, as illustrated in Figure 1.

The tester starts with running 1 CSU, obtaining the benchmark score and the processor utilization of the

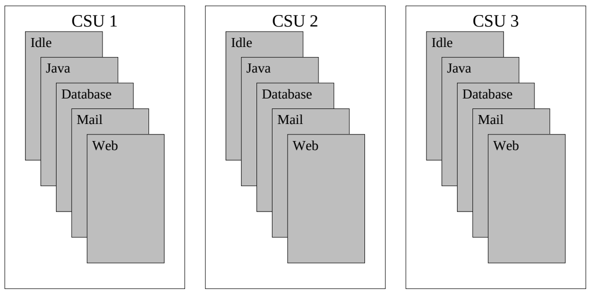


Figure 1: Consolidation stack units

system. The tester does three iterations and then uses the median score and its processor utilization. The tester incrementally adds additional CSUs, recording the benchmark score and processor utilization, until the benchmark score for the set of N CSUs is less than the score for N-1 CSUs, or until all the system processors are fully utilized. The final benchmark score is the maximum of the scores reported along with the number of CSUs and the processor utilization for that score.

The vConsolidate benchmark score is calculated by first summing each of the component benchmark scores across the individual CSUs. The sums are then normalized against the score of the benchmark running on a reference system, giving a ratio of the sum compared to the score of the reference platform. The reference system scores can be obtained from a 1 CSU run on any system. It is Intel's desire to define a "golden" reference system for each profile. The vConsolidate score for the test run is the geometric mean of the ratios for each of the benchmarks. Figure 2 shows sample results from a vConsolidate test run. In this example, the maximum score was achieved at 4 CSUs with a processor utilization of 78.3%.

The reporting of the processor utilization along with the score is not common. Most standard benchmarks simply report the benchmark score and are not concerned with the processor utilization. The processor utilization, however, is a useful metric in characterizing the performance of the system running the consolidated workloads. It can also be useful in spotting performance issues in other areas of the system (for example, disk I/O, network), for example, when the score starts dropping off before the processors get fully utilized, as seen in Figure 2.

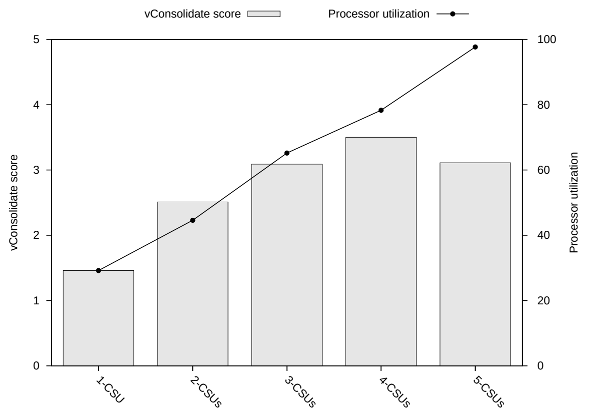


Figure 2: Sample results for vConsolidate

2.1.1 Benchmark Implementation

vConsolidate specifies which benchmarks are run for each of the workloads: WebBench™ [7] from PC Magazine for the web server, Exchange Server Load Simulator (LoadSim) [8] from Microsoft for the mail server, SysBench [9] for the database server, and a slightly modified version of SPECjbb2005 [10] for the Java server.

WebBench is implemented with two programs—a controller and a client. The controller coordinates the running of the WebBench client(s). vConsolidate uses only one client program, which runs eight engine threads. The client and the controller can be run on the same system because neither is processor intensive. The only interface to WebBench is through its GUI, making it difficult to automate.

vConsolidate indirectly specifies which mail server to run. Microsoft's LoadSim only works against a Microsoft Exchange Server, therefore, the mail server must be Exchange Server running on the Windows operating system. Although it runs in a GUI, LoadSim can be easily automated because it can be started from the command line.

vConsolidate runs a version of SysBench that has been modified so that it prints out a period after a certain number of database transactions. The output is then redirected to a file, which is processed by vConsolidate to calculate the throughput score.

vConsolidate has instructions for modifying the SPECjbb2005 source to add a think time, so that the test doesn't run full bore, and to have it print out some statistics at a periodic interval. The output is then redirected to a file that is processed by vConsolidate to calculate the throughput score. The benchmark specifies the version of Java to run: BEA® JRockit® 5.0.

One observation of the benchmark implementation is that most of the workloads are modified or configured such that the benchmark does not run all out, but simulates a server running with a given load. LoadSim is configured to run with 500 users. SysBench is configured to run only four threads. SPECjbb2005 is modified to add a delay. However, WebBench is not modified to limit its load. The delay time and think time are both zero. It may be that this is an oversight of the benchmark configuration. Or it may be that even with no delay and no think time that WebBench does not generate enough load to consume the network and/or processor utilization on the server. That is, WebBench may generate a moderate load even with the delay and think time set to zero.

Certain components of vConsolidate are portable to other applications and other operating systems. WebBench makes standard HTTP requests, consequently it doesn't matter which web server software is run nor the OS on which it runs. The POSIX version of SysBench has support for MySQL, PostgreSQL, and Oracle® database, making it conceivable that vConsolidate could be easily modified to use those databases. SPECjbb will run on any OS that has a Java Virtual Machine. SPECjbb will also run on any Java Virtual Machine (JVM), so although vConsolidate specifies JRockit 5.0, it is conceivable that it could run with any other JVM. Other components of vConsolidate are not portable, e.g., the mail benchmark is LoadSim which requires Microsoft Exchange Server. vConsolidate could be made more portable by not requiring specific software, e.g., JRockit 5.0, and by using more portable, industry standard bench marks, such as SPECweb2005 for the web server and SPECmail2008 for the mail server.

However, vConsolidate has achieved its purpose of getting the discussion started on benchmarking virtualization. Intel is not concerned with trying to make vConsolidate and industry standard benchmark. Comments on how to improve the benchmark are now being fed to the Standard Performance Evaluation Corporation (SPEC) and they are developing an industry standard benchmark for virtualization. In fact, the portability issue is still up for debate among the SPEC members. Some are of the mind that the benchmark should specify the software and benchmarks used so that fair comparisons can be made between different hardware platforms. Others see that specifying the software and benchmarks favors the software selected, does not allow for comparisons to be made between software stacks, and reduces the concept of the benchmark being open.

2.1.2 Running the Benchmark

As mentioned above, the test setup requires client machines to run LoadSim and to run the WebBench controller and client. In the current version of vConsolidate (1.1), each client runs one instance of LoadSim and one instance of the WebBench controller and client. Essentially, one machine runs all of the client drivers for one CSU. This is an improvement over the previous version of vConsolidate that specified separate clients for LoadSim and WebBench. Having one client machine per CSU makes for an easier test setup, not to mention the savings in hardware, energy, and rack space.

The test setup also requires one machine to run the vConsolidate controller. The controller coordinates the start and stop of each test run so that all benchmarks start at the same time via a daemon that runs on the LoadSim and WebBench clients, and on the database and Java servers. The controller kicks off the LoadSim clients first and then delays some time to let the LoadSim clients finish logging into the mail servers before starting the other benchmarks. The controller collects the benchmarks' results when the run is complete.

The vConsolidate controller, LoadSim client, and WebBench controller and client are all Windows applications. As with the servers, this prevents a person from running a single OS other than Windows for the test bed. When testing a Linux environment one must still deal with Windows clients.

vConsolidate has nice automation of the benchmarks. The Linux daemons kickoff and kill a `run.sh` script to start and stop the test, respectively. The `run.sh` script can can have any commands in it. We were able to add commands to `run.sh` to kickoff and kill profilers. The Windows daemon is capable of starting and stopping LoadSim and is able to press the "Yes" button on the WebBench GUI to start WebBench.

On the other hand, vConsolidate itself is hard to automate. Much of this arises from a common mentality when writing a Windows application, which is to assume that everything is controlled by a user sitting in front of the screen. Many Windows benchmarks are written with GUIs that are nice for user interaction but terrible for automating the benchmark. We have a sophisticated, well-developed automation framework for running benchmarks that allows us to, for example, kick off a batch of benchmarks to be run over night and come back in the morning and look at the results. It is difficult to run GUI-based benchmarks from the scripted automation framework.

The vConsolidate controller uses a GUI interface. Thus, you cannot automate a test run suite of 1, 2, and 3 CSUs, for example. Each test run must be started by a user pressing the "Start!" button on the vConsolidate GUI. It would be nice if vConsolidate could be started from a command line, thus enabling it to be scripted.

Our test runs of vConsolidate used a version prior to 1.1 which required the user manually to press a "Stop!" button when the WebBench clients finished. The tester had to keep an eye on the WebBench tests and when they finish, go to the vConsolidate controller and push the "Stop!" button. If the tester was not alert, he could miss the end of the test run and end up with bad results because the other benchmarks would have continued to run beyond the end of the WebBench test and would have logged throughput numbers including the time when WebBench was not running. We managed to automate the test termination by making use of Eventcorder™ [11], a Windows program for automating keyboard and mouse input. vConsolidate version 1.1 addressed this issue and will stop the tests by itself.

As mentioned above, vConsolidate uses WebBench to test the web server. WebBench is also a GUI-based Windows benchmark. For each test run, and for each client, the tester must manually setup the test on the WebBench controller up to the point of clicking the "Yes" button to start the test. The vConsolidate controller does coordinate the pushing of the Yes button on all the WebBench controllers at the same time so the tests start at the same time. However, the tester must manually setup each WebBench controller before each test run. This could be avoided by using a more recent benchmark (WebBench is six years old) that is not GUI-based, such as SPECweb2005.

vConsolidate is still a work in progress. Some issues, such as test termination, have been addressed. Other issues have been deferred to the SPEC virtualization work group. We hope to see many of the issues raised here addressed in future releases of the benchmark.

2.2 VMmark

We must note up front that we have not had any hands-on experience with VMmark™. The following analysis is based on the paper *VMmark: A Scalable Benchmark for Virtualized Systems* [12].

VMmark was developed by VMware®. VMmark was the first performance benchmark for virtualization. Like vConsolidate, VMmark is a benchmark to measure the performance of a system running consolidated workloads. And like vConsolidate, VMmark runs a benchmark for a web server, a mail server, a database server, and a Java server, and has an idle server. VMmark adds another benchmark for a file server. In VMmark's terminology, the combination of the six guests is called a *tile*.

VMmark constrains each of its benchmarks so that it runs at less than full capacity so that it emulates a server that is usually running at less than full capacity. A given tile should then generate a certain amount of load on the system. The tester starts with a test run on a single tile and then incrementally adds tiles until the system is fully utilized.

VMmark specifies which benchmarks to use: Exchange Server Load Simulator (LoadSim) from Microsoft for the mail server, a slightly modified version of SPECjbb2005 for the Java server, SPECweb2005 [13] for the web server, SysBench for the database server, and a slightly modified version of dbench [15] for the file server. The paper mentions using Oracle database and Oracle's SwingBench benchmark, but the latest version of VMmark specifies MySQL and SysBench.

A normal run of LoadSim increases the number of users until a maximum is reached. The benchmark score is the maximum number of users. VMmark is concerned with maintaining a specific load, therefore it keeps the number of users set at 1000 and instead measures the number of transactions executed by the server.

VMmark uses a modified version of SPECjbb2005. SPECjbb2005 is designed to do short runs over an increasing number of warehouses. The VMmark version of SPECjbb2005 is set to run eight warehouses for a long period of time. It is also modified to report periodic status instead of a final score at the end of a run. As with vConsolidate, VMmark requires BEA® JRockit® 5.0.

VMmark uses a modified version of SPECweb2005. The VMmark version of SPECweb changes the think time from ten seconds to two seconds to generate the desired load. The run rules for SPECweb benchmark specify three separate runs, each with a warm up and a cool down period. VMmark, however, does one long run to keep a consistent load on the system. VMmark makes use of the internal polling feature of SPECweb to get periodic measurements of the number of pages accessed. VMmark runs both the SPECweb2005 backend simulator and the web server in the same guest "to simplify administration and keep the overall workload tile size down."

VMmark does not need to modify SysBench. The desired workload can be obtained by setting the number of threads. (The same is true of SwingBench. In the paper, SwingBench is configured for 100 users.)

VMmark uses a modified version of dbench. The benchmark is modified to run repeatedly so that it keeps running during the full VMmark run. The benchmark is also modified to connect to an external program that keeps track of the benchmark's progress and controls the benchmark's resource use so that it generates a predictable load. VMmark also runs a small program that allocates and mlocks a large block of memory to keep the page cache small and force most of the dbench I/O to go to the physical disk.

As with vConsolidate, VMmark requires one client machine per tile. The client machines run Microsoft Windows Server 2003 Release 2 Enterprise Edition (32-bit).

A VMmark test runs at least three hours. Periodic measurements are taken every minute. After the system has achieved a steady state, the benchmark is run for two hours. The two hours are split into three 40 minute runs. The median score of the three runs is used to calculate the score for the tile. This method has an advantage over vConsolidate, since the tester only has to start the test once instead of having to start three separate runs.

The overall benchmark score is determined by the number of tiles run and the individual scores of the benchmarks. Similar to vConsolidate, the individual benchmark scores are first normalized with respect to a reference system. The score for a tile is the geometric mean of the normalized scores for each benchmark in the tile. The overall VMmark score is the sum of the scores for each tile. This is different from vConsolidate, which sums the normalized scores across the CSUs for each benchmark and then takes the geometric mean of each of the benchmark sums for the overall score.

VMmark is less portable than vConsolidate. It specifies which operating systems, server software, and benchmarks are to be used. And, of course, it only runs on VMware® ESX Server. Apparently VMmark is only concerned with comparing hardware platforms.

It is not clear how much effort VMware will spend on updating VMmark. VMware is on the SPEC Virtualization subcommittee and is spending effort there to help build an industry standard virtualization benchmark.

3 Other Benchmark Studies

These studies focus on virtualization benchmarks that do not implement a reference standard, and did not attempt to be adopted as such. They can be considered "ad-hoc," but certainly could be adapted or replicated.

3.1 Server Consolidation on POWER5

In 2006, we conducted a study [17] to help understand the effectiveness of server consolidation using an IBM System p5™ 550 server and Linux guests. The goal was to see how many servers tasked with common services such as web, file and e-mail could be consolidated. New methods were constructed to capture representative loads for such servers and to measure the server consolidation capacity of a virtualization solution.

3.1.1 Workload Definition

A goal of this project was to devise a metric that was tailored to a virtualization solution, not just an application solution. Typically, a single benchmark is designed to be ramped-up to achieve peak throughput while maintaining a specified quality-of-service level, and the metric is a measure of throughput. This study instead used many benchmarks, each configured to inject a fixed load with no ramp-up. Similar to some server consolidation benchmarks, aggregate load for the host was increased by adding more guests and benchmarks. This study achieved this with three benchmark types, targeting a mail, web, and file server. Initially, one instance of the three benchmark types was used (concurrently), injecting load to three guests. Host load was increased iteratively by adding another instance of the three benchmark types along with three more guests. This was repeated until either 50% of host processor utilization was reached or quality of service from any benchmark instance could not be maintained.

In this study, there was a desire to have each server that was consolidated represent a load that might be typical on a stand-alone server. This was achieved by taking a common x86 server, and for each benchmark type, injecting a low load, then ramping up the load until the server reached 15% processor utilization. The load level to achieve 15% processor was our reference load. This reference load was then used to target guests on the POWER5™ system.

3.1.2 Resource Characterization

This workload exhibited a significant amount of disk and network I/O due to both file server and web server benchmarks involved. This flows logically to/from guest and its client. However, there is significant traffic between the guests and the guest designated as the Virtual I/O Server (VIOS). Unless a guest has an I/O adapter dedicated to its exclusive use, the VIOS must handle I/O requests for the guests. This requires an efficient method to move this data as well as adequate processor resources for the VIOS.

3.1.3 Issues and Challenges

This study required an effective way to ensure that the host did not exceed 50% processor utilization. This limitation was placed on this study to ensure ample headroom, should a guest or multiple guests need to accommodate spikes in load. One could have monitored system utilization tools, but there was a concern that all processor cycles may not be accounted for. For example, processor cycles used by the hypervisor, which may not be attributed to one particular guest, may not be accounted for in a host utilization tool. To avoid this problem, half the processors were disabled, ensuring no chance of exceeding 50% host processor utilization

When dealing with benchmarks that have different strategies to begin injecting load, warming up, entering a measurement period, ramping down, and stopping, a method was required that allowed one to accurately measure the load of all three benchmark types. Ideally, one would have individual benchmarks that have their measurement period coincide at the exact same time. In absence of this, one must record the results of each benchmark type individually, while ensuring the load of the other two benchmarks is the load desired and is generated throughout the entire measurement period of the first benchmark. For example, to get results for the web server benchmark, one must ensure the file and mail server benchmarks' steady state began before, and ended after, the web server benchmark's measurement period. This technique should be followed for mail and file server benchmark measurements as well. Because each of these benchmarks can align their measurement period with benchmarks of the same type, all instances of that server type can be measured concurrently. For example, if one is testing 10 sets of consolidated servers (10 web, 10 file, 10 e-mail), three passes of measurements are made. The first pass has all web benchmarks execute in unison, while mail and file benchmarks' steady state begin before, and end, after the web benchmarks' measurement period occurs. The second pass has measurements from file benchmarks, and the third pass has measurements from mail benchmarks.

3.2 LAMP Server Consolidation

One of the primary workloads that we have targeted for consolidation has been the underutilised LAMP (Linux Apache MySQL PHP/Perl/Python/etc.) servers, similar to what a web-hosting company might have. Traditionally, these types of servers are some of the lowest utilized and therefore have a high capacity for consolidation. Additionally, as detailed further when discussing the workload characteristics, these workloads stress many parts of the system stack and are therefore more representative as a generic workload than something that stresses one area.

In order to compare the consolidation capabilities of various virtualization solutions (Xen, VMware, PowerVM, and KVM) on various hardware platforms (x86, x86_64 and POWER), a consolidation benchmark was developed using a LAMP stack application and additional open-source tools. In order to ensure cross platform availability, all stack components were built from source (with the exception of the Linux kernel and the distribution being tested). The benchmark consists of two data collection steps that combine to form a consolidation metric which is the number of underutilized servers that can be consolidated as guests on a single virtualized system. The first data collection step is to run the target workload on a system that is representative of the historically underutilized servers that are to be virtualized. Existing surveys and reports by industry consulting services provide metrics such as the average processor utilization of the underutilized servers; these processor utilization metrics are used to determine the injection rate (the amount of work for the benchmark drivers to "inject" into the test system) that the client drivers should use to drive the baseline system. This injection rate combined with the workload itself defines the baseline workload that is then run simultaneously on each of the virtualized guests on the test system in the second data collection step. The more guests that a test system can host while maintaining similar performance and quality-of-service to the baseline system, the higher the consolidation metric it achieves.

In order to understand the workload characteristics that the virtualized system must be capable of handling, the characteristics of the workload when running on the baseline system must first be understood. A LAMP stack application consists of the Apache web server, the MySQL database, and an interpreted language (PHP in this particular example) all running on top of the Linux operating system. These applications inherently have properties that need to be understood.

The Apache web server accepts connections from client systems and responds appropriately depending on the request. Servicing a simple request with non-dynamic code will usually consist of reading the requested object from disk or fetching it from cache, and returning it to the client. Servicing a more complicated request, such as PHP code, will involve fetching the script from disk or cache, invoking the interpreter, and then returning the generated content to the client. All requests are logged, which consists of a sequential I/O write that will eventually be forced to flush to disk.

When the dynamic code interpreter is invoked, the execution possibilities are quite expansive, but there are some basic concepts that can be summarized: first, the dynamic code may have never been requested before, which means that it will have to be compiled before execution; second, if the dynamic code has been executed before and was not cached, such as by a PHP accelerator, it will have to be compiled again; third, in a LAMP scenario, the dynamic code will most likely involve connecting to the database and waiting for data to be returned or processed. Any compilation of dynamic code will require processor cycles so caching of the compiled code is desirable in order to reduce processor utilization.

When requests are made to the database from the dynamic code execution, a combination of reads and writes are likely, and the I/O pattern can vary depending on the operation required. Database read queries will likely result in small random, read I/O operations. Database write queries will likely consist of a combination of random I/O writes for the data and sequential I/O writes for the log.

When you combine the characteristics of the various stack components you get the following: TCP/IP socket connections handled by the web server, disk reads and writes by the web server, processor intensive compilation and execution of dynamic code, and disk reads and writes (both sequential and random) from the database engine. For an underutilized system, the magnitude of these characteristics is relatively low and of little concern, however, in a consolidation scenario that can change.

When large numbers of guests are consolidated on a single system, the workload properties of the consolidated guests are stacked on top of each other. In the case of the TCP/IP connections, due to the fact that these are lightly loaded servers being consolidated, the number of requests are quite low and therefore not an overriding factor in the test. The processor utilization is cause for some concern, but it is one of the finite resources that consolidation is trying to maximize, so it will inherently run out at some point anyway. The real area of concern is the disk I/O that the workload drives. In an ideal world of spinning disks, all I/O would be sequential in order to minimize the penalty of head seeks. Unfortunately, this is not the case, and most workloads, such as this LAMP stack application, have a mix of sequential and random I/O patterns. Virtualization exacerbates the problem, though because due to the fact that all guests have their own carved out disk space and the drive heads are forced to seek more and more with each added guest. This means that drive I/O capacity will decrease with each added guest until the I/O pattern becomes completely random, and the ability to complete requests will be bounded by the random I/O capabilities of the storage system. In some of the consolidation studies we have done, on systems with large amounts of processor power (large numbers of fast processor cores) the maximum consolidation factor could be not reached due to the I/O capacity of available storage systems being maxed out well before processor horsepower was exhausted. A reality of today's storage systems is that systems capable of high random I/O performance are quite expensive, and for workloads with large amounts of I/O, success of consolidation will depend greatly on the ability to pair the target virtualization system with the the proper storage.

3.3 Processor Scalability Workloads

For the 2006 Ottawa Linux Symposium, we conducted several scalability tests [18]. These workloads were designed specifically to measure and improve the scalability of hypervisors and the guests they manage. Unlike other workloads discussed here, these are more synthetic, and are designed to isolate and study specific scalability problems. They do not necessarily strive to represent typical virtualization scenarios, but try to target extreme scalability situations.

3.3.1 Two Types of Scalability

Typically, one would measure scalability by testing a scenario with N resources (in our case, processors), then testing again with N*M resources, and observing the relative increase in throughput. A benchmark would normally run "all out" to maximize the use of the resource. Without virtualization, this is fairly straight forward. One would boot with one processor enabled, run a test (for example dbench), record the result, then boot with N processors and run the test again. The scalability would be the N-way throughput divided by the 1-way throughput. There are, of course, variations of this theme—for example, using 1 and N sockets or NUMA nodes instead of processors. For virtualization scalability, we alter this method slightly in order to study two types of scenarios: scalability of a single guest and scalability of many guests.

3.3.2 Single Guest Scalability

Scaling just one guest involves assigning the guest one processor resource (or socket, or NUMA node), testing, then assigning the guest N processors (or sockets, NUMA nodes) and testing again. This is probably the easiest way to test virtualization, as it follows the methodology that one would use for traditional scalability testing. With just one guest, there is no plurality of benchmarks to manage and synchronize, and no sets of results to aggregate. One complication is that there may be a service and/or I/O guest which should also be accounted for.

3.3.3 Multiple Guest Scalability

For scaling many guests, we start with one guest, assign a fixed processor resource (core, socket, etc.), then test with N guests, running the same benchmark at the same time, each assigned the same resource amount (core, socket, etc.), such that we have enough guests to maximize that resource. Our goal is to analyze the scalability of the hypervisor, and not necessarily the scalability of the guests. In the previous scenario, any scalability inhibitor within the guest could affect the overall scaling, while in this scenario that is not true. In this type of scalability test, we do have a little more work to do. Because we are running many guests, one must ensure that all of the benchmarks begin and end at the same time. One must also sum the benchmarks' results into one result.

3.4 SPEC and Virtualization

The Standard Performance Evaluation Corporation (SPEC) is a non-profit corporation that creates, maintains and endorses a standardized set of benchmarks. Members of SPEC include many computer hardware and software manufacturers across the world. Recently, SPEC has formed a new sub-committee, Virtualization, to create and maintain virtualization benchmarks.

3.4.1 SPECvirt_sc2009

SPECvirt_sc2009 is the first virtualization benchmark from SPEC. As of this writing, this benchmark is still under development. The characteristics, methodologies, and run rules described here are subject to change. The SPEC virtualization sub-committee contains members who have been involved in many of the virtualization benchmarks outside of SPEC, including the benchmarks mentioned earlier in this paper. As such, SPECvirt_sc2009 takes influence from these benchmarks, both from their methodologies and from the lessons learned.

SPECvirt_sc2009 follows a server consolidation scenario, similar to other benchmarks described here. The benchmark uses the same *tile* concept as VMmark, similar to the vConsolidate CSU and server sets on the POWER5 Server Consolidation Study. The benchmark metric is the number of tiles one can run while maintaining the quality of service of all benchmarks participating. However, there are some attributes that differentiate this benchmark from the others previously mentioned

SPEcvirt_sc2009 also addresses the issue of portability. There are no restrictions on the type of architecture or operating system. All services tested can be implemented with proprietary and/or open-source solutions. The client driver also makes no requirement of architecture or operating system.

The proposed SPECvirt_sc2009 tile consists of 6 guests: a web server, mail server, application server, database server, infrastructure server, and an idle server. To drive a single tile, three SPEC benchmarks are used: SPECweb2005, SPECmail2009, and SPECjAppServer2004. SPECweb2005 injects requests to a web server guest. A back-end database simulator, or Besim, resides on the infrastructure guest, simulating database requests from the web guest. In addition, the web server has part of its document root NFS mounted from the infrastructure server, so some HTTP requests are served by just the web guest, and some involve the infrastructure guest as well. The SPECmail2009 benchmark injects requests using the IMAP mail protocol to the mail server, driving load on the mail server guest only. The SPECjAppServer2005 benchmark injects requests to the application server. The application server guest requires a database, located on the database server guest. The SPECjAppServer2004 benchmark drives load on both the application server and database server. The idle guest is used to represent the overhead of a guest which is running but not actually doing anything. It does not interact with any of the other tile components.

One of the big issues for other virtualization benchmarks, which had heterogeneous guests and load generators, was synchronization. When using the original SPEC benchmarks to prototype SPECvirt, one encounters similar issues. It is possible to configure each benchmark type such that they begin and end their measurement period at nearly the same time, however it can not be done with a high level of confidence. The SPEC virtualization sub-committee is changing these benchmarks to support a more tightly controlled apparatus, ensuring the benchmarks' start and stop times coincide exactly.

SPECvirt_sc2009 aims to introduce a characteristic which is not too prominent on most of the other virtualization benchmarks: intra-guest network communication. The benchmark introduces dependencies which will require network communication between guests within a tile. A portion of the document root for the web server guest is served by the infrastructure server guest, generating a significant amount of NFS traffic between these two guests. SPECjAppServer is required to use two guests, one for the application server and one for the database server, so that communication between the two services is between guests, compared to both services on the same guest communicating over loopback.

4 Considerations for Future Virtualization Benchmarking

After working with these benchmarks, we would like to propose some ideas for consideration when designing, running, and analysing virtualization workloads. These ideas are not strictly for benchmarking or marketing collateral, but also for general testing of virtualization.

4.1 Simplification

Many of today's benchmarks are already quite complicated. Configuring a multi-tier benchmark such as SPECweb2005 or SPECjAppServer2004, can be quite a task by itself. Combining several instances of benchmarks like this can be daunting at first. We propose leveraging the concept of virtual appliances for both the server under test and the client driver system. Maintaining a library of virtual appliances allows the user to spend more time on evaluation and analysis of the total solution instead of dealing with details of service implementation, OS configuration, and related tasks.

4.2 New Workload types

Our strategy here is to test more of the emerging features that virtualization provides. Currently we have embarked on just one of the common scenarios for virtualization but have not fully explored that area yet. For example, most of the server consolidation workloads use benchmarks that have a fixed load level. However, in real-world situations, all guests do not run a constant load all the time. The dynamic nature of guests' resource requirements needs to be explored. A hypervisor's ability to accommodate these changes may vary greatly from one solution to another.

Another area to look into could center around the RAS features that virtualization offers. Scenarios that include BIOS and other software updates, requiring the live migration of guests during such operations, could be tested. Other serviceability scenarios could be considered, like impact of guest provisioning on hosts that have active guests. Benchmarking scenarios like these may not be traditionally covered, but with a much more dynamic data-center, these situations need to be studied more closely.

In addition, one might want to explore the concept of whole data-center management using virtualization. This builds on previous concepts like server consolidation and availability, but takes it further. For example, a benchmark may evaluate the performance per watt of an entire data-center. A virtualization management solution and the hypervisors in use can significantly impact the performance and power consumption, especially when the load is dynamic, driving actions like migration to and from hosts to meet quality of service guarantees while conserving the most energy possible.

Another area that probably needs attention is the desktop virtualization scenario. This solution is quickly becoming very competitive, and drawing any conclusions from a server consolidation benchmark may not be prudent. Desktop virtualization has significantly different characteristics than a server consolidation scenario. Desktop users' perceived performance, rather than throughput, may be far more important to this solution.

5 Conclusions

This paper surveys various virtualization benchmarks, comparing their strengths and weaknesses. The art of virtualization benchmarks is in its infancy, however, we believe it is making progress. We are still seeing growing pains in most implementations, as we try to go beyond what traditional benchmarking scenarios have done. Issues such as overall complexity, heterogeneous workload control, quality of service guarantees, and verification all need improvement. These benchmarks have also just begun to simulate the vast number of present and future usage cases that virtualization introduces. We are confident, as long as there is a need for improving virtualization, there will be a drive to improve these benchmarks.

6 Trademarks and Disclaimer

References

[1] IBM Corporation, Virtualization, `http://www-03.ibm.com/servers/eserver/zseries/virtualization/features.html`

[2] VMware Corporation, Build the Foundation of a Responsive Data Center, `http://www.vmware.com/products/vi/esx/`

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, Andrew Warfield, *Xen and the Art of Virtualization* SOSP'03, October 19–22, 2003, Bolton Landing, New York, USA.

[4] Ian Pratt, Keir Fraser, Steven Hand, Christain Limpach, Andrew Warfield, *Xen 3.0 and the Art of Virtualization,* Ottawa Linux Symposium 2005

[5] Microsoft Corporation, Virtualization and Consolidation, `http://www.microsoft.com/windowsserver2008/en/us/virtualization-consolidation.aspx`

[6] Jeffrey P. Casazza, Michael Greenfield, Kan Shi, *Redefining Server Performance Characterization for Virtualization Benchmarking,* `http://http://www.intel.com/technology/itj/2006/v10i3/7-benchmarking/6-vconsolidate.htm`

[7] Ziff Davis Media, PC Magazine benchmarks, `http://www.lionbridge.com/lionbridge/en-US/services/outsourced-testing/zdm-eula.htm`

[8] Microsoft Exchange Server 2003 Load Simulator, `http://www.microsoft.com/downloads/details.aspx?FamilyId=92EB2EDC-3433-47CA-A5F8-0483C7DDEA85&displaylang=en`

[9] Alexey Kopytov, SysBench: a system performance benchmark, `http://sysbench.sourceforge.net`

[10] Standard Performance Evaluation Corporation (SPEC), SPECjbb2005, `http://www.spec.org/jbb2005`

[11] CMS Eventcorder, `http://www.eventcorder.com`

[12] Vikram Makhija, Bruce Herndon, Paula Smith, Lisa Roderick, Eric Zamost, Jennifer Anderson, *VMmark: A Scalable Benchmark for Virtualized Systems,* `http://www.vmware.com/pdf/vmmark_intro.pdf`

[13] Standard Performance Evaluation Corporation (SPEC), SPECweb2005, `http://www.spec.org/web2005/`

[14] Oracle Corporation, SwingBench, `http://www.dominicgiles.com/swingbench.html`

[15] Andrew Tridgell, dbench benchmark, `http://samba.org/ftp/tridge/dbench/`

[16] VMware, Measure Virtualization Performance with Industry's First Benchmark, `http://www.vmware.com/products/vmmark/`

[17] Yong Cai, Andrew Theurer, Mark Peloquin *Server Consolidation Using Advanced POWER Virtualizatin and Linux,* `http://www-03.ibm.com/systems/p/software/whitepapers/scon_apv_linux.html`

[18] Andrew Theurer, Karl Rister, Orran Krieger, Ryan Harper, Steve Dobbelstein, *Virtual Scalability: Charting the Performance of Linux in a Virtual World*, Ottawa Linux Symposium 2006

Thermal Management in User Space

Sujith Thomas
Intel Ultra-Mobile Group
`sujith.thomas@intel.com`

Zhang Rui
Intel Open Source Technology Center
`zhang.rui@intel.com`

Abstract

With the introduction of small factor devices like Ultra Mobile PC, thermal management has gained a higher level of importance. Existing thermal management solutions in the Linux kernel lack a standard interface to user space. This mandates that all the thermal management policy control needs to reside in the kernel. As more and more complex algorithms are being developed for thermal management, it makes sense to allow moving the *policy control decisions* part into user space and allow the kernel to just facilitate these decisions.

In this paper, we will introduce a generic solution for Linux thermal management, which usually contains a user application for policy control; a generic thermal sysfs driver, which provides a set of platform-independent interfaces; native sensor drivers; and device drivers for thermal monitoring and device throttling. We will also take a look at the software stack of Intel's Menlow platform, where this solution is already enabled.

1 Thermal Modeling

Even though there are many thermal modeling concepts out there, the crux still remains the same.

- There are sensors associated with devices. The devices have various throttle levels and by putting the device into a lower performance state, the temperature of the device as well as the overall platform will decrease.
- There may be provisions for programming sensor trips to send notifications to the monitoring application.
- There may be a fan in the platform and it may have multi-speed control.

Provided these hardware features are available, how can the software manage thermals for a platform? This can be implemented either in kernel space or in user space. The kernel-space implementation by using the framework is enough as long as there are only a few thermal contributors, and mainly the CPU.

But with the Ultra Mobile PCs (UMPC) and Mobile Internet Devices (MID), the CPU is no longer the major thermal contributor. There may be cases where, over a period of time, multiple devices' contributions cause the platform temperature to rise significantly. So, the real challenge here is to choose the right device(s) and to pick up the right performance levels.

2 The Concept of 'Thermal management in User Space'

Thermal management in user space would imply that all the policy decisions will be taken from user space and the kernel's job would be only to facilitate those decisions. This model gives us the these advantages:

- The algorithms can scale well from simple scripts to complex algorithms involving neural networks.
- The kernel is freed from consuming CPU cycles for non-critical tasks. Response to a non-critical thermal scenario, which is the most common case, is not immediately required—that is, we don't have to account for decisions which are in milliseconds or microseconds. This is because raising the platform temperature about one degree Celsius takes around 20s–30s on most platforms. So moving thermal management from kernel space, where we have critical things to do, is the right thing to do.
- This also guarantees that the same application will work on different platforms even though the thermal modeling is different at the hardware level.

But for the thermal management to shift to user space, applications still need to get support from the kernel. That's the role the generic thermal management framework plays.

3 ACPI vs. Generic Thermal Management

The ACPI 2.0 thermal model was a good start for thermal management. But for new platforms with small form factors like the Mobile Internet Devices (MID), this model is no longer sufficient. Some of the reasons are:

- ACPI proposes active trip points, but there may not be any fans on the handhelds.
- ACPI assumes that the CPU is the major thermal contributor and doesn't discuss other thermal contributors.
- ACPI doesn't support sensors with programmable AUX trip points.

Even with these limitations, many platforms still use ACPI because of its other benefits. As a matter of fact, the generic thermal management is not a thermal model itself; instead it complements existing models like ACPI.

The generic thermal management solution was designed to support thermal models (like ACPI 2.0) and to go beyond, to complement such models with proprietary platform-based sensors and devices. Intel's Menlow platform is a good example of using ACPI as the backbone for thermal management. Along with that, it uses sensors with programmable AUX trip points and it can even throttle the memory controller. A case study in the latter part of this paper illustrates this solution.

4 Generic Thermal Management Architecture

The generic thermal management has these key components:

- Thermal zone drivers for thermal monitoring and control.
- Cooling device drivers for device throttling.
- An event framework to propagate the platform events to user-space applications.
- A generic thermal sysfs driver which provides a set of platform-independent interfaces.

Figure 1 shows the software stack of the generic thermal solution.

4.1 Thermal Zone Drivers

A thermal zone, by definition, not only gives the temperature reading of a thermal sensor, but also gives the list of cooling devices associated with a sensor. The driver or application may in turn control these devices to bring down the temperature of this thermal zone. The thermal zone driver abstracts all the platform-specific sensor information and exposes the platform thermal data to the thermal sysfs driver. This may include data like temperature and trip points. In addition, it also binds cooling devices to the associated thermal zones. This driver is also responsible for notifying user space about thermal events happening in the platform.

4.2 Cooling Device Drivers

The cooling device drivers are associated with thermal contributors in the platform. The cooling device drivers can register with the generic thermal sysfs driver, thus becoming the part of platform thermal management. By registering, these drivers provide a set of thermal ops that they can support, like the number of cooling states they support and the current cooling state they are in. The generic thermal sysfs driver will redirect all the control requests to the appropriate cooling device driver when the user application sets a new cooling state. It is up to the cooling device driver to implement the actual thermal control.

4.3 Eventing Framework

Events will be passed from kernel to user space using the netlink facility. The applications may use `libnetlink` to receive these events and to do further processing.

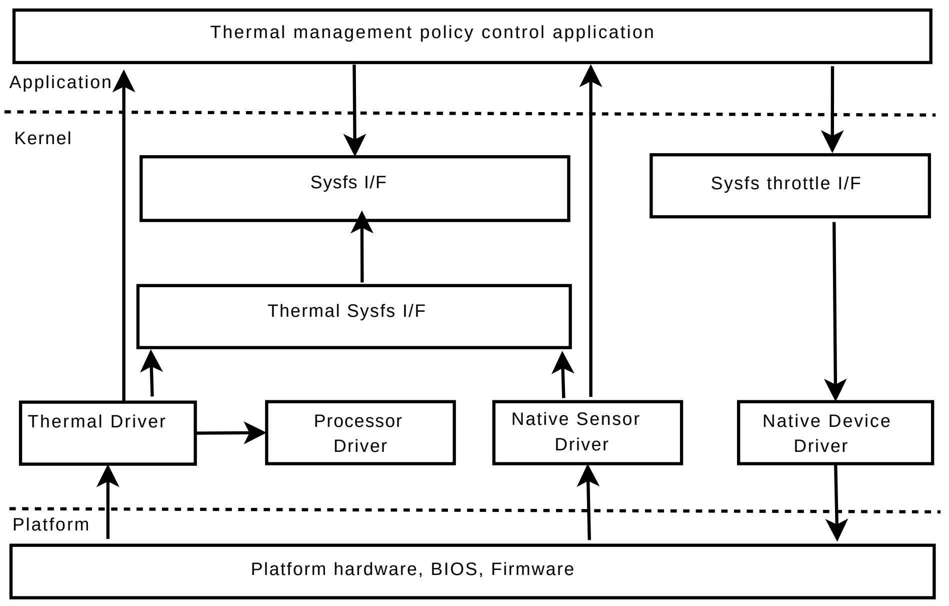


Figure 1: Generic thermal management architecture

4.4 Generic Thermal sysfs Driver

The generic thermal sysfs driver is a platform-independent driver which interacts with the platform-specific thermal zone drivers and cooling device drivers. This driver, in turn, builds a platform-independent sysfs interface (or I/F) for user space application. It mainly works on device management and sysfs I/F management for registered sensors and cooling devices.

4.4.1 Device Management

The thermal sysfs driver exports the following interfaces

- `thermal_zone_device_register()`
- `thermal_zone_device_unregister()`
- `thermal_cooling_device_register()`
- `thermal_cooling_device_ unregister()`
- `thermal_zone_bind_cooling_ device()`
- `thermal_zone_unbind_cooling_ device()`

for the thermal zone drivers and cooling device drivers to register with the generic thermal solution and to be a part of it. The thermal sysfs driver creates a sysfs class during initialization and creates a device node for each registered thermal zone device and thermal cooling device. These nodes will be used later on to add the thermal sysfs attributes.

The `bind/unbind` interfaces are used by the thermal zones to keep a mapping of the cooling devices associated with a particular thermal zone. Thermal zone drivers usually call this function during registration, or when any new cooling device is registered.

4.4.2 Sysfs Property

The generic thermal sysfs driver interacts with all platform-specific thermal sensor drivers to populate the standard thermal sysfs entries. Symbolic links are created by the generic thermal driver to indicate the binding between a thermal zone and all cooling devices associated with that particular zone. Table 1 gives all the attributes supported by the generic thermal sysfs driver.

The generic thermal management uses a concept of *cooling states*. The intent of a cooling state is to define thermal modes for supporting devices. The higher the cooling state, the lower the device/platform temperature would be. This can be used for both passive and active cooling devices. It's up to the cooling device driver to

Sysfs	Location	Description	RW
type	/sys/class/thermal/thermal_zone[0-*]	The type of the thermal zone	RO
mode	/sys/class/thermal/thermal_zone[0-*]	One of the predefined values in [kernel, user]	RW
temp	/sys/class/thermal/thermal_zone[0-*]	Current temperature	RO
trip_point_[0-*]_temp	/sys/class/thermal/thermal_zone[0-*]	Trip point temperature value	RO
trip_point_[0-*]_type	/sys/class/thermal/thermal_zone[0-*]	Trip point type	RO
type	/sys/class/thermal/cooling_device[0-*]	The type of the cooling device	RO
max_state	/sys/class/thermal/cooling_device[0-*]	The maximum cooling state supported	RO
cur_state	/sys/class/thermal/cooling_device[0-*]	The current cooling state	RW
cdev[0-*]	/sys/class/thermal/thermal_zone[0-*]	Symbolic links to a cooling device node	NA
cdev[0-*]_trip_point	/sys/class/thermal/thermal_zone[0-*]	The trip point that this cooling device is associated with	RO

Table 1: Thermal sysfs file structure

implement the cooling states. In most of the cases, it may map directly to the power modes of the device. But in some other cases, it may not. The CPU is the example of when power modes are controlled by P states, but thermal is controlled by a combination of P and T states.

Besides the generic thermal sysfs files, the generic thermal sysfs driver also supports the `hwmon` thermal sysfs extensions. The thermal sysfs driver registers an `hwmon` device for each type of registered thermal zones. With the `hwmon` sysfs extensions, an attempt has been made to support applications which use the `hwmon` style of interfaces. Currently, using this interface, the temperature and critical trip point of the platform are exposed.

Table 2 shows the `hwmon` thermal sysfs extensions.

5 User Space Policy Control

The generic thermal management framework enables thermal management applications to collect all the relevant thermal data. The data will be pre-processed and then passed on to the intelligent algorithm where the throttling decisions are taken. The algorithm may consider user preferences before executing any decisions, again through the generic thermal management framework.

Here are the operations which applications can perform using the generic thermal management framework.

- Enumerate the list of thermal sensors in the platform.
- Enumerate the list of thermal contributors (CPU, Memory, etc.) in the platform.
- Enumerate the list of active cooling devices (fans) in the platform.
- Enumerate the thermal zones (to get device sensor associations) in the platform.
- Get sensor temperatures and trip points of various sensors.
- Set the threshold trip points if the underlying platform supports this feature.
- Get notifications on thermal events happening in the platform.
- Get exclusive control of any thermal zone in the platform.

6 Thermal Management on Intel's Menlow Platform

Menlow is Intel's handheld platform for the 2008 time frame. It is a small form-factor device (screen size of about 5 inches), which makes its thermal management a challenge. The goal of the solution was that at any time, the skin temperature (top and bottom) should be below 45°C. The other challenge was that the CPU was not the major thermal contributor. There are other devices, like the memory controller and communication devices, which contributed equally to the platform's skin temperature. There was clearly a need for a complex algorithm to perform the thermal management by throttling the devices at the same time, while not compromising the performance.

6.1 Why ACPI Was Not Enough…

Menlow's thermal management solution leverages many of the ACPI standards available on the platform. But

Sysfs	Location	Description	RW
Name	/sys/class/hwmon/hwmon[0-*]	Same as the thermal zone 'type'	RO
Temp[1-*]_temp	/sys/class/hwmon/hwmon[0-*]	Current temperature value	RO
Temp[1-*]_crit	/sys/class/hwmon/hwmon[0-*]	Critical temperature value	RO

Table 2: `Hwmon` support file structure

relying only on the ACPI standards was not enough because sensors available in the platform were capable of doing more things than in ACPI 2.0. Hence the concept of *generic thermal management* was proposed.

The Menlow platform has many thermal sensors attached to the platform's embedded controller. The embedded controller firmware was in charge of reading the temperature from the sensors. These sensors had the additional capability of programming the AUX trips, wherein the application can program the upper and lower thresholds, based on the current temperature. Whenever the temperature exceeds any of these thresholds, the application will get an event and can make a decision based on the user policy. ACPI 2.0 didn't have support for AUX trip point programming. Generic thermal management was used to complement the ACPI standards.

6.2 How the Generic Thermal Management Works on Menlow Platform

Menlow's thermal management is the first use of the generic thermal solution. Thermal management on Menlow is made up of these components.

- An intelligent user-space application which can make throttling decisions based on thermal events it receives.
- ACPI thermal management, which has its thermal zone driver (ACPI thermal driver) and cooling device drivers (processor, fan, and video driver) registered with the thermal sysfs driver.
- `intel_menlow` platform driver, which provides required, extra thermal management, such as memory controller throttling and AUX trip point programming.
- ACPI BIOS which has objects for controlling the processor's P and T states.
- Embedded controller firmware which reads the sensor temperature and programs the temperature thresholds.

Figure 2 shows the thermal sysfs architecture on the Menlow platform.

6.3 Thermal Zone Driver on Menlow

The ACPI thermal driver plays a key role on Menlow, which is registered with the thermal sysfs driver to export the temperature and trip point information to user space. The ACPI thermal driver does the thermal management to some extent—that is, it controls the processor P and T states whenever the temperature crosses the configured `_PSV` temperature. The generic thermal management provides a way for a user-space application to override kernel algorithm using the sysfs-exported file named `mode`. If ACPI thermal zones' modes are set to "user," ACPI thermal zones will no longer follow thermal policy control. Instead, they will only export temperature change events to user space through netlink. Whenever the application exits, it can give back the thermal management control by writing "kernel" into the file `mode`. This was needed to guarantee the mutual exclusivity of the thermal management between the kernel and the user-space application.

6.4 Menlow's Native Driver

`Intel_Menlow` is a platform-specific driver which handles:

- AUX trip point programming for platform thermal sensors.
- Throttling of memory controller.

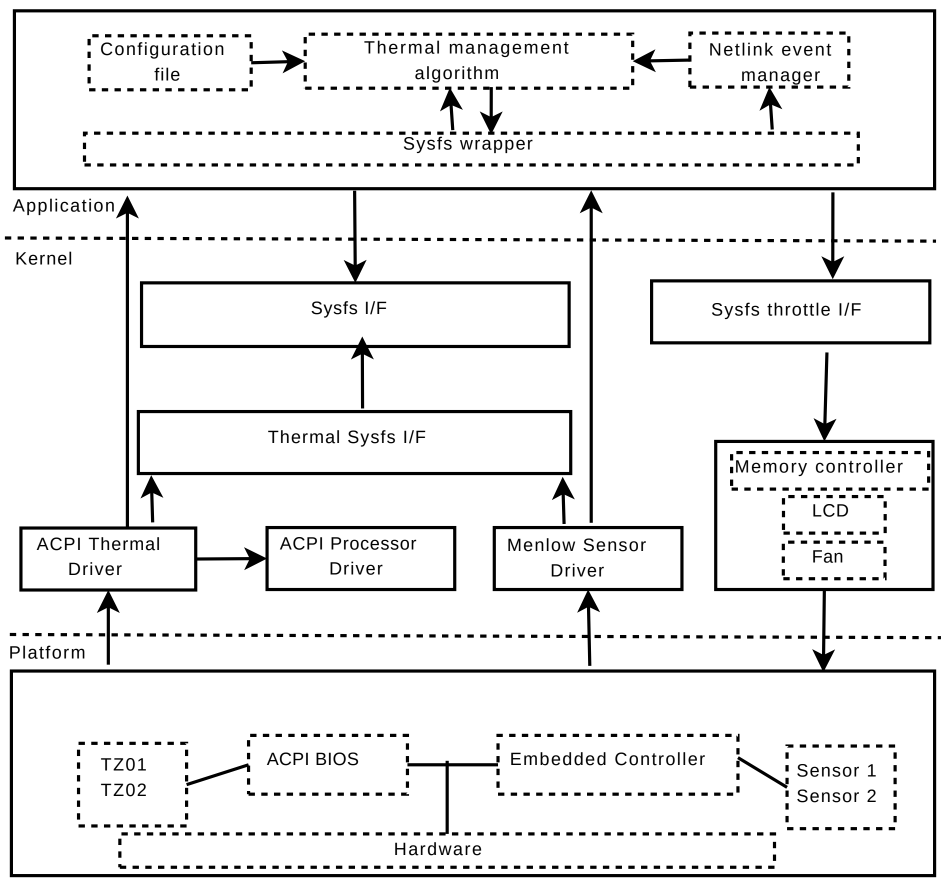


Figure 2: Menlow using the generic thermal management

6.5 Cooling device driver on Menlow

The following cooling devices are registered with the thermal sysfs driver on Menlow:

- Memory controller, which controls the temperature by throttling the memory bandwidth.
- ACPI processor cooling state is a combination of the processor P-state and T-state. The ACPI CPU frequency driver prefers to reduce the frequency first, and then to throttle.
- The ACPI fan driver supports only two cooling states: state 0 means the fan is off, state 1 means the fan is on.
- ACPI video throttles the LCD device by reducing the backlight brightness levels.

7 Conclusion

For handheld devices it is viable to move the thermal management to user space applications. By doing so the application are given the freedom to implement the algorithm that would be the best to handle thermals for a particular class of devices. The job of the kernel would be limited to delivering events and exposing device specific throttle controls. This approach can be used on platforms with ACPI, without ACPI, or to compliment thermal models like ACPI.

8 Acknowledgment

This paper is based on "Cool Hand Linux—Handheld Thermal Extensions" co-written by Len Brown and Harinarayan Seshadri. We would also like to acknowledge Nallaselan Singaravelan, Vinod Koul, and Sailaja Bandarupali of the Ultra Mobility Group, Intel Corporation, for their valuable contributions.

9 References

- "Cool Hand Linux—Handheld Thermal Extensions" by Len Brown and Harinarayan Seshadri, *Proceedings of the Linux Symposium*, Ottawa, Canada, 2007.

- ACPI Hewlett-Packard, Intel, Microsoft, Phoenix, Toshiba. *Advanced Configuration and Power Interface 3.0b*, October, 2006. `http://www.acpi.info`

A Model for Sustainable Student Involvement in Community Open Source

Chris Tyler
Seneca College
chris.tyler@senecac.on.ca

Abstract

A healthy community is the lifeblood of any open source project. Many open source contributors first get involved while they are students, but this is almost always on their own time. At Seneca College we have developed an approach to sustainably involving students in open source communities that has proven successful in a course setting.

This paper outlines Seneca's approach and discusses the results that have been obtained with it. I will examine the key factors for successful student integration into open source communities and steps that educational institutions and open source projects can each take to improve student involvement.

1 The Challenge

To effectively teach Open Source, it's necessary to move each *student* into the role of *contributor*. At first blush this appears straightforward, but it ultimately proves to be an enormous challenge because Open Source is as much a social movement as a technical one and because many Open Source practices are the exact opposite of traditional development practices.

1.1 Barriers to Teaching Open Source Development

Many attempts to involve students in Open Source within a course have failed because everyone is overwhelmed:

- The students, because they're suddenly facing an established codebase several orders of magnitude larger than any they have previously encountered in their courses, a community culture that they do not understand, and principles and ideals which are the opposite of what they've learned in other courses—for example, that answers and solutions should not be openly shared on the web [15], that building on other's work by pasting it into your own is academically dishonest, and that it's wrong to deeply collaborate with peers on individual projects.
- The professor and institution, because they're dealing with a continuously-changing, amorphous environment.
- The Open Source project, because it is very difficult to deal with a sudden influx of students who tie up other contributors' time with questions and yet are unlikely to become long-term participants.

1.2 Distinctive Qualities of Open Source Development

In order to develop an effective approach to Open Source development, it's important to understand the qualities which make it unique:

- Open Source development is based around *communities*. These are generally much larger and more geographically diverse than closed-source development teams, and they are enabled and empowered by the web, leading to an increased focus on communication tools and internationalization and localization issues. Social issues become significant, and there is a productive tension between the need to maintain group discipline for coherence and the possibility of provoking a fork. Often, the culture of the community is not the culture of any particular member, but a synthetic intermediate culture.
- The *codebases* managed by the larger communities range up to millions of lines in size and can

date back many years or even decades. Furthermore, they often use tools and languages that are different from those taught in post secondary institutions, or employ common languages in unexpected ways—for example, using custom APIs that dwarf the language in which they are written (such as Mozilla's XPCOM and NSPR). These codebases require specialized, heavy-duty tools such as bug tracking systems, code search tools, version control systems, automated (and sometimes multi-platform) build and test farms and related waterfall and alert systems, toolchains for compiling and packaging each of the source languages used in the project, and release and distribution systems. Smaller Open Source projects which do not maintain their own infrastructure use some subset of these tools through a SourceForge account [20], fedorahosted.org Trac instance [8], or other mechanism.

- Most Open Source systems have an *organic architecture*. Since it's impossible to anticipate the eventual interests and use-cases of the community—including downstream needs—at the inception of a project, the project requirements and development direction change over time and the project grows into its final form (I've never seen UML for an Open Source project!). Although the lack of top-down design can be a disadvantage, the flexible, modular, and extensible architecture that often results has many benefits.

1.3 Turning Challenges into Strengths

Each of these distinctive qualities presents a challenge to a traditional lecture-and-assignment or lecture-and-lab format course, but can be a strength in a community-immersed, project-oriented course. Carefully applied, these strengths can be used to overcome the barriers identified above.

2 Preparing to Teach Open Source

2.1 Select a Faculty Member

A prerequisite for teaching Open Source effectively is a professor who has one foot firmly planted in the Open Source community and the other in the educational world. In order to turn students into contributors, you need a dedicated conduit and liaison who can introduce students to the right people within the Open Source community.

On the academic side, the professor needs to connect with students on a personal level and to be aware of and able to navigate within the learning and administrative context of the educational institution. On the Open Source side, the professor must have deep contacts (and friendships!) within the community, understand the community culture, and know what matters to the community so that projects selected for the students have traction. She must also know and effectively use the community's tools—for example, knowing when to use IRC (Internet Relay Chat), when to use Bugzilla, and when to use e-mail to communicate. The faculty member must have bought-in to Open Source principles, and use the community's products in a production environment ("eating your own dogfood")—there's no credibility to lecturing about bugzilla.mozilla.org using Safari, or presenting PowerPoint slides about OpenOffice.org.

The massive size of most large Open Source codebases prevent any one person from effectively knowing the entire codebase in detail, a problem that is compounded when multiple languages, layers, or major components are involved. This leads to the need to be *productively lost* in the code—moving beyond being overwhelmed and becoming effective at searching, navigating, and reading code. The professor must demonstrate how to cope in this state instead of pretending to know each line, and this includes pulling back the curtain and showing the students how she uses community resources and contacts to find answers to questions. There is no textbook for this; it is behavior that must be modeled.

2.2 Select an Open Source Community

An effective Open Source course also requires the support of a large Open Source project. This selection of project is usually informed by the involvements of the faculty member(s) who will be teaching the course.

The Open Source community selected must have a sufficiently large scope to provide opportunities for many different types and levels of involvement. Its products must also have many angles and components, so students can innovate in corners that aren't being touched by the mainline developers. This likely narrows the list of potential candidates down to the top one hundred or so Open Source projects, which includes the major desktop applications, graphical desktop environments, key server applications, kernels, and community-based Linux distributions.

The reasons for selecting a larger community are straightforward:

- A large community can absorb a large number of students spread across the various components and sub-projects within the community. This enables students with a broad range of interests and skills to get involved in a way that interests them, using the Open Source model of having people work on things they are passionate about. It also spreads the student contact across the community so that few developers will have direct contact with more than one or two students, preventing overload of the existing contributors. At the same time, working within a single community provides a level of coherence that makes it much easier to hold class discussions and plan labs and lectures than if the students' involvement was spread across a number of smaller, independent Open Source communities.
- The project's infrastructure has usually been scaled up to the point where it will readily support the extra contributors.
- Large projects tend to have broad industry support, opening up possibilities for spin-off research projects and broadening the value of the students' experience.

To make this work, you will need the support of the community; they must buy into the idea of teaching students to become productive contributors—not a hard sell, because most communities are hungry for contributors—and there must be open lines of communication with the community's leaders.

It is counter-intuitive to select a large community because it seems easier to manage the students' involvement in a smaller project—but the key is to select something so big that the professor cannot directly manage the students and they are forced to interact with the community in order to succeed.

2.3 Select Potential Student Projects

Open Source communities know what is interesting and valuable within their own space and are in the best position to suggest potential student projects. They're not always able to verbalize these projects, so the professor may need to poke and prod to shake out good ideas, but the community will recognize the value of ideas as they are proposed.

Some of the best project ideas as ones that existing community members would like to pursue, but can't due to a lack of available time (or in some cases, a lack of appropriate hardware). These issues should not be blocker bugs or critical issues that will directly affect release timelines or major community goals, but they may be of significant strategic value to the community. Each person proposing a project idea should be willing to be a resource contact for that project.

Potential projects can include a wide range of activities: feature development, bug fixing, performing testing, writing test cases, benchmarking, documenting, packaging, and developing or enhancing infrastructure tools.

The projects must then be screened for viability within the course context:

- Are they the right size for the course? This does not mean that the project should be fully completed during the course; we've taken an idea from Open Source—the "dot release"—to replace the idea of "complete work," and we look for projects that are not likely to be completed but which can be developed to a usable state in three months.
- Are the necessary hardware and software resources available?
- Is the level of expertise required appropriate for the type of student who will be taking the course? Ideally, each project should make the student reach high, but be neither stratospherically difficult nor trivially easy.

2.4 Prepare the Infrastructure

Each Open Source community has its own set of tools, and it's crucial that students use those native tools so that community members can share with, guide, and encourage the new contributors. The existing community mailing lists, wikis, IRC channels, version control systems, and build infrastructure should be used by the students as they would by any other contributor.

Most academic institutions have their own computing and communications infrastructure, including tools such as Moodle, Blackboard, version control systems, instant messaging systems, forums and bulletin boards, and so forth. It's tempting to use these resources because they are familiar and to avoid placing a burden on the community's resources, but doing so draws a fatal line between the students and the rest of the community. Students can learn to use any tools, but the community will continue to use the tools they have established; when the students meet them there, as fellow contributors, great interaction takes place.

However, there's a certain amount of additional infrastructure needed to support an Open Source course, including:

- A course wiki for schedules, learning materials, labs, project status information, and student details. If this wiki is compatible with the community's wiki (using the same software and similar navigation), it will be easier for the community to contribute to learning materials.
- An IRC channel set up in parallel to the community's developer channel(s), on the same network or server. We have established #seneca channels on irc.freenode.net and irc.mozilla.org, for example; these provide a safe place for students to ask the sorts of questions which may provoke intense flaming in developer channels.
- A blog planet to aggregate the student's blog postings so that all community members, including the students themselves, can easily stay up-to-date on what all of the students are doing. This should be separate from the community's main planet because some of the material will be course-specific. (It's a good idea for the professor to feed the community planet to keep the community up-to-date with what the students are doing.)
- Server farms and/or development workstations (as appropriate to the projects undertaken), to ensure that the students have access to all relevant hardware and operating system platforms.

3 Teaching the Course

We start our Open Source courses by briefly teaching the students the history and philosophy of Open Source. We do this using classic resources such as *The Cathedral and the Bazaar* [24] and the film *Revolution OS* [23], but we don't spend a lot of time on this topic because the philosophy will be explained and modeled in every aspect of the course.

3.1 Communication

Since Open Source is by its very nature *open*, we get students communicating immediately so that they get used to working in the open. They are required to establish a blog (on their own website, or on any of the blogging services such as blogger.com or livejournal) and submit a feed to the course planet. Almost all work is submitted by blogging, and students are expected to enter comments and to blog counterpoints to their colleagues' postings.

All course materials and labs are placed on the course wiki, and both students and community members are encouraged to expand, correct, and improve the material. These resources and the knowledge they represent grow over time and are not discarded at the end of each semester. This body of knowledge eventually becomes valuable to the entire community. Students are also required to get onto IRC. Since the main developers' channels can be daunting to use, students are initially encouraged to lurk in those channels while communicating with classmates and faculty on the student channel. The parallel channel enables students (and faculty) to provide commentary on #developers chatter in real time without annoying the developers, and it provides an appropriate context for course-related discussion. Since the student channel is on the same network/server as the developers' channels, some existing community developers will join the student channel.

3.2 Project Selection

At the very start of the course, students begin reviewing the potential project list, and are required to select a project by the third week. As part of the selection process, students will often use IRC or e-mail to contact the community member who proposed a project that they are interested in. This is the first direct contact between the student and a community member, and since the student is expressing interest in something that the member proposed, the contact is usually welcome. It is critical that students choose projects that are important to the community and attract community support, so we prohibit them from proposing their own projects. Students do find it intimidating to select from the potential project list, since the things that matter to the community are big, hard, and mysterious (or at least appear that way). The professor will often need to serve as a guide during project selection.

We strongly prefer that each student select an individual project, with some rare two-person groups where warranted by the project scope; larger groups are almost always less successful. Students need to collaborate in the community—both inside the class community and within the larger Open Source community—instead of doing traditional, inward-focused academic group work. Students claim a specific project from the potential project list by moving it to the active project list and creating a project page within the course wiki.

3.3 Learning How to Build

Each community has a unique build process. This is often the first non-trivial, cross-platform build that students have encountered, so it's a significant learning experience, and one that has a gratifying built-in reward. There's a lot of easy experimentation available here, so students often go to great lengths testing different build options and approaches (discovering, for example, that a particular build takes 8 or more hours on an Windows XP system, but only about 40 minutes on a Linux VM under that same XP system). The students also learn how to run multiple versions of the software for production and test purposes.

One of the challenges with building is finding an appropriate place to build, since many of the laptop computer models favored by students may have low CPU "horsepower" or memory, while student accounts on lab systems may not have sufficient disk space or student storage may be shared over a congested institutional network. Possible solutions include using external flash or disk drives with lab systems, or providing remote access to build systems.

3.4 Tools and Methodologies

As the students start work on their project, the course topics and labs teach the tools and methodologies used within the community. In most cases, the bug or issue tracking system (such as Bugzilla) drives the development, feature request, debugging, and review processes, providing an effective starting point. It's best that student projects have a bug/issue within the community tracking system, so students must either take on an existing bug or create a bug/issue for each project.

One useful exercise at this stage is to have the students "shadow" an active developer; on Bugzilla, a student can do this by entering that developer's e-mail address in their watch list [4], which forwards to the student a copy of all bugmail sent to the developer. After coming to grips with the e-mail volume, students learn a lot about the lifecycle of a bug through this process.

Next, the students need to learn how to cope with being *productively lost* by using code search tools (such as LXR [9], MXR [10], and OpenGrok [11]), learning to skim code, and most importantly, learning who to talk to about specific pieces of code, including module and package owners and community experts. By working shoulder-to-shoulder with community members, particularly on IRC, they learn the ins-and-outs of the development process, including productivity shortcuts and best practices. The professor can keep his finger on the pulse of the activity through IRC, guiding students when they get off track and connecting them with appropriate community members as challenges arise.

As with all of the activity in the course, students are expected to blog about their experiences on a regular basis, and all of the students benefit from this shared knowledge (as does the community, which does not have to answer the same questions over and over again). At the same time, differences between the student projects prevents one student from riding entirely on the coattails of other students.

3.5 Meeting the Community

Guest lectures by community developers have an enormously powerful impact on students: meeting a coding legend on IRC is great, but talking to him face-to-face and seeing a demonstration of how he works or hearing first-hand about the direction the software is headed has exceptional value.

We film these meetings and share the talks under open content licenses, making them available to people around the world. We've been surprised at the number of views these videos have received, and who is viewing them: for example, we've found that new Mozilla employees often read our wiki and view the videos of our Mozilla developer talks as they come up to speed on the Mozilla codebase.

3.6 Releases

Following the "release early, release often" mantra, students are required to make releases on a predetermined schedule: for the first Open Source course, three releases from 0.1 to 0.3 are required, and for the follow-on course, six biweekly releases from 0.4 to 1.0.

We define the 0.3 release as "usable, even if not polished," reflecting the fact that a lot of Open Source software is used in production even before it reaches a 1.0 state. This means that the 0.3 release should be properly packaged, stable, and have basic documentation, although it may be missing features, UI elegance, and comprehensive user documentation. The slower release rate in the first course is due to the initial learning curve and the fact that setting up a project and preparing an initial solution are time-consuming.

3.7 Contribution to Other Projects

As active members of an Open Source community, students are required to contribute to other Open Source projects, either those of other students or other members within the community. This contribution—which can take the form of code, test results, test cases, documentation, artwork, sample data files, or anything else useful to the project—accounts for a significant portion of the student's mark. Each project is expected to acknowledge external contributions on their wiki project page, and to welcome and actively solicit contributions from other students and community members. This in turn requires that they make contribution easy, by producing quality code, making it available in convenient forms, and by explicitly blogging about what kind of contributions would be appreciated.

Students are often surprised to find community members contributing to their projects (and community members are sometimes unsure whether doing so is permissible from an academic point of view), but that is part of the authentic Open Source experience; it's important not to choke off collaboration for the sake of traditional academics.

In order to receive credit for contribution, students must blog about their contributions to other projects. At first this seems immodest to students, but the straight-facts reporting of work accomplished is a normal part of open development.

4 Seneca's Experience

4.1 History

Seneca College has been involved with Open Source for over 15 years, starting with Yggdrasil Linux installations in 1992. In 1999 we started a one-year intensive Linux system administration graduate program; in 2001 we introduced the Matrix server cluster and desktop installation, converting all of hundreds of lab systems to a dual-boot configuration, which enabled us to teach the Linux platform and GNU development toolchain to students right from their first day at the college. In addition, a number of college faculty members released small Open Source software packages, including *Nled* [25], *VNC#* [22], and *EZED* [21].

In 2002, John Selmys started the annual Seneca Free Software and Open Source Symposium [17], which has since has grown to a two-day event attracting participants from across North America.

In 2005, an industry-sponsored research project on advanced input devices created the need to modify a complex application. The lead researcher on this project, David Humphrey, contacted Mozilla to discuss the possibility of modifying Firefox. This contact led to a deep relationship between Mozilla and Seneca which outlasted that research project and led to the eventual development of the Open Source teaching model described here.

Seneca College's DPS909/OSD600 *Open Source Development* course [19] implemented this model within the Mozilla community. David subsequently developed the *Real World Mozilla* seminar, which packs that course into an intensive one-week format, and the DPS911/OSD700 continuation course was eventually added to enable students to continue development on their Open Source projects and take them to a fully-polished 1.0 release with faculty support.

4.2 Failures

The unpredictable nature of working within a functioning Open Source community poses peculiar challenges. We've had situations where a developer appears unexpectedly and posts a patch that fully completes a student's half-done project. Sometime students encounter reviewers who can't be bothered to do a review, stalling a student's work for weeks at a time, and some module and package owners have a complete lack of interest in the students' work.

On the other hand, we've also had students drop the ball on high-profile work, or fail to grasp how to leverage the community and end up just annoying other contributors. In both cases our relationship with the community has taken a beating.

We've found that most students rise to the challenge presented to them in the Open Source development courses. This has meant that, properly supported, students thrive when presented with big challenges. Conversely, trying to protect students by coddling them in terms of project scope or expectations ("throwing them into the shallow end of the pool") almost certainly leads to failure.

4.3 Successes

By and large, the Open Source Development courses have been successful for the majority of students. Notable projects successes by Seneca students include:

- APNG [1] – Animated PNG format, an extension of the PNG [14] high-colour-depth, full-alpha graphic format. While the PNG Development Group favored the use of MNG as the animated version of PNG, that standard had proven to be large and difficult to implement effectively, and Mozilla wanted to try a lightweight, backward-compatible animated PNG format. Andrew Smith implemented this format [2] and his work has been incorporated into Firefox 3; Opera now also supports APNG.
- Buildbot integration – The Mozilla build system was adapted to work with the BuildBot automation system by Ben Hearsum [6].
- Plugin-Watcher – Many Firefox performance problems are believed to originate with 3rd-party binary plugins such as media players and document viewers. Fima Kachinski (originally working with Brandon Collins) implemented an API to monitor plugin performance, and created a corresponding extension to provide a visual display of plugin load [13].
- DistCC on Windows – A distributed C compilation tool originally written to work with GCC. Tom Aratyn and Cesar Oliveira added support for Microsoft's MSVC compiler, allowing multi-machine builds in a Windows environment [5].
- Automated Localization Build Tool – There are many localizations that deviate in a very minor way from another localization (for example, en_US and en_CA). Rueen Fiez, Vincent Lam, and Armen Zambrano developed a Python-based tool that will apply a template to an existing localization to create the derivative version, which eliminates the need for extensive maintenance on the derivative [3].

In addition, 4 out of 25 student interns at Mozilla this summer are from our courses, and a number of graduates are now employed full-time by Mozilla and companies involved in Open Source as a result of their work.

The Open Source courses have also led to a number of funded research projects in collaboration with Open Source projects and companies.

4.4 What We've Learned

There are many lessons which students repeatedly take away from the Open Source development courses:

- It's important to persevere.
- It's OK to share and to copy code (within the context of the applicable Open Source licenses) instead of guarding against plagiarizing or having your code "stolen."

- Work in public instead of in secret.
- Tell the world about your mistakes instead of publicizing only your successes—there's a lot of value in knowing what does not work.
- You are a full community member, which makes you a teacher as well as a student. Write down what you've done, and it will become a resource. (It's interesting to note that many of the Google searches which the students are performing now return our own course wiki and blogs.)
- Ask for help instead of figuring things out on your own.
- Key figures in this industry do not stand on pedestals—they are real people and are approachable. Relationships are important and communication is critical.
- Code is alive.

We've also learned that Open Source is definitely not for everyone. The least successful students are those who do not engage the community and who attempt to work strictly by themselves. However, even students who don't continue working with Open Source take an understanding of Open Source into their career, along with an understanding of how to work at scale—which is applicable even in closed-source projects.

Finally, we've learned that Open Source communities and companies have a huge appetite for people who know how to work within the community.

4.5 Where We're Headed

The OSD/DPS courses are growing and will continue to work within the Mozilla project. In addition, we will also be working with OpenOffice.org [12] this fall.

Our Linux system administration graduate program (LUX [18]) is being revised to incorporate many of the principles that we've used in the other Open Source courses. LUX students will be working directly with the Fedora project [7], but on a much larger scale than the Mozilla and OpenOffice.org projects: LUX projects will span three courses across two semesters.

One other course is in development: a build automation course, scheduled to be introduced into our system administration and networking programs in January 2009. This course will also be based on work within the Fedora project.

In order to effectively leverage our Open Source teaching, research projects, and partnerships, we've created the Seneca Centre for the Development of Open Technology (CDOT) [16] as an umbrella organization for this work.

5 Steps an Open Source Community Can Take to Improve Student Involvement

Most Open Source communities actively welcome new contributors, but don't always make it easy to join. Many of the steps a project will take to encourage contributors of any sort will improve student involvement:

- Make it easy for new contributors to set up your build environment. Create an installable kit of build dependencies, generate a metapackage, or provide a single web page with links to all of the required pieces.
- Create a central web page with links to basic information about your project that a new contributor will need, such as build instructions, communication systems, a list of module owners, a glossary or lexicon of community-specific technical terms and idioms, and diagrams of the software layers and components used in your products. It's challenging for new contributors to even map IRC nicks to e-mail addresses and blog identities!
- Create sheltered places or processes to enable new people to introduce themselves and get up to speed before being exposed to the full flaming blowtorch of the developer's lists and channels. This might include an e-mail list for new-contributor self-introductions or a process for self-introduction on the main lists, or an IRC channel for new developers.

In addition, in a course context:

- Ensure that the community is aware of the course and course resources.

- Feel free to join the student IRC channel, contribute to student projects as you would any other project, and read the student planet.
- Contribute to learning materials on the course wiki.
- Apart from recognizing the students as new community members, treat them as any other contributor.

6 Conclusion

Open Source development is dramatically different from other types of software development, and it requires some radically different pedagogical approaches. A community-immersed, fully-open, project-oriented approach led by professor who is also a member of the Open Source community provides a solid foundation for long-term, sustainable student involvement in that Open Source community.

7 Acknowledgments

I would like to acknowledge the pioneering work of my colleague David Humphrey in establishing the Open Source Development courses at Seneca, and for his thoughtful review of this paper.

References

[1] *Animated PNG Information Site*. `http://animatedpng.com/`.

[2] *APNG project page*. `http://zenit.senecac.on.ca/wiki/index.php/APNG`.

[3] *Automated Localization Build Tool project page*. `http://zenit.senecac.on.ca/wiki/index.php/Automated_localization_build_tool`.

[4] *Bugzilla Watch Lists*. `http://www.bugzilla.org/docs/3.0/html/userpreferences.html#emailpreferences`.

[5] *DiscCC with MSVC project page*. `http://zenit.senecac.on.ca/wiki/index.php/Distcc_With_MSVC`.

[6] *Extending the Buildbot project page*. `http://zenit.senecac.on.ca/wiki/index.php/Extending_the_Buildbot`.

[7] *Fedora Project*. `http://fedoraproject.org/`.

[8] *Fedorahosted Trac instances*. `https://fedorahosted.org/web/`.

[9] *LXR*. `http://lxr.linux.no/`.

[10] *MXR*. `http://mxr.mozilla.org/`.

[11] *OpenGrok*. `http://opensolaris.org/os/project/opengrok/`.

[12] *OpenOffice.org*. `http://openoffice.org/`.

[13] *Plugin-watcher project page*. `http://zenit.senecac.on.ca/wiki/index.php/Plugin-watcher`.

[14] *PNG - Portable Network Graphics*. `http://www.libpng.org/pub/png/`.

[15] *The Ryerson Facebook Dilemma*. `http://www.wikinomics.com/blog/index.php/2008/03/12/the-ryerson-facebook-dilemma/`.

[16] *Seneca Centre for Development of Open Technology (CDOT)*. `http://cdot.senecac.on.ca/`.

[17] *Seneca Free Software and Open Source Symposium*. `http://fsoss.senecac.on.ca/`.

[18] *Seneca LUX Graduate Program*. `http://cs.senecac.on.ca/?page=LUX_Overview`.

[19] *Seneca Open Source Development Wiki*. `http://zenit.senecac.on.ca/wiki/`.

[20] *Sourceforge*. `http://sourceforge.net/`.

[21] John Flores. *EZED - Easy Editor*. `http://cdot.senecac.on.ca/software/ezed/`.

[22] David Humphrey. *VNC#*. `http://cdot.senecac.on.ca/projects/vncsharp/`.

[23] J. T. S. Moore. *Revolution OS*, 2001. `http://www.revolution-os.com/` (available online at `http://video.google.com/videoplay?docid=7707585592627775409`).

[24] Eric Raymond. *The Cathederal and the Bazaar*, 2000. `http://catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/`.

[25] Evan Weaver. *NLED—Nifty Little Editor*. `http://cdot.senecac.on.ca/software/nled/`.

A Runtime Code Modification Method for Application Programs

Kazuhiro Yamato
Miracle Linux Corporation
`kyamato@miraclelinux.com`

Toyo Abe
Miracle Linux Corpration
`tabe@miraclelinux.com`

Abstract

This paper proposes a runtime code modification method for application programs and an implementation. It enables the bugs and security problems to be fixed at runtime. Such software is notably useful for applications used in telecom, which cannot be stopped because of the need to maintain the required level of system availability. The advantages of the proposed method are short interruption of the target application and easy maintenance using trap instructions and utrace.

This paper also shows evaluation results with three conditions. The interruption times by the proposed method were comparable to, or shorter than those by existing similar software, *livepatch* and *pannus*. In a certain condition, our implementation's interruption time was three orders of magnitude shorter in comparison.

1 Introduction

Although there have been a number of activities to improve software quality, there is no way to completely prevent bugs and security problems. These are generally fixed by rebuilding the program with patches to the source code. This fix naturally requires termination and restarting of the program. The termination of the program not only interrupts the service, but also loses various data such as variables, file descriptors, and network connections. It is impossible to recover these properties unless a recovery mechanism is built in the program itself.

Fixing with a termination is a serious problem especially for application programs used in telecom, because they cannot easily be terminated to keep the required level of system availability.[1] Therefore, the problems should be fixed at runtime with binary patches. In addition, interruption time to apply binary patches should be short because long interruption degrades the quality of voice and video, which are major services of telecom.

In this paper, we call an application program to be fixed by RBP (Runtime Binary Patcher) a *target*. Two major open source RBPs, *livepatch* [1] and *pannus* [2], already exist. However, *livepatch* potentially interrupts the execution of a target for a long time. The maintenance of *pannus* doesn't seem to be easy.

This paper proposes a runtime code modification method for application programs, and, an implemented RBP. It achieves short interruption and easy maintenance using the trap instruction and the utrace APIs [3].

2 Existing Methods

2.1 `ptrace` system call and *gdb*

The `ptrace` system call provides debug functions, such as reading/writing memory in the target's virtual memory space, acquisition/modification of the target's registers, and catching signals delivered to the target. These functions are enough to realize runtime code modification. Actually, we can modify a target's memory with *gdb* [4], which is one of the most popular debuggers in the GNU/Linux environment and also a typical application using `ptrace`.

For example, the *gdb* command `"set variable *((unsigned short*)0x8048387)=0x0dff"` overwrites `0x0dff` (`dec` instruction on i386) at address `0x8048387` by calling `ptrace(PTRACE_POKEDATA, pid, addr, data)`, where `pid`, `addr`, and `data` are the process ID of the target, the address to be overwritten, and the address of data to overwrite, respectively.

However, this approach potentially causes long interruption of target execution when the target has many

[1]The CGL (Carrier Grade Linux) specification requires 99.999% availability.

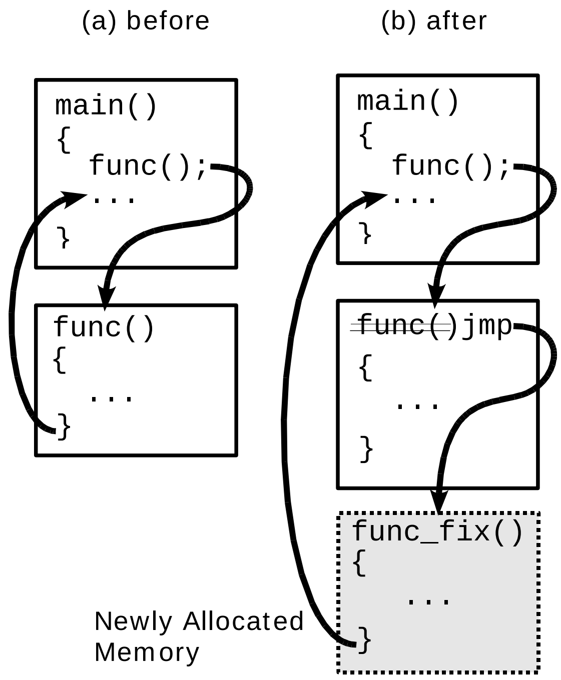


Figure 1: Function call path before and after applying a binary patch

threads or the patch is large. *gdb* frequently stops all threads in a target. As the number of threads increases, interruption time increases, too. The writing size of `PTRACE_POKEDATA` is a 'word,' which is architecture-dependent—four bytes on i386.

Some practical cases will require additional memory to apply a binary patch whose size is greater than the original code. `ptrace` doesn't provide a direct function to allocate memory. However, *livepatch* solves this by making a target execute instructions to allocate memory as described in Section 2.2.1.

2.2 Open Source RBPs

livepatch and *pannus* are two major open source RBPs. They both fix problems by adding a binary patch in the target's virtual memory space and overwriting the `jmp` instruction to the binary patch at the top of the function to be fixed (we call it the target function) as shown in Figure 1. This means problems are fixed by the function unit. Thus a patch is provided as a function (fixed-function) in an ELF shared library (patch file). The basic processes of *livepatch* and *pannus* are similar and roughly divided into the following four stages.

1. Preparation: opens a patch file, obtains the size of the patch, gives addresses to unresolved symbols, and so on.

2. Memory Allocation: allocates memory for the patch in the target.

3. Load: loads the patch into the allocated memory.

4. Activation: overwrites an instruction to jump to the patch at the top of the target function.

```
int prot = PROT_READ|PROT_WRITE;
int flags = MAP_PRIVATE|MAP_ANONYMOUS;
mmap(NULL, size, prot, flags, -1, 0);
asm volatile("int $3");
```

Figure 2: The code written in the target stack

2.2.1 livepatch

livepatch [1] was developed by F. Ukai, and consists only of a utility in user space, which is about 900 lines of code. In the preparation stage, *livepatch* gets the information about the patches in the ELF file with `libbfd`. In the memory allocation stage, *livepatch* first obtains the stack pointer of the target using `PTRACE_GETREGS`. Then it writes the machine code corresponding to the source shown in Figure 2 on the stack. The code is executed by setting the program counter to the top address of the stack using `PTRACE_SETREGS`, followed by `PTRACE_CONT`. The `mmap()` in the code allocates memory in the target, because the target itself calls the `mmap()`. The assembler instruction '`int $3`' generates the `SIGTRAP` to bring back the control to *livepatch*, which is sleeping by `wait(NULL)` after the `PTRACE_CONT`.

In the load stage, *livepatch* writes the patch in the allocated memory by repeating `PTRACE_POKEDATA`. In the activation stage, `PTRACE_POKEDATA` is also used to overwrite the instruction to jump.

2.2.2 pannus

pannus [2] was developed by a group from NTT Corporation. It consists of a utility in user space and a kernel patch. In the preparation stage, *livepatch* analyzes a patch file by itself without external libraries and obtains necessary data. The memory allocation is mainly performed by a `mmap3()` kernel API which is provided by the kernel patch. Actually it also plays a role in a portion of the load stage, because the `mmap3()` maps the patch file. The `mmap3()` is an enhanced version of `mmap2()`, which is a standard kernel API. It enables other processes to allocate memory for the specified process, directly accessing the core kernel structures such as `mm_struct`, `vm_area_struct`, and so on. Because members of the structures or access rules are often changed, the maintenance of the patch doesn't seem to be easy.

In the load stage, the `access_process_vm()` kernel API is used via the kernel patch to set relocation information in the allocated memory. The API reads/writes any size of memory in the specified process.

In the activation stage, *pannus* also uses `access_process_vm()` to overwrite the instruction to jump. Note that *pannus* checks whether the status is safe before the overwriting. The safety means that the number of threads whose program counters point the region to be overwritten is zero. The program counter is obtained by `PTRACE_GETREGS`, after the target was stopped by `PTRACE_ATTATCH`. If the status is not 'safety,' *pannus* resumes the target once by `PTRACE_DETATCH`, and tries to check again. If the result of the second check is not also safety, *pannus* aborts the overwriting. The probability that this situation happens increases with the call frequency of the target function.

3 Proposed Method and its Implementation

3.1 Patching process

We propose a new patching method, which is used in our project *kaho*, which means *Kernel Aided Hexadecimal code Operator*. The patching process of *kaho* is divided into four stages similarly to *livepatch* and *pannus*. They are shown in Figure 3 with detailed steps. Its implementation consists of a user-space utility program and a kernel patch. The white boxes in the figure are processing by the utility. The shaded steps are processed in the kernel via IOCTLs. The supported architectures of *kaho* are x86_64 and i386 at this moment. All these steps are described here and details of IOCTLs and 'Safety check' are explained in the following sections.

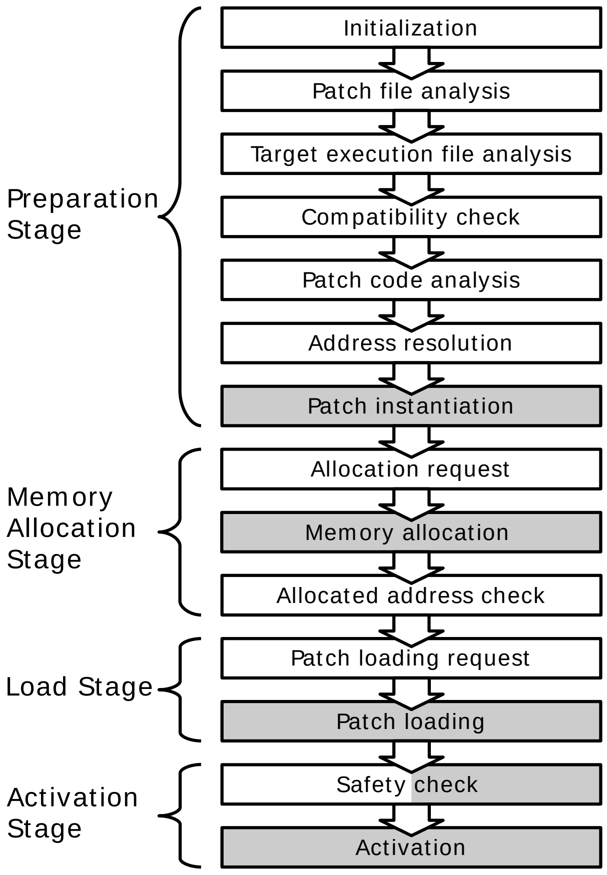


Figure 3: The patching process of *kaho*

The *Preparation Stage* consists of seven steps. *Initialization* interprets the command line options, which include information about the target and the command file. The command file defines the name of a target, a fixed function, a patch file, and a map file. *Patch file analysis* opens the patch file and reads ELF information such as ELF header, section headers, program headers, symbol table, dynamic sections, and versions. *Target execution file analysis* finds the executable from `/proc/<PID>/exe` and acquires ELF information. *Compatibility check* confirms that byte order (endian), file class, OS ABI,[2] and ABI version of the patch file, which are contained in the ELF header, are identical to those of the target execution file. *Patch code analysis* searches for an entry whose `st_name` member is identical to the name of the fixed function from the symbol table, and its file position from the `st_value`. *Address resolution* works out the addresses of unresolved symbols using a target executable, the depending libraries, and `/proc/<PID>/maps`. When symbols

[2]such as UNIX – System V, UNIX – HP-UX, UNIX – NetBSD, GNU/Hurd, UNIX – Solaris, *etc.*

are stripped out in the executable and libraries, the map file must be specified in the command file, because the map file lists function names and the corresponding addresses. *Patch instantiation* allocates the data structure to manage binary patches in the kernel space, and generates a unique handle for every a patch instance.

The *Memory Allocation Stage* consists of three steps. *Allocation request* finds a vacant-address range near the target function using `/proc/<PID>/maps` and calls the *Memory allocation* IOCTL. *Allocated address check* confirms that the address[3] is within a ±2GB range from the target function. Note that *Allocated address check* is needed for the x86_64 architecture only, because the immediate operand of the `jmp` instruction is relative-32-bit despite having a 64-bit accessible memory space.

Load Stage consists of *Patch loading request* and *Patch loading*. *Patch loading request* calls an IOCTL with a patch instance handle, the fixed function's address in *kaho* utility virtual memory, size of the fixed function, and the target function's virtual memory address. *Patch loading* is an IOCTL to load the fixed function in the target.

Activation Stage consists of *Safety check* and *Activation*. *Safety check* confirms that no threads are executing code on the region to be overwritten by *Activation*. If this is not checked, threads may fetch illegal instructions. There are two modes to check safety, standard mode and advanced mode. In the standard mode, the check is performed by the *kaho* utility program. In the advanced mode, *Activation* performs it. *Activation* overwrites the instruction to jump to the fixed function at the top of the target function.

In fact, *kaho* can deactivate and remove the activated patches. In addition, *kaho* can modify the data in the target. In the case, *Loading Stage* stores the data from the utility in the kernel. *Activation Stage* overwrites the data with `access_process_vm()`.

3.2 Patch instantiation

The *Patch instantiation* IOCTL receives the process ID of the target and the number of patches. It first takes an available handle from its own handle pool and creates the data structure to manage patches, which is named *patch instance*. Patch instance's member variables include the number of patches, the pointer to the target's `task_struct`, addresses of the target functions, sizes of the patches, and so on. Then it gets the pointer to target's `task_struct` with `find_task_by_pid()`, adds the patch instance to the dedicated list named patch-instance list, and attaches the target by calling `utrace_attach()` with the callbacks shown in Figure 4. After the target is attached, `utrace_set_flags()` with flag `UTRACE_EVENT(EXEC)|UTRACE_EVENT(REAP)`[4] is called to delete the patch instance from the patch-instance list and release it for the case in which the patch becomes no longer needed. Finally, *Patch instantiation* returns the handle to be used in the other IOCTLs.

```
static const struct
utrace_engine_ops kaho_utrace_ops =
{
  .report_exec = kaho_report_exec,
  .report_quiesce = kaho_report_quiesce,
  .report_reap = kaho_report_reap,
};
```

Figure 4: *kaho*'s `utrace` callbacks

3.3 Memory Allocation

Memory Allocation IOCTL receives a handle, a request address, and a request size. It first finds that the patch instance which has the handle is in the patch-instance list and stores the request address and the request size in the patch instance. Then it enables the callback `kaho_report_quiesce()` by calling `utrace_set_flags()` with flags `UTRACE_ACTION_QUIESCE | UTRACE_EVENT(QUIESCE)`. Shortly after the flags are set, `utrace` executes the callback specified in `.report_quiesce` (that is `kaho_report_quiesce()` in this case) on the target context as shown in Figure 5.

`kaho_report_quiesce()` calls `do_mmap()` with the requested address and size on the target context. As a result, memory is allocated in the target's virtual memory space. The basic idea is similar to

[3] The address in which the fixed-function is loaded in a precise sense.

[4] The flag enables callbacks specified in `.report_exec` and `.report_reap` to be called when the target calls the `exec()` family and when the target terminates, respectively.

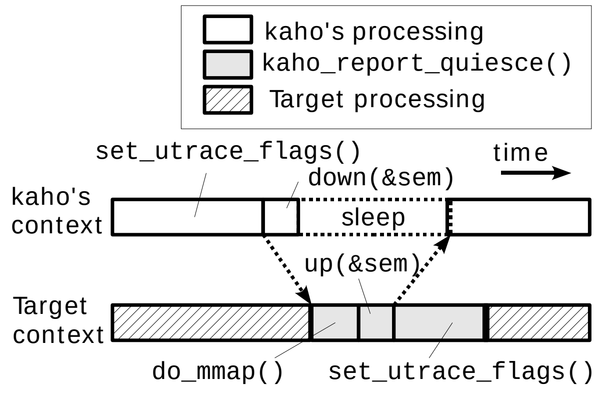


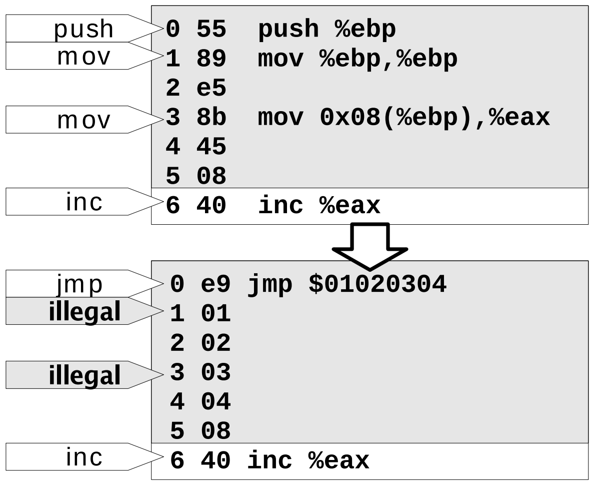
Figure 5: Mechanism of memory allocation

that of *livepatch*. Next `kaho_report_quiesce()` executes `up(&sem)` to wake up the *kaho* utility program which is sleeping by `down(&sem)`. Finally, `kaho_report_quiesce()` stores the address of the allocated memory in the patch instance and calls `utrace_set_flags()` without flags `UTRACE_ACTION_QUIESCE|UTRACE_EVENT(QUIESCE)` to disable this callback and resume the target's processing. After the sleep finishes, the address is returned.

3.4 Patch Loading

Patch Loading IOCTL receives the handle, the address at which the fixed function is in the *kaho*'s memory space, the address to be loaded in the target's memory space, the address of the target function in the target's memory space, the size to be loaded, and the subpatch ID. The sub patch ID is the sequential number to identify patches which are applied at one time. After *Patch Loading* stores them in the patch instance, it loads the fixed functions into the target by calling `access_process_vm()`.

`access_process_vm()` is the standard kernel API which reads or writes the memory in the specified process. It is also used from the `ptrace` system call and some kernel functions to handle `/proc/<PID>/mem`. This means that memory in the target can be written by calling `ptrace` and writing `/proc/<PID>/mem`. However, This works only when the target is in a traced state; namely, the target must be stopped. Although this is necessary to prevent unexpected results in usual cases, we don't access memory loading fixed functions. Therefore, *Patch Loading* calls `access_process_vm()` without stopping the target.



```
0 55  push %ebp
1 89  mov %ebp,%ebp
2 e5
3 8b  mov 0x08(%ebp),%eax
4 45
5 08
6 40  inc %eax
```

```
0 e9 jmp $01020304
1 01
2 02
3 03
4 04
5 08
6 40 inc %eax
```

Figure 6: The example of the safety and danger

3.5 Safety check

Safety check confirms that program counters of the all threads in the target don't point the region to be fixed. If a thread's program counter points to such a region,[5] it gets illegal instructions after the instruction to jump is overwritten, as shown in Figure 6. *kaho* has two modes to check the safety. One is the standard mode in which the *kaho* utility program checks. The other is the advanced mode in which the *Activation* IOCTL checks before the instruction to jump to the fixed function is overwritten in kernel space. The interruption of the target in the advanced mode is shorter than that in the standard mode, because the target is not stopped in the advanced mode. However, the number of fixed functions which are applied at one time in the advanced mode is limited to only one.

3.5.1 Standard mode

Safety check in the standard mode consists of *Quick check* and *Forced displacement*. *Quick check* attaches the target with `ptrace` and checks the value of the program counter with `PTRACE_GETREGS`. When the program counters of all threads do not point the region to be overwritten, the check successfully finishes. Otherwise, the check is retried. If the failures continue a few times, *Forced displacement* is executed as the threads are attached.

[5]The first byte of the region is exempted, because some sort of valid instruction should be overwritten.

Forced displacement tries to bring about a safety state using the trap instruction (`int 3`). It first overwrites a trap instruction at the top of the target function. Then it resumes only the threads whose program counters point the region to be fixed with `PTRACE_CONT`. After that, some threads will stop by the trap instruction. Consequently such threads become safe. Other threads including threads which do not run on the code with trap instruction are checked periodically with `PTRACE_ATTACH` and `PTRACE_GETREGS`. If safety of all threads is confirmed, the check successfully finishes. Otherwise, it fails.

3.5.2 Advanced mode

Safety check in the advanced mode is further broken down into four steps. These consist of *Preparation*, *Trap handler*, *Thread safety check*, and *Target safety check*. The basic strategy is to mark the thread which has executed the overwritten trap instruction at top of the target function for safety. This method is inspired by `djprobes` [5]. *Preparation* first makes a checklist in which pointers to the task struct of all threads and corresponding check statusus are contained and adds the pointer of the target's task struct to the list called *trapped-targets*. Then it overwrites the trap instruction at top of the target function. After this, the check in the *Trap handler* becomes active as described below. Finally, it calls `utrace_attach()` and `utrace_set_flags()` to execute the callback specified in `.report_quiesce` for all threads, which is *Thread safety check* in this case.

Trap handler is registered by `register_die_notifier()` when a system is initialized. It is called when any process or a thread in the system executes a trap instruction. Therefore it must confirmed whether the thread which executes the trap instruction and the address are included in the trapped-targets list. If it is included, *Trap handler* marks the thread as safety. Then it changes the program counter of the thread to the address of the fixed function by setting `rip` or `eip` in `struct pt_regs` given by the lower-common layer of kernel, as shown in Figure 7. This means that the thread executes the fixed function.

Thread safety check is called on the context of the target thread. It first reads the check list and confirms that the safety of the thread was already checked by *Trap handler*. If the thread is 'safety,' it finishes immediately.

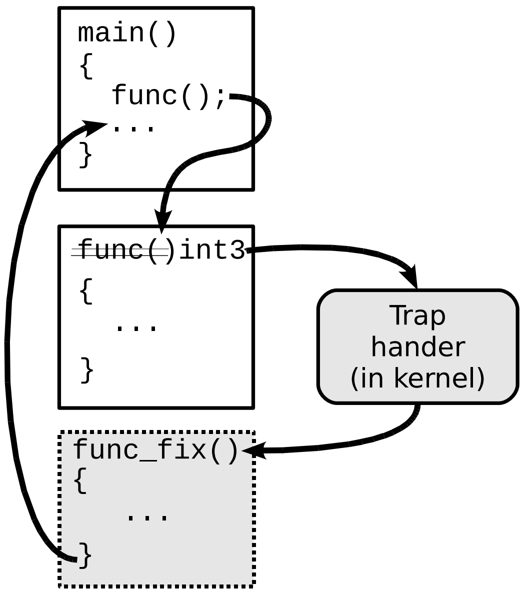


Figure 7: Function call path during safety check in advanced mode

Otherwise, it checks for safety, reading program counters in user space, which is saved in the kernel mode stack. The result is written in the check list. After the *Thread safety check* is executed on the all threads, *Target safety check* reads the check list and deletes safe threads from the list. When the check list becomes empty, *Target safety check* successfully finishes. Otherwise, it repeats *Thread safety check* for the remaining threads in the check list.

3.6 Activation

Activation IOCTL receives the handle and the flag. It overwrites instructions to jump to the fixed function at the top of the target function using `access_process_vm()`. The addresses of the fixed function, address of the target function, and the sizes of the fixed functions are obtained from the patch instance. The examples of the instructions to be overwritten are shown in Figure 8 and Figure 9. In Figure 8, the `jmp` instruction which takes a 32-bit relative address is used and its total size is 5 bytes. The instructions shown in Figure 9 are used only when the architecture is x86_64 and `KAHO_X86_64_ABS_JMP` is set in the flag. The condition in which the instructions in Figure 9 is needed is rare. Code, data, and stack regions in the memory space are usually sparsely placed in memory.

```
        jmpq  $0x12345678
```

Figure 8: Example of instruction to be written

```
    pushq $0x12345678
    movl  $0x9abcdef0, 4(%rsp)
    ret
```

Figure 9: Example of instructions to be written when the flag `X86_64_ABS_JUMP` is specified on the 86_64 architecture

3.7 Usage of *kaho* utility program

We introduce brief usage for the *kaho* utility program named `kaho`. The usage is greatly influenced from that of *pannus*. First of all, environment variable `KAHO_HOME` must be set. It specifies the base directory in which the command file, the patch file, and the map file are placed. Then users perform the following three steps.

1. Prepare a command file, a patch file, and a map file.
2. Execute `kaho` with the load option.
3. Execute `kaho` with the activation option.

3.8 Limitations

3.8.1 Handling of C++ exception

kaho cannot apply a binary patch to code which can potentially generate C++ exceptions. When the exception happens, the code compiled by *g++* tries to find the frame to be caught with the `.eh_frame_section` in the ELF file which contains the target function. However, the information about the fixed function is not in the `.eh_frame_section`. As a result, the exception is not caught correctly.

3.8.2 Static variables

When a target function contains static variables and a patch uses them, the patch may not work correctly. The reason is that the patch refers not to the variable in the target, but in itself. This can be avoided by replacing the `static` keyword with the `extern` keyword and adding the address of the static variable in the target function to the map file.

3.8.3 Loading new shared libraries

kaho fails to load patches when the fixed functions in the ELF file require additional functions which are not in the ELF file itself, the target executable file, or already-loaded shared libraries.

3.8.4 Multiple binary patch in advanced mode

The present algorithm of the safety check in the advanced mode allows only one fixed function to be applied at a time. However, there is a certain situation in which multiple fixed functions should be applied at one time. Therefore, we plan to extend the number of fixed functions which are applied at one time in advanced mode.

4 Evaluation

4.1 Evaluation method

We evaluated performance of RBPs (*livepatch*, *pannus*, and *kaho*) by the interruption time in target execution. This section describes our definition of the interruption time, our measuring method, the target programs used for the measurement, and our evaluation environment.

4.1.1 Interruption time

We defined five categories of interruption; *Allocation*, *Load*, *Check*, *Trap*, and *Setup*. *Allocation* is the interruption due to memory allocation for a binary patch. All RBPs but *pannus* allocate memory in the target context. *pannus* does it from the outside of the target via `mmap3()`. *Load* is the interruption due to loading the binary patch. Only *livepatch* does it in the target context. *Check* is the interruption due to the safety check for the target to be safely patched. *pannus*, *kaho* in the standard mode (*kaho-std*), and *kaho* in the advanced mode (*kaho-adv*) have safety check mechanisms, which work in the target context. *livepatch* doesn't have such a step. *Trap* is the interruption due to the in-kernel trap handler; it also happens in the safety check step. Only *kaho-adv* uses this functionality. *Setup* is the interruption due to processing the `_init` function in the binary patch. Only *pannus* executes it in the activation.

4.1.2 Measurement

We placed probe points statically in the kernel to determine the interruption time in each category listed in Section 4.1.1. Also, we measured the interruption time in the *Trap* category from the target by getting processor's cycle counter.

The followings are the probe points we placed in kernel.

(A) `ptrace`-triggered stop/cont point (i.e., `TASK_TRACED` stop/cont). The `ptrace` is used for the *Allocation*, *Load*, or *Check* category.

(B) *kaho*'s utrace quiesce handler entry/exit for memory allocation. This is used for the *Allocation* category in *kaho-std* and *kaho-adv*.

(C) *kaho*'s utrace quiesce handler entry/exit for safety check. This is used for the Check category in *kaho-adv*.

(D) Setupper entry and restorer exit of the `_init` function in a patch. This is used used for the *Setup* category in *pannus*.

The following one is in user space (i.e., the target).

(E) Right before calling the fixed function and at the top of the fixed function. This is used for the *Trap* category in *kaho-adv*.

We conducted our measurements a hundred times per target. We took the mean value as a result.

4.1.3 Targets for the evaluation

To determine the performance and the characteristics of each RBP, three typical target programs described below were used.

- Single: Single-threaded application. The target function is continuously called and immediately returns. Because safety checks are very easy, this type of target can be used to determine the shortest time of the interruption.

- Multi-I: Multi-threaded application. The target function is frequently called from all of the threads. A hundred threads simultaneously access the function without any control. There is no outstanding resource contention across the threads in this target. Safety checks are very tough work, so this type of target can be used to determine the longest time. This is the most unfavorable condition for *kaho-adv* because it uses the trap handler for the check.

- Multi-II: Multi-threaded application and the target function is never called from any of the threads. A hundred threads run without accessing the function. There is no outstanding resource contention across the threads in this target. This target is the most favorable condition for *kaho-adv* because no one hits its trap handler. We used this target to estimate the effect of the trap on the interruption time.

4.1.4 Evaluation environment

We conducted our measurements on a Dual 2.66GHz Intel Quad-Core CPU machine with 4GB of RAM, which was installed with the Fedora Core 6 Linux distribution. When testing *pannus*, the 2.6.15.1 kernel plus the *pannus* patch was running on the machine, because the latest patch of *pannus* is against this kernel version. When testing *livepatch* and kaho, the 2.6.24 version of the `utrace` kernel plus the *kaho* kernel module was running on it. Because *livepatch* supports only the i386 architecture, we evaluated *livepatch* in ia32e mode on the x86_64 kernel.

4.2 Results

The evaluation results are summarized in Table 1 and in Figures 10-12. We can see that the results depend largely on the target, at first glance. For any targets, the interruptions by *kaho-adv* were shorter than those by *kaho-std* and *pannus*. The interruptions by *kaho-std* were of the same order as those by *pannus*. Although the results of *livepatch* are also presented, it is naïve to compare the results with *pannus* and *kaho* in terms of

Target	RBP	Total Interupt. (μs)	*Allocation*	*Load*	*Check*	*Trap*	*Setup*
Single	*livepatch*	2486	569 (A)	1916 (A)	–	–	–
Single	*pannus*	48	–	–	29 (A)	–	19 (D)
Single	*kaho-std*	43	5 (B)	–	37 (A)	–	–
Single	*kaho-adv*	10	6 (B)	–	3 (C)	0 (E)	–
Multi-I	*livepatch*	502253	100055 (A)	402197 (A)	–	–	–
Multi-I	*pannus*	4673323	–	–	4657529 (A)	–	15794 (D)
Multi-I	*kaho-std*	1034518	119 (B)	–	1034399 (A)	–	–
Multi-I	*kaho-adv*	1017455	111 (B)	–	594 (C)	1016750 (E)	–
Multi-II	*livepatch*	513139	99107 (A)	414032 (A)	–	–	–
Multi-II	*pannus*	5040145	–	–	5020773 (A)	–	19372 (D)
Multi-II	*kaho-std*	895451	121 (B)	–	895330 (A)	–	–
Multi-II	*kaho-adv*	634	112 (B)	–	522 (C)	0 (E)	–

Table 1: Total and break-down interruption time. '–' means N/A. The characters in parentheses indicate the probe point listed in Section 4.1.2.

the total interruption time. Because *livepatch* doesn't have a safety check, which is an necessary function to prevent unexpected results, the interruptions are short, especially for Multi-I and Multi-II. Therefore, we discuss the results without *livepatch* in the following sections.

4.2.1 Single-threaded target

The interruption-time distribution for the 'Single' case is shown in Figure 10. The interruptions by *kaho-adv*, *kaho-std*, and *pannus* were 10μs, 43μs, and 48μs respectively. The interruption by *kaho-adv* was about a quarter of that by *kaho-std* and *pannus*. We think the reason is that context switch of the target doesn't happen in *Check* by *kaho-adv*, which utilizes `utrace`. On the other hand, because *kaho-std* and *pannus* use `PTRACE_ATTACH` and `PTRACE_DETACH` for *Check*, the context switch happens at least twice.

4.2.2 Multi-threaded target I

The interruption-time distribution for Multi-I is shown in Figure 11. The interruptions by *kaho-adv*, *kaho-std*, and *pannus* were 1.02s, 1.03s, and 4.67s, respectively. This shows that the performance of all RBPs is comparable in this situation.

The interruption due to *Check* by *kaho-adv* was 594μs. This is three orders of magnitude shorter than that of *pannus* and *kaho-std*. However, almost all of the interruption by *kaho-adv* was consumed by *Trap*. When the fixed function was called via *Trap*, the time was 2.2μs longer than the time via direct jump in our evaluation. Even so, *Trap* happened about 440,000 times until *Activation* was completed. As a result, about 1s of interruption happened in total.

In the *kaho-std* and *pannus*, over 99% of interruptions were consumed by *Check*. Although their safety-check algorithms are almost the same, the interruption by *kaho-adv* is about a quarter of *pannus*'s result. The reason is unclear. It may be due to the difference of the base kernel versions.

4.2.3 Multi-threaded target II

The interruption-time distribution for Multi-II is shown in Figure 12. The interruptions by *kaho-adv*, *kaho-std*, and *pannus* were 634μs, 0.90s, and 5.04s, respectively. The interruptions by *kaho-std* and *pannus* differed little from the interruptions for Multi-I. However, the interruption by *kaho-adv* was notably reduced, because *Trap* was not used at all for the target. This means *kaho-adv* is much better than *kaho-std* and *pannus* when the target function is not called frequently.

5 Conclusion

This paper has proposed a runtime code modification method for application programs and an implementation

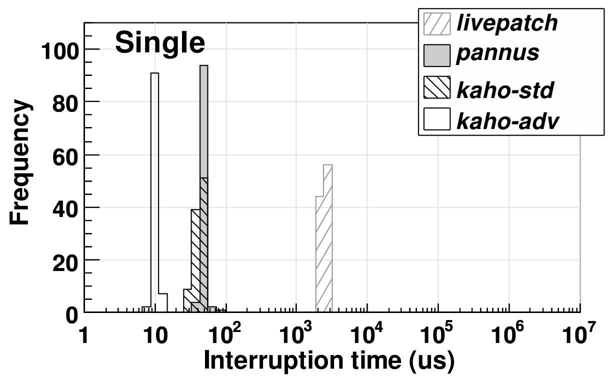


Figure 10: Interruption-time distribution for the single-thread target

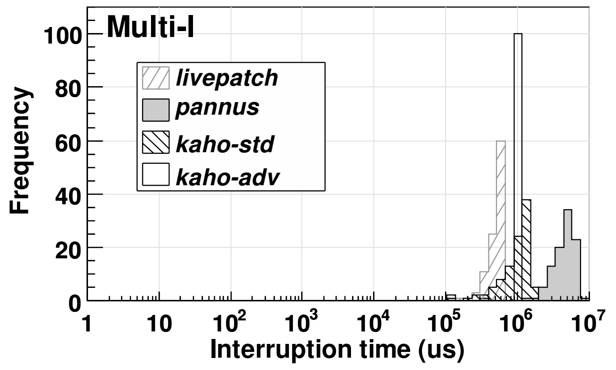


Figure 11: Interruption-time distribution for the multi-thread target I

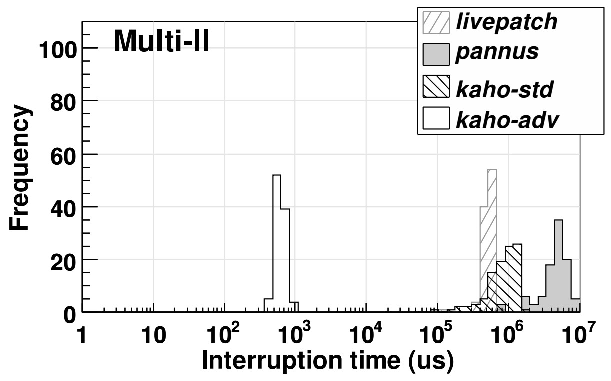


Figure 12: Interruption-time distribution for the multi-thread target II

named *kaho*. Such software is called Runtime Binary Patcher (RBP). The RBP is notably useful for applications used in telecom, which must continue running to keep the required level of system availability. The basic process of *kaho* is based on that of existing open source RBPs, *livepatch* and *pannus*. However, *kaho* has two major advantages. One is short interruption of target execution by the safety check using the trap instruction, which is inspired by `djprobes`. The other is easy maintenance using the `utrace` kernel APIs instead of the `ptrace` system call.

This paper also has shown the evaluation results with three conditions. Although the results depended largely on the condition, the interruptions by *kaho* were comparable to or shorter than interruptions by *livepatch* and *pannus* in all conditions. In a certain condition, the interruption by *kaho* was three orders of magnitude shorter than that by *livepatch* and *pannus*.

Acknowledgement

I wish to express my special thanks to Mr. I. Shikase and Mr. A. Kato of AIR Co., Ltd., who have developed the *kaho* utility program.

References

[1] `http://ukai.jp/Software/livepatch/`

[2] `http://pannus.sourceforge.net/`

[3] `http://people.redhat.com/roland/utrace/`

[4] `http://sourceware.org/gdb/`

[5] `http://lkst.sourceforge.net/djprobe.html`

SynergyFS: A Stackable File System Creating Synergies between Heterogeneous Storage Devices

Keun Soo Yim and Jae C. Son
Samsung Advanced Institute of Technology
{keunsoo.yim, jcson}@samsung.com

Abstract

Hybrid storage architecture is one efficient method that can optimize the I/O performance, cost, and power consumption of storage systems. Thanks to the advances in semiconductor and optical storage technology, its application area is being expanded. It tries to store data to the most proper medium by considering I/O locality of the data. Data management between heterogeneous storage media is important, but it was manually done by system users.

This paper presents an automatic management technique for a hybrid storage architecture. A novel software layer is defined in the kernel between virtual and physical file systems. The proposed layer is a variant of stackable file systems, but is able to move files between heterogeneous physical file systems. For example, by utilizing the semantic information (e.g., file type and owner process), the proposed system optimizes the I/O performance without any manual control. Also as the proposed system concatenates the storage space of physical file systems, its space overhead is negligible. Specific characteristics of the proposed systems are analyzed through performance evaluation.

1 Motivation

The primary design objectives of storage systems include high performance, low cost, and low power consumption. In practice there exists no single storage device satisfying this design requirement. The faster the access speed of storage device is, the higher the cost per bit ratio. Despite the efforts toward an ideal storage device, we now have storage devices, partially ideal and biased to the requirement of specific applications. This stresses the importance of hybrid storage architecture because this can utilize the strong points of storage devices and hide the weaknesses in an efficient manner.

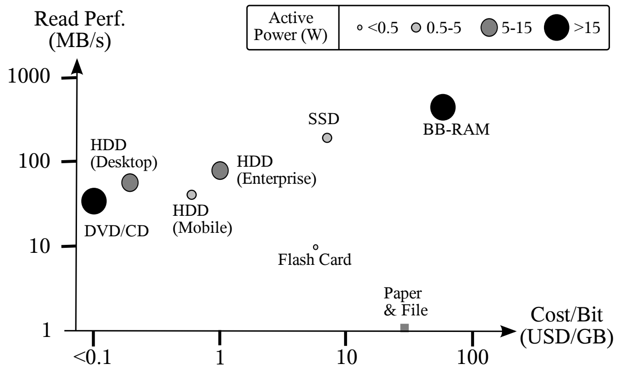


Figure 1: Characteristics of modern storage devices

2 Heterogeneous Storage Devices

The advances in semiconductor and optical storage technology are magnifying the efficiency of hybrid storage architecture. Figure 1 shows the spectrum. Semiconductor storage devices (e.g., solid-state disk, flash card, and battery-backed DRAM) are being used as a mass storage medium [1]. This is because the price of NAND flash becomes cheaper than the price of paper and film, often considered as an entry point of mass storage, and that of DRAM gets closer to this barrier. Also, optical storage (e.g., CD/DVD+RW) can be used for a mutable storage solution thanks to its rewrite support. Not only that, there are several different types of HDD and each has a different characteristic that the hybrid storage system can take advantage of.

The advent of novel storage technology facilitates the use of hybrid storage systems, especially in desktops and entry-level servers where Linux is popular. Let us assume that a device *A* has *x* times higher I/O operations per second (IOPS) than device *B*. If frequently accessed data are stored in *A*, both I/O performance and the ensuing user experience will be improved. The high cost-per-bit ratio of *A* can be compensated by its performance, as device *A* can replace *x* number of device *B* in terms of IOPS. Also the energy consumption can be optimized. If *A* and *B* are used together and *A* absorbs a significant portion of I/O traffic, *B* faces a longer idle period, which gives more chances to the power management technique applied in *B*. If *A* consumes less energy than the energy saving of *B*, the overall energy consumption is lowered.

3 Previous Management Techniques

The efficiency of hybrid storage depends heavily on the I/O locality of the I/O pattern of the storage systems, and how this locality is exploited when placing data between heterogeneous storage devices. Strong locality is typically found in desktops and servers, as they are controlled by humans and human behavior causes this. For example, a file used in yesterday is likely to be used today, implying temporal locality, and a software package has its own working directory where most of its I/O operations are done, producing spatial locality. By storing files having strong locality to a faster device, the overall performance, cost, and energy consumption of a hybrid storage system can be optimized.

Until recent years, data placement between heterogeneous storage media was explicitly controlled by computer administrators. For example, a software package is installed to a fast medium if its expected usage frequency is high. Similarly the administrator stores large multimedia files to a slow but cost-efficient medium. Conventional users are not good at placing and migrating files. Even when they are good at this, automatic management is still desired, as it is more convenient. Automatic management techniques shall abstract the address spaces of heterogeneous storage devices in order to provide a unified view of them to applications and users.

In the previous automatic management techniques, abstraction was done by using the memory hierarchy shown in Figure 2(a). Specifically, a faster medium is used as a nonvolatile cache memory, and block-level management technique is used to place data into nonvolatile memory and evict the data to a permanent storage device. Both in Vista [2] and Solaris ZFS [3], NAND flash storage is utilized as a buffer memory of HDD. Vista identifies blocks used for booting and application launching, and the identified blocks are kept in the flash storage. In ZFS, two SSD devices are used as a write buffer and the storage of file system log data, respectively.

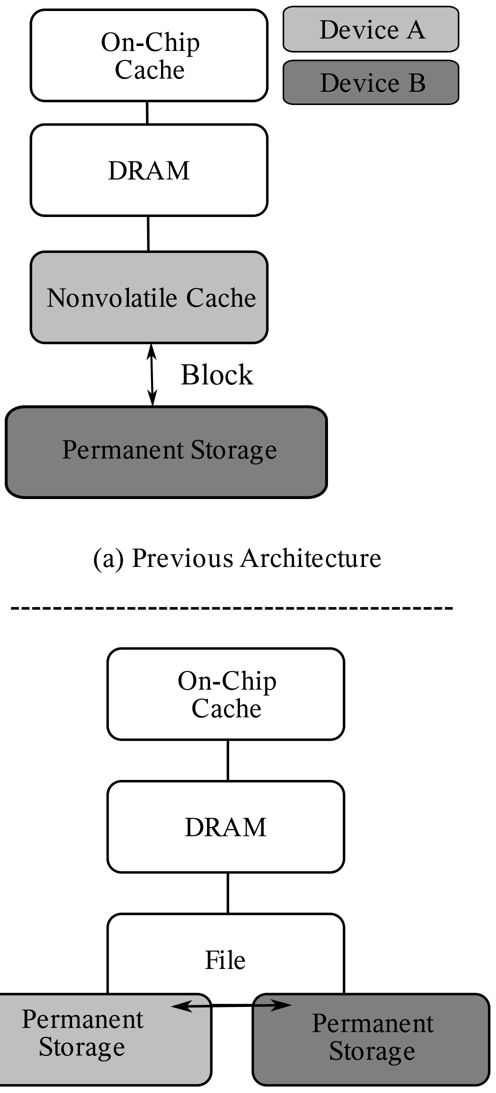


Figure 2: Hybrid storage architectures

4 Proposed Synergy File System

This paper presents the design and analysis of synergy file system (SynergyFS) that can manage hybrid storage architecture in an efficient manner. The following is a summary of the key features of the proposed technique. Because of this, SynergyFS is able to create synergies between heterogeneous file systems and also heterogeneous storage devices.

First, the memory hierarchy assumed by SynergyFS is shown in Figure 2(b). Both faster medium and cheaper medium are used as a permanent storage. Each medium has its own physical file system that is particularly optimized for its I/O characteristics. This is helpful, espe-

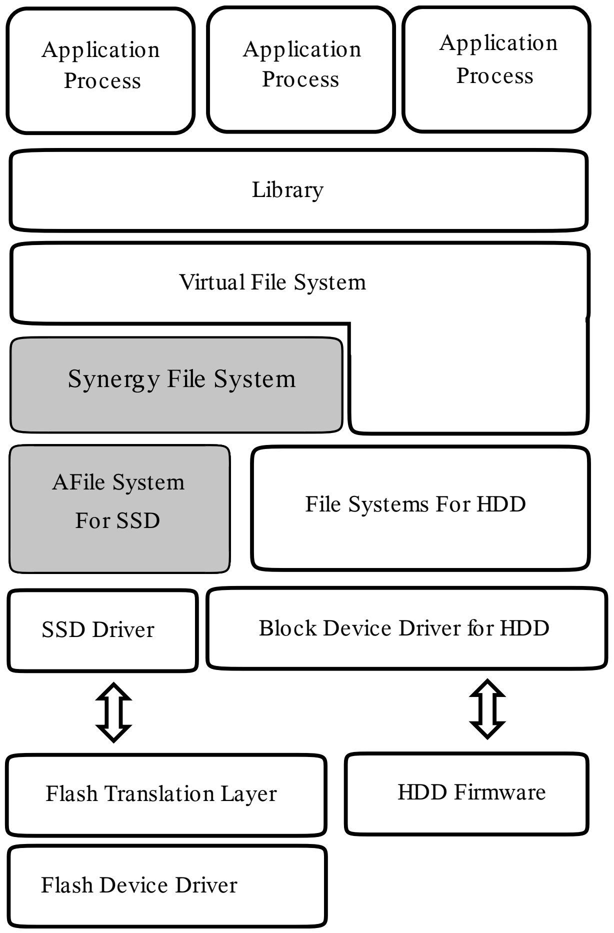


Figure 3: The proposed software architecture

cially in novel storage devices having distinct I/O characteristics.

Second, SynergyFS is a descendant of the stackable file system; thus it unifies the multiple physical file systems it manages. Figure 3 shows the software stack. It has ability to place and move files between the physical file systems. This ability is not available in the existing stackable file systems, as they were not designed for this purpose. Since SynergyFS is an intermediate software layer between virtual and physical file systems, modification of existing kernel modules is not required.

Third, the proposed SynergyFS supports file-level data placement and migration, and this enables the utilization of semantic information (e.g., file type and owner process) that were not accessible in the previous block-level management techniques. It decides where a file will be stored in by taking into account the locality observed in the access pattern of file. On the other hand, the block-level management techniques need a large size of memory to maintain the mapping information and a relatively long lookup time is involved in their I/O operations. Also the block-level management is not good at exploiting the spatial locality. For example, a file can be spread all over the storage address space, but it is difficult for the block-level technique to know which physical blocks belong to which file.

Fourth, SynergyFS concatenates the storage space of physical file systems and thus its space overhead is negligible. However, in the previous techniques, the non-volatile cache capacity is not visible to users and is not added up to the whole storage capacity. As the non-volatile cache size becomes increasingly larger, the proposed scheme is more efficient in terms of user-visible storage capacity.

5 Effectiveness Evaluations

In order to evaluate the effectiveness of hybrid storage architecture, we have analyzed the user experience (UX) of computer systems using an HDD, an SSD, and a device combining these two. The UX affected by storage device includes cost per bit ratio, boot-up time, file system I/O performance, power consumption, and storage device portability. Figure 4 shows the evaluation results of these factors where a generic Linux-based PC system was used.

First, in terms of cost per bit ratio both HDD and SSD/HDD hybrid drive are better than SSD. Currently an SLC(Single-Level Cell) SSD is about five times more expensive than a 2.5-inch HDD having the same capacity. As the price of NAND flash memory is being dropped quickly and high-density NAND flash memory technologies (e.g., Multi-Level Cell) are being commercialized, the price gap between HDD and SSD will be

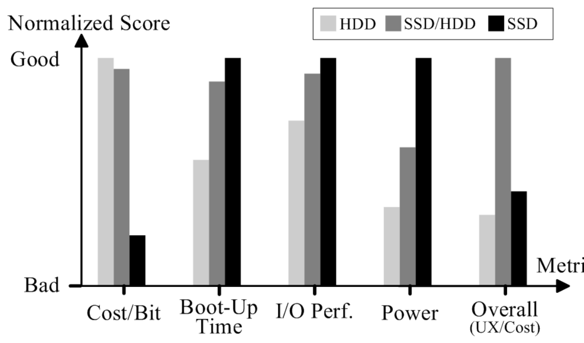


Figure 4: Performance evaluation results

alleviated. In this analysis, the cost per bit ratio of the hybrid drive is about four percent higher than that of HDD. This gap in fact depends on the ratio of SSD and HDD storage capacities used in the hybrid drive. The hybrid drive used in this analysis had a 16GB SSD and a 320GB HDD. SSD capacity was limited to 16GB because this capacity is sufficient to hold all files related to operating system and basic application programs. This shows that with only a small amount of additional cost, hybrid storage devices can be realized in practice.

Second, for the performance evaluation, the system boot-up time and file system I/O performance were analyzed. Application launching time was not taken into accounted as it can be predicted by using the boot-up time analysis result. SSD provides superb I/O performances as compared to HDD thanks to its internal I/O parallelism. The performance of the hybrid drive is comparable to that of SSD, and this shows the effectiveness of the proposed SynergyFS. Specifically, the hybrid drive had a short boot-up time because most files required for boot-up were stored in the SSD part. This drive was also able to provide high-performance file I/Os as the SSD part maintained files that were frequently-accessed. The hybrid drive used in this experiment [4] has a dedicated data path between SSD and HDD. This is helpful at reducing the time required for copying data from one device to the other and the I/O traffics observed in the host system I/O bus.

Third, the energy consumption of storage device was analyzed. Unlike HDD having power-hungry mechanical arm and motor, SSD has no mechanical parts and thus it consumed significantly lower power than HDD. Even the hybrid drive consumed lower power than HDD. In the hybrid drive, most I/O operations were handled by the SSD part. This means that its HDD part had more chances to stay in either idle or stand-by power mode.

On the other hand, in terms of portability SSD is better than both HDD and the hybrid drive. Portability of storage device means the weight, volume, shock resistance, and power consumption. Although the hybrid drive uses a small size of SSD, the volume and weight of the hybrid drive is slightly larger than that of HDD. Tight-integration of HDD and SSD using system-on-chip (SoC) and multi-chip packaging (MCP) technologies can address this problem [4].

Overall effectiveness of these three storage devices is analyzed. The UX score is calculated by multiplying the normalized scores of boot-up time, file system I/O performance, and power consumption. The UX score is then divided by the normalized score of cost per bit ratio, and this is the overall score depicted in Figure 4.

These evaluation results clearly suggest the application area of each type of devices. The hybrid drive has a high potential to replace both HDD and SSD in desktop and workstation computers because it gives better user experience and the extra cost it brings out is tolerable in this market. Note that the hybrid drive consists of both HDD and SSD and thus the amount of HDDs supplied to these computers will not be reduced. Also as HDD is one of the most cost effective solution, it will be widely used in large-capacity storage clusters continuously. On the other hand, SSD is appropriate for high-performance servers where the high cost per bit ratio can be compensated by the fast I/O performance. Portable computers are another application area for SSD because these computers use relatively small size of storage device that can hide the high price of SSD.

6 Conclusion

In this paper, we reviewed the requirement of storage systems and suggested the hybrid storage architecture as a solution. An automatic management technique for the hybrid storage architecture was presented and its effectiveness was analyzed by using an SSD/HDD hybrid storage drive. The proposed SynergyFS can be effectively used for the desktop and workstation computers in order to improve the user experience of the system in a cost-efficient manner.

References

[1] Samsung Electronics, "Samsung Solid-State Drive (SSD)," `http://www.samsung.com/global/business/semiconductor/products/flash/ssd`

[2] Microsoft Corporation, "Windows PC Accelerators: Performance Technology for Windows Vista," `http://www.microsoft.com/whdc/system/sysperf/accelerator.mspx`

[3] Sun Microsystems, "ZFS: the last word in file systems," `http://www.sun.com/2004-0914/feature`

[4] Samsung Electronics, "Samsung S-Drive: Imagine Revolutionary Convergence," `http://www.samsung.com`

Live Migration with Pass-through Device for Linux VM

Edwin Zhai, Gregory D. Cummings, and Yaozu Dong
Intel Corp.
{edwin.zhai, gregory.d.cummings, eddie.dong}@intel.com

Abstract

Open source Linux virtualization, such as Xen and KVM, has made great progress recently, and has been a hot topic in Linux world for years. With virtualization support, the hypervisor de-privileges operating systems as guest operating systems and shares physical resources among guests, such as memory and the network device.

For device virtualization, some mechanisms are introduced for improving performance. Paravirtualized (PV) drivers are implemented to avoid excessive guest and hypervisor switching and thus achieve better performance, for example Xen's split virtual network interface driver (VNIF). Unlike software optimization in PV driver, IOMMU, such as Intel® Virtualization Technology for Directed I/O, AKA VT-d, enables direct passing through of physical devices to guests to take advantage of hardware DMA remapping, thus reducing hypervisor intervention and achieving high bandwidth.

Physically assigned devices impose challenges to live migration, which is one of the most important virtualization features in server consolidation. This paper shows how we solve this issue using virtual hot plug technology, in addition with the Linux bonding driver, and is organized as follows: We start from device virtualization and live migration challenges, followed by the design and implementation of the virtual hotplug based solution. The network connectivity issue is also addressed using the bonding driver for live migration with a direct assigned NIC device. Finally, we present the current status, future work, and other alternative solutions.

1 Introduction to Virtualization

Virtualization became a hot topic in Linux world recently, as various open source virtualization solutions based on Linux were released. With virtualization, the hypervisor supports simultaneously running multiple operating systems on one physical machine by presenting a virtual platform to each guest operating system. There are two different approaches a hypervisor can take to present the virtual platform: full virtualization and paravirtualization. With full virtualization, the guest platform presented consists of all existing components, such as a PIIX chipset, an IDE controller/disk, a SCSI controller/disk, and even an old Pentium® II processor, etc. which can be already supported by modern OS without any modification. Paravirtualization presents the guest OS with a synthetic platform, with components that may not have existed in the real world to date, and thus are unable to run a commercial OS directly. Instead, paravirtualization requires modifications to the guest OS or driver source code to match the synthetic platform, which is usually designed to avoid excessive context switches between guest and hypervisor, by using the underlying hypervisor knowledge, and thus achieving better performance.

2 Device Virtualization

Most hardware today doesn't support virtualization, so device virtualization could only rely on pure software technology. Software based virtualization shares physical resources between different guests, by intercepting guest access to device resource, for example trapping I/O commands from a native device driver running in the guest and providing emulation, that is an emulated device, or servicing hypercalls from the guest front-end paravirtualized drivers in split device model, i.e. a PV device. Both sharing solutions require hypervisor intervention which cause additional overhead, which limits performance.

To reduce this overhead, a pass-through mechanism is introduced in Xen and KVM (work in progress) to allow assignment of a physical PCI device to a specific guest so that the guest can directly access the physical resource without hypervisor intervention [8]. A pass-through mechanism introduces an additional requirement for the DMA engines. A DMA engine transaction requires the host physical address but a guest can only provide the guest physical address. So a method must be invoked to convert a guest physical address to a host physical address for correctness in a non-identical mapping guest and for secure isolation among guests. Hardware IOMMU technologies, such as Intel® Virtualization Technology for devices, i.e. VT-d [7], are designed to convert guest physical addresses to host physical addresses. They do so by remapping DMA addresses provided by the guest to host physical addresses in hardware via a VT-d table indexed by a device requestor ID, i.e. Bus/Device/Function as defined in the PCI specification. Pass-through devices have close to native throughput while maintaining low CPU usage.

PCI SIG I/O Virtualization based Single Root I/O Virtualization, i.e. SR-IOV, is another emerging hardware virtualization technology which specifies how a single device can be shared between multiple guest via a hardware mechanism. A single SR-IOV device can have multiple virtual functions (VF). Each VF has its own requestor ID and resources which allows the VF to be assigned to a specific guest. The guest can then directly access the physical resource without hypervisor intervention and the VF specific requestor ID allows the hardware IOMMU to convert guest physical addresses to host physical addresses.

Of all the devices that are virtualized, network devices are one of the most critical in data centers. With traditional LAN solutions and storage solutions such as iSCSI and FCoE converging on to the network, network device virtualization is becoming increasingly important. In this paper, we choose network devices as a case study.

3 Live Migration

Relocating a virtual machine from one physical host to another with very small down-time of service, such as 100 ms [6], is one major benefit of virtualization. Data centers can use the VM relocation feature, i.e. live migration, to dynamically balance load on different hosting platforms, to achieve better throughput. It can also be used to consolidate services to reduce the number of hosting platforms dynamically to achieve better power savings, or be used to maintain the physical platform after running for a long time because each physical platform has its life cycles, while VMs can run far longer than the life cycle of a physical machine. Live migration, or its similar features, like VM save and VM restore, is achieved by copying VM state from one place to another including memory, virtual devices, and processor states. The virtual platform, where the migrated VM is running, must be the same as the one where it previously ran, and it must provide the capability that all internal states can be saved and restored, which depends on how the devices are virtualized.

The guest memory subsystem, making up the guest platform, is kept identical when the VM relocates, assigning the same amount of memory in the target VM with the same layout. The live migration manager will copy contents from the source VM to the target, using an incremental approach, to reduce the service outage time [5], given that the memory a guest owns may vary from tens of megabytes to tens of gigabytes, and even more in the future, which means a relatively long time to transmit even in a ten gigabit Ethernet environment.

The processor type the guest owns and features the host processor have usually need to be the same across VM migration, but certain exceptions can be taken if all the features the source VM uses exist in the target hosting processor, or if the hypervisor could provide emulation of those features which do not exist on the target side. For example, live migration can request the same CPUID in host side, or just hide the difference in host side by providing the guest a common subset of physical features. MSRs are more complicated, except that the host platform is identical. Fortunately, today's guest platform presented is pretty simple and won't use those model-specific MSRs. The whole CPU context size saved at the final step of live migration is usually in the magnitude of tens of kilobytes, which means just several milliseconds of out of service time.

On the device side, cloning source device instances to the target VM after live migration is much more complicated. If the source VM includes only those software emulated devices or paravirtualized devices, identical platform device could be maintained by generating exactly the same configuration for the target VM startup, and the device state could be easily maintained since the hypervisor knows all of its internal state. Those devices are called migration friendly devices. But for guests who have pass-through devices or SR-IOV Virtual Functions on the source VM side, things are totally different.

3.1 Issues of Live Migration with pass-through device

Although guest with pass-through device can achieve almost native performance, maintaining identical platform device after migration may be impossible. The target VM may not have the same hardware. Furthermore, even if the target guest has the identical platform device as the source side, cloning the device instance to target VM is also almost impossible, because some device internal states may not be readable, and some may be still in-flight at migration time, which is unknown to the hypervisor without the device-specific knowledge. Even without those unknown states, knowing how to write those internal states to the relocated VM is another big problem without device-specific knowledge in the hypervisor. Finally, some devices may have unique information that can't be migrated, such as a MAC address. Those devices are migration unfriendly.

To address pass-through device migration, either the hypervisor needs to have the device knowledge to help migration or the guest needs to do those device-specific operations. In this paper, we ask for guest support by proposing a guest hot plug based solution to request cooperation from the guest to unplug all the migration unfriendly devices before relocation happens, so that we can have identical platform devices and identical device states after migration. But hot unplugging an Ethernet card may lead to network service outage, usually in the magnitude of several seconds. The Linux bonding driver, originally developed for aggregating multiple network interfaces, is used here to maintain connectivity.

4 Solution

This section describes a simple and generic solution to resolve the issue of live migration with pass-through device. This section also illustrates how to address the following key issues: save/restore device state and keeping network connectivity for NIC device.

4.1 Stop Pass-through Device

As described in the previous section, unlike emulated devices, most physical devices can't be paused to save and restore their hardware states, so a consistent device state across live migration is impossible. The only choice is to stop the guest from using physical devices before live migration.

How to do it? One easy way is to let the end user stop everything using a pass-through device including applications, services, and drivers, and then restore them on the target machine after the hypervisor allocates a new device. This method works, but it's not generic, as different Linux distributions have different operations. Moreover, a lot of user intervention is needed inside the Linux guest.

Another generic solution is ACPI [1] S3 (suspend-to-ram), in which the operating system freezes all processes, suspends all I/O devices, then goes into a sleep state with all context lost except system memory. But this is overkill because the whole platform is affected, besides the target device, and service outage time is intolerable. PCI hotplug is perfect in this case, because:

- Unlike ACPI S3, it is a device-level, fine-grained mechanism.
- It's generic, because the 2.6 kernel supports various PCI hotplug mechanisms.
- No huge user intervention, because PCI hotplug can be triggered by hardware.

The solution using PCI hotplug looks like the following:

1. Before live migration, on the source host, the control panel triggers a virtual PCI hot removal event against the pass-through device into the guest.
2. The Linux guest responds to the hot removal event, and stops using the pass-through device after unloading the driver.
3. Without any pass-through device, Linux can be safely live migrated to the target platform.
4. After live migration, on the target host, a virtual PCI hot add event, against a new pass-through device, is triggered.
5. Linux guest loads the proper driver and starts using the new pass-through device. Because the guest reinitializes a new device that has nothing to do with the old one, the limitation described in 3.1 doesn't hold.

4.2 Keeping Network Connectivity

The most popular usage model for a pass-through device is assigning a NIC to a VM for high network throughput. Unfortunately, using PCI NIC hotplug within live migration breaks the network connectivity, which leads to an unpleasant user experience. To address this issue, it is desired that the Linux guest can automatically switch to a virtual NIC after hot removal of the physical NIC, and then migrate with the virtual NIC. Thanks to the powerful and versatile Linux network stack, the Ethernet bonding driver [3] already supports this feature.

The Linux bonding driver provides a mechanism for enslaving multiple network interfaces into a single, logical "bonded" interface with the same MAC address. Behavior of the bonded interfaces depends on modes. For instance, the bonding driver has the ability to detect link failure and reroute network traffic around a failed link in a manner transparent to the application, which is active-backup mode. It also has the ability to aggregate network traffic in all working links to achieve higher throughput, which is referred to as trunking [4].

The active-backup mode can be used for an automatic switch. In this mode, only one slave in the bond is active, while another acts as a backup. The backup slave becomes active if, and only if, the active slave fails. Additionally, one slave can be defined as primary that will always be the active while it is available. Only when the primary is off-line will secondary devices be used. This is very useful when bonding pass-through device, as the physical NIC is preferred over other virtual devices, for performance reasons.

It's very simple to enable bonding driver in Linux. The end user just needs to reconfigure the network before using a pass-through device. The whole configuration in the Linux guest is shown in Figure 1, where a new bond is created to aggregate two slaves: the physical NIC as primary, and a virtual NIC as secondary. In normal conditions, the bond would rely on the physical NIC, and take the following actions in response to hotplug events in live migration:

- When hot removal happens, the virtual NIC becomes active and takes over the in/out traffic, without breaking the network inside of the Linux guest.
- With this virtual NIC, the Linux guest is migrated to target machine.
- When hot add is complete on the target machine, the new physical NIC recovers as the active slave with high throughput.

In this process, no user intervention is required to switch because the powerful bonding driver handles everything well.

4.3 PCI Hotplug Implementation

PCI hotplug plays an important role in live migration with a pass-through device. It should be implemented in the device model, according to the hardware PCI hotplug spec. Currently, the device model of most popular Linux virtualization solutions such as Xen and KVM, are derived from QEMU. Unfortunately, QEMU did not support virtual PCI hotplug when this solution was developed, so we implemented a virtual PCI hotplug device model from scratch.

4.3.1 Choosing Hotplug Spec

The PCI spec doesn't define a standard hotplug mechanism. Here are the three existing categories of PCI hotplug mechanisms:

- **ACPI Hotplug**: This is a similar mechanism as the ACPI dock hot insert/ejection, where some ACPI control methods work with ACPI GPE to service the hotplug.
- **SHPC [2] (Standard HotPlug Controller)**: It's the spec from PCI-SIG to define a complicated controller to handle the PCI hotplug.
- **Vendor-specific**: There are other vendor-specific standards, such as Compaq and IBM, which have their own hardware on servers for PCI hotplug.

Linux 2.6 supports all of the above hotplug standards, which gives us more choices to select a simple, open, and efficient one. SHPC is a really complicated device, so it's hard to implement. Vendor-specific controllers are not well supported in other OS. ACPI hotplug is best suited to being emulated in the device model, because interface exposed to OSPM is very simple and well defined.

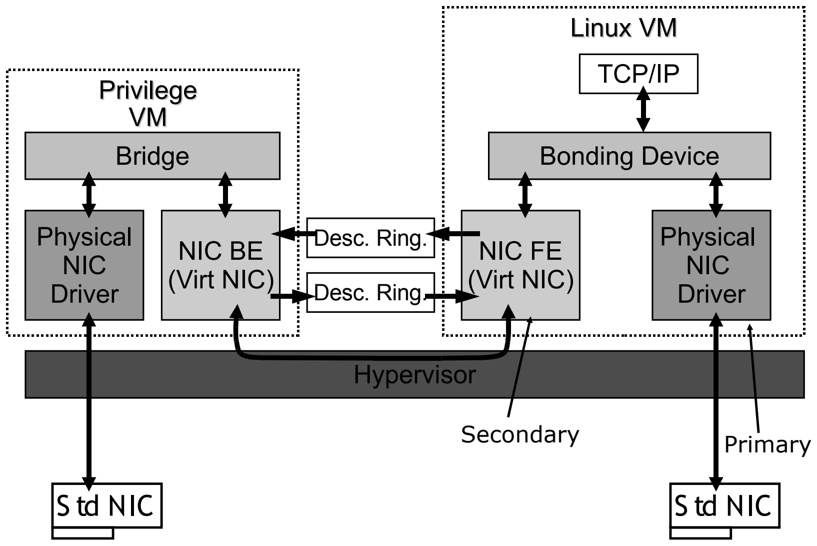


Figure 1: Live Migration with Pass-through Device

4.3.2 Virtual ACPI hotplug

Making an ACPI hotplug controller in device model is something like designing a hardware platform to support ACPI hotplug, but using software emulation. Virtual ACPI hotplug needs several parts in the device model to coordinate in a sequence similar to native. For system event notification, ACPI introduces GPE (General Purpose Event), which is a bitmap, and each bit can be wired to different value-added event hardware depending on design.

The virtual ACPI hotplug sequence is described in Figure 2. When the end user issues the hot removal command for the pass-through device, analogous to pushing the eject button, the hotplug controller updates its status, then asserts the GPE bit and raises a SCI (System Control Interrupt). Upon receiving a SCI, the ACPI driver in the Linux guest clears the GPE bit, queries the hotplug controller about which specific device it needs to eject, and then notifies the Linux guest. In turn, the Linux guest shuts down the device and unloads the driver. At last, the ACPI driver executes the related control method `_EJ0`, to power off the PCI device, and `_STA` to verify the success of the ejection. Hot add is similar to this process, except it doesn't call the `_EJ0`.

In the process shown above, it's obvious that following components are needed:

- **GPE**: A GPE device model, with one bit wired to the hotplug controller, is described in the guest FADT (Fixed ACPI Description Table).
- **PCI Hotplug Controller**: A PCI hotplug controller is needed to respond to the user's hotplug action and maintain the status of the PCI slots. ACPI abstracts a well-defined interface so we can implement internal logic in a simplified style, such as stealing some reserved ioports for register status.
- **Hotplug Control Method**: ACPI control methods for hotplug, such as `_EJ0` and `_STA`, should be added in the guest ACPI table. These methods interact with the hotplug controller for device ejection and status check.

5 Status and Future Work

Right now, hotplug with a pass-through device works well on Xen. With this and the bonding driver, Linux guests can successfully do live migration. Besides live migration, pass-through device hotplug has other useful usage models, such as dynamically switching physical devices between different VMs.

There is some work and investigation that needs to be done in future:

- **High-level Management Tools**: Currently, hotplug of a pass-through device is separated from generic live migration logic for a clean design, so

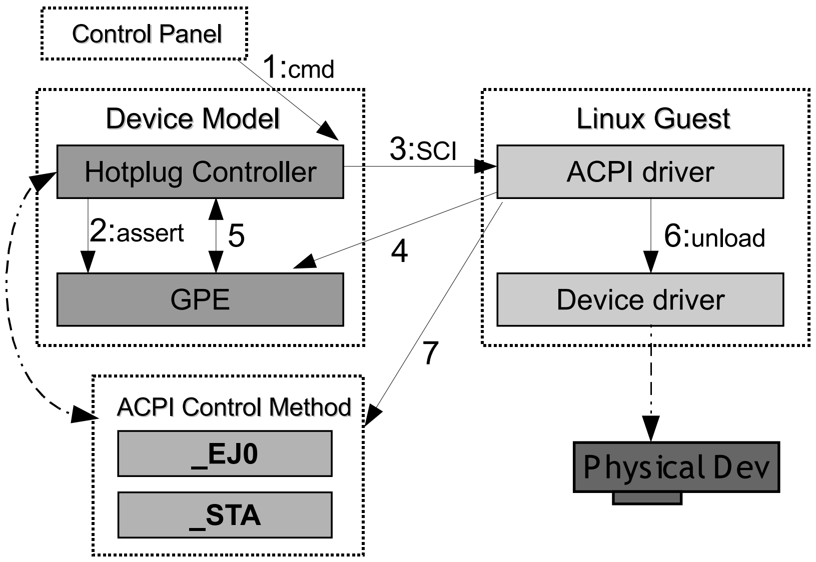


Figure 2: ACPI Hotplug Sequence

the end user is required to issue hotplug commands manually before and after live migration. In the future, these actions should be pushed into high level management tools, such as a friendly GUI or scripts, in order to function without user intervention.

- **Virtual S3**: The Linux bonding driver works perfectly for a NIC, but bonding other directly assigned devices, such as graphics cards, is not as useful. Since Linux has good support for ACPI S3, we can try virtual S3 to suspend all devices before live migration and wakeup them after that. Some drawbacks of virtual S3 need more consideration:

  – All other devices, besides pass-through devices, go into this loop too, which takes more time than virtual hotplug.

  – With S3, the OS is in sleep state, so a long down time of the running service is unavoidable.

  – S3 has the assumption that the OS would wake up on the same platform, so the same type of pass-through devices must exist in the target machine.

  – S3 support in the guest may not be complete and robust.

  Although virtual S3 for pass-through device live migration has its own limitation, it is still useful in some environments where virtual hotplug doesn't work, for instance, hot removal of pass-through display cards which are likely to cause a guest crash.

- **Other Guest**: Linux supports ACPI hotplug and has a powerful bonding driver, but other guest OS may not be lucky enough to have such a framework. We are in the process of extending support to other guests.

6 Conclusion

VM direct access of physical device achieves close to native performance, but breaks VM live migration. Our virtual ACPI hotplug device model allows VM to hot remove the pass-through device before relocation and hot add another one after relocation, thus making pass-through devices coexist with VM relocation. By integrating the Linux bonding driver into the relocation process, we enable continuous network connectivity for directly assigned NIC devices, which is the most popular pass-through device usage model.

References

[1] "Advanced Configuration & Power Specification," Revision 3.0b, 2006, Hewlett-Packard, Intel, Microsoft, Phoenix, Toshiba. `http://www.acpi.info`

[2] "PCI Standard Hot-Plug Controller and Subsystem Specification," Revision 1.0, June, 2001, `http://www.pcisig.info`

[3] "Linux Ethernet Bonding Driver," T. Davis, W. Tarreau, C. Gavrilov, C.N. Tindel *Linux Howto Documentation, April, 2006.*

[4] "High Available Networking," M. John, *Linux Journal, January, 2006.*

[5] "Live Migration of Virtual Machines," C. Clark, K. Fraser, S. Hand, J.G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfiled, In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation*, 2005.

[6] "Xen 3.0 and the Art of Virtualization," I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Mallick, In *Proceedings of the Linux Symposium* (OLS), Ottawa, Ontario, Canada, 2005.

[7] "Intel Virtualization Technology for Directed I/O Architecture Specification," 2006, `ftp://download.intel.com/technology/computing/vptech/Intel(r)_VT_for_Direct_IO.pdf`

[8] "Utilizing IOMMUs for Virtualization in Linux and Xen," M. Ben-Yehuda, J. Mason, O. Krieger, J. Xenidis, L.V. Doorn, A. Mallick, and J. Nakamima, In *Proceedings of the Linux Symposium*, Ottawa, Ontario, Canada, (OLS), 2006.