# Proceedings of the Linux Symposium

July 13th–17th, 2009
Montreal, Quebec
Canada

# Contents

**Autotest — Testing the Untestable** 9
*John Admanski & Steve Howard*

**Increasing memory density by using KSM** 19
*Andrea Arcangeli*

**Sandboxer: Light-Weight Application Isolation in Mobile Internet Devices** 29
*R. Banginwar, M. Leibowitz, & T. Tanaka*

**Dynamic Debug** 39
*Jason Baron*

**Measuring Function Duration with Ftrace** 47
*Tim Bird*

**The Simple Firmware Interface** 55
*A. Leonard Brown*

**The Corosync High Performance Shared Memory IPC Reusable C Library** 61
*Steven C Dake*

**GStreamer on Texas Instruments OMAP35x Processors** 69
*D. Darling, C. Maupin, & B. Singh*

**From Fast to Predictably Fast** 79
*Dominic Duval*

**Combined Tracing of the Kernel and Applications with LTTng** 87
*Pierre-Marc Fournier*

**Twenty Years Later: Still Improving the Correctness of an NFS Server** 95
*R. Gardner, S. D'Angelo, & M. Sears*

**Memory Migration on Next-Touch** 101
*Brice Goglin & Nathalie Furmento*

**Non Privileged User Package Management:**
**Use Cases, Issues, Proposed Solutions** 111
*François-Denis Gonthier & Steven Pigeon*

**GeoDNS—Geographically-aware, protocol-agnostic load balancing at the DNS level** **123**
*John Hawley*

**Porting to Linux the Right Way** **131**
*Neil Horman*

**Tracing the HA Cluster of Guests with VESPER** **141**
*S. Kim, S. Moriya, & S. Oshima*

**Hardware Breakpoint (or watchpoint) usage in Linux Kernel** **149**
*Prasad Krishnan*

**Shoot first and stop the OS noise** **159**
*Christopher Lameter*

**Tuning 10Gb network cards on Linux** **169**
*B.H. Leitao*

**A day in the life of a Linux kernel hacker...** **185**
*John W. Linville*

**Transcendent Memory and Linux** **191**
*Dan Magenheimer*

**Incremental Checkpointing for Grids** **201**
*John Mehnert-Spahn*

**Putting LTP to test—Validating both the Linux kernel and Test-cases** **209**
*Subrata Modak*

**Linux-based virtualization for HPC clusters** **221**
*L. Nussbaum, F. Anhalt, O. Mornard, & J.-P. Gelas*

**I/O Topology** **235**
*Martin K. Petersen*

**Step two in DCCP adoption: The Libraries** **239**
*L.M. Sales, H. Stuart, H.O. Almeida, & A. Perkusich*

**Programmatic Kernel Dump Analysis On Linux** **251**
*Alex Sidorenko*

**Online Hierarchical Storage Manager** **263**
*S.K. Sinha, R.B. Agrawal, V. Agarwal, R. Vashist, R.K. Sharma, & S. Hendre*

**Effect of readahead and file system block reallocation for LBCAS** **275**
*K. Suzaki, T. Yagi, K. Iijima, N.A. Quynh, & Y. Watanabe*

**Scaling software on multi-core through co-scheduling of related tasks** **287**
*Srivatsa Vaddagiri*

**Converged Networking in the Data Center** **297**
*Peter P. Waskiewicz Jr.*

**How to (Not) Lose Your Data** **303**
*Ric Wheeler*

**Testing and verification of cluster filesystems** **311**
*Steven Whitehouse*

**Fixing PCI Suspend and Resume** **319**
*Rafael J. Wysocki*

**Real-Time Performance Analysis in Linux-Based Robotic Systems** **331**
*H. Yoon, J. Song, & J. Lee*

## Conference Organizers

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

## Programme Committee

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

James Bottomley, *Novell*
Bdale Garbee, *HP*
Dave Jones, *Red Hat*
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Alasdair Kergon, *Red Hat*
Matthew Wilson, *rPath*

## Proceedings Committee

Robyn Bergeron
Chris Dukes, *workfrog.com*
Jonas Fonseca
John 'Warthog9' Hawley

**With thanks to**
John W. Lockhart, *Red Hat*

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

# Autotest — Testing the Untestable

John Admanski
*Google Inc.*
jadmanski@google.com

Steve Howard
*Google Inc.*
showard@google.com

## Abstract

Increased automated testing has been one of the most popular and beneficial trends in software engineering. Yet low-level systems such as the kernel and hardware have proven extremely difficult to test effectively, and as a result much kernel testing has taken place in a manual and relatively ad-hoc manner. Most existing test frameworks are designed to test higher-level software isolated from the underlying platform, which is assumed to be stable and reliable. Testing the underlying platform itself requires a completely new set of assumptions and these must be reflected in the framework's design from the ground up. The design must incorporate the machine under test as an important component of the system and must anticipate failures at any level within the kernel and hardware. Furthermore, the system must be capable of scaling to hundreds or even thousands of machines under test, enabling the simultaneous testing of many different development kernels each on a variety of hardware platforms. The system must therefore facilitate efficient sharing of machine resources among developers and handle automatic upkeep of the fleet. Finally, the system must achieve end-to-end automation to make it simple for developers to perform basic testing and incorporate their own tests with minimal effort and no knowledge of the framework's internals. At the same time, it must accommodate complex cluster-level tests and diverse, specialized testing environments within the same scheduling, execution and reporting framework.

Autotest is an open-source project that overcomes these challenges to enable large-scale, fully automated testing of low-level systems and detection of rare bugs and subtle performance regressions. Using Autotest at Google, kernel developers get per-checkin testing on a pool of hundreds of machines, and hardware test engineers can qualify thousands of new machines in a short time frame. This paper will cover the above challenges and present some of the solutions successfully employed in Autotest. It will focus on the layered system architecture and how that enables the distribution of not only the test execution environment but the entire test control system, as well as the leveraging of Python to provide simple but infinitely extensible job control and test harnesses, and the automatic system health monitoring and machine repairs used to isolate users from the management of the test bed.

## 1 Introduction

Autotest is a framework for fully automated testing of low-level systems, including kernels and hardware. It is designed to provide end-to-end automation for functional and performance tests against running kernels or hardware with as little manual setup as possible. This automation allows testing to be performed with less wasted effort, greater frequency, and higher consistency. It also allows tests to be easily pushed upstream to various developers, moving testing earlier into the development cycle.

Using Autotest, kernel and hardware engineers can achieve much greater test coverage than such components usually receive. This typical lack of effective low-level systems testing comes with good reason: automated testing of such systems is a difficult task and presents many challenges distinct from userspace software testing. This paper introduces the requirements Autotest aims to meet and some of the unique challenges that arise from these requirements, including robust testing in the face of system instability, scaling to thousands of test machines, and minimizing complexity of test execution and test development. The paper will discuss solutions for each of these challenges that have been employed in Autotest to achieve effective, fully automated low-level systems testing.

## 2 Background

High-quality automated testing is a necessity for any large, long-lived software project to maintain stability

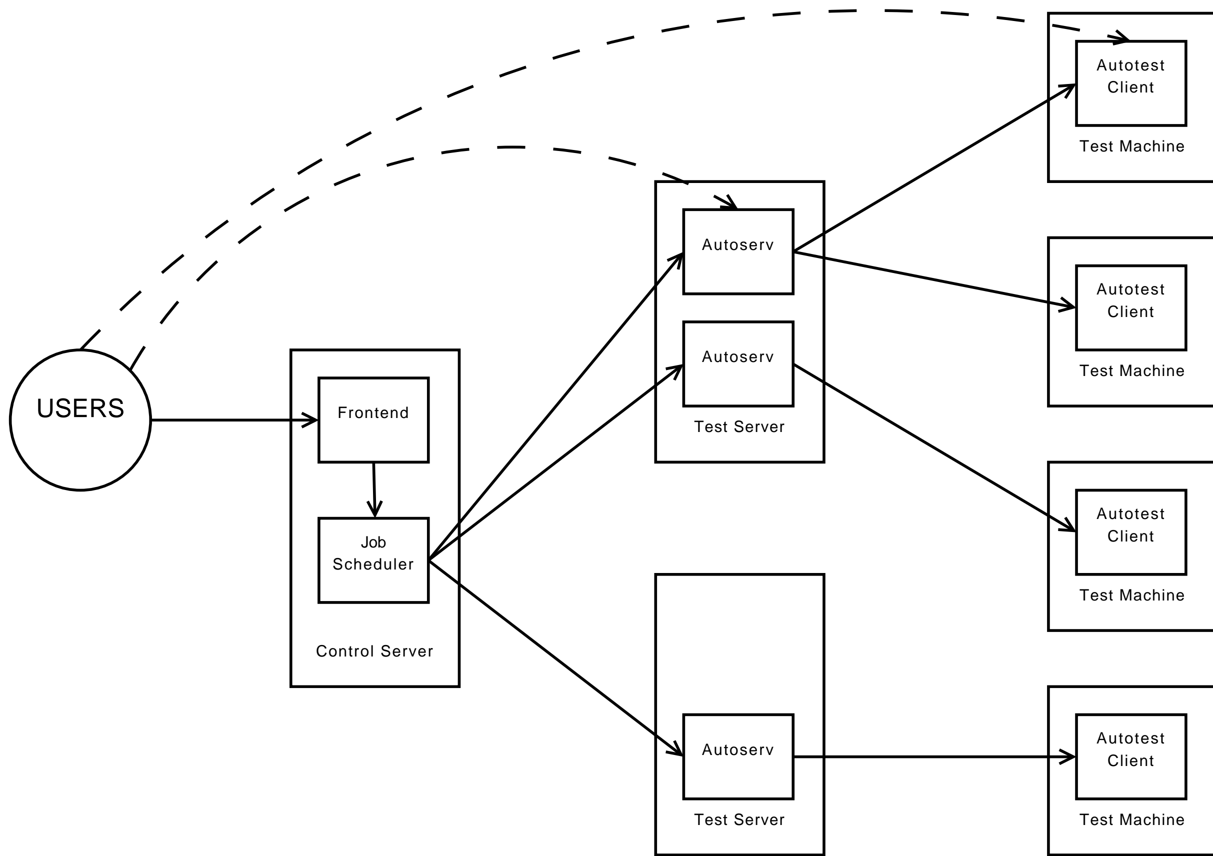



Figure 1: High level operation of a complete Autotest system

while permitting rapid development. This is as true for the Linux kernel and other system software as it is for user-space software. However, so far the benefits of automated testing have been most successfully realized within user-space applications.

Most existing test automation frameworks are targeted at software running on top of the platform provided by the hardware and operating system, the realm in which nearly all software operates. By taking advantage of the assumption that an application is running in a reliable standardized environment provided by the platform, a framework can abstract away and simplify most of the underlying system. When attempting to provide the same services for kernel (and hardware) testing, this assumption is no longer reasonable since the underlying system is an integral component of what is being tested. This was part of the original motivation for the development of the first versions of Autotest and its predecessor, IBM Autobench[5][4].

Autotest begins with the goal of testing the underlying platform itself, and this goal engenders a unique set of requirements. Firstly, because the platform on which Autotest runs is itself under test, Autotest must be built from the ground up to assume system instability. This requires graceful handling of kernel panics, hardware lockups, network failures, and other unexpected failures. In addition, tasks such as kernel installation and hardware configuration must be simple, commonplace activities in Autotest.

Secondly, because the platform under test cannot be easily virtualized, every running test requires a physical machine. Hardware virtualization may be used for basic kernel testing, but as it fails to produce accurate performance results and can mask platform-specific functional issues it is useful only for the most basic kernel functional verification. Autotest is therefore built to run every test on a physical machine, both for kernel and hardware testing. This makes coordination among multiple machines a core necessity in Autotest and furthermore implies that scaling requires distribution of testing among hundreds or even thousands of machines. This additionally creates a need for a system of efficient sharing of test machines between users to maximize utilization over such a large test fleet.

Finally, Autotest must fulfill the generic requirements of any testing framework. In particular, Autotest must minimize the overhead imposed on test developers. It must be trivial to incorporate existing tests, easy to write simple new tests, and possible to write complex multi-process or multimachine tests, all within the same basic framework. Furthermore, developing tests should be a simple, familiar process, requiring interaction with only a small subset of the available infrastructure. Tests must therefore be easily executable by hand and simultaneously pluggable into a large-scale scheduling system. These levels of abstraction are broken down into distinct modules discussed in more detail throughout this paper.

As illustrated in Figure 1, the lowest layer of the system is the Autotest client, a simple test framework that runs on individual machines. The next layer, Autoserv, is designed to run on centralized test servers to automatically install and execute clients and to coordinate multi-machine tests. The outermost layer consists of a single frontend and job scheduler to allow multiple users to share a single test fleet and results repository. Note that the dependencies go in only one direction making the design more modular and allowing users to interact with the system on multiple levels. On a large scale users can push a button on a web interface to launch a complete test suite on a large cluster of machines while on a small scale users can run a single test on a local workstation by executing a shell command.

### 2.1 Related work

The Linux Test Project "has a goal to deliver test suites to the open source community that validate the reliability, robustness, and stability of Linux"[1]. It is a collection of functional and stress tests for the Linux kernel and related features as well as a client infrastructure for test execution. The client infrastructure eases the execution of a many tests (there are over 3,000 tests included), supports running tests in parallel, can generate background stress during test execution, and generates a report of test results at the end of a run. LTP is not, however, intended to be a general-purpose, fully-automated kernel testing framework. There are a number of Autotest goals that are specifically non-goals of LTP[8]. It is essentially a collection of tests and is therefore suitable for inclusion into Autotest as a test, and indeed such inclusion has been easily done.

An automation framework called Xentest was developed for testing the Xen virtualization project. David Barrera *et al.* note that "testing Linux under Xen and testing Linux itself are very much alike" and perform part of their testing by "running standard test suites under Linux running on top of Xen", including LTP[3]. Since testing Xen is much like testing the underlying hardware itself the goals of Autotest share much in common with those of Xentest, both from a kernel testing and a hardware testing point of view. Xentest is a collection of scripts with support for building and booting Xen, running tests under it, and gathering results logs together. It does not support any automated analysis of test results to determine pass/fail conditions. Test runs are configurable by a control file using the Python ConfigParser module. This provides simple configuration but lacks any programmatic power within control files. Finally, Xentest is built closely around Xen and does not aim to be generic framework for kernel or hardware testing. On the other hand, Autotest could be used to perform Xen testing much like Xentest does and some work has been done on this in the past.

Crackerjack is another test automation system, one designed specifically for regression testing[10]. It focuses on finding incompatible API changes between kernel versions. This is valuable testing but is a narrower focus from that of Autotest.

Two frameworks that address the problem of distributed kernel testing are PyReT[6] and ANTS[2]. The former depends on a shared file system for all communications while the latter uses a serial console. Both of these requirements on test machines were deemed too restrictive for Autotest, which relies solely on an SSH connection for communications. ANTS is quite robust to test machine failures, as it configures all test machines from scratch using network booting and is capable of using remote power control to reset and recover machines that have become unresponsive. The system additionally includes a machine reservation tool so that machines can be shared between developers and the automated system without conflict. These are all important features that have found their way into Autotest. However, the system is built strictly for nightly testing and does not support a general queue of user-customizable jobs. It includes very limited results analysis in the form of an email report upon completion of the night's tests. It runs a number of open-source tests (including LTP) but does not support more complex, multimachine tests. Finally, the system is proprietary and therefore of little direct

utility to the community.

For distributed performance testing of the kernel there exist systems presented by Alexander Ufimtsev[9] and Tim Chen[7]. In both systems, test machines operate autonomously, running a client harness which monitors the kernel repository, building and testing new releases as they appear. In this sense, the systems are built around the specific purpose of per-release testing, although the latter system includes support for testing arbitrary patches on any kernel. Both systems' clients transmit results to a central repository, a remote server in the former case and a shared database in the latter. The former system includes some automated analysis for regression detection based on differences from previous averages, a task not yet implemented in Autotest. The latter system includes a web frontend displaying graphs of each benchmark over kernel versions, with support for displaying profiler information, rerunning tests or bisecting to find the patch responsible for a regression. Autotest includes partial support for these features but could benefit from improvements in this area.

## 3 Autotest Client

The most basic requirement that Autotest is intended to fulfill is to provide an environment for running tests on a machine in a way that meets the following criteria:

1. The lowest, most bare-metal access must be available.
2. Test results are available in a standard machine-parseable way.
3. Standard tests developed outside of the framework can be easily run within it.

The first of the criteria, low-level system access, seems fairly self-evident when writing tests which are aimed at the kernel and the hardware itself. To test a particular component of a system, the test must be written using tools that have access the standard API for that component. Since C is the lingua franca of the systems world, a C API can generally be counted on as being available, but even that isn't always the case. When creating a file system during a test, `mkfs` is going to be the easiest and most readily available mechanism; so as well as being able to easily incorporate custom C the framework must also make it easy to work with external tools.

This initial requirement could have been satisfied by writing the framework itself in C, but that would ultimately have conflicted with the other requirements that Autotest was expected to meet. First, this would've made calling out to external applications ultimately more difficult; while functions like `fork`, `exec`, `popen` and `system` provide all the basic mechanisms needed to launch an external process and collect results from it, working with them in C requires a relatively large amount of boilerplate compared to a higher-level scripting language such as Perl or Python. This only becomes more true if the output of the executed process needs to be manipulated and/or parsed in any way. The second requirement that test results be logged in a standard way almost guarantees that the test will need to do string manipulation, another task simplified by using a scripting language.

To meet these somewhat conflicting requirements, the Autotest framework itself was written in Python, with utilities provided to simplify the compilation and execution of C code. Tests themselves are implemented by creating a Python module defining a test subclass, satisfying a standardized, pre-defined interface. Individual tests are packaged up in a directory and can be bundled along with whatever additional resources are needed, such as data files, C code to be compiled and executed or even pre-compiled binaries if necessary.

This also satisfies the third of the three requirements, the ability to run standard tests written independently of Autotest. All that is required is to bundle the components necessary for the test with a simple Python wrapper. The wrapper is responsible for setting up any necessary environment, executing the underlying test, and translating the results from the form produced by the test into Autotest standard logging calls. The wrappers are generally quite simple; the median size of a test wrapper in the current Autotest distribution is only 38 lines.

Using Python for implementing tests also provides an easy mechanism for bundling up suites of tests or customizing the execution of specific tests. Tests themselves are executed by writing a "control file" which is simply a Python script executing in a predefined environment. It can be a single line saying "execute this test", a more complex script that executes a whole sequence of tests, or even a script that conditionally executes tests depending on what hardware and kernel are running on the machine. The environment provided by Autotest contains additional utilities that allow control

files to put the machine into any state necessary for executing tests, even if it requires installing a kernel and rebooting the machine. Having the full power of Python available allows test runners to perform limitless customization without having to learn a custom job control language.

This power does come with one major drawback, though. Due to the dynamic nature of Python and the power available to control files, it is impossible to statically determine much information about a job. For example, it is impossible to know in advance what tests a job will run, and indeed the set of tests run may potentially be nondeterministic. This limitation has not been severe enough to outweigh the benefits of this approach.

### 3.1 Installation Problems

As this system was put into use at Google, the installation of Autotest onto test machines quickly became a serious performance issue. Allowing test developers to bundle data, source code and even binaries with their tests made it easy to write tests but allowed the installation size to grow dramatically. The situation could be somewhat alleviated by minimizing how often an install was necessary, but in practice this only helps if the test framework can be pre-installed on the systems.

The solution to this problem is a fairly standard one: rather than treating Autotest and its test suite as a single, monolithic package, break it up into a set of packages:

- a core package containing the framework itself
- packages for the various utilities and dependencies such as profilers, compilers and any non-standard system utilities that would need to be installed
- packages for the individual tests

Each package is able to declare other packages as dependencies. The core package can be installed everywhere and is fairly lightweight, consisting only of a set of Python source files without any of the more heavyweight data and binaries required by some tests. When executing a job, the framework is then able to dynamically download and install any packages needed to execute a specific test.

## 4 Autotest Server

### 4.1 Distributing test runs across machines

The Autotest client provides sufficient infrastructure for running low-level tests but it only executes tests and collects results on a single machine. To test a kernel on multiple hardware configurations, a tester would need to install the test client on multiple machines, manually run jobs on each of these machines, and examine the results scattered across these systems.

This deficiency led to the development of Autoserv, an Autotest Server, a separate layer designed around the client. It allows a user to run a test by executing a server process on a machine other than the test machine. The server process will connect to the remote test machine via SSH, install an Autotest client, run a job on the client, and then pull the results back from the test machine. Localizing these server runs to a single machine allows users to run test jobs on arbitrary sets of machines while collecting all the results into a central location for analysis.

### 4.2 Recovering failed test systems

Once users start running tests on larger sets of machines, dealing with crashed systems becomes a much more common occurrence. As the number of test machines increases, bad kernels (and random chance) are going to result in more failed systems. When testing on a single machine, manual intervention is the simplest method of dealing with failure, but this does not scale to hundreds or thousands of machines. Automation becomes necessary with two major requirements:

- Automatically detect and report on test machine failures
- Provide a mechanism for repairing broken systems

Handling these requirements entirely within the client running on the test machine is impractical; detecting and reporting a kernel panic or hardware failure will not even be possible when the crash kills the test processes on the machine. Similarly, repair may require re-imaging a machine which will wipe out the client itself.

With job execution controlled from a remote machine, handling these requirements becomes feasible. Autoserv implements support for monitoring serial console output, network console output and general syslog output in `/var/log`. It can also interact with external services that collect crash dumps and even power cycle the machine if that capability is available. In the very worst case the server process can at least clearly log the failure of the job (and any tests it was running) along with the last known state of the failed test machine.

Automated repair can also be performed. This is implemented in Autoserv in an escalating fashion, first by making several attempts to put the machine back into a known good state, then by optionally calling out to any local infrastructure in place to carry out a complete reinstallation of the machine, and finally, if necessary, by escalating the repair process to a human. Testing on large numbers of machines now becomes much more practical when systems broken by bad kernels (or bad tests) can be put back into a working state with a minimum of human intervention.

### 4.3 Multi-machine tests

Remote control of test execution also introduces the opportunity to run single tests that span multiple machines. While this could be done with the Autotest client alone by running the client on a master test system and having it drive other slave test systems, this would require duplicating most of the "remote control" infrastructure from the server directly into the client. This could also be problematic from a security point of view since, rather than routing control through a single server, the test machines would require much more liberal access to one another.

Since Autotest already established the need for a separate server mechanism, it was natural to extend it to support "server-side" testing. Instead of only providing a fixed set of server operations (install client and run job, repair, etc.), Autoserv allows testers to supply a Python control file for execution on the server, just like on the client. This can be used to implement, for example, a network test with the following flow:

- Install Autotest client on two machines
- Launch "network server" job on one machine
- Launch "network client" job on one machine
- Wait for both jobs to complete and collect results

No single-machine networking test can duplicate the same results, particularly when attempting to quantify networking performance and not just test the stability of the network stack.

This also allows for execution of larger-scale cluster testing. Although this begins to creep beyond the scope of systems testing it still has significant value, not as a way to test the cluster applications but rather as a way of testing the impact of kernel and hardware changes on larger-scale applications. A smaller-scale cluster test can follow a workflow similar to that for network testing. Alternatively, a server job can make use of preexisting cluster setup and management tools, simply driving the external services and collecting results afterwards.

### 4.4 Mitigating Network Unreliability

While one of the primary goals of Autoserv is to increase reliability, it also introduces new unreliabilities as an unfortunate side effect. The primary issue is that it introduces a new point of failure, the connection between the server and the client machines. Working directly with the client, a user can launch a job on a machine and return after expected completion, and any transient network issues will not affect the test result. This is no longer the case when the job is being controlled by a remote server that continuously monitors the test machine. The problem can be alleviated somewhat by periodically polling the remote machine rather than continually monitoring it, but ultimately this only reduces susceptibility to the problem.

Implementing more reliable communications over OpenSSH ultimately proved too difficult, primarily due to the lack of control over and visibility into network failure modes. One alternative considered was to use a completely separate communication mechanism, but this was rejected as impractical. Using SSH provides Autotest with a robust and secure mechanism for communication and remote execution, without requiring the large investment of time and labor required to invent a custom protocol that would then need to be installed on every test machine.

Instead the solution was to add an alternative SSH implementation that uses a Python package (paramiko[1])

[1] `http://www.lag.net/paramiko/`

instead of launching an external OpenSSH process. Using an in-process library allowed tighter integration and communication between Autoserv and the SSH implementation, allowing the use of long-lived SSH connections with automatic recovery from network failure. At the same time modifications were made to the Autotest client to allow it to be run as a detachable daemon so that the automatic connection recovery could re-attach to clients with no impact on the local testing.

Adding paramiko support had the additional benefit of reducing the overhead of executing SSH operations from Autoserv by performing them in-process, as well as simplifying the use of multi-channel SSH sessions to avoid the cost of continually creating and terminating new sessions. Within Autoserv this is implemented in such a way that the paramiko-based implementation can be used as a drop-in replacement for the OpenSSH-based one, allowing testers to make use of whichever is better suited to their needs. OpenSSH works better "out of the box" with most Linux configurations, while paramiko, which requires more setup and configuration, ultimately allows for more reliable, lightweight connections.

## 5 Scheduler and Frontend

### 5.1 Shared machine pool

Autoserv provides a convenient and reliable way for individual users to test small numbers of platforms. As a standalone application, however, it cannot possibly fulfill the requirement of scaling to thousands of machine and achieving efficient utilization of a shared machine pool. To address these needs the Autotest service architecture provides a layer on top of Autoserv that allows Autotest to operate as a shared service rather than a standalone application. Rather than execute the Autotest client or server directly, users interact with a central service instance through a web- or command-line-based interface. The service maintains a shared machine pool and a global queue of test jobs requested by users. There are three major components that make this usage model possible. The Autotest Frontend is an interface for users to schedule and monitor test jobs and manage the machine pool. The Autotest Scheduler is responsible for executing and monitoring Autoserv to run tests on machines in the pool in response to user requests. Finally, the results analysis interface, not discussed in this paper, provides a common interface to view, aggregate and analyze test results.

The Autotest Frontend is a web application for scheduling tests, monitoring ongoing testing, and managing test machines. It operates on a database which takes the available tests, the machines in the shared test bed, and the global queue of test jobs that have been scheduled by users. The scheduler interacts with the frontend through this database, executing test jobs that have been scheduled and updating the statuses of jobs and machines based on execution progress.

The frontend supports a number of features to help users organize the machine pool. First, the system supports access control lists to restrict the set of users that can run tests on certain machines. Some machines may be open for general testing, but some users, particularly hardware testers, will have dedicated machines that cannot be used by others. Second, the system supports tagging of machines with arbitrary labels. The most common usage of this feature is to mark the platform of a machine, which is often important for both job scheduling and results analysis. Labels can additionally be used to declare machine capabilities, such as remote power control, or to group together large numbers of machines for easier scheduling.

The scheduler is a daemon running on the server whose primary purpose is to execute and monitor Autoserv processes. The scheduler continuously matches up scheduled test jobs with available machines, launches Autoserv processes to execute these jobs, and monitors these processes to completion. It updates the database with the status of each job throughout execution, allowing the user to track job progress. Upon completion, the scheduler executes a parser to read Autoserv's structured results logs into a database of test results. The user can then perform powerful analysis of these results through a special results analysis interface.

An important feature of the scheduler is its statelessness. While it maintains plenty of in-memory state, all important state can be reconstructed from the database. This is exactly what happens upon scheduler startup, ensuring that when the scheduler needs to restart, all tests will continue running uninterrupted and machine time won't be wasted. This is critical for minimizing user impact during deployments of new Autotest versions or after a scheduler crash.

In addition, as the test fleet scales to thousands of machines, automated fleet health management becomes critical. To this end, the scheduler takes advantage of Autoserv's machine diagnosis and repair functionality. The scheduler launches special Autoserv processes to verify machine health before each job and perform repairs as necessary. Machines that cannot be repaired are marked as such in the database, from which a machine health dashboard can read and summarize machine health data. Additionally, the scheduler performs periodic reverification of known dead machines to catch any manual repairs that may have occurred.

### 5.2 Distributed execution for scalability

When all Autoserv processes are running on a single server, serious performance degradation tends to set in around 1,000 simultaneous machines under test. The scheduler supports global throttling of running processes to avoid bringing the system to a halt, but this still leaves a scalability limit imposed by the hardware itself. To alleviate the problem and allow for further scaling, the scheduler supports distributing Autoserv processes among a pool of servers.

A single scheduler coordinates execution among multiple servers and all results are centralized on a single archive server after execution completes. Each server can support roughly 1,000 machines under test, and to date no Autotest installation has reached a limit on the number of servers that can be utilized in the system. In addition to increasing scalability, distributed execution increases system reliability. Since execution servers are completely independent of each other, each can fail completely without bringing the entire service to a halt. With this distributed execution model, the Autotest service at Google has scaled to approximately 5,000 simultaneous machines under test.

### 5.3 Automatic generation of control files

To run a single test, users of Autoserv can run one of the existing control files written for each test. However, in order to run multiple tests within a single execution the user must write a custom control file. While control files have been kept as simple as possible, writing a custom control file still presents a major barrier to entry for new users. To this end, the Autotest Frontend simplifies the process of running multiple tests by support automatic generation of control files.

Creating a job through the frontend consists of selecting a number of tests, a number of machines, and a variety of job options. The user can select tests from a list, which includes a description of each test, and the frontend will automatically generate a control file to run the selected tests. Users may also specify a kernel to install and select profilers to enable during testing and the generated control file will incorporate all of these options. This allows users to run moderately complex jobs through Autotest with ease, without requiring any knowledge of control files. Machines can be similarly selected from a list, either one-by-one or in bulk based on filtering by hostname or platform (or any other machine label). Furthermore, users may request that the job run on any machine of a particular platform and allow the scheduler to select one at run time. This feature helps increase utilization of shared test machines and makes it particularly easy to run automated jobs without a static, dedicated set of machines.

### 5.4 Support for high-level automation

The bulk of the work for the web frontend is performed on the web server, which operates primarily as an RPC server. It is written in Python using the Django[2] web framework and communicates with a MySQL[3] database. The web interface is a fully-fledged application running in the browser implemented using Google Web Toolkit[4]. It communicates with the server solely through the RPC interface. There is also a command-line interface, implemented in Python, which communicates with the server through the same RPC interface. This is made possible by the use of the lightweight JSON[5] data-interchange format which is easily implemented in either language. Furthermore, custom scripts can be written that access the RPC interface directly, providing the full capabilities of the web frontend through a simple interface. This supports powerful and easy high-level automation, allowing users to extend the functionality of Autotest with external scripts layered on top of the frontend.

[2] `http://www.djangoproject.com/`
[3] `http://www.mysql.com/`
[4] `http://code.google.com/webtoolkit/`
[5] `http://www.json.org`

## 6 Future Directions

Autotest has made great strides in automating the execution of kernel and hardware tests. But test execution usually occurs in the context of a qualification process, and the full qualification process remains a tedious and rather mechanical ordeal. Qualifying a new kernel generally involves running a collection of functional and performances tests over a large population of machines representing a range of hardware platforms. The choice of tests to execute may be dependent on the outcome of earlier tests. The results must then be compared to those for a known stable kernel to find statistically significant deviations. Furthermore, a continuous testing system would like to execute this entire process in a fully-automated fashion, reporting deviations on a per-change basis. Qualifying a collection of new machines involves a similar, but not identical, process. In particular, individual machines will must be tracked through a cycle of testing, triaging, and repairing by either updating system software or manipulating hardware components. At the same time, this individualized tracking must scale to hundreds or thousands of machines, and the process must culminate in a report of significant deviations from a known stable platform.

While Autotest abstracts away many of the low-level issues involved in these processes, it does little to automate these higher-level processes. Successful automation of such processes is one of the major unsolved problems for the Autotest project. Fortunately, the high-level automation support provided by the frontend makes it possible to prototype solutions to these problems. Such solutions can be built on top of the Autotest architecture without requiring modifications to Autotest itself, and indeed a number of such solutions have been built to satisfy needs of particular Autotest users. These prototypes provide a useful path forward to incorporate such automation into the Autotest system.

In addition, improved reporting remains an area of great opportunity for Autotest. Autotest's current reporting interface can generate a variety of reports, potentially spanning multiple jobs, but it still requires a significant manual effort to draw useful high-level conclusions and it still makes triage of failures a difficult task. To aid the former task, Autotest needs to support better automated folding of larger amounts of data into smaller, more concise reports which highlight significant quality deviations and hide the rest of the data. For easier triaging of failures, Autotest needs to better categorize and organize test output and more efficiently guide users to the places where failure details are most likely to be found.

## 7 Conclusion

A significant amount of developer time has been invested in Autotest to enable the continuous execution of small- and large-scale tests on thousands of machines. This effort has successfully overcome numerous problems with reliability and scalability inherent in testing low-level systems such as the kernel and hardware components. While further work remains to be done to improve and automate the high-level testing workflow, the fundamental components are in place and already usable for large-scale testing today.

## Acknowledgements

We would like to thank Martin Bligh for his input to and his reviews of drafts of this paper.

## Legal Statement

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

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

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

## References

[1] Linux Test Project. `http://ltp.sourceforge.net`.

[2] Jason Baietto. Linux Quality Assurance Utilizing An Automated Nightly Test System. `http://www.ccur.com/isddocs/ANTS.pdf`.

[3] David Berrera, Li Ge, Stephanie Glass, and Paul Larson. Testing the Xen Hypervisor and Linux Virtual Machines. In *Linux Symposium*, volume 1, pages 271–288, 2005.

[4] Kamalesh Bibulal and Balbir Singh. Keeping the Linux Kernel Honest. In *Linux Symposium*, volume 1, pages 19–29, 2008.

[5] Martin Bligh and Andy P. Whitcroft. Fully Automated Testing of the Linux Kernel. In *Linux Symposium*, volume 1, pages 113–125, 2006.

[6] Aaron Bowen, Paul Fox, James M. Kenefick Jr., Ashton Romney, Jason Ruesch, Jeremy Wilde, and Justin Wilson. Automated Regression Hunting. In *Linux Symposium*, volume 2, pages 27–35, 2006.

[7] Tim Chen, Leonid I. Ananiev, and Alexander V. Tikhonov. Keeping Kernel Performance from Regressions. In *Linux Symposium*, volume 1, pages 93–102, 2007.

[8] Subrata Modak and Balbir Singh. Building a Robust Linux kernel piggybacking The Linux Test Project. In *Linux Symposium*, volume 2, pages 91–100, 2008.

[9] Alexander Ufimtsev and Liam Murphy. Automatic System for Linux Kernel Performance Testing. In *Linux Symposium*, volume 2, pages 403–408, 2006.

[10] Hiro Yoshioka. Regression Test Framework and Kernel Execution Coverage. In *Linux Symposium*, volume 2, pages 285–296, 2007.

# Increasing memory density by using KSM

Andrea Arcangeli, Izik Eidus, Chris Wright
*Red Hat, Inc.*
aarcange@redhat.com, ieidus@redhat.com, chrisw@redhat.com

## Abstract

With virtualization usage growing, the amount of RAM duplication in the same host across different virtual machines possibly running the same software or handling the same data is growing at a fast pace too. KSM is a Linux Kernel module that allows to share equal anonymous memory across different processes and in turn also across different KVM virtual machines. Thanks to the KVM design and the mmu notifier feature, the KVM virtual machines aren't any different from any other process from the Linux Virtual Memory subsystem POV. And incidentally all Guest physical memory is allocated as regular Linux anonymous memory mappings. But KSM isn't just for virtual machines.

The KSM main task is to find equal pages in the system. To do that it uses two trees, one is the stable tree the other is the unstable tree. The stable tree contains only already shared and not changing KSM generated pages. The unstable tree contains only pages that aren't shared yet but that are tracked by KSM.

The content of the pages inserted into the two trees is the index of the tree, but we don't want to write-protect all the pagetables that points to the pages in the unstable tree. So we allow the content of the pages (so the tree index) to change under KSM and without knowledge of the tree balancing code. Thanks to the property of the red black trees that can keep a tree balanced without checking the node index value, even if the tree becomes unusable, the tree still remains balanced and the worst case insertion/deletion remains O(log(N)), to guarantee the ksm-tree algorithm not to degenerate in corner cases.

To reduce the number of false negative from the unstable tree lookups, a checksum is used to insert into the unstable tree only pages whose checksum didn't change recently, but in the future the checksum can be replaced by checking the dirty bit of the pagetables and shadow pagetables (not with current EPT though). After a full scan of all pages tracked by KSM, the unstable tree is rebuilt from scratch to reset all lookup errors introduced by the pages changing content during the scan.

Whenever KSM finds a match in the stable or unstable tree, it proceeds to write-protecting the pagetables that mapped to the old not shared anonymous page, and it makes them map the new shared KSM page as read-only. If any KVM shadow pagetable was mapping the page, it is updated and write-protected through the mmu notifier mechanism with a newly introduced change_pte method.

## 1 Nomenclature

The name of this Linux Kernel feature might change. For the scope of this document, the term *KSM* (as in *Kernel Shared Memory* or *Kernel Samepage Merging* if you wish) will be used, even if it may be renamed to *Memory Merging* in the future.

## 2 KSM objective

The objective of KSM is to increase memory density. KSM is generating shared pages by merging equal pages, and in turn it is making free memory available allowing to run more virtual machines or applications on the same system, than otherwise would be possible without KSM.

## 3 KSM API

The API to use in KSM has been one of the most discussed parts of the feature on mailing lists, but it's also the least interesting part for the scope of this document and will be only covered briefly here.

While it would be possible for KSM to scan every single anonymous page in the system, it would be wasteful to

scan virtual areas where we don't expect to find any significant amount of equal pages. It would be wasteful not only in CPU terms but in RAM terms too; to keep track of the pages, KSM has to make some slab allocation. The amount of slab allocations increases linearly with the size of the virtual areas registered. Usually Linux applications try to be intelligent in sharing memory either with shared librarians or through fork. Not all applications are generating memory regions with lots of equal anonymous pages in a way that cannot be shared without the KSM feature, so it's worth scanning only the virtual areas that are likely to contain lots of equal pages that cannot be shared by other means.

Processes (through the KSM API) shall simply have the option to register which virtual memory areas should be scanned by the kernel thread that has the task of merging equal physical pages of memory.

The kernel thread that scans the registered virtual ranges can be controlled through sysfs at /sys/kernel/mm/ksm (but if the KSM name changes, supposedly the location is subject to change too). Writing 1 or 0 in *run* respectively starts and stop the kernel thread. *pages_to_scan* and *sleep* control how CPU intensive the scan of the memory will be. The more pages scanned per wakeup and the more frequent the wakeups, the more CPU the kernel thread will take, and the faster the equal virtual memory will be shared. *sleep* is in usec units, pages is in PAGE_SIZE units. *pages_shared* is a read only statistic field showing how many KSM pages are allocated in the system at any given time. *max_kernel_pages* can limit the number of KSM pages, this can be useful especially in the short term because in its first version the KSM pages aren't swappable yet (swapping KSM pages is possible similarly to how tmpfs swaps and it will be addressed shortly).

At time of this writing, it seems likely the final memory merging API that applications can use will be implemented through the madvise syscall with a new MADV_(UN)MERGEABLE *advice* parameter.

In the current implementation, only anonymous pages (like the ones generated by malloc) can be merged with KSM, but perhaps in the future this could be extended to other kind of pages.

There will likely be an option to avoid compiling the KSM code into the kernel to save kernel .text for those embedded systems where the KSM feature won't be required, in which case madvise will fail if passed the relevant memory merging advice parameter.

## 4 KSM and KVM

One of the primary users of KSM is the Linux *Kernel Virtual Machine*. When the same guest OS and guest applications are running in different virtual machines, lots of equal anonymous memory will be generated on the host/hypervisor system. So it is ideal to always keep KSM enabled with parameters like sleep = 5000 pages_to_scan = 60, so that around 12000 virtual pages are scanned each second, allowing a max memory merging rate of 46.87MB/sec (the max rate would materialize only if all virtual pages scanned during a second of time, are found to be equal to some other page tracked by KSM). With this setting the KVM kernel thread should use around 10% of one CPU core. On very large systems, however, more aggressive settings can be used, up to dedicating a CPU core to the KSM kernel thread.

Although at present KSM is only capable of merging equal anonymous memory on the host system, KVM virtualization allows KSM running on the host to share pagecache, tmpfs, or any other type of memory allocated in the guest, because all guest memory is backed by host anonymous memory.

## 5 KSM at CERN

We had great feedback from CERN and Lawrence Berkeley National Laboratory related to the computations they're running to crunch the LHC generated data. Their scientific reconstruction jobs generate lots of equal pages while they run. With KSM enabled they achieve memory sharing rates up to 750MB if they run two similar 2GB jobs (without KSM the sharing is limited to 250MB). They conclude they are able to run 3 jobs in parallel on a 4GB machine, instead of only 2 before. This cumulatively saves a very significant amount of memory, given the number of nodes involved in the computations.

They're not yet using virtualization on top of Linux, so to use KSM, their application has to call into the KVM API directly (by using a malloc wrapper). If they were to use KVM as a hypervisor (instead of proprietary hypervisor solutions running underneath of Linux) they wouldn't need to change their application at all, and all memory would be merged transparently at the host kernel level.

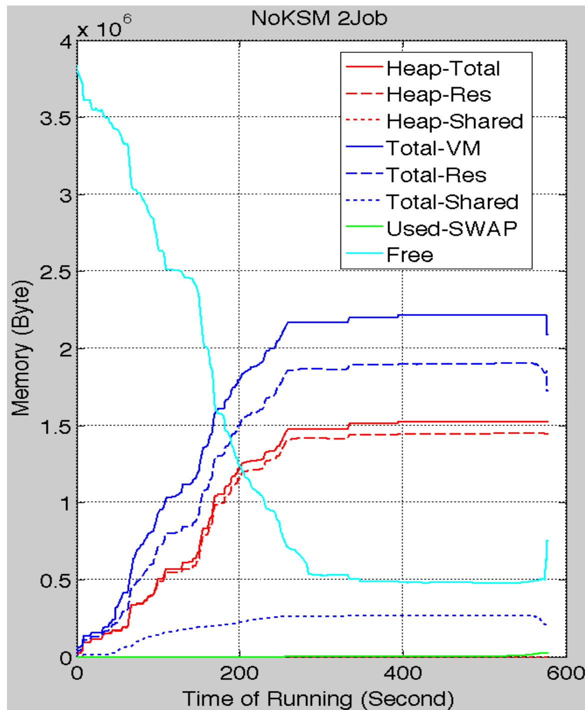



Figure 1: 2 LHC reconstruction jobs without KSM

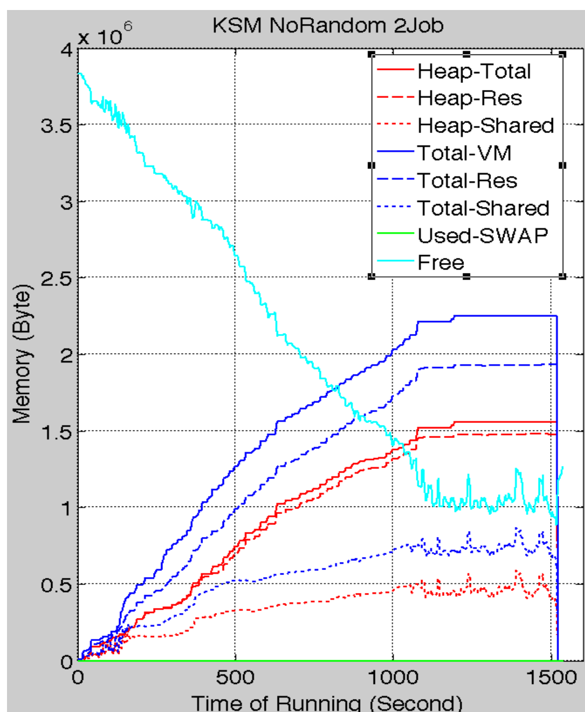



Figure 2: 2 LHC reconstruction jobs with KSM

## 6 KSM and embedded

KSM is suitable to be run on embedded systems too; the important thing is not to register in KSM regions that won't likely have equal pages. For each virtual page scanned, KSM has to allocate some rmap_item and tree_item, so while these allocations are fairly small, they can be noticeable if lots of virtual areas are scanned for no good.

Furthermore, these KSM internal rmap/tree data structures are not allocated in high memory. To avoid early out of memory conditions, it is especially important to limit the amount of lowmem allocated on highmem 32bit systems that might have more than 4GB of memory, but these shouldn't fit in the embedded category in the first place.

## 7 KSM and swap size

When KSM merges pages, it frees memory. However, it must be clear that the shared KSM pages remains shared only as the virtual machines using them are only reading from and not writing to those pages. So there is no actual guarantee that the memory freed by KSM as result of creating shared KSM pages will remain free. To obviate this problem administrators must tune the swap size appropriately, to ensure that even if the amount of shared memory would decrease significantly (if the workload of the virtual machines suddenly changes) the host Linux Kernel will not run out of physical memory.

## 8 KSM tree algorithm

The KSM tree algorithm is built around the concept that to find equal pages we add each page in the registered virtual memory areas to a Red Black Tree. The index of the tree is the content of the page itself. The function that searches the tree to find an equal page, will check the *memcmp()* return value to decide if to go left, right, or if we already found an equal page indexed into the tree.

## 9 KSM pass

A *KSM pass* for the scope of this document is intended as a entire full scan of all virtual areas marked VM_MERGEABLE by madvise, so registered in KSM.

## 10 Computational complexity

The usual page size for x86 architectures (and most other architectures) is 4096 bytes. But on average the *memcmp()* function will break out of its inner loop before processing all 4096 bytes. This is because the pages are unlikely to be all equal except for the last bits. The cost of finding an equal page, will be the cost of *memcmp()* multiplied by the number of levels in the tree. Thanks to the rbtree, the computation complexity of all insert/search/delete functions is O(log(N)) (where N is the total number of pages scanned by KSM). So even if we hit the absolute worst case where the first 4092 bytes of all pages scanned by KSM are equal, and only the last 4 bytes differs, the KSM tree algorithm will not degrade too much.

## 11 Stable and unstable trees

The KSM tree algorithm uses two rbtrees, one called *stable tree* (as in Figure 3) and one called *unstable tree*. Using two trees is an optimization and also increases the probability of quickly sharing the pages that are the most likely to be good candidates for sharing as well as reducing the instability of the *unstable tree*. The algorithm flow chart is visible in Figure 4.

For each anonymous page scanned, the kernel thread proceeds searching a match first in the *stable tree* that only contains already shared pages (shared so in turn write protected, hence their content is stable). If a match is found in the *stable tree*, the anonymous page is merged with the KSM page found in the *stable tree*.

If no match is found in the *stable* tree, KSM checks if the anonymous page has changed content recently using a checksum.

If the checksum changed since the last *KSM pass*, KSM updates the checksum and will defer the search of the *unstable tree* to the next *KSM pass* (assuming that the checksum won't change again). This is to avoid merging or adding to the *unstable tree* pages that changes content frequently.

If instead the checksum didn't change KSM proceeds searching the *unstable tree* that only contains anonymous pages scanned previously but not merged by KSM yet. If a match is found in the *unstable tree* KSM merges the anonymous page under scan, with the anonymous page in the *unstable tree*, and the resulting KSM merged page is added to the stable tree (the anonymous page found in the *unstable tree* is removed from the *unstable tree* and freed). If a match is not found in the *unstable tree* KSM adds the page to the *unstable tree*.

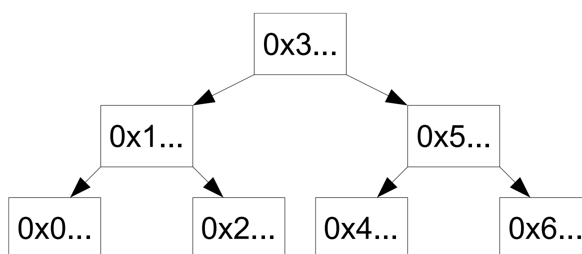



Figure 3: KSM *stable tree*

In the future, instead of the checksum, a dirty bit in the pte (and spte) can signal KSM if a page is worth adding to the *unstable tree* or not, or special instructions can be used if provided by the CPU to compute a checksum faster than jhash2.

This 'checksum' here has really nothing to do with the KSM Tree algorithm itself. The 'checksum' is not used to find equal pages to share; rather, it's only an heuristic to try to keep the *unstable tree* more stable and to avoid wasting time with bad sharing candidates. Even if we eliminate the checksum, the algorithm would still work.

If a page changes content frequently, besides risking the generation of false negatives from the *unstable tree* lookups, we'll likely only waste CPU by sharing it, because a copy-on-write page fault will likely happen soon enough, breaking the sharing.

## 12 When the *unstable tree* becomes unstable

We must avoid write protecting pages that aren't shared yet, or the whole virtual memory scanned by KSM would be write protected most of the time, in turn leading to a flood of copy-on-write page faults. The *stable tree* only contains shared KSM pages, and we know all pages inside it aren't going to change content because they have to be write protected in the pagetables and shadow pagetables in the first place in order to be shared. So a lookup in the *stable tree* is fully reliable and can't return false negatives. It's just like a lookup of any other regular rbtree in the kernel, where the index doesn't change under the tree after the node is indexed

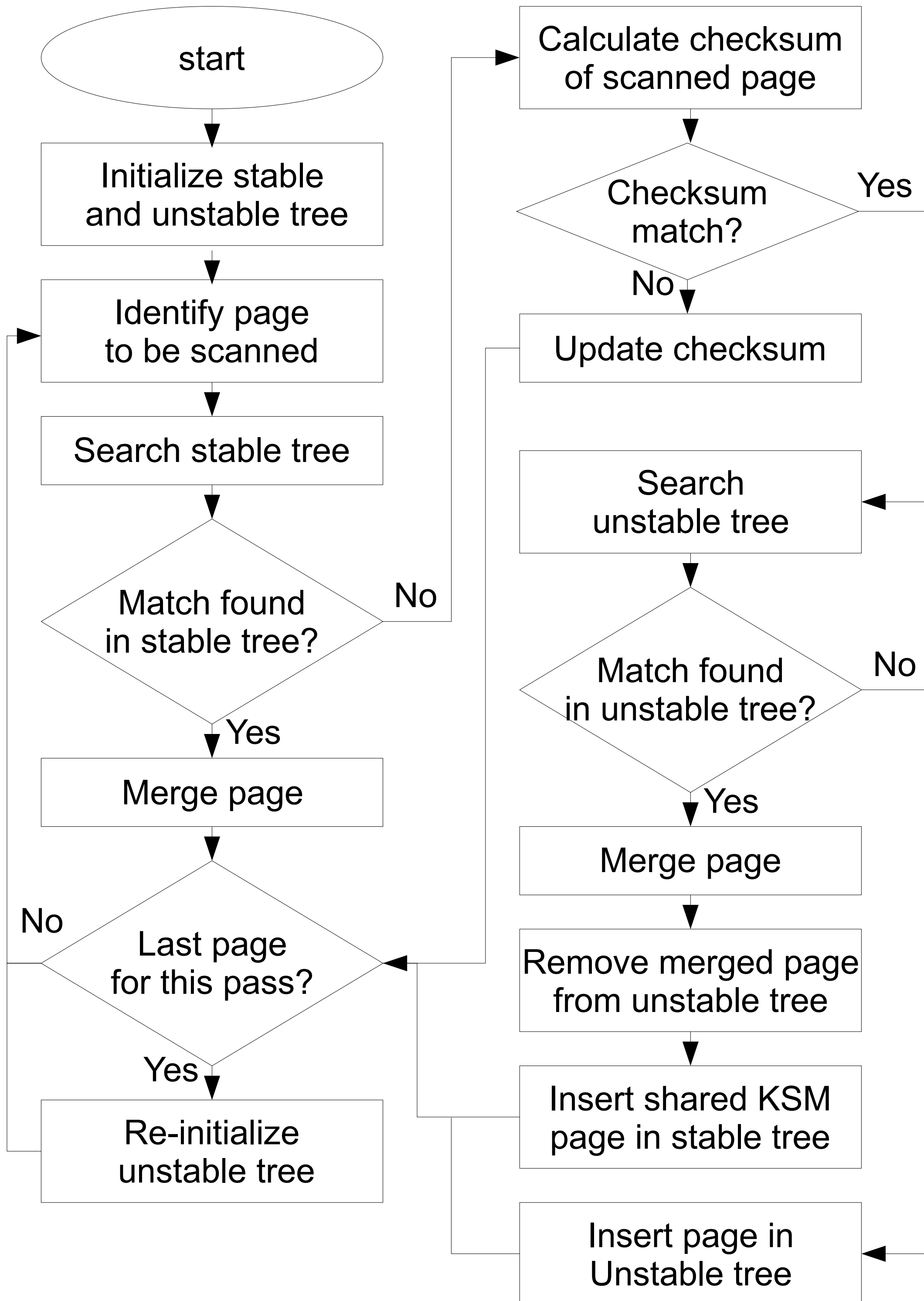



Figure 4: KSM Tree algorithm flowchart

into the tree. The problem is the lookup of the *unstable tree* because the *unstable tree* only contains regular anonymous pages not shared yet, that can be still written to by applications.

Because the rb_insert()/rb_erase() functions that balance the rbtree while inserting and deleting an element from the tree are unaware of the index value, we're guaranteed the rbtree will remain well-balanced regardless of where we insert any new node in the tree. We are also guaranteed that all insert, search, and delete operations will not degrade in terms of computational complexity, even after the *unstable tree* becomes really unstable.

An example of the *unstable tree* while it's still stable can be seen in Figure 5. If an application writes to a page indexed in the *unstable tree* that had the first byte set to 0x03 when it was inserted in the stable tree, and it changes it to 0x07 afterwards, the *unstable tree* might become unstable as in Figure 6 and lookups might start to generate false negatives.

To avoid the instability and the resulting false negatives to be permanent, KSM re-initializes the *unstable tree* root node to an empty tree, at every *KSM pass* (i.e. after completing a full scan of all virtual areas registered in KSM). This way, a new *unstable tree* is rebuilt from scratch at every *KSM pass* and the false negatives won't be sticky.

To further decrease the probability of false negatives from the *unstable tree* lookups, we could also remove pages from the *unstable tree* if we find a dirty bit set or the checksum being not uptodate anymore during the tree walk, even though we're not doing it in the current implementation as it'd make the search in the *unstable tree* slower than just a *memcmp()* for each level of the tree.

Because all long-term important sharable pages are going in the stable tree over time, the *stable tree* guarantees us that the important sharable pages are going to be merged without any risk of false negatives, regardless of any temporary instability of the *unstable tree*.

If all goes well and there are no false negatives, while inserting an anonymous page in the *unstable tree*, KSM will find a page with equal content already indexed in the *unstable tree*, so KSM will merge them together, it will create a new KSM page with equal content added to the *stable tree* and remove the indexed anonymous page from the *unstable tree*, and finally will free both anonymous pages.

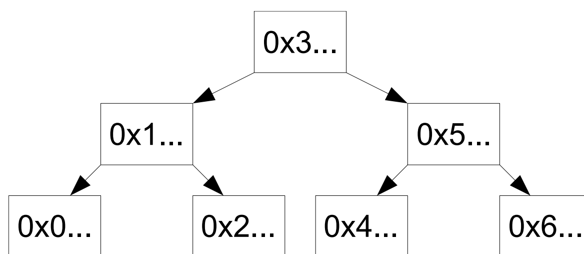



Figure 5: KSM unstable tree while still stable

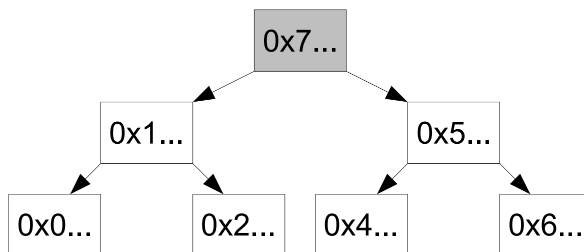



Figure 6: KSM unstable tree gone unstable after application write

## 13 Page merging

The procedure used for page merging involves two functions: *page_wrprotect()* and *replace_page()*. The former write protects all pagetables mapping the page passed as parameter (and sptes too through change_pte() mmu notifier discussed below) method; the latter merges two pages by updating the pagetables accordingly (and sptes too through change_pte() mmu notifier), and then by freeing the merged anonymous page that no pagetable (or spte) maps anymore.

One final *memcmp()* is required after *page_wrprotect()* returns to be sure both pages being compared cannot change while *memcmp()* runs. Only if the final *memcmp()* succeeds (returning zero) *replace_page()* is called to merge the two pages.

If there is a match in the *stable tree*, the KSM page already in the *stable tree* is merged with the anonymous page under scan.

If there is a match in the *unstable tree*, a new KSM page is allocated and the content of one of the anonymous pages is copied to it, and both anonymous pages are merged with it.

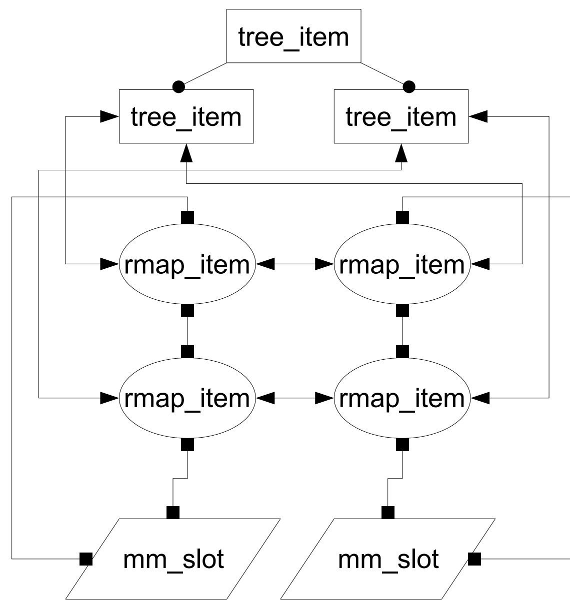



Figure 7: rmap_item and tree_item relation

The merge path is a fairly slow path: if it would run all the time, it would mean that all virtual addresses scanned by KSM are sharable all the time which certainly isn't the case most of the time. In the future we could however optimize the *unstable tree* merge path to transform an anonymous page into a KSM page in place to avoid one page copy and to optimize away some minor pte/spte mangling for one of the two anonymous pages being merged together.

## 14 KSM rmap_item and tree_item

A physical page is represented in the the *stable* and *unstable* trees by the tree_item structure. The tree_item is a rmap structure that contains the head of a list that links all virtual addresses that map each physical page. The virtual addresses (the list elements) themself are represented by a rmap_item structure. Their relation is shown in Figure 7.

So during the *stable* and *unstable* tree lookups KSM, walks the tree_item list to find the virtual address (in the rmap_item) to call get_user_pages and to obtain the physical page address to run *memcmp()* against.

## 15 KSM rmap_item and tree_item out of sync with the Linux VM

The reason get_user_pages() is called during the tree walk is that we're tracking virtual addresses instead of page frame numbers in the tree_item. This is because the tree_item and the rmap_item are maintained out of sync with the Linux VM. This means that if an anonymous page is swapped out or unmapped, we'll find out only during the tree lookups or during the KSM scan on the virtual areas registered. Whenever we find a virtual address not mapped in the pagetables, we drop the respective rmap_item and if that was the last rmap_item in the tree_item linked list, we also drop the tree_item.

## 16 KSM rmap_item and tree_item in sync if KSM would register its mmu notifier methods

We considered to change KSM to avoid the get_user_pages() call during the tree walk by storing a pointer to the physical page directly in the tree_item by using mmu notifiers that would notify us whenever a rmap_item should be dropped. However, in addition to making the code more complicated, that would require global spinlocks that would serialize the rbtrees lookups against mmu_notifier invalidate methods, and it might lead to applications to scale worse because every time a virtual area is unmapped that global lock would be taken. We want KSM to run in the background without affecting the regular runtime of applications as much as possible. Furthermore, replace_page() used by KSM to merge the pages would then re-enter KSM again through a mmu notifier invocation in replace_page() after it mangles the pte, a case that would require some special handling and perhaps rmap_item refcounting. Because keeping the rmap_item and tree_item fully synchronized isn't required to efficiently find equal pages, we think it's simpler to maintain them out of sync, with the main disadvantage of the tree walk requiring get_user_pages calls, but we prefer KSM itself to be a bit slower in merging memory and not to risk slowing down the actual applications with global locks.

Strictly speaking for the unstable tree a per-mm *unstable tree* protected by a per-mm lock would be feasible but the *stable tree* spinlock would need to be global if we want to share memory system-wide.

There are various implementations but likely the way KSM will implement the rmap_item scan over the vmas with the madvise API is to keep an list ordered by address of rmap_item for each mm with vmas with VM_MERGEABLE set, and to resync the rmap_item

list in a inner loop, with the outer loop being the vma->vm_next loop. Any rmap_item instantiated in a previous *KSM pass* but found not anymore in the range of any VM_MERGEABLE vma will be dropped, and new rmap_item will be created for each new virtual address that has a pagetable pointing to an anonymous page in a VM_MERGEABLE vma. This way the madvise syscall will not have to call KSM, and it will only have to split vmas if needed and set or clear the VM_MERGEABLE flag in the virtual areas passed as parameter to madvise MADV_MERGEABLE.

## 17 MMU notifier change_pte() method

When KSM merge pages, we don't want to teardown all secondary pagetables (e.g. the VT shadow pagetables instantiated by KVM). To avoid that a new change_pte() method is used by *replace_page* that will update all sptes that pointed to the old anonymous page to point to the location of the new KSM page.

If we used the invalidate_page() method instead of introducing a new change_pte() mmu notifier method, KSM would have destroyed the sptes in turn requiring KSM to take minor faults to recreate them later as the guest returns to access those guest virtual addresses, by re-reading the kernel pagetables.

Side note: due to some short term limitation right now KVM will always trigger write faults as far as the Linux VM is concerned even if the guest issued a read memory operation, so lack of change_pte() method would have prevented the shared KSM pages to be mapped by any shadow pagetable at all. This limitation in the KVM page fault will however be addressed in the future, but change_pte() will still remain an useful optimization even then, by preventing KVM to vmexit to rebuild invalidate sptes (even if the sptes in the future could be rebuilt by KVM with readonly permissions without triggering copy-on-write faults in the Linux VM if the guest issued a read access on a KSM page).

change_pte() takes the Linux pte as parameter and it makes sure the sptes are marked readonly if the pte passed as parameter is readonly.

change_pte() is also used in the Linux VM write protect page faults triggering on KSM pages as an optimization to avoid tearing down sptes (do_wp_page()).

Not all MMU notifier users are required to implement the new change_pte() method; if not implemented, it will simply fallback to the invalidate_page() backwards compatible behavior, which is safe but less efficient for users like KVM. For the MMU notifier users that don't manage real secondary pagetables, but only a secondary tlb (like GRU), implementing the change_pte() method is unnecessary.

## 18 KSM not working on pages under GUP

It is interesting to note that page_wrprotect() has to fail for any page that is temporarily pinned by get_user_pages() users (to avoid generating I/O corruption on the drivers that accesses the pinned pages directly) and it will only function on drivers that uses MMU notifier and that can unpin the pages immediately after get_user_pages() returns. So to allow KSM to work on KVM guest physical memory, we had to remove the page pinning on the shadow pagetable mappings (in short that means calling put_page() immediately after get_user_pages() returns, and entirely relaying on mmu notifiers methods to teardown shadow pagetables before the corresponding virtual address is teardown by the Linux VM, either because of VM pressure or userland action).

All get_user_pages() pins shall be temporary; if not, the pinned pages cannot be paged out by the VM in case of memory pressure. So if the pins are temporary as they've to be, KSM will simply be able to write protect those pages (and then possibly to merge them) in one the next *KVM passes*. Drivers that use the pages returned by get_user_pages() in a persistent way like KVM must use MMU notifiers and release the page pins to be transparent to the Linux VM and in turn to allow KSM to merge pages on those memory regions too.

## 19 KSM multi threading

In the future it will be possible to add more than one KSM kernel thread by adding a read-write mutex or spinlock that protects each tree. Starting more than one KSM kernel thread will be helpful if somebody wants to dedicate more than 100% of one CPU core at merging pages.

## 20 KSM swapping

KSM pages cannot be swapped at this time; KSM pages are effectively nonlinear entities mapped in the middle

of linear anonymous vmas and the Linux VM swap logic cannot cope with them at this point in time.

Because KSM to function requires its own internal rmap logic and because we surely don't want to hurt the Linux Kernel VM memory footprint when KSM is not enabled, likely an external rmap functionality shall be implemented to allow the Linux VM to call into KSM to unmap all pagetables mapping the shared KSM pages. The swapin path will also require some change because the anonymous fault won't be suitable for swapping-in KSM pages if they've been swapped-out, similarly to how tmpfs swaps out the tmpfs shared pages.

## 21 Reduce *memcmp()* length in tree lookup maintaining rbtree cumulative info

It should be possible to add to the tree_item some rbtree related metadata information on the status of the left and right nodes. This metadata information can tell the tree lookup function the offset of the first byte that differs between the current node physical page, and the two physical pages in the right and left nodes. That will require adding a callback to rb_insert() and rb_erase() called with proper information during each tree balancing rotation of the nodes, so that this metadata can be recalculated at every rebalance of the tree. With this information, we should be able to significantly reduce the cumulative amount of memory compared by the *memcmp()* function during a worst case of tree lookup.

If rb_insert() and rb_erase() will be extended like above, the rebalancing callback could also be used by get_unmapped_area() to allow it to work in O(log(N)) instead of the current O(N) (where N is the number of vmas in the mm_struct).

## 22 KSM benchmark

We run all test cases using a Linux 2.6.30-rc6 kernel, with a fairly recent KVM external module and the KSM patchset posted on the Linux Kernel Mailing List on 20 April 2009 with Message-ID: 1240191366-10029-1-git-send-email-ieidus@redhat.com (which still uses the old ioctl API and not madvise yet). To merge memory at the fastest possible peace, KSM clearly has been tuned so that the single threaded kksmd kernel thread runs at 100% CPU load (sleep = 0, pages_to_scan = 1000000). The hardware used is on a common and cheap Intel Q9300 Core 2 Quad at 2.50GHz with 4 GigaByte of 800mhz DDR2 memory.

We intend to measure here the max speed of KSM in merging pages under best, worst and real life cases. The number of MegaBytes per sec of memory merged by KSM when KSM runs at full CPU utilization is a relevant parameter to measure, because it shows how fast KSM is at merging pages. In real life environments it's unlikely KSM will be tuned to run at full CPU utilization (with the exception of very large servers with many CPUs and several dozen GigaBytes of RAM), but the fastest KSM is at merging pages at full CPU utilization, the lower CPU KSM will take when tuned for real life environments. We could have statistically measured the average CPU utilization instead, but measuring the amount of RAM merged per second and maxing out the CPU utilization allows for a much more reliable measurement of the efficiency of the algorithm under different workloads. The forth column of the output from *'vmstat 1'* will be used to monitor the progress KSM does in merging memory.

The worst case for KSM that should practically never materialize in practice (unless of course malicious users can run their own malicious applications) can be exercised with an application that allocates one gigabyte of memory and that makes all pages equal except for the last 4 bytes of each page. The first copy of this application called *ksmpages* will fully populate the *unstable tree*. Running a second copy will merge all pages in the *unstable tree* and it will create equal amounts of KSM shared pages in the *stable tree* and free one gigabyte of memory in the process. The *stable* and *unstable* trees generated will have many levels and the *memcmp()* will not break before at least 4092 bytes have been read for each level of the tree.

The best case for KSM can be exercised with an application that allocates one gigabyte of memory and initializes all pages to the same value. KSM when started will quickly free one gigabyte of memory minus one KSM page that will be the only one indexed in the *stable tree*. The unstable tree will not be empty only before the very first merge. A single *memcmp()* and a single level of the *stable tree* has to be walked in order to merge the pages.

The real life case for KSM can be measured by running two copies of a popular proprietary guest OS in KVM with 1G of memory each, wait both of the to finish booting, and finally start KSM and see how fast the memory

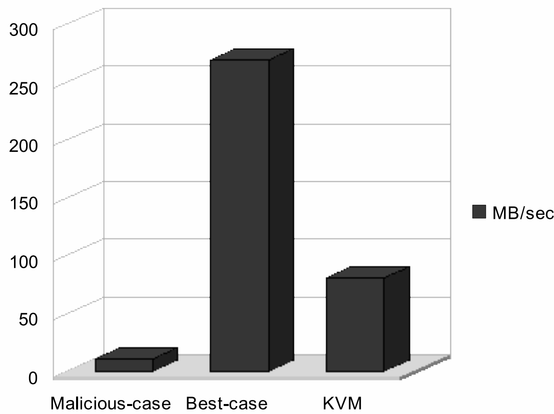



Figure 8: KSM *benchmark*

is merged (i.e. freed). Then we stop the kksmd thread, we start a third VM of the same OS, and we start kksmd again.

## 23 Conclusions

Considering that even the worst possible malicious case on one of the cheapest workstation hardware configurations with very cheap motherboard and northbridge, definitely makes progress at 10.62 MegaBytes merged per second (note that the only side effect of malicious behavior is an higher CPU utilization), that the fixed cost of the virtual address scanning and page merging is CPU bounded at 269.05 MegaBytes per second, and that the real life KVM case merges pages at 80.70 MegaBytes per second, we're comfortable this algorithm (even without the future possible further optimizations) in the background will be able to merge pages efficiently in virtualization and scientific environments and in embedded systems as well.

## 24 References

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Chapter 13—Red-Black Trees* Introduction to Algorithms, Second Edition. The MIT Press, September, 2001.

Vincenzo Innocente, Summary of the evaluation of KSM for sharing memory in a multiprocess environment, `https://twiki.cern.ch/twiki/bin/view/LCG/EvaluationKSM0409`, 20 May 2009.

Izik Eidus, KSM version used for benchmark, `http://kerneltrap.org/mailarchive/linux-kvm/2009/4/20/5521504`, 20 April 2009.

Andrea Arcangeli, ksmpages.c source and some benchmark, `http://kerneltrap.org/mailarchive/linux-kvm/2009/3/31/5349904`, 31 March 2009.

# Sandboxer: Light-Weight Application Isolation in Mobile Internet Devices

Rajesh Banginwar
*Intel Corporation*
rajesh.banginwar@intel.com

Michael Leibowitz
*Intel Corporation*
michael.leibowitz@intel.com

Thomas Tanaka
*Department of Computer Engineering*
*San Jose State University*
thomas.tanaka@gmail.com

## Abstract

In this paper, we introduce sandboxer, an application isolation mechanism for Moblin based Mobile Internet Devices (MIDs). MIDs are expected to support the open but secure device model where end users are expected to download applications from potentially malicious sources. Sandboxer allows us to safely construct a system that is similar to the conventional `*NIX` desktop, but with the assumption that applications are malicious. Sandboxer uses a combination of filesystem namespace isolation, which provides a secure `chroot()` like jail; `UID/GID` separation, which provides IPC isolation; and `cgroups` based resource controllers, which provides access control to devices as well as dynamic limits on resources. By combining these facilities, we are able to provide sufficient protection to the user and system from both compromised applications that have been subverted as well as malicious applications while maintaining a very similar environment to the traditional `*NIX` desktop. The mechanism also provides facility for applications to hide the local data from rest of the applications running in their own sandboxes.

## 1 Introduction

Mobile internet devices (MIDs) have become an increasingly popular choice of device that people use every day as part of their daily routines. The recent released of Intel Atom processor which targets computer systems with small form factor such as MIDs and comes with the capability to deliver full internet experiences to mobile devices; further adds to a roadmap of more powerful processors powering MIDs in the near future. User will thus be able to enjoy high quality entertainment such as game or working towards their business related tasks on their mobile devices. With the increase in the computational power, more complex software applications will be developed to run in mobile devices. This could potentially lead to an increase in the security exploit of the device due to bugs and other possible software design flaws. Malicious software that has successfully penetrated the device will have the available resources to tap into user's privacy, which could be in the form of personal data (e.g., phone number) or sensitive phone conversations.

The majority of the user groups will not be necessarily equipped with sufficient knowledge to identify a possible malicious website or application. Therefore, designing and managing a strict security measure for mobile device is a necessary first step to ensure a safe operating environment. We have thus proposed sandboxer, the security tool that will provide a mechanism to protect mobile device in the event of malicious attacks. The basic sandboxing technique provides a concealed environment in which an application can be run, and in the event of malicious attack, damage to the system is greatly minimized. There have been similar works in the sandboxing design by several researchers. Nevertheless, their respective work has been focusing more on delivering a complete and efficient sandboxing solution that intends to minimize the possibility of an exploit in a vulnerable desktop/server like system rather than targeting specifically on mobile devices. Savitha and Kolar have proposed the use of hardware base solution to create the fine grained sandboxing by utilizing the privilege level adjustment that is available in today's processors [1]. West and Gloudon have proposed to monitor

system calls that required the modification of the kernel codes [2]. Yee *et al.* have developed a novel approach that utilizes the system interaction in terms of software fault isolation and controlling the runtime environment securely [11]. On the other hand, Chang *et al.* have implemented user level sandbox that uses resource monitoring and restrictions on applications specifically on Windows platform [10].

Our implementation differs in that we specifically focus on implementing the application sandboxing in a Linux platform, as part of the Moblin.org open source project [15]. Moblin is an open source Linux based operating system specifically targeted for MIDs. The unique features of our designs are as follows:

- The use of available and simple yet robust filesystem and privilege isolation techniques that are available as part of the Linux platform.
- User level implementation that does not require any modification to the Linux kernel only relying on the existence of a stable kernel and system.
- The ability to further extend the functionality of the sandbox through the use of plugins.
- The use of `cgroups` as a plugin to further enhance the sandboxing capability to include a tool that capable of enforcing a policy base resource control mechanism on the system.

With this, we will begin our discussion of the overall sandbox architecture design. We will then proceed on how we isolate the filesystem and privileges. Finally, we will proceed with the brief discussion of `cgroups` and specifically which features of `cgroups` that is currently included in our overall sandbox design.

## 2 Design and Implementation

Our design principle is based on the following key objectives:

1. To guarantee that a compromised application could not take ownership of the whole system. In other words, an attacker will not be able to use a possible vulnerable application as a springboard to launch a premeditated attack.
2. To provide the ability to hide information or data associated with an application from the rest of the applications running on the platform.
3. To provide the way to restrict access to the part of the system that an application does not require to accomplish its task.
4. To provide the ability to customize extended functionality of the sandbox by providing software hooks that could be developed and installed as a plugin.

Based on the above objectives, we have the overall high level system architecture of our design as shown in Figure 1.

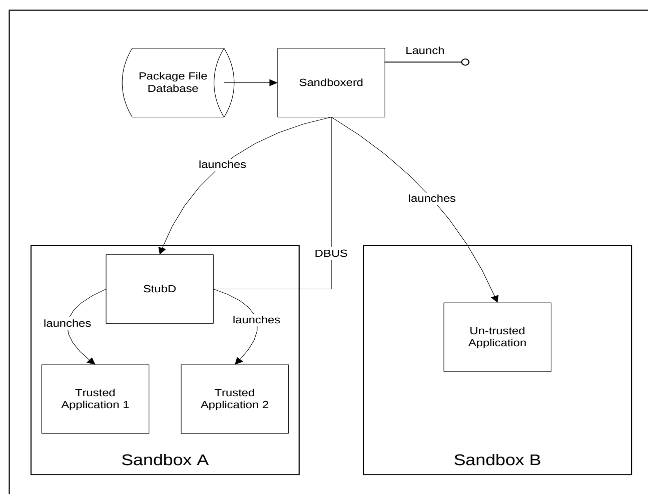



Figure 1: Architecture Overview

Our design consists of three functional components: Package File Database, Sandboxer Daemon, and Stub Daemon, as shown in Figure 1 above. The roles of these three functional components are as follows:

- Package file database decides whether to create a new sandbox or to use the existing one for the newly invoked application.
- Sandboxerd responds to request from the new application to execute.
- Stub Daemon is a daemon that only launches within a sandbox that contains multiple trusted applications. It specifically handles the request from the Sandboxerd where a new trusted application needs to run in the existing sandbox.

| **Trusted Domains** |
|---|
| - Packages are installed in the `/usr` hierarchy as per Filesystem Hierarchy Standard (FHS) recommendations |
| - Binary files/directories are owned by (root, root) |
| - Binaries are run as <unique_uid>, <unique_gid> |
| - Multiple binaries may be run in the same sandbox |
| **Untrusted Domains** |
| - Packages are installed in `/opt/<package-name>` as per FHS recommendations |
| - Files/directories are owned by <unique_uid>, <unique_gid> |
| - Binaries are run as <unique_uid>, <unique_gid> |

Table 1: Assumptions based on trusted and untrusted domains

Notice that we emphasize the notion of trusted applications. Trusted applications are verified as safe applications and from trusted domains. The assumptions between trusted and untrusted domains are summarized in Table 1.

The package file database's primary role is to provide a mapping between trusted binaries to sandbox in the form of configuration file or database. All applications to be run in a sandbox are configured here, both trusted and untrusted. The distinction from trusted and untrusted operation is the configuration of the sandboxes, rather than the flag in the database. Care must be exercised during the creation of such entries. The format is illustrated in Table 2 below.

```
[Sandbox]
SandboxName=shared_sandbox
PackageName=firefox
Users=firefox
ExecPaths=/usr/lib/firefox-3.0.8/firefox
[Sandbox]
SandboxName=shared_sandbox
PackageName=gcalctool
Users=gcalctool
ExecPaths=/usr/bin/gcalctool
[Sandbox]
SandboxName=xterm_sandbox
PackageName=xterm
Users=xterm
ExecPaths=/usr/bin/xterm.bin
```

Table 2: Example .package files

The application `firefox` and `gcalctool` will therefore share the sandbox which will be referred to as `'shared sandbox'`, while `xterm` will use the new sandbox referred to as `'xterm_sandbox'`.

### 2.1 Filesystem and privilege isolation method

By filesystem isolation we mean that the sandboxed application runs in the pre-defined subset of filesystem that it cannot escape from. This is commonly referred to as "jail" and is most commonly accomplished with `chroot()`. Exploits that will compromise a simple `chroot()` are well known [14]. Our implementation does not use `chroot()` directly. Instead, we uses the `CLONE_NEWNS` flag introduced in the Linux 2.4.19 as a flag to create a new filesystem namespace with the `unshare()` system call. Once a process has entered its own namespace, `mount()` and `umount()` only affect the namespace of the current process and not the parent. Thus, manipulation to the root filesystem is possible that is specific to a certain process. Bind mounts (Linux 2.4 onwards) allow a sub-tree of the filesystem to be mounted as though it were a filesystem on a path. Using this mechanism, one can, for example, bind mount `/foo/bar` to `/baz` with `mount("/foo/bar", "/baz", NULL, MS_BIND, NULL)`. With these two tools, a secure jail can be constructed simply with:

```
chdir("/jail");
unshare(CLONE_NEWNS);
mount("/jail","/jail",0,MS_BIND,0);
pivot_root("/jail","/jail/old_root");
chdir("/");
mount("/old_root/bin","bin",0,MS_BIND,0);
mount("/old_root/usr","usr",0,MS_BIND,0);
mount("/old_root/lib","lib",0,MS_BIND,0);
umount2("/old_root",MNT_DETACH);
/* drop privilege omitted */
exec(application);
```

For privilege isolation, we use conventional `UNIX` users and groups. It is expected that individual applications will run with individual `UID` and `GIDs`. This allows

traditional isolation among users, which UNIX systems provide to keep applications distinct from each other. Several unprivileged applications will likely be put in their own sandbox. Certain applications will remain outside of sandboxes. These generally include privileged applications and daemons as well as applications that need unfettered access to the whole filesystem to work correctly (such as the IDS system). Of course, X is outside of a sandbox.

It may be desirable to put a PIM application and the web browser in separate sandboxes because both process' considerable input from the outside. It would be undesirable if an arbitrary code execution's flaw in the web browser exposed all of user's email. Likewise, the damage the compromised web browser will do should be limited.

## 2.2 Sandoxerd and Stub Daemon

Sandboxerd uses D-Bus as the communication medium among client applications. Sandboxerd exists as a daemon that manages the creation of sandbox and verify the policy that comes with that particular sandbox. It exposes an interface that is roughly analogous to vfork() and wait() usages. Note that this is API usage. A helper application is provided that can be directly vfork'd and waited. The helper utility is called sandbox and the usage of such utility would be:

```
sandbox <cmd> [args]
```

The calling application can directly use vfork()/exec() and waitpid() on this helper utility. With such a utility, enabling the use of sandboxing to applications is a mere configuration change rather than a code change.

The Stub Daemon exists for the case where two or more trusted applications are to share a sandbox. This can be determined in advance by cross-indexing the application name to the sandbox name in the configuration database during the mapping procedure. Because a sandbox is a premised on a filesystem namespace (CLONE_NEWNS), the only way to add a process to an existing sandbox is with fork(). For this scenario, the Sandboxerd vfork() the Stub Daemon. The Stub Daemon does the unshare() and bind mounts filesystems available to the sandbox and waits for commands from Sandboxerd. When all the children have exited, the daemon exits, thus destroying the sandbox. The following pseudocode further illustrates the mechanism.

```
loop {
  wait for command from Sandboxerd() {
    pid= fork;
    if (pid) {
       children << pid;
       send child pid to Sandboxerd;
    } else {
       setgid();
       setruid();
       exec();
    }
  }

  if children is empty
  exit;
}
```

To better understand how these blocks function together, two flow examples are provided side by side. Note that although we use the UML sequence notation, each lifeline represents a process rather than an object. Referring to Figure 2, in the first use case (the left figure), one application in a sandbox (App1), requests for the launching of a second application in a sandbox (App2), and each application exists in its own sandbox.

## 2.3 Plugins architecture

Our implementation of sandbox provides strategically placed hooks. Referring to our method of creating a secure "jail" in the previous 2.1 section, the following hooks in order:

- INIT – initialization state; run as *sandboxer user*
- PRE_FORK – state before fork() system call; run as *sandboxer user*
- PRE_UNSHARE – state before unshare() system call; run as *root*
- PRE_PIVOT_ROOT – state before pivot_root() system call; run as *root*
- PRE_UMOUNT – state before unmounting old_root; run as *root*
- PRE_SETUID – state before dropping privileges; run as *root*
- PRE_EXEC – state before exec() system call; run with privilege specified in .package file
- FINALIZE – final state; run as *sandboxer user*

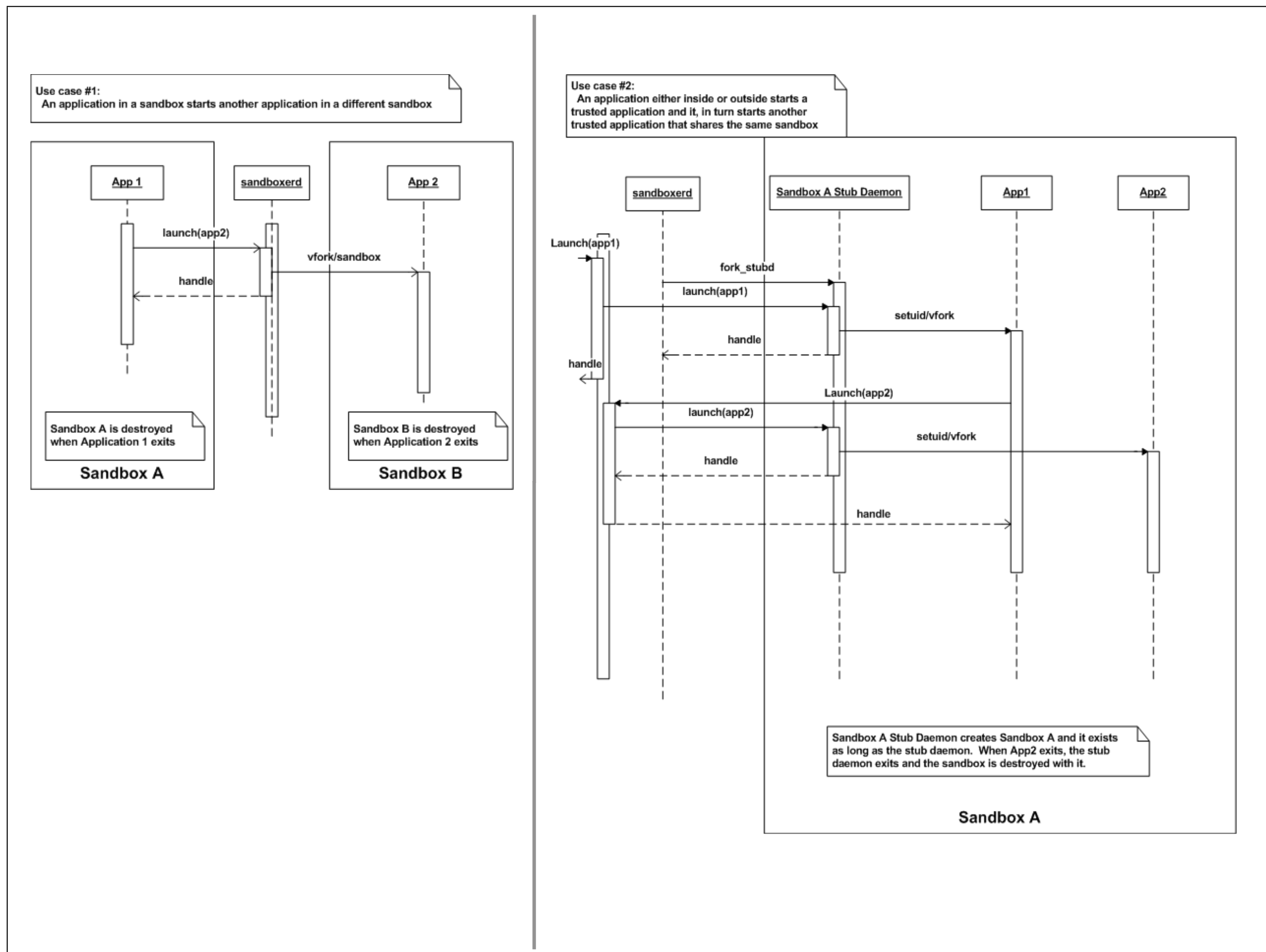



Figure 2: One application run on one sandbox (**left**), and semantics of Stub Daemon with two applications run inside the same sandbox (**right**)

Various plugins will be instantiated and registered within the Sandboxerd and Stub Daemon through configuration files. When new process is to be run, the Sandboxerd and Stub Daemon will invoke a particular function provided by the plugins, i.e. before `unshare()` a filesystem, the `PRE_UNSHARE` plugins function will be called to accomplish the necessary task related to that particular process. The hook function will provide information of the respective parent of particular process and the sandbox environment, i.e. its confined filesystem. Notice that at each of the hook, uid `(UID)` and gid `(GID)` will be different depending on which stage of the sandbox creation process a particular plugin function is invoked.

With the adaption of the plugins architecture towards our sandbox design, it enables the flexibility in extending beyond the simple "jail" mechanism that the basic sandbox provides. The need to expand the security features of the sandbox will be available to a developer simply by implementing a plugin. We have chosen to demonstrate the use of plugin to enhance our sandbox by integrating a resource control mechanism `(cgroups)` via a plugin.

## 2.4 Resource control

Resource control enables us to establish policy in regards to memory usage or devices available in the system. We have integrated `cgroups` into our sandbox to achieve this goal. `Cgroups` also known as control groups is Linux kernel mechanism that is currently a work in progress, which provides a way to partition tasks and their respective children into a hierarchical groups [6]. It was originally developed with the intention to become a Linux container. `Cgroups` by itself provides a simple job tracking mechanism available inside the kernel. It comes with several subsystems

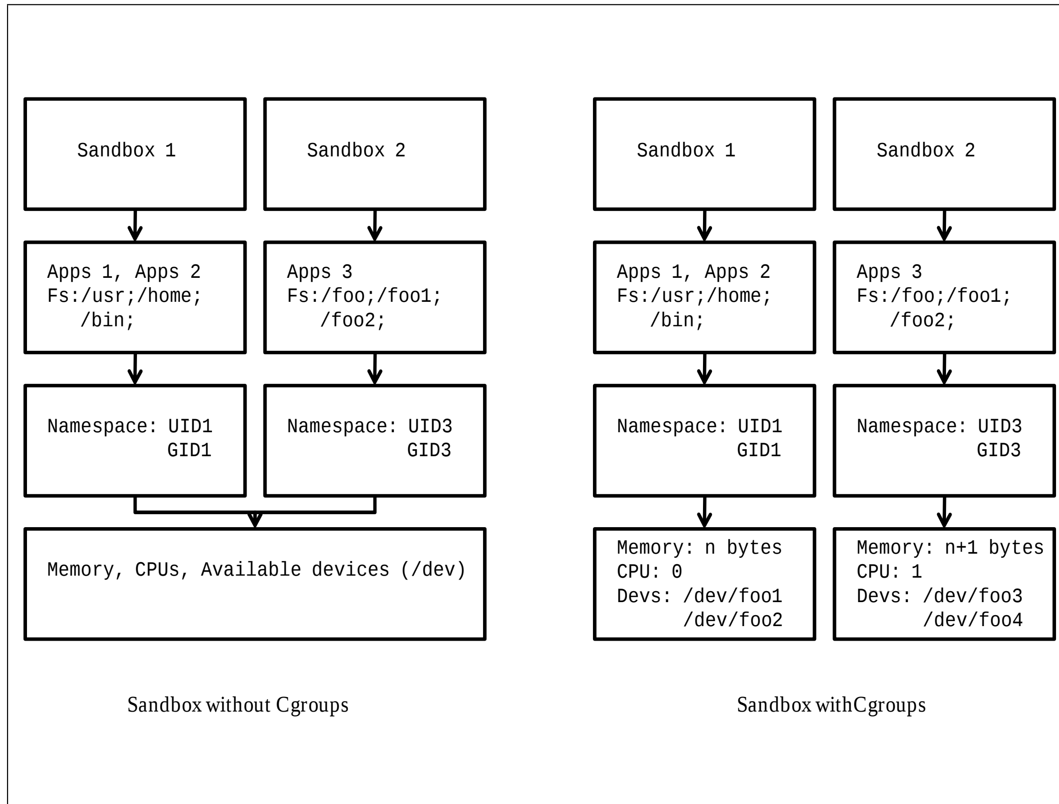



Figure 3: High level comparison on sandbox design with and without `cgroups`

which uses this basic functionality to extend the ability to provide resource controls. Currently there are ten or more `cgroups` subsystems being developed and experimented. Since this is a work in progress, more stable and new subsystems will be available in the near future. By enabling `cgroups` subsystems into our sandbox design, we are able to provide policies that capable of directly controlling the limit on the usage of system resources. Currently, we have included only a total of three subsystems (`memory, memrlimit` and `devices`) to be included as part of our sandbox design.

`Memory` subsystem provides the availability to limit the available memory per sandbox [8]. `Memrlimit` subsystem provides the same functionality as `memrlimit()` system call and this limit applies in per sandbox context (regardless of the number of processes residing inside particular sandbox) as oppose to per process context as in the original system call. `Device` subsystem provides the ability to restrict access to devices available in the system as available in `/dev` filesystem. It provides ability to track and enforce `open` and `mknod` restrictions on device files [7].

The `memrlimit` capability provides the sandbox an ability to automatically keep tracks of the total memory limit (available address space or number of bytes allocated via `malloc()`, `sbrk()`, `mmap()`). It will automatically fail any attempt to allocate dynamic memory beyond the specified allocated limit as per sandbox configuration policy. `Device` subsystem provides a way to enable and disable the available devices (device whitelists) to a particular sandbox. Customize policy that distinguished between the availability of device between a trusted sandbox and those which serves un-trusted/possible malicious application.

With these three subsystems implemented in our sandbox, we could further provide a more targeted policy control for specific applications. One example is the ability to enforce a strict policy to target sandbox usage for a potential un-trusted/malicious application. Although the filesystem and privilege isolation is a defense mechanism in the event of an attack; the system resources are still available to be exploited, specifically memory limit. As depicted in Figure 3, system's memory and cpu(s) are shared among the sandbox in the absence of `cgroups`. With `cgroups`, memory and

cpu(s) usage and access policy is enforced. The need to always ensure sufficient memory availability is a necessity for a device that has telephonic capability. In the event of an emergency call such as 911, we have to provide a high level of assurance that the call will not be interrupted in the possible event of depleted memory availability consumed by some malicious processes. With resource control functionality, thus we are able to create a policy that will pre-allocate sufficient system resources to ensure the emergency call will proceed uninterrupted.

### 2.5 `Cgroups` as a plugin

The first prerequisite in enabling the `cgroups` resource capability is to compile the Linux kernel with required `cgroups` subsystem enabled. The configuration for each sandbox object comes with the lists of `cgroups` variable of interest. The running system is also required by default to create a virtual filesystem that `cgroups` will use to operate. The hooks for the `cgroups` functionality need only to be called during the `PRE_FORK` and `PRE_SETUID` state. Initialization and setting up necessary parameters such as memory limits and lists of allowable devices are accomplished at the `PRE_FORK` state by reading available package files. At the `PRE_SETUID`, the pid of the running process in a particular sandbox will be registered with the `cgroups` system. `Cgroups` will then perform an accounting and monitoring activity.

## 3 Sandbox confines

For most applications, some shared variable is required as well as some Inter Process Communication (IPC). To the end user, the sandbox should be transparent and they should not see individual fields of data that cannot be merged. For most applications to work correctly, several environment variables must be set up in the sandbox. Examples of such environment variables are `PATH, USER, HOME, HOSTNAME,` and `GTK+` themes. However, manipulation of environment variables by a nefarious caller can lead to compromise. As such, the environment variables used inside the sandbox must be taken from a source other than the caller. Since the environment of the Stub Daemon is trusted, it serves as the source for environment variables to be used inside sandboxes.

Since the sandboxes are file-system based, most forms of IPC are possible across sandboxes. Most IPC semantics require some handle to be present for one application to know the IPC method and path to be used to connect to. For example, `D-Bus` uses the environment variable `DBUS_SESSION_ADDRESS`. For these IPC mechanisms to be readily available inside the sandbox, they need to be copied over. At present, `Xauth` cookies and `D-Bus` handles are copied over. Plugin-type architecture will allow for flexible manipulation of the sandbox environment at creation. Mechanics for sharing files between sandboxed applications can be done simply for trusted applications. Standard `POSIX` permissions and groups offer an appropriate method. For example, suppose we wish for all trusted applications to be able to read and write files in the directory `/usr/share/foo`. If all trusted applications are in the "trusted" group and `/usr/share/foo` is set `GID foo` with mode `770`, then all trusted applications can read, write, create, and execute files out of `/usr/share/foo`.

Although the sandbox environment provides almost all of the same functionality as a normal Linux programming environment, certain exceptions and caveats are present. The most notable difference for the programmer is the absence of `/proc` and `/sys` inside the sandboxes. The removal of `proc` and `sys` effectively limit the visibility of sandboxed applications to see the true environment. Most end-user applications function without `proc` being present; however application writers should be cognizant of this omission. Additionally, the `/dev` and `/etc` directories are present, but are not true copies of the respective system directories. They are "shims" with only the relevant files or sections of files present. Unlike the bind mounts for the other root directories, the shims are selective copies that are created on demand, although they can be cached. In `/etc`, for example, the full `passwd` database will not be present. However, an abridged version will be created on demand that only contains the relevant user and group information for the application(s) that run in that sandbox. Similarly, most host information available in `/etc` are not copied over. In `/dev` only devices that need to be present need to be created. For most applications, only non-hardware devices will be present, such as `random, null,` and `zero`. For trusted applications, some devices may need to be present as specified in the package file database.

Additionally, top level directories cannot be removed from within sandboxes. For example if there was a top level directory `/foo` and it was mode `777` with a bind-mounted directory, then it could not be removed from either inside or outside the sandbox. Although files can be removed from `/foo`, the directory `/foo` itself is unremovable. There are two examples where expected results may occur if the developer is not aware of the sandboxed environments. The first is the unintentional launching of an application within the same sandbox. For example, if the browser wishes to launch the email client (to handle a mailto: url), and it uses `vfork/exec` of the email executable directly instead of the convenience wrapper, it will inadvertently start the email client within the same sandbox. This will most likely lead to a non-functional email client, but could be an exploitable condition. Care should therefore be exercised. Similarly, care must be exercised with `D-Bus` activation. Since the session bus is shared amongst all sandboxes outside of their respective sandbox. An application that wishes to use `D-Bus` activation will use the convenience wrappers in its activation procedure to allow proper functionality. Failure to do so will result in activation failing. With these sandboxing approaches, we feel that we can limit the damage of application subversion, lessen the risks of disclosure of sensitive data, as well as reduce opportunities for privilege escalation.

## 4 Related work

Many implementations of sandboxing technologies focused on almost many different approaches, both software and hardware utilization. One of the hardware implementations from the work of Sahita and Kolar that proposed the use of Virtual Machine Monitor (VMM) is included in the hardware virtualization support inside Intel's CPU [1]. The implementation consisted of creating a monitoring scheme that utilized a VMM and a kernel service that will particularly monitor the request for memory usage. The monitoring agent will allocate a range of linear addresses for the new application. Memory access needs to be requested via a communication with the monitoring agent that runs as VMM. Trusted and untrusted application will run on separate isolation of memory addressing that will be determined by a policy access of a particular sandbox. The similarity in our implementation is in the policy that we have enforced for a particular sandbox which has lists of variables such as memory limit and devices white list. Our implementation is only restricting the amount of memory instead of restricting access to a range of linear addresses. With a fine grained control of range of accessible linear addresses, malicious application will not be able to gain access to the protected memory regions thus reducing the damage caused to the system. However, the tradeoff will be the complexity in the integration with the system hardware. Our implementation focused on providing a simple yet robust solution that provides the sandbox properties that will work on all platforms capable of running Linux operating system regardless of the underlying hardware design.

The work of Chang *et al.* in the implementation of user-level resource constrained sandbox is closely related to our work [10]. The implementation covered the ability to constrain the system usage of CPU, memory and network. The focus of their implementation was specifically on the Windows NT platform. Although Linux platform was included in the experimental testing, the paper lacks the discussion on the details of the implementation. CPU resource constrained is accomplished by implementing a monitoring scheme that scheduled processes based on the priority level. A process that exceeded the limit will be penalized by lowering its priority level. Memory constrained is accomplished with the way of sampling the memory usage by intercepting memory allocation API. Those applications that exceeded its limit will be penalized by having the extra memory pages marked, such that access to the marked pages will result in page fault. The difference in our implementation is that we have used the resource constrained capability-`cgroups` as part of the kernel. `Cgroups` provide a simple yet robust framework for resource control capability within the Linux kernel. Instead of penalizing a process that exceeded its limit, i.e. `malloc()` request, `cgroups` will simply return a fail for any attempt to request for resource usage beyond its preset sandbox limit. Instead of per process restriction, our `cgroups` implementation provides per sandbox policy restriction. Without the complexity of monitoring each and every resources allocation API, `cgroups` keep a simple accounting routine that will check if the policy limit has been exceeded and thus requires less processing overhead. It therefore translated into a lower total power consumption. The needs for more complex solution that involves kernel functionality could always be extended as part of the `cgroups` subsystem.

West and Gloudon proposed a user level sandbox to

provide protection for extensible system [2]. Their approach modifies a process address space to contain one or more shared pages. The extension codes will then be mapped into this shared page that comes with access privilege as a protection mechanism during transition from the user to kernel space. Through changing the access privilege of the shared page, kernel is able to maintain the integrity over the newly implemented extension codes. Though not particularly focusing on an extensible system, our design provides the capability of running a completely new or existing sandbox setup, in which a new extension of the system is desire. The sets of policy could be reused or recreated to accommodate the changes. It may contain a total set of filesystems or various sandbox variables to better accommodate and provide security isolation to the newly modified application.

Yee *et al.* introduced Native Client, the protection mechanism to run un-trusted native code specifically on the x86 based system [11]. Native Client provides a secure runtime protection and software fault isolation. A two layer of sandbox is introduced, with the inner layer provides memory reference constraint through the use of x86 segmented memory capability. The outer layer compares each request of a system call with the database of trusted system call. Our implementation does not intercept against any system call made by an application. However, we restrict the namespace of the particular application, so that it could only have the visibility of a sufficient set of filesystems to accomplish its task. Filesystems that contain system information such as `proc, sys, etc.` may not be available to the application. The application will execute with sufficient privilege to accomplish its task. In the event that software fault that could trigger an attack, such as buffer overflow, the compromised application will be confined to its own sandbox environment with its default privilege, thus gaining a root access privilege will be difficult.

Most of the isolation solutions that we have seen so far tend towards the use of restriction against access to specific memory range to protect sensitive data. With such a fine grained control of memory access, it will be able to prevent any unintended access towards a particular memory area; however, a complex modification in both hardware and software is almost a major requirement. The cost of implementing and maintaining will increase correspondingly with respect to the the complexity of the system design. System call interception is also another common approach. Our main goal of creating the sandbox solution is to use the simple approach that already existed in the system especially in the Linux platform and by integrating a kernel level control that is provided by a framework such as `cgroups` that has low system overhead. By going with a simplistic approach we are not sacrificing any security measure, since we are using a Linux system mechanism that has been thoroughly used and tested overtime in terms of its stability and security. Additionally, our goal is to enable the porting of our implementation across the many different hardware platforms with minimal difficulty.

## 5 Future work

We hope to get insightful feedback and contribution from the open source community and integrate it into our future design. With the inclusion of `cgroups`, we are always capable of improving and adding the resource control functionalities by including the new `cgroups` subsystem. The plugin capability also allows the user of our sandboxer to add an extra functionality as required.

## 6 Conclusion

This paper provides an overview of our design of the user level sandboxing design which provides filesystem namespace isolation, flexibility to extend sandbox capability through innovative design of plugins architecture, and utilization of a resource control capability through Linux kernel `cgroups`. We have described in this paper how we achieve filesystem and namespace isolation, and how we utilized our sandbox plugin capability by making `cgroups` a plugin. With the ability to contain and run predefined sets of policy, we are thus preventing a compromised application to invoke a significant damage in a system such as MIDs. This project is also part of the moblin.org open source project initiative for Linux based operating system specifically targeted for MIDs.

## 7 References

[1] R. Sahita and D. Kolar, *Beyond Ring-3: Fine Grained Application Sandboxing.* World Wide Web Consortium (W3C), December 2008.

[2] R. West and J. Gouldon, *User-Level Sandboxing: a Safe and Efficient Mechanism for Exensibility.* Technical Report, 2003-014, Boston University, June 2003.

[3] R. West and J. Gouldon, *QoS Safe' kernel extensions for real-time resource management.* The 14th EuroMicro International Conference on Real-Time Systems, June 2002.

[4] I. Goldberg, D. Wagner, R. Thomas and E. Brewer, *A secure environment for untrusted helper applications.* In Proceedings of 6th USENIX Security Symposium, July 1996.

[5] S. Miwa, T. Miyachi, M. Eto, M. Yoshizumi and Y. Shinoda, *Design Issues of an Isolated Sandbox Used to Analyze Malwares.* In Lecture Notes in Computer Science: Advances in Information and Computer Security, Heidelberg, 2007.

[6] `Cgroups` documentation. `/linux-2.6.X/Documentation/cgroups/cgroups.txt`

[7] `Cgroups` devices documentation. `/linux-2.6.X/Documentation/controllers/devices.txt`

[8] `Cgroups` memory documentation. `/linux-2.6.X/Documentation/controllers/memory.txt`

[9] Robert N. M. Watson, *Exploiting concurrency vulnerabilites in system call wrappers.* In 1st USENIX Workshop on Offensive Technologies, August 2007.

[10] F. Chang, A. Itzkovitz and V. Karamcheti, *User-level Resource-constrained Sandboxing.* USENIX Windows Systems Symposium, August 2000.

[11] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula and N. Fullagar, *Native Client: A Sandbox for Portable, Untrusted x86 Native Code.* Technical paper Google Inc, 2008.

[12] S. Santhanam, P. Elango, A. A-Dusseau and M. Linvy, *Deploying Virtual Machines as Sandboxes for the Grid.* Proceedings of the 2nd conference on Real, Large Distributed Systems-Volume 2, San Francisco 2005.

[13] B. Ford and R. Cox, *Vx32:Ligthweight User-level Sandboxing on the x86.* 2008 USENIX Annual Technical Conference, June 2008.

[14] Using `chroot()` Securely. `http://linuxsecurity.com/content/view/117632/49/`

[15] Moblin. `http://moblin.org`

# Dynamic Debug

Jason Baron
*Red Hat*
jbaron@redhat.com

## Abstract

The kernel is sprinkled with debug statements that are only available by individually re-compiling the various subsystems of the kernel. In addition, each subsystem has its own rules and methods for expressing these debug statements - `dprintk()`, `DEBUGP()`, `pr_debug()`, etc. *Dynamic debug*, introduced in kernel 2.6.28, organizes these debug statements and makes them available at run-time. Statements can be enabled on an individual basis, or via higher level organizations such as per-module. *Dynamic debug* can be thought of as a verbose mode for the kernel. We explore the design, usage, and performance impact of this new feature. We also highlight issues that have been debugged with this methodology and future work.

## 1 Introduction

The kernel contains many debug statements. In kernel 2.6.29 there are 3058 `pr_debug()` calls, 3158 `dev_dbg()` calls, 5618 `dprintk()` calls, 206 `DEBUGP()` calls, and countless additional debugging statements. Some of these statements are activated by defining `DEBUG` in the corresponding .c files, while others are activated by enabling subsystem specific configuration parameters. Some subsystems have developed sophisticated debug statement frameworks, allowing fine grain control via bit flags and debug levels. Thus, when a problem occurs that requires additional debugging information, a kernel would typically need to be re-compiled in order to obtain this additional debugging information.

According to Linux Device Drivers, "The most common debugging technique is monitoring, which in applications programming is done by calling *printf* at suitable points. When you are debugging kernel code, you can accomplish the same goal with *printk*"[4]. Many, if not most user space programs have a verbose mode. So, why doesn't the kernel have such a mode?

Really, the kernel does already have such a mode. We currently label *printk* messages, from most severe, `KERN_EMERG`, to least severe, `KERN_DEBUG`. Thus, a *printk* of level `KERN_DEBUG` already exists. Why don't we simply convert the 10,000 or so debug statements previously mentioned, to *printk*s of level `KERN_DEBUG` and be done with it?

First, enabling all those debug statements would clutter up the logs. These debug statements are often found in high frequency code paths, and thus would make noise to signal ratio of the logs rather high. Second, even when a *printk* message doesn't make its way to the console and/or the system logs, we still 'render' every *printk* in the kernel, which is an expensive operation. In 'rendering' a *printk*, we format the messages into a buffer with locks held and irqs disabled. In fact, we saw an 86% tbench performance degradation, when enabling all the debug statements, but not logging any of the messages to the system logs or console. Thus, there is an enormous cost associated with simply turning these messages into *printk* statements of level `KERN_DEBUG`.

*Dynamic debug* addresses these two core concerns. The verbosity is tackled using a unique language, which allows expression for fine grain control of each debug statement, while also permitting coarser control, using for example, per-module enabling. This control language was originally developed by Greg Banks at SGI and was incorporated into *dynamic debug*[1]. The run-time performance concerns are addressed while making use of a bloom filter[2].

There has been a lot of work recently in the general area of kernel tracing. Ftrace[6], LTTng[5], and Systemtap[3] can all be used to trace the kernel. We view *dynamic debug* as a complementary technology that can be used in conjunction with these other tools. There may also be ways for *dynamic debug* to leverage and/or combine with some of these tools, which we will explore.

In the next section, we will look at the implementation of *dynamic debug*. We will then look at its usage and some examples. Next, we will analyze the impact of *dynamic debug* in terms of its size and performance. We then introduce a subtle variation on *dynamic debug* which can handle more complex debugging statements. Finally, we conclude by commenting on the organization of kernel debug statements, and suggesting areas for future work.

## 2 Implementation

As mentioned, the two central goals of the implementation are controlling what ends up in the system logs or filtering, and efficiency or minimal run-time cost. The initial implementation focused on two functions - `pr_debug` and `dev_dbg`. These functions are defined centrally, thus, 'overwriting' their definition was contained to two source files.

We then associate meta data with each debug statement. This meta data records the containing .c file, line number, containing module, and associated format string. This information enables the user to understand and control which statements are enabled. The size of the meta data is then proportional to information stored for each debug statement and the number of debug statements. We explore the meta data associated with each debug statement, and the run-time control, in the following two sections entitled 'data structures', and 'bloom filters' respectively.

### 2.1 Data Structures

We hook into the `pr_debug` macro with some simple macro magic. Prior, to the introduction of *dynamic debug* we had:

```
#ifdef DEBUG
#define pr_debug(fmt, arg...) \
    printk(KERN_DEBUG fmt, ##arg)
#else
#define pr_debug(fmt, arg...) \
    ({ if (0) printk(KERN_DEBUG fmt, ##arg); 0; })
#endif
```

We re-define this construct as follows:

```
#if defined(DEBUG)
#define pr_debug(fmt, ...) \
    printk(KERN_DEBUG pr_fmt(fmt), ##__VA_ARGS__)
#elif defined(CONFIG_DYNAMIC_DEBUG)
#define pr_debug(fmt, ...) do { \
    dynamic_pr_debug(fmt, ##__VA_ARGS__); \
    } while (0)
#else
#define pr_debug(fmt, ...) \
    ({ if (0) printk(KERN_DEBUG pr_fmt(fmt),
                            ##__VA_ARGS__); 0; })
#endif
```

Thus, if `DEBUG` is defined, we continue to get an inlined *printk* statement, while no code is generated when `DEBUG` is not defined. The new case that we have introduced results from defining the new configuration parameter `CONFIG_DYNAMIC_DEBUG`. When this configuration parameter is defined, we hook into the new dynamic debug code. Note, that `DEBUG` takes precedence over `CONFIG_DYNAMIC_DEBUG`. In this way, code that defines `DEBUG` continues to work as was previously expected. The `dev_dbg()` hook is implemented in a similar manner.

The `dynamic_pr_debug()` macro expands to store debug statement data in the `struct _ddebug` data structure, which follows.

```
struct _ddebug {
    /*
     * These fields are used to drive the user
     * interface for selecting and displaying
     * debug callsites.
     */
    const char *modname;
    const char *function;
    const char *filename;
    const char *format;
    char primary_hash;
    char secondary_hash;
    unsigned int lineno:24;
    /*
     * The flags field controls the behaviour
     * at the callsite. The bits here are
     * changed dynamically when the user
     * writes commands to
     * <debugfs>/dynamic_debug/control
     */
#define _DPRINTK_FLAGS_PRINT   (1<<0)
#define _DPRINTK_FLAGS_DEFAULT 0
    unsigned int flags:8;
} __attribute__((aligned(8)));
```

The modname, function, filename and lineno fields are populated using the C code definitions `KBUILD_MODNAME`, `__func__`, `__FILE__`, and `__LINE__`, respectively. The format field is filled using the supplied in format string. The `primary_hash` and `secondary_hash` fields are explained in subsequent

bloom filter section. Finally, the flags field is used as a per-call site control variable.

Thus, the `struct _ddebug` data structure is 40 bytes in size. With a combined 6,000 call sites for `pr_debug` and `dev_dbg`, that amounts to 23k(6000*40) in addition to the size of associated strings. We share results of the kernel image size increases in the results section.

When the kernel boots or as new modules are inserted and removed we create a new `struct ddebug_table` structure for each logically module, which follows.

```
struct ddebug_table {
        struct list_head link;
        char *mod_name;
        unsigned int num_ddebugs;
        unsigned int num_enabled;
        struct _ddebug *ddebugs;
};
```

The pointer to the `_ddebugs` structures which are associated with a particular module are then assigned to the `_ddebug` field of the associated `ddebug_table` data structure. The `ddebug_table` structures are then linked together in a linked list. Thus, using this list we can easily look up entries and display them. A linked list works fine since looking up these entries when setting values, is not a hot path.

### 2.2 Bloom Filter

We associate two hash values with each instrumented debug statement. Both hashes are in the range 0-64. These are stored in the `primary_hash` and `secondary_hash` fields of the `_ddebug` structure. We use the djb2 and the r5 hash algorithms. The input to the hash function is the Linux source code directory and the module name. Thus, modules of the same name that are located in different source directories likely have different hash values. Thus, hash bits 'n' and 'm' are associated with each debug statement.

Two global variables of type `long long` are also introduced. They are `dynamic_debug_enabled` and `dynamic_debug_enabled2`. When we wish to enable a debug statement, we set bit 'n' and 'm' in the global variables `dynamic_debug_enabled` and `dynamic_debug_enabled2` respectively. Thus, each debug statement is conditioned on having bits 'n' and 'm' set in the global variables `dynamic_debug_enabled` and `dynamic_debug_enabled2` respectively. This is a variation on a bloom filter[2]. There can be false positives, and thus we use the `flags` field of the `struct _ddebug` field as a third, and definitive check before calling into the associated *printk* statement.

Thus, the pseudo-code for the above described case is:

```
bit1 = hash1(kernel path + module name)
bit2 = hash2(kernel path + module name)
if (bit1 & dynamic_debug_enabled &&
    bit2 & dynamic_debug_enabled &&
    _ddebug field is set)
    (do the printk)
```

While there may be more complex implementation which involve live code patching, such as the immediate variable work, we find this implementation to be a good trade off between complexity and speed. Notice we have no locking or synchronization of any kind. Thus in the off case we expect to execute only a couple of additional instructions, and because we are relying only two global variables, we expect the code to exhibit good caching property. We can also further tweak the properties of the bloom filter by creating additional levels of hashing. Notice we could also fold the hashing into one global variable as well.

Next we turn to hash collisions. In building the v2.6.29 kernel with these options, when all module are disabled, both global variables are set to 0, and thus we have no false positives. When we turn all debugging on, we set both of the globals to all 1s, and thus we have no false positives in this case either. When one module is enabled, we also have no false positives. This is not necessarily true, but is true for v2.6.29 kernel that was tested. This kernel produced eighty separate modules, and thus eighty unique sets hashes. We did not compute three way collisions.

Thus, when no, one, or all modules are enabled we have no false positives. Even in the case where we do have a false positive, we do not call through to a function we simply check the unique variable associated with the corresponding debug statement.

## 3 Usage and Examples

### 3.1 Controlling Dynamic Debug Behavior

The behaviour of `pr_debug()` and `dev_debug()` are controlled by writing to a control file in the debugfs filesystem. Thus, you must first mount the debugfs filesystem, in order to make use of this feature. Subsequently, we refer to the control file as: `<debugfs>/dynamic_debug/control`. For example, if you want to enable printing from source file svcscok.c, line 1603 you simply do:

```
# echo 'file svcsock.c line 1603 +p' >
<debugfs>/dynamic_debug/control
```

If you make a mistake with the syntax, the write will fail thus:

```
# echo 'file svcsock.c blah 1 +p' >
<debugfs>/dynamic_debug/control
-bash: echo: write error: Invalid argument
```

### 3.2 Viewing Dynamic Debug Behavior

Viewing the current configuration is done with a simple read. See Figure 1

### 3.3 Command Language Reference

At the lexical level, a command comprises a sequence of words separated by whitespace characters. Note that newlines are treated as word separators and do not end a command or allow multiple commands to be done together. So these are all equivalent:

```
# echo -c 'file aio.c line 1603 +p' >
<debugfs>/dynamic_debug/control
# echo -c ' file aio.c     line  1603 +p  ' >
<debugfs>/dynamic_debug/control
# echo -c 'file aio.c\nline 1603 +p' >
<debugfs>/dynamic_debug/control
# echo -n 'file aio.c line 1603 +p' >
<debugfs>/dynamic_debug/control
```

Commands are bounded by a write() system call. If you want to do multiple commands you need to do a separate "echo" for each:

```
# echo 'file aio.c line 1603 +p' >
       <debugfs>/dynamic_debug/control;\
> echo 'file svcsock.c line 1563 +p' >
       <debugfs>/dynamic_debug/control
```

or even:

```
# (
> echo 'file svcsock.c line 1603 +p' ;\
> echo 'file svcsock.c line 1563 +p' ;\
> ) > <debugfs>/dynamic_debug/control
```

At the syntactical level, a command comprises a sequence of match specifications, followed by a flags change specification.

```
command ::= match-spec* flags-spec
```

The match-spec's are used to choose a subset of the known debug statements to which to apply the flags-spec. Think of them as a query with implicit ANDs between each pair. Note that an empty list of match-specs is possible, but is not very useful because it will not match any debug statement call sites.

A match specification comprises a keyword, which controls the attribute of the call site to be compared, and a value to compare against. Possible keywords are:

- *match-spec* ::= 'func' string | 'file' string | 'module' string | 'format' string | 'line' line-range
- *line-range* ::= lineno | '-'lineno | lineno'-' | lineno'-'lineno // Note: line-range cannot contain space, e.g. // "1-30" is valid range but "1 - 30" is not.
- *lineno* ::= unsigned-int

The meanings of each keyword are:

- *func.* The given string is compared against the function name of each callsite. Example:

  func `svc_tcp_accept`
- *file.* The given string is compared against either the full pathname or the basename of the source file of each callsite. Examples:

  file svcsock.c
  file sched.c

You can view the currently configured behaviour of all the debug statements via:

```
# cat <debugfs>/dynamic_debug/control
# filename:lineno [module]function flags format
fs/sysfs/file.c:147 [file]sysfs_read_file - "%s: count = %zd, ppos = %lld, buf = %s\012"
fs/sysfs/dir.c:788 [dir]__sysfs_remove_dir - "sysfs %s: removing dir\012"
fs/sysfs/bin.c:110 [bin]read - "offs = %lld, *off = %lld, count = %d\012"
fs/debugfs/inode.c:217 [debugfs]debugfs_create_file - "debugfs: creating file '%s'\012"
```

You can also apply standard Unix text manipulation filters to this data, e.g.:

```
# grep -i aio <debugfs>/dynamic_debug/control  | wc -l
10
# grep -i security <debugfs>/dynamic_debug/control | wc -l
163
```

Note in particular that the third column shows the enabled behaviour flags for each debug statement call site. The default value, no extra behaviour enabled, is "-". So you can view all the debug statement call sites with any non-default flags:

```
[root@mets dynamic_debug]# awk '$3 != "-"' control
# filename:lineno [module]function flags format
fs/aio.c:77 [aio]aio_setup p "aio_setup: sizeof(struct page) = %d\012"
fs/aio.c:222 [aio]__put_ioctx p "__put_ioctx: freeing %p\012"
fs/aio.c:1788 [aio]sys_io_cancel p "calling cancel\012"
fs/aio.c:1698 [aio]sys_io_submit p "EINVAL: io_submit: invalid context id\012"
fs/aio.c:1604 [aio]io_submit_one p "EINVAL: io_submit: overflow check\012"
fs/aio.c:1594 [aio]io_submit_one p "EINVAL: io_submit: reserve field set\012"
fs/aio.c:1335 [aio]sys_io_destroy p "EINVAL: io_destroy: invalid context id\012"
fs/aio.c:1303 [aio]sys_io_setup p "EINVAL: io_setup: ctx %lu nr_events %u\012"
fs/aio.c:248 [aio]ioctx_alloc p "ENOMEM: nr_events too high\012"
fs/aio.c:1022 [aio]aio_complete p "added to ring %p at [%lu]\012"
```

Figure 1: Viewing current dynamic debug status

- *module*. The given string is compared against the module name of each callsite. The module name is the string as seen in "lsmod", i.e. without the directory or the .ko suffix and with '-' changed to '_'. Examples:

  module sunrpc
  module nfsd

- *format*. The given string is searched for in the dynamic debug format string. Note that the string does not need to match the entire format, only some part. Whitespace and other special characters can be escaped using C octal character escape notation, e.g. the space character is
  040. Alternatively, the string can be enclosed in double quote. Examples:

  format svcrdma: // many of the NFS/RDMA server dprintks
  format readahead // some dprintks in the readahead cache
  format nfsd:
  040SETATTR // one way to match a format with whitespace
  format "nfsd: SETATTR" // a neater way to match a format with whitespace
  format 'nfsd: SETATTR' // yet another way to match a format with whitespace

- *line*. The given line number or range of line numbers is compared against the line number of each debug statement call site. A single line number matches the call site line number exactly. A range of line numbers matches any call site between the first and last line number inclusive. An empty first number means the first line in the file, an empty line number means the last number in the file. Examples:

  line 1603 // exactly line 1603
  line 1600-1605 // the six lines from line 1600 to line 1605
  line -1605 // the 1605 lines from line 1 to line 1605
  line 1600- // all lines from line 1600 to the end of the file

The flags specification comprises a change operation followed by one or more flag characters. The change operation is one of the characters:

- - remove the given flags
- + add the given flags
- = set the flags to the given flags

The flags are:

- *p* Causes a `printk()` message to be emitted to dmesg

Note the regexp `^[-+=][p]+$` matches a flags specification. Note also that there is no convenient syntax to remove all the flags at once, you need to use "-p".

### 3.4 Examples

In the figure 2 below we show two examples, to give a flavor of the output. The first example shows enabling all messages. The second example shows enabling kobject module output while the cifs module is loaded.

## 4 Size and Performance

The kernel used for testing was v2.6.28 compiled for x86_64. An Intel quad core machine running at 1.6 GHz with 2GB of RAM was used for all tests.

| test case | tbench throughput |
|---|---|
| `CONFIG_DYNAMIC_DEBUG` disabled | 773.054 MB/sec |
| `CONFIG_DYNAMIC_DEBUG` enabled | 773.913 MB/sec |
| `CONFIG_DYNAMIC_DEBUG` enabled and all debug statements enabled | 79.664 MB/sec |

Table 1: performance results

Thus, the run-time cost of having `CONFIG_DYNAMIC_DEBUG` enabled, but none of the debug statements printing, is negligible. However, when we enable all of the debugging statements, the system throughput drops quite dramatically. Thus, simply converting all of these high frequency debug statements to *printk* at `KERN_DEBUG` level is not viable. This also suggests that alternate methods for 'rendering' the format strings might be worth investigating. We discuss this further in the future work section.

In terms of kernel code size growth, the kernel increased 2% when enabling `CONFIG_DYNAMIC_DEBUG`.

## 5 More Complex Debugging Statements

Thus far, we've looked at debug statements that are binary - they are either enabled or disabled. However, several kernel subsystems have developed more complex debugging facilities based on 'levels' or 'flags'. The 'levels' model is employed by the CPU frequency subsystem, where messages above a configurable level *n* are emitted. Currently, the level is set via module parameters. Thus, to change the level, one would need to unload and re-load the CPU frequency modules. The NFS filesystem uses a 'flags' style of debugging, where each 'flag' or bit in an integer toggles on or off a set of debugging statements.

If we extend the *dynamic debug* construct somewhat, we can accommodate both the 'level' and 'flags' debugging style. For 'flags' we can create the following general function (pseudo-code):

```
#define debug_enabled_flag
        (flag_bit, subsys_set_bits)
if (normal dynamic debug checks) {
    if (flag_bit & subsys_set_bits) {
        return 1;
   }
}
return 0;
```

The 'flag_bit' refers to the bit associated with this particular debug statement. The 'subsys_set_bits' refers to the global integer which is associated with this subsystem. The 'normal dynamic debug checks' have the associated hash bits set corresponding module if any of the flag bits are set. We can then design a subsystem specific macro for any subsystem as follows:

```
#define subsystem_foo_level_debug
        (flag_bit, fmt, va_args)
if (debug_enabled_flag(flag_bit,
                       subsys_set_bits)) {
    print(fmt, va_args);
}
```

Subsystem can thus pass in any subsystem specific print information. Also, by designing this interface in this manner, subsystems could easily perform any additional checks that they wish where the 'print' statement is located. Thus, we can imagine *dynamic debug* being used for more than just printing information. We've

```
# cut -f2 -d"[" control | cut -f1 -d"]" | xargs -i echo 'module {} +p' > control

Apr 14 15:17:49 mets kernel: [ 3883.017536] nf_conntrack:tcp_in_window: START
Apr 14 15:17:49 mets kernel: [ 3883.017539] nf_conntrack:tcp_in_window: <7>nf_conntrack:seq=376950469
ack=3577053373 sack=3577053373 win=1803 end=376950469
Apr 14 15:17:49 mets kernel: [ 3883.017549] nf_conntrack:tcp_in_window: sender end=376950469
maxend=376964293 maxwin=115392 scale=6 receiver end=3577053373 maxend=3577168717 maxwin=13824 scale=7
Apr 14 15:17:49 mets kernel: [ 3883.017555] nf_conntrack:tcp_in_window: <7>nf_conntrack:seq=376950469
ack=3577053373 sack =3577053373 win=1803 end=376950469
Apr 14 15:17:49 mets kernel: [ 3883.017565] nf_conntrack:tcp_in_window: sender end=376950469
maxend=376964293 maxwin=115392 scale=6 receiver end=3577053373 maxend=3577168717 maxwin=13824 scale=7
Apr 14 15:17:49 mets kernel: [ 3883.017571] nf_conntrack:tcp_in_window: I=1 II=1 III=1 IV=1
Apr 14 15:17:49 mets kernel: [ 3883.017577] nf_conntrack:tcp_in_window: res=1 sender end=376950469
maxend=376964293 maxwin=115392 receiver end=3577053373 maxend=3577168765 maxwin=13824
Apr 14 15:17:49 mets kernel: [ 3883.017582] nf_conntrack:tcp_conntracks: <7>nf_conntrack:syn=0 ack=1
fin=0 rst=0 old=3 new=3
Apr 14 15:17:50 mets kernel: [ 3883.110062] file:sysfs_read_file: count = 4096, ppos = 0,
buf = 00000000,0000000f
Apr 14 15:17:50 mets kernel: [ 3883.110116] file:sysfs_read_file: count = 4096, ppos = 0,
buf = 00000000,00000001
Apr 14 15:17:50 mets kernel: [ 3883.110164] file:sysfs_read_file: count = 4096, ppos = 0,
buf = 00000000,00000005
Apr 14 15:17:50 mets kernel: [ 3883.110204] file:sysfs_read_file: count = 1, ppos = 0,
buf = 1

# echo 'module kobject +p' > /mnt/debugfs/dynamic_debug/control
# /sbin/modprobe cifs

Apr 14 15:54:45 mets kernel: [  184.968002] kobject:kobject: 'cifs' (ffffffffa007e0f0):
kobject_add_internal: parent: 'module', set: 'module'
Apr 14 15:54:45 mets kernel: [  184.970204] kobject:kobject: 'holders' (ffff880073c34580):
kobject_add_internal: parent: 'cifs', set: '<NULL>'
Apr 14 15:54:45 mets kernel: [  184.970225] kobject:kobject: 'cifs' (ffffffffa007e0f0):
fill_kobj_path: path = '/module/cifs'
Apr 14 15:54:45 mets kernel: [  184.970267] kobject:kobject: 'notes' (ffff880073c34440):
kobject_add_internal: parent: 'cifs', set: '<NULL>'
Apr 14 15:54:45 mets kernel: [  184.970761] kobject:kobject: 'cifs_inode_cache' (ffff880075d230a8):
kobject_add_internal: parent: 'slab', set: 'slab'
Apr 14 15:54:45 mets kernel: [  184.970807] kobject:kobject: 'cifs_inode_cache' (ffff880075d230a8):
fill_kobj_path: path = '/kernel/slab/cifs_inode_cache'
Apr 14 15:54:45 mets kernel: [  184.970862] kobject:kobject: ':0016512' (ffff880075d214a8):
kobject_add_internal: parent: 'slab', set: 'slab'
```

Figure 2: Dynamic debug output examples

recently re-named this work to *dynamic debug* from the original *dynamic printk*, to make this clear. The above `debug_enabled_flag()` macro could easily accommodate the 'level' style debugging, by replacing the bit check with a greater than check. Prototypes have already implemented and will be proposed in the near future.

There are a number of modules that set module debugging levels using module parameters Thus, we would propose system wide 'standard' module name parameters that *dynamic debug* can implement. For example, as dynamic_debug_level=n, dynamic_debug_flag=0101, and dynamic_debug=enabled/disabled.

## 6 Debug Statement Organization

As mentioned at the outset of the work, there are a myriad of styles and macros for debug printing. We propose that the various subsystems make use of the 'core' debugging functions to the extent that they suit their needs. For example, if you are just printing out text use `pr_debug()`, or `dev_dbg()` if you are in a driver. If you are doing 'flag' or 'level' style debugging use the corresponding *dynamic debug* macros. For example, in kernel/module.c, `DEBUGP()` is used. We should convert it to `pr_debug()` so that it can tie into the *dynamic debug* infrastructure.

The question also becomes when should one use `pr_debug()` and when should `printk(KERN_DEBUG)`

be used? Obviously, given the test results posted one can not simply sprinkle `printk(KERN_DEBUG)` everywhere. Thus, for frequently used codepaths need to use `pr_debug()`. Additionally, now that `pr_debug()` can be compiled in, `pr_devel()` can be used for those cases where you wouldn't want *dynamic debug* to pick up the debug statement.

## 7 Future Work

Clearly, converting more kernel code to use the standard debug statements of `pr_debug()` and `dev_dbg()` is desired. Further, along these lines would be converting subsystems that have more complex debugging styles to the proposed framework. Also, adding command line control parameters, module parameters, and a simple enable and disable all debug statements control mode would be desirable.

As mentioned in the results section, when all the debug statements are enabled, the system performance drops significantly. Although, this is not the 'hot path', it might be nice to improve this case. By simply attaching the backend of these patches to the new ring buffer code we should drastically speed things up. Perhaps, its an option as I think getting the information out via the normal printk path is important as well. There might also be a chance for further integration with Ftrace code specifically the event code[7].

## 8 Conclusion

'Dynamic Debug' has successfully made high frequency debug statements available at run-time in a manner that does not degrade performance. It has already been used by SGI to help resolve NFS problems, and it is planned to be incorporated into upcoming enterprise kernel releases. We hope that this paper will further understanding of this new feature, in hopes that it can be further adopted and expanded.

## 9 Acknowledgements

SGI independently developed a very similar debugging system which tied into *dprintk()* and has been used for a number of years to help diagnose customer issues[1]. Greg Banks submitted this work upstream shortly after I submitted the *dynamic debug* work. *Dynamic debug* owes its control language to this work. Section 3, Usage and Examples, is largely taken from the documentation that Greg Banks wrote for the *dprintk()* work.

## References

[1] Greg Banks. activate & deactivate dprintks individually and severally. `http://marc.info/?l=linux-kernel&m=123241522202638&w=2`.

[2] Pei Cao. Bloom filters - the math. `http://pages.cs.wisc.edu/~cao/papers/summary-cache/node8.html`.

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

[4] Greg Kroah-Hartman Jonathan Corbet, Alessandro Rubini. *Linux Device Drivers*. O'Reilly, 2005. ISBN 0-596-00590-3.

[5] Michel Dagenais Mathieu Desnoyers. Lttng: Tracing across execution layers, from the hypervisor to user-space. In *Proceedings of the Ottawa Linux Symposium 2006*, 2006.

[6] Steve Rostedt. ftrace tracing inftrastructure. `http://lwn.net/Articles/270971/`.

[7] Steven Rostedt. event tracer. `http://marc.info/?l=linux-kernel&m=123550413414913&w=2`.

# Measuring Function Duration with Ftrace

Tim Bird
*Sony Corporation of America*
tim.bird@am.sony.com

## Abstract

FTrace is a relatively new kernel tool for tracing function execution in the Linux kernel. Recently, FTrace added the ability to trace function exit in addition to function entry. This allows for measurement of function duration, which adds an incredibly powerful tool for finding time-consuming areas of kernel execution.

In this paper, the current state of the art for measuring function duration with FTrace is described. This includes recent work to add a new capability to filter the trace data by function duration, and tools for analyzing kernel function call graphs and visualizing kernel boot time execution.

## Introduction

Analyzing a running operating system kernel can be a difficult task. In the 2.6.27 version of the kernel, a powerful tracing mechanism called Ftrace was added to mainline Linux. Ftrace provides some very nice facilities for instrumenting the kernel, recording trace data, and outputting the data to user space.

The Ftrace system provides a generic tracing framework in the kernel, upon which several different kinds of tracers can be implemented. Different kinds of tracers utilize different methods of instrumenting the kernel code and different data collection algorithms.

Ftrace supports the ability do basic *function* tracing, which consists of recording information at the time of entry to every function executed in the kernel. Additionally, on some architectures, Ftrace supports the ability to perform *function graph* tracing, which involves tracking not just function entry but also function exit, and the ability to measure function duration. This is is useful to find performance problems and latency problems in the kernel.

This paper presents work by the author to add function graph tracing to the ARM architecture. This includes a description of the mechanisms used and some of the issues involved on the ARM architecture.

Also, this paper describes the author's efforts to add duration filtering to the function graph tracer. Even on a relatively slow processor, the kernel executes many thousands of functions per second. Without filtering, the length of time that data can be captured in the trace log without loss is very limited. By adding duration filtering, it is possible to greatly extend the duration of a trace, to capture more events of interest and to help isolate problem areas.

## 1 Overview of Ftrace Operation

### 1.1 Instrumentation

Ftrace operates by adding tracepoints to the Linux kernel. The insertion into the Linux kernel of locations where tracing information is recorded is referred to as *instrumentation*. Instrumentation comes in two main forms—explicitly declared tracepoints, and implicit tracepoints. Explicit tracepoints consist of developer-defined declarations which specify the location of the tracepoint, and additional information about what data should be collected at a particular trace site. Implicit tracepoints are placed into the code automatically by the compiler, either due to compiler flags or by developer redefinition of commonly used macros.

Function tracing and function graph tracing utilize implicit instrumentation. The kernel consists of many thousands of C functions, and it would be extremely impractical to maintain explicit tracepoint definitions for all of them. To instrument functions implicitly, when the kernel is configured to support function tracing, the kernel build system adds `-pg` to the flags used with the compiler. This causes the compiler to add code to

```
00000570 <sys_sync>:
 570:  e1a0c00d   mov     ip, sp
 574:  e92dd800   stmdb   sp!, {fp, ip, lr, pc}
 578:  e24cb004   sub     fp, ip, #4  ; 0x4
 57c:  e3a00001   mov     r0, #1  ; 0x1
 580:  ebfffa0    bl      408 <do_sync>
 584:  e3a00000   mov     r0, #0  ; 0x0
 588:  e89da800   ldmia   sp, {fp, sp, pc}
```

Figure 1: ARM code *without* call to mcount

```
00000570 <sys_sync>:
 570:  e1a0c00d   mov     ip, sp
 574:  e92dd800   stmdb   sp!, {fp, ip, lr, pc}
 578:  e24cb004   sub     fp, ip, #4  ; 0x4
 57c:  e1a0c00e   mov     ip, lr
 580:  ebfffffe   bl      0 <mcount>
 584:  00000028   andeq   r0, r0, r8, lsr #32
 588:  e3a00001   mov     r0, #1  ; 0x1
 58c:  ebffff9d   bl      408 <do_sync>
 590:  e3a00000   mov     r0, #0  ; 0x0
 594:  e89da800   ldmia   sp, {fp, sp, pc}
```

Figure 2: ARM code *with* call to mcount

the prologue of each function, which calls a special assembly routine called `mcount`. This compiler option is specifically intended to be used for profiling and tracing purposes.

Figures 1 and 2 show the ARM assembly code generated when compiling the short routine `sys_sync()` both with and without the `-pg` compiler flag. The assembly code was produced from the compiled object file with the command: `arm-eabi-objdump -S fs/sync.o >fs/sync.S`. Comparing the two shows that the mcount call only takes a few extra instructions.

The `mcount` routine is written in platform-specific assembly, located in the file `arch/arm/kernel/entry-common.S`, for the ARM platform. It is called every time a function is entered. Because of this, it is important that the routine have very low overhead, especially when tracing is disabled[1].

Another issue with use of `mcount` is that it is incompatible with certain kinds of compiler optimizations. `mcount` must be called with a consistent stack frame and frame pointer, in order for it to operate correctly. Some compiler optimizations produce stack frames, frame pointers, or call sequences that would cause `mcount` to be inaccurate, or worse, to function incorrectly. For example, on the ARM platform, the kernel must be compiled to use frame-pointers in order for function tracing to work correctly. That is, you cannot use the `-fomit-frame-pointers` compiler option.

Luckily, when the `-pg` compiler option is used, the gcc compiler automatically disables several optimizations which it might normally perform. Also, the kernel configuration system automatically adjusts compiler flags at build time to avoid conflicts between tracing options and optimization options.

### 1.2 Tracing at Runtime

At runtime, tracing is disabled until enabled by the user. In this situation, the `mcount` routine returns as quickly as possible to the instrumented function, and kernel processing continues. When tracing is enabled, `mcount` calls the function corresponding to the user-selected tracer, which then records information and makes an entry in the trace log.

Tracing can be enabled by the user by the manipulation of pseudo-files in the debug file system. The user can select what tracer to activate, and also set various tracing parameters. Files in the `Documentation/trace` directory describe the pseudo-files that are presented by Ftrace, the different tracers, and what parameters can be used by each one. In general, there are files for initiating and suspending a trace, adjusting the trace log size, for setting parameters for trace-time filtering, and for customizing the format of the trace log output.

### 1.3 Trace Data Capture

The trace log is kept in a new kernel data structure called the ring buffer. This data structure is specifically designed for holding trace data, for quick and lockless data entry, and for simultaneous reader and writer access to the buffer.

The ring buffer provides automatic management of timestamps used with the trace data. Also, it provides page-aligned, per-cpu buffers for holding trace data. A

[1] Note that on some platforms, Ftrace includes the capability to use “dynamic tracepoints,” whereby the tracepoints are replaced with ’nop’ instructions at runtime, to reduce overhead when not tracing. This is a very neat capability, which dramatically reduces overhead and makes it feasible to leave tracing configured on even for some production systems. However, detailed discussion of this capability is outside the scope of this paper.

more detailed description of the ring buffers is outside the scope of this document, but see `Documentation/trace/ring-buffer-design.txt` for more information.

Note that to avoid locking operations, data entry into the ring buffer is done in steps. First, the data position is reserved in the buffer, using the function `ring_buffer_lock_reserve()`. The data position is reserved in an atomic fashion, to avoid a costly lock operation. (Note that the word *lock* in the function name is misleading.)

Next the data for the trace event is filled in. If the trace data is to be saved (the normal case), then `ring_buffer_unlock_commit()` is called to commit the data to the buffer. If for some reason the event data should not be saved, then `ring_buffer_discard_commit()` can be called to eliminate the event from the buffer. If no other data has been written to the buffer, the discard_commit operation can remove the data from the buffer. However, if other data has been written, `ring_buffer_discard_commit()` just marks the data so that it is ignored by the tracer output system. In the case of filtering, it is highly desirable to not merely mark the data, but to actually remove it from the buffer, to free up space for other event data. This will be discussed in more depth in Section 2.1.

### 1.4 Trace output

Finally, a user can access the trace data via more debugfs pseudo-files. Trace data is formatted in plain text, and intended to be easily readable by humans, as well as easily processable by post-trace analysis tools.

Trace data can be accessed either after a trace has completed, or during a trace run.

### 1.5 Function Graph Tracing

Function graph tracing is a form of function tracing where both the function entry and exit are tracked by the tracer. With "regular" function tracing, only function entry is traced. When both the entry and exit of functions are available, it is possible to see the relationship between functions. It is possible to reconstruct the complete graph of function calls for a particular operation in the kernel. This is very helpful to understand the operation of the kernel, and also to detect anomalies in kernel operation. Also, by measuring both entry and exit, it is possible to measure the duration of each function.

Function graph tracing utilizes the same compiler instrumentation as function tracing. However, using the `mcount` mechanism to capture the exit of a function requires some tricky manipulation of the stack and call sequence. Since the `-pg` compiler option only adds instrumentation for function entry, the Ftrace system needs to adjust the register and stack conditions before returning to execute the instrumented function so that Ftrace can regain control when the function exits.

It does this with a return "trampoline." This is shown in Figure 3. When Ftrace is called on function entry, it records the real return address (the address that the instrumented function was called from) and saves it in the process' task structure. Because multiple functions will nest before the returns are processed, these are kept in a stack of return addresses. After Ftrace calls the function graph tracer, it replaces the return address (either on the stack or in a register, depending on the architecture and ABI being used) with the address of an Ftrace routine to handle the return trace. Then Ftrace returns to the instrumented routine so that it can execute. When the instrumented routine finishes and returns, instead of returning to it's original caller, it returns to Ftrace. Ftrace then calls the function graph tracer again, with the function exit tracepoint data. Then Ftrace retrieves the real return address from the task structure, and returns to the real caller.

## 2 Adding Function Graph Tracing to ARM

Function graph tracing was originally developed on the x86 architecture. This section describes some of the issues encountered while adding support for this feature to the 2.6.30 Linux kernel, for the ARM platform.

Here is the list of problems encountered, and the solutions implemented to fix them.

1. Basic function tracing was supported for the ARM architecture, but testing revealed that the system hung when it was activated on my particular platform.

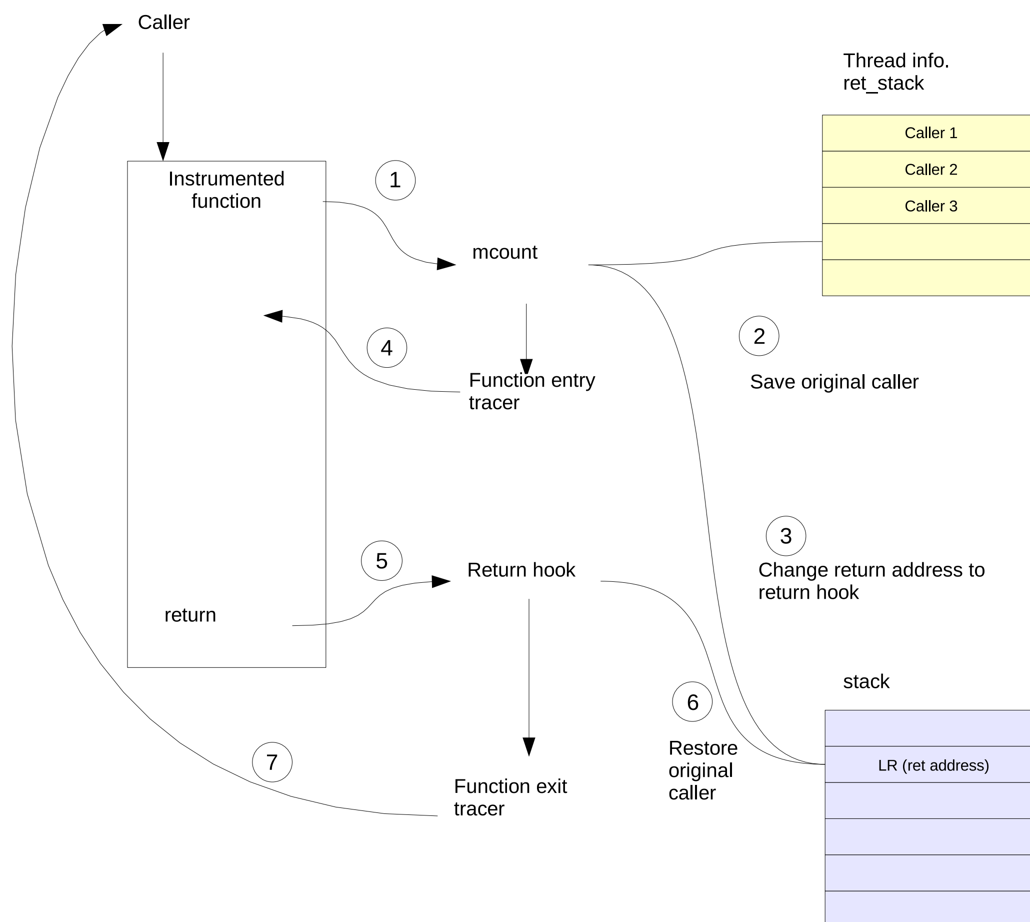



Figure 3: Mcount Handling and Return Trampoline

This was eventually determined to be an issue with recursion in the tracing code, due to some routines in the timestamping code path being instrumented. For this, I added the 'notrace' attribute to my platform-specific `sched_clock()` and all possible nested functions called by that routine.

2. Function graph tracing was implemented by doing the following:

    - Extending the ARM mcount routine in `arch/arm/kernel/entry-common.S` to check for and call a registered graph tracer function.
    - Adding a return trampoline for Ftrace for ARM.
    - Adding the return stack data structure to the task structure for processes.
    - Adding an interrupt segment to the ARM linker directive file.

      This was required because portions of the function graph display code examine routines to see whether they are interrupt functions. They do this by checking whether the function resides in the "interrupt segment." Note that I did not actually declare any routines to be interrupt routines, which is done with a qualifier on the function declaration.

3. I also modified the code to use a higher-resolution clock source for timestamps. The default clock source on my OMAP platform produced timestamps with a resolution of only 31 microseconds. This did not allow trace event times to be distinguished accurately. Luckily, there was another clock source (the `MPU_TIMER`, in my case) available that had higher resolution.

To use this clock source for trace timestamps, I modified the OMAP `sched_clock()` to use the different clock read routine for this timer.

4. I added duration filtering, using the existing `tracing_thresh` debugfs pseudo-file. The duration, calculated on function exit, was checked against this threshold and events discarded if the threshold was not met.

5. I optimized the duration filtering by adding routines to the ring buffer code to allow for discarding already-committed events. This change is discussed in the next section.

### 2.1 Optimizing the Discard of Trace Events

The function graph tracer places two events in the trace log for each function call. One event is logged for trace entry, and one for trace exit. The duration of the function is recorded in the trace exit event. In the first version of the duration filtering code, function exit events were discarded using `ring_buffer_discard_commit()`, and the function entry events were discarded using `ring_buffer_discard_event()`. `ring_buffer_discard_commit()` can usually back up the write pointer for the log, resulting in complete removal of the event from the trace buffer. However, `ring_buffer_discard_event()` just marks an entry as padding, and leaves it in the buffer.

This means that even though a trace log entry is not used in the trace output, it still occupies space in the trace log buffer, reducing the total number of events that can be held in the buffer at once.

Normally, previously committed entries in the trace log cannot be removed from the log, since subsequent entries cannot be moved to reclaim the space in the log without adding unacceptable overhead to the trace operation. So for post-commit filters, the only option is to mark the entry to be discarded as pad and leave it in the log.

However, the case of a duration filter is special, in that if a function is less than the duration threshold, all nested functions will also be less than the threshold. This means that, when using per-cpu trace buffers, and processing an exit event, if all nested function entry and exit events have been eliminated from the trace buffer, the last event in the trace buffer will always be the function entry event for the function that is currently exiting.

This observation allows for optimization of the ring buffer discard operation. If no other events follow the event to be discarded in the ring buffer, then it is possible to back up the commit and write pointers for the event (avoiding the unacceptably costly move operation), and eliminate the function entry event completely from the buffer.

I implemented a new ring buffer routine, called `ring_buffer_rewind_tail()` to do this more intelligent discard. In order to validate that `rewind_tail()` improved the length of the trace, compared to a regular `discard_event()`, I measured the amount of time I could capture in a trace, using different duration filter values with the different routines. All tests were performed with a buffer size of 1408KB.

The results of this testing are found in Table 1.

| **Discard operation** | **Duration filter value** | **Total function count** | **Time covered by trace** | **Trace event count** |
|---|---|---|---|---|
| discard_event | 0 | 3.292M | 0.39 s | 27392 |
| discard_event | 1000 | 3.310M | 1.29 s | 26630 |
| discard_event | 100000 | 3.309M | 1.34 s | 26438 |
| rewind_tail | 0 | 3.295M | 0.39 s | 27316 |
| rewind_tail | 1000 | 3.327M | 31.26 s | 35565 |
| rewind_tail | 100000 | 3.328M | †79.44 s | 1669 |

†*The test only lasted 79 seconds—extrapolating the results yields a trace coverage time of 27 minutes*

Table 1: Comparison of Discard Operations

The results clearly show the efficacy of the optimized discard operation. When function entry events were left in the trace log, the log filled up after approximately 1.3 seconds, no matter what the duration filter value was. The low value (1669) for the event count in the last row of the table indicates that the test completed before the log became full. When almost all filtered function entry events are removed from the log using the `rewind_tail()` operation, the buffer can hold almost as many events of interest as the size of the buffer allows.

## 3 Example of Use

In this section, I describe use of the function graph tracer with duration filtering. For this example, I piped data

between two Linux commands operating on file system data. The sample program is busybox, running the 'ls' and 'sed' commands, with sed executing a trivial character replacement script. This was run in a loop 10 times.

Steps:

```
$ mount debugfs -t debugfs /debug
$ cd /debug/tracing
$ cat available_tracers \
   function_graph function sched_switch nop
$ echo 0 >tracing_enabled
$ echo 1000 >tracing_thresh
$ echo function_graph >current_tracer
$ echo 1 >tracing_enabled
$ for i in `seq 1 10` ; do \
   ls /bin | sed s/a/z/g ; done
$ echo 0 >tracing_enabled
$ echo funcgraph-abstime >trace_options
$ echo funcgraph-proc >trace_options
$ cat trace
```

Figure 4 shows the first 25 lines of function graph trace output. Note that for this example I turned on the funcgraph-abstime and funcgraph-proc trace output options. Duration times for the functions are shown in units of microseconds, on the line containing the closing brace indicating the function exit.

Note that all functions in the log output took longer than 1000 microseconds to complete. Other functions which took less time than the tracing_thresh were filtered at runtime from the log.

### 3.1 Using 'ftd' to Analyze Data

To analyze system data, a post-processing tool called *ftd* was written. ftd stands for *function trace dump*, and it is a script with the capability to show call counts and cumulative time for functions in a trace log. ftd is written in Python. If you are analyzing a trace log from an embedded target, it is recommended to move your trace log to a development host and run ftd there, rather than on the target.

ftd currently requires the absolute time and process information per trace line in the trace log, in order to work properly. Make sure these display options are set before retrieving the trace log data and using ftd on the data.

To retrieve the trace log data, use:

```
$ cat trace >/tmp/trace-data.txt
```

To see a list of functions, sorted by total time spent in them, use:

```
$ ftd /tmp/trace-data.txt
```

The first 10 lines of results for this command on some sample data are shown in Figure 5.

Other useful tasks that ftd can be used for include:

- Sorting the function list by function count—the number of times that the function was called during the trace.
- Examining the *local time* of a function. The local time of a function is the elapsed time between the start and end of the function, minus the time spent in all functions called between the start and end of the function. Note that this includes not just children function called by this function, but also interrupts. Local time also includes the time spent in user space, and in other processes' kernel functions (i.e. when the function's process is scheduled out.) So local time should be interpreted cautiously, with this understanding.
- Finding the subroutines called by functions the most times.

See ftd -h for usage help documenting the command line options to use for these tasks.

## 4 Performance Measurements

The performance of various Ftrace configurations was measured, to get a sense of how much overhead tracing caused during kernel execution.

All results are for an OMAP 5912 processor running at 192 MHZ. The program I traced was a simple shell script consisting of:

```
for i in `seq 1 10` ; do
   echo $i ; find /sys >/dev/null ;
done
```

```
# tracer: function_graph
#
#      TIME       CPU  TASK/PID        DURATION                  FUNCTION CALLS
#       |          |    |    |           |   |                     |   |   |   |
  193.719625 |   0)     ls-556     |               |  sys_lstat64() {
  193.719641 |   0)     ls-556     |               |    vfs_lstat() {
  193.719650 |   0)     ls-556     |               |      vfs_fstatat() {
  193.719660 |   0)     ls-556     |               |        user_path_at() {
  193.719722 |   0)     ls-556     |               |          do_path_lookup() {
  193.719755 |   0)     ls-556     |               |            path_walk() {
  193.719777 |   0)     ls-556     |               |              __link_path_walk() {
  193.719826 |   0)     ls-556     |               |                do_lookup() {
  193.719855 |   0)     ls-556     |               |                  nfs_lookup_revalidate() {
  193.719883 |   0)     ls-556     | ! 1028.500 us |                    _text();
  193.719946 |   0)     ls-556     | ! 1189.500 us |                  }
  193.719965 |   0)     ls-556     | ! 1258.500 us |                }
  193.719986 |   0)     ls-556     | ! 1775.167 us |              }
  193.720016 |   0)     ls-556     | ! 1874.333 us |            }
  193.720045 |   0)     ls-556     | ! 2018.167 us |          }
  193.720069 |   0)     ls-556     | ! 2143.000 us |        }
  193.720099 |   0)     ls-556     | ! 2397.000 us |      }
  193.720108 |   0)     ls-556     | ! 2415.167 us |    }
  193.720139 |   0)     ls-556     | ! 2478.334 us |  }
  193.720315 |   0)     ls-556     |               |  sys_lstat64() {
  193.720337 |   0)     ls-556     |               |    vfs_lstat() {
  193.720346 |   0)     ls-556     |               |      vfs_fstatat() {
  193.720357 |   0)     ls-556     | ! 1094.500 us |        user_path_at();
  193.720410 |   0)     ls-556     | ! 1738.167 us |      }
  193.720419 |   0)     ls-556     | ! 1758.500 us |    }
  193.720452 |   0)     ls-556     | ! 1825.500 us |  }
```

Figure 4: A function graph trace, with a duration filter of 1000 microseconds

I found that this sequence was CPU-bound and spent most of its time in the kernel. Raw data is not provided here, but the results of my testing showed that the overhead for function graph tracing is quite large. My tests generated approximately 3 million kernel function calls. The overhead per call, when tracing was active, was approximately 18.9 microseconds per call. The average time to execute a kernel function call during the test was 1.7 microseconds, so this represents a significant overhead. It should be noted that function graph tracing requires 2 calls through the tracer code per function called (one each for entry and exit).

I found that the overhead per function with tracing disabled was about .3 microseconds per function. This added, on average, 19% overhead to kernel execution. The overhead for when function graph tracing was active was approximately 1100%. (That's right, over *one thousand* percent).

It should be noted that these are microbenchmarks, operating on a test designed to be kernel-function intensive, using non-blocking operations. The CPU utilization of these tests was always close to 100%. The overhead of using Ftrace on a system with a real user-space workload and real I/O would not be this high.

| Tracer Status | Elapsed Time | Function count† | Time per function | Overhead per function |
|---|---|---|---|---|
| TRACE=n | 9.25 s | 2.91M | 1.72 us | - |
| nop | 10.30 s | 2.92M | 2.05 us | 0.33 us |
| graph disabled | 19.85 s | 2.98M | 5.22 us | 3.50 us |
| graph active | 72.15 s | 3.29M | 20.61 us | 18.89 us |

†*Function counts were estimated, using data from other testing*

Table 2: Overhead of Function Graph Tracing

## 5 Future Work

The primary motivation for adding these features to Ftrace on ARM is to use them to help find problem areas in early boot. The next step in developing these features is to make it possible to use them during early kernel startup, to see which functions are taking a long time to execute, or which functions are called excessively during kernel startup.

Unfortunately, it may prove difficult to utilize Ftrace

```
Function                       Count Time         Average    Local
------------------------------ ----- ------------ ---------- ------------
schedule                          70  1353560.333  19336.576  1337519.333
pipe_wait                          1   526363.500 526363.500       56.535
preempt_schedule                 320   414278.260   1294.620     3870.986
preempt_schedule_irq              17   294134.456  17302.027      -82.004
_text                            465   278833.987    599.643   -58897.146
handle_IRQ_event                 436   239268.153    548.780    88927.501
handle_mm_fault                  396   228733.980    577.611     5986.491
local_bh_enable                 1342   220684.604    164.445    16004.635
do_DataAbort                     304   197972.822    651.226    61489.333
sys_wait4                          4   144681.433  36170.358   144681.433
```

Figure 5: Output of `ftd` command

during early boot. Some of the requirements for doing this are listed below.

Requirements for using Ftrace in early boot:

- Early clock – The tracing systems depends on the availability of a clock source for timestamps very early in the boot sequence. On many platforms (X86, MIPS, and PPC), cpu registers are available from power-on which can be used for this purpose. On ARM, clocks are not initialized until after the kernel has already started running. This would limit how early tracing could start on ARM.

- Static trace parameters – Trace parameters, such as the start location for the trace, and the duration threshold, would have to be specified at compile time to be available from the earliest kernel execution points (i.e. `start_kernel()`.)

- Static ring buffer – Possibly the most difficult problem is pre-initializing the ring buffer data structures to prepare them for receiving trace data. Other early-accessible data structures in the kernel, such as the kernel's printk log buffer, are much simpler and their initialization state can be prepared by the compiler.

Another area that should be worked on is performance. The overhead of Ftrace should be reduced. The generality of the Ftrace system and utilization of generic clock routines and ring buffer code add substantial overhead to a system that should be lightweight. Currently, Ftrace adds approximately 6 times more overhead, on the same hardware, than a function graph tracing system that the author used previously.[2].

Finally, this work should be submitted (again) to the kernel mailing list for review and consideration for mainlining. The patches for the 2.6.31-rc1 kernel and the `ftd` may currently be found at `http://elinux.org/Ftrace_Function_Graph_ARM`.

## 6 Conclusion

The Ftrace system continues to be enhanced with new features and capabilities. This new duration filtering feature should help kernel developers continue to enhance the operation of the kernel. This effort is particularly focused on finding and reducing latencies in early boot, so that the Linux kernel can continue to be improved in the area of fast booting.

[2]Kernel Function Trace—see `http://elinux.org/Kernel_Function_Trace`

# The Simple Firmware Interface

A. Leonard Brown
*Intel Open Source Technology Center*
len.brown@intel.com

## Abstract

The Simple Firmware Interface (SFI) was developed as a lightweight method for platform firmware to communicate with the Operating System.

Intel's upcoming "Moorestown" hand-held platform will be deployed using SFI.

Here we summarize the motivation for SFI, summarize the contents of the SFI specification, and detail choices made in the Linux kernel implementation.

## 1 Introduction

The SFI project home page is `http://simplefirmware.org`.

This paper starts by briefly summarizing the site's content, including the content of the SFI specification. Then we describe the implementation of SFI on Linux.

For more details, readers are encouraged to look over the specification, to read and participate on `sfi-devel@simplefirwmare.org`, and to review and suggest enhancements to the source code.

## 2 Motivation

Intel's upcoming Moorestown hand-held platform is the reason that SFI exists. However, SFI is intended to be both general and open, such that it could be re-used for other platforms.

While Moorestown contains an Intel® Atom™ processor and PCI Express®, it does not contain the legacy elements of a system that make it PC compatible, or ACPI[1] compatible.

Moorestown cannot run in ACPI mode because its chipset does not include the required ACPI hardware, and it cannot run in legacy mode because the PC-compatible elements of the system simply do not exist.

## 3 SFI vs. ACPI

System platforms are either "SFI-platforms" supporting SFI firmware tables, or "ACPI-platforms" supporting ACPI tables.

An Operating System (OS) kernel that supports SFI is an "SFI-OS." An OS that supports ACPI is an "ACPI-OS."

An SFI-platform requires an SFI-OS to boot and run optimally. An ACPI-platform requires an ACPI-OS to boot and run optimally.[2]

A single OS binary can boot and run optimally on both SFI-platforms and ACPI-platforms. It simply includes the capabilities of the SFI-OS and ACPI-OS, making an "ACPI-SFI-OS."

It is conceivable to build an ACPI-SFI-platform, and such a lab prototype is useful for testing. However, it makes little sense to ship such a system as a product. Were an ACPI-SFI-OS to boot on an ACPI-SFI-platform, the SFI-platform support would simply be ignored in favor of the ACPI-platform.

That said, SFI-platforms can provide access to selected ACPI-defined and ACPI-reserved tables. However, extending SFI with ACPI tables does not make the platform into an ACPI-platform.

[1] Advanced Configuration & Power Interface, `http://www.acpi.info`

[2] ACPI platforms can often also boot in legacy PC mode, but no known SFI platforms are able to boot in legacy PC mode.

# 4 SFI and UEFI

SFI is agnostic as to whether a platform supports UEFI[3] or not.

However, for platforms that choose not to implement UEFI, SFI does define a static "MMAP" table that returns the information defined by UEFI's GetMemoryMap() API.

# 5 SFI Tables

SFI tables are simply a data structure in memory populated by system firmware for the benefit of the OS.

## 5.1 SFI Table Header

All SFI tables share a common table header format shown in Figure 1. The format is a proper sub-set of

| Signature (4) |
|---|
| Length (4) |
| Revision (1) |
| Checksum (1) |
| OEMID (6) |
| OEM Table ID (8) |
| Table Payload<br>... |

Figure 1: SFI Common Table Format

ACPI's static table format[4] and the semantics and use of the fields in SFI is exactly the same as in ACPI.

However, even though they share a similar format, SFI table signatures are entirely independent of ACPI table signatures. Were a future version of the specifications to define a table signature used by the other, they would refer to two entirely different tables, unless explicitly defined to refer to the same table.

Today SFI's "XSDT" explicitly refers to the exact same XSDT as defined by ACPI. Indeed, the XSDT is the mechanism used by SFI to prevent name-space collisions between SFI and ACPI.

[3]UEFI, Unified Extensible Firmware Interface, `http://www.uefi.org`

[4]SFI deleted the OEM Revision, Creator ID, and Creator Revision because they had no apparent function.

## 5.2 SFI System Table (SYST)

The payload of the SFI System Table (SYST) is an array of pointers to other tables.

While the SYST must reside within a fixed memory region, using an array of pointers allows system firmware the flexibility to locate the actual tables and any convenient address.

It is not uncommon, however, for all of the tables shown in Figure 2 to reside on the same physical page of memory.

## 5.3 SFI CPUS Table

The optional CPUS table is an array of 32-bit Local APIC IDs, enumerating all the logical processors in the system.

## 5.4 SFI MMAP Table

The optional MMAP table describes the RAM present in the system. It contains memory descriptors as defined in UEFI's GetMemoryMap() API.

## 5.5 SFI (IO) APIC Table

The optional APIC table is an array of physical addresses of the IO-APICs in the system.

## 5.6 SFI FREQ Table

The optional FREQ table describes the available processor frequencies in the system, in addition to the transition latency and the actual control word used for native hardware performance-state control.

The entries in the FREQ table apply to all processors in the system. The table applies to every logical processor in the system. If there are topology dependencies between processors, the OS must discover those via native hardware methods.

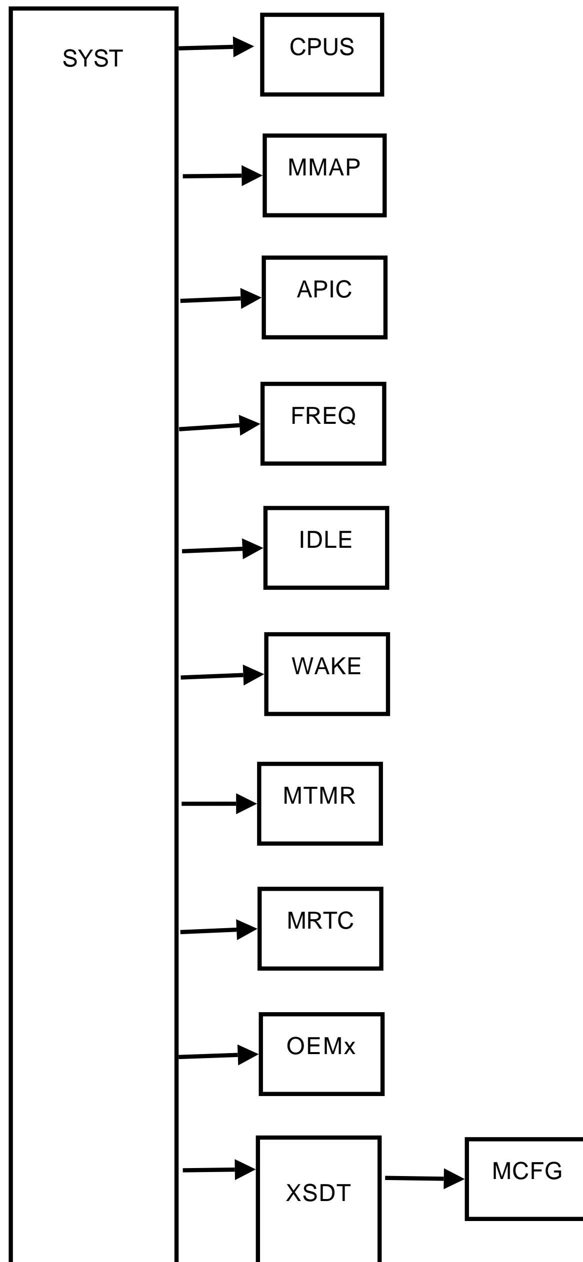



Figure 2: SFI 0.6 table structure

### 5.7 SFI IDLE Table

The optional Idle Table describes the power saving CPU idle states (e.g., ACPI C-states) available to the OS. These are accessed via the native hardware MWAIT instruction. The IDLE table also enumerates the worst-case exit-latency for each state.

The table applies to every logical processor in the system. If there are topology dependencies between processors, the OS must discover those via native hardware methods.

### 5.8 SFI WAKE Table

The optional WAKE vector table contains the 64-bit physical address of the location where the OS writes its resume vector.

### 5.9 SFI MTMR Table

The optional MTMR table describes the location, frequency, and IRQ of the platform timers present in the Moorestown chip set.

### 5.10 SFI MRTC Table

The optional MRTC table describes the location and IRQ of the real time clock present in the Moorestown chip set.

### 5.11 SFI OEMx Table

The optional OEMx table allows OEMs to define vendor-specific SFI tables while avoiding name-space collisions with other platform vendors. The OS and drivers search for tables not only on their base signature, but also using the "6-byte" OEM-id and 8-byte "OEM table id."

OEMx is intended to mean OEM1, OEM2, OEM3, etc. But the reality is that if a unique OEM-id and OEM-table-id are used in a table search, any arbitrary table signature would work. However, to avoid confusion in the table signature name-space, it is highly encouraged that the OEMx signature be used for vendor specific tables.

### 5.12 SFI XSDT Table

The optional SFI XSDT is a standard ACPI XSDT. A standard ACPI XSDT can appear in the SYST as a valid SFI table because the SFI table header is a proper subset of the ACPI table header. (SFI simply views the extra ACPI header fields as part of the table body.)

The purpose of the XSDT is to allow SFI to be extended by access to tables and table signatures defined and reserved by the ACPI specification in their standard format. It is not meant to imply that the same system should support both SFI and ACPI at the same time.

### 5.13 ACPI Tables, and the PCI MCFG

The PCI Memory Configuration Table (MCFG) is defined by the PCI Firmware Specification. It is shown in Figure 2 as an example of a standard ACPI table accessed via SFI.

## 6 Linux SFI Implementation

The SFI tables can be classified based on when in the boot process they are accessed.

### 6.1 Early Boot time

First the SYST is located in a reserved region of physical memory. The SYST must be properly aligned and must not cross a 4 KB boundary, which also puts an upper bound on its length.

Linux has several methods to discover the machine's physical memory map, including BIOS e820, UEFI, or boot parameters. If none of those are available, SFI SYST can point to an MMAP table, which must be located and parsed before the MMU is enabled.

### 6.2 Early OS Initialization

Parts of the kernel will parse SFI tables during the period after the MMU is enabled, but before the OS can set up permanent virtual mappings with `ioremap()`. During this period, the tables are temporarily mapped via `early_ioremap()` for the duration of the parsing routine.

`sfi_init()` is responsible for sanity checking all the SFI tables. It also prints out the table headers to the console.

Linux takes several steps to harden itself against firmware bugs. For a given table signature and version number, it will compare the table length to that listed in the specification before calculating the check-sum.

If any SFI tables fail to check-sum properly, SFI is disabled (and the system will likely not boot).

Linux parses the CPUS and (IO) APIC tables during this period, to enable the processors and interrupts.

### 6.3 Late OS Initialization

SFI tables can be parsed after the system is up and running and `__init` memory has been freed. Indeed, the main table parsing entry point is exported by the SFI core code such that drivers can parse SFI tables at any time.

### 6.4 Implementation Choices

In the original prototype, we copied the table headers into a static array in kernel `.data` to make scanning for table signatures fast and compact. However, at Andi Kleen's suggestion, the SFI core no longer copies any tables. Instead they are all parsed in place. The reason is that in the common case, the tables all reside on the same page of memory, so scanning the headers in-place requires no MMU operations and is thus the same speed as doing compares in a data structure optimized for that purpose. Also, most of the tables are scanned at boot and initialization time and never accessed again, so there seems little justification to keep a copy of all the headers around in kernel memory for the up-time of the system.

Of course the driver supplied parsing routine is still free to do whatever it wants with the table, including copying its data into local data structures.

Earlier we mentioned that Linux will sanity check each table signature, version, length, and compute a checksum. However, old versions of Linux must be able to handle tables that are defined by new versions of the specification. Obviously, it can not look up a future table's signature and version number to check its length. So for unknown tables, Linux uses an arbitrary 1 MB length limit before it check-sums a table.

### 6.5 Source Code

The core SFI patch is about 1,000 lines of code.

This includes the basic SFI table parsers. Drivers that consume SFI tables will provide their own table-specific parsers.

The source code is targeted to go upstream in Linux-2.6.32.

## 7 Conclusion

The Simple Firmware Interface is indeed simple.

The Linux Kernel patches to implement SFI have been public since late-June. They are currently running on Moorestown hardware, and are expected to be upstream in Linux-2.6.32.

To get involved, please go to the SFI home page, `http://simplefirmware.org`. Review the latest specification, join the mailing list, review and comment on the source code.

## 8 Acknowledgements

The author thanks Jacob Pan for prototyping the initial Linux SFI support, Feng Tang, for writing most of the final code—and testing it on pre-production hardware—Ingo Molnar, Andi Kleen, and everybody on the lists for their thoughtful code review.

# The Corosync High Performance Shared Memory IPC Reusable C Library

Steven Dake
*Red Hat, Inc.*
sdake@redhat.com

## Abstract

The Corosync coroipc reusable C libraries providing high performance client server communication are presented. The rationale for this effort is provided. An overview of the coroipc features are given. The programming API is described in enough detail to provide developers with a complete understanding of how to develop a client server application. Finally performance results are provided.

## 1 Introduction

The Corosync Cluster Engine project was created in July 2008 to address the needs of the Linux clustering community. As part of this effort, the project implemented and qualified a high performance client server interprocess communication system called coroipc.

Throughout the history of client server applications, every project implemented a unique IPC system. These IPC systems each contain a unique set of defects, performance characteristics, security model, thread safety, and portability support. After developing coroipc, the Corosync community determined coroipc could be modified to be reusable by third party client server applications.

By making coroipc reusable, coroipc enables consuming projects to focus on their strengths. Further by centralizing development effort on one IPC system, a larger community of experienced designers can provide support for that IPC system. Finally since coroipc is built into a significant portion of the Linux community's cluster infrastructure, it provides a perfect environment for ensuring the software has a sanitary design model and is defect free.

## 2 Features

### 2.1 Security

The coroipcs library provides a mechanism to ensure only users with specific user id or group id access the IPC system, and by inference, the server. This is enforced on all platforms which support the ability to retrieve the uid or gid of a connecting socket from a platform-specific system call.

### 2.2 High Performance

The coroipc client and server use almost exclusively the `mmap()` system call to map shared memory. As a result, in most cases there is no copy into the kernel, or from the kernel to userspace for communications. Notification of new messages occurs through a system V semaphore.

### 2.3 Portability

The coroipc system is dependent upon a Posix API, a coherent `mmap()` system call, and system V semaphores. Nearly all modern Posix platforms provide these features. The coroipc system has been ported and tested on Linux, BSD, Darwin, and Solaris.

### 2.4 Thread Safety

The coroipc client library is thread safe and requires no special attention by the client library users to ensure thread safety. Thread safety is implemented using reference counting on the identifier used for a client IPC connection. The reference counting critical sections are protected by spinlocks on platforms which support them, or a mutex on platforms without spinlocks.

### 2.5 Zero Copy

Clients may allocate a zero copy buffer which removes one copy from client requests. Allocating a zero copy buffer is an expensive operation and is reserved for buffers with a consistent size which are consistently reused.

### 2.6 Support for External Poll Systems

The coroipc server allows the server developer to use customized polling mechanisms. Currently there are no examples of using third party polling systems beyond the coropoll API provided with the software. We expect a glib example to be available in the community.

### 2.7 Asynchronous Client Delivery

The coroipc client blocks when the waiting for a server response. If the server takes long periods to process requests, it may prefer to issue an asynchronous response to unblock the client. The coroipc system supports the delivery of these messages through a special channel called the dispatch channel.

## 3 Architecture Overview

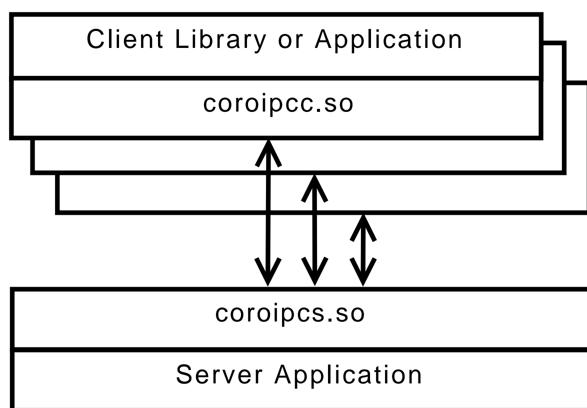



Figure 1: Example client-server application

The coroipc system is composed of two major components. The client component is composed of a client header file called coroipcc.h and client shared library called coroipcc.so. The server component is composed of a server header file called coroipcs.h and server shared library called coroipcs.so. Figure 1 an example client server application with multiple clients communicating to one server.

The global header file `coroipc_types.h` is shown in Listing 1. Every request message sent by library coroipcc clients should begin with a `coroipc_request_header_t`. The `size` parameter should be set to the size of the message and the `id` parameter should be set to the message identifier.

The server coroipcs handlers should format a message with a header of `coroipc_response_header_t`. The coroipcc clients should expect to receive a message with the `coroipc_response_header_t` header.

```
typedef struct {
        int size;
        int id;
} coroipc_request_header_t;

typedef struct {
        int size;
        int id;
        cs_error_t error;
} coroipc_response_header_t;
```

Listing 1: The coroipcc Types Definition

## 4 coroipcc

The coroipcc library provides lifecycle operations, dispatch operations, request and reply operations, and zero copy buffer operations. The full API is shown in Listing 2.

### 4.1 Lifecycle Operations

Clients connect to servers using the `coripcc_service_connect()` API. When a client connects, the client and server both `mmap()` several files into memory shared by the client and server. Finally a semaphore set is created to provide signalling between client and server of new messages.

Several files are mapped using the `mmap()` system call into the address space of both the client and server. The first of these files is the control buffer which is used internally for communication between the client and server. A unique file is also mapped for client to server requests and server to client responses. Finally an

```
extern cs_error_t
coroipcc_service_connect (const char *socket_name, unsigned int service,
        size_t request_size, size_t respnse_size, size_t dispatch_size,
        hdb_handle_t *handle);

extern cs_error_t
coroipcc_service_disconnect (hdb_handle_t handle);

extern cs_error_t
coroipcc_fd_get (hdb_handle_t handle, int *fd);

extern cs_error_t
coroipcc_dispatch_get (hdb_handle_t handle, void **buf, int timeout);

extern cs_error_t
coroipcc_dispatch_put (hdb_handle_t handle);

extern cs_error_t
coroipcc_dispatch_flow_control_get (hdb_handle_t handle,
        unsigned int *flow_control_state);

extern cs_error_t
coroipcc_msg_send_reply_receive (hdb_handle_t handle, const struct iovec *iov,
        unsigned int iov_len, void *res_msg, size_t res_len);

extern cs_error_t
coroipcc_msg_send_reply_receive_in_buf_get (hdb_handle_t handle,
        const struct iovec *iov, unsigned int iov_len, void **res_msg);

extern cs_error_t
coroipcc_msg_send_reply_receive_in_buf_put (hdb_handle_t handle);

extern cs_error_t
coroipcc_zcb_alloc (hdb_handle_t handle, void **buffer, size_t size,
        size_t header_size);

extern cs_error_t
coroipcc_zcb_free (hdb_handle_t handle, void *buffer);

extern cs_error_t
coroipcc_zcb_msg_send_reply_receive (hdb_handle_t handle, void *msg,
        void *res_msg, size_t res_len);
```

Listing 2: The coroipcc C API

asynchronous dispatch buffer is mapped twice to avoid copies during dispatch operations.

Clients may disconnect via the `coroipcc_service_disconnect()` API. A disconnect doesn't actually occur until all references of the ipc connection have been released.

### 4.2 Dispatch Operations

Client server applications may desire asynchronous communication. In coroipc, these are called dispatch operations.

To determine when a dispatch operation is available, the `poll()` system call should be used on a file descriptor obtained with `coroipcc_fd_get()`.

To retrieve the current dispatch buffer contents, the `coroipcc_dispatch_get()` API is called. The dispatch buffer is implemented internally as a circular buffer. To avoid copies, the operating system virtual memory system is used to provide a circular buffer mapping. The `coroipcc_dispatch_get()` operation pins the dispatch message. It can be then be released

with coroipcc_dispatch_put().

The design of coroipcc requires there is only one thread of execution which executes a coroipcc_dispatch_get() and coroipcc_dispatch_put() operation. This model is consistent with the user of coroipc providing an API to handle asynchronous dispatching of events and using the internal coroipcc dispatch operation functions.

The coroipcc_fd_get() API does not have to be used for those client applications which don't need to multiplex input/output operations in the client. Instead coroipcc_dispatch_get() may be used directly and will use semaphores to avoid busy spins.

### 4.3 Request and Reply Operations

The coroipcc library provides message request and reply operations to allow requests to be sent synchronously to a server and a reply to be received from the server. The common API is coroipcc_msg_send_reply_receive() which copies the response into a user supplied buffer. The remaining two APIs allow zero copy reading of the response buffer by pinning the response buffer into memory. Pinning is done via coroipcc_msg_send_reply_receive_in_buf_get() and an unpin operation occurs via coroipcc_msg_send_reply_receive_in_buf_put().

### 4.4 Zero copy buffer operations

To provide zero copy requests, the client must allocate memory in both the client and server and share it via mmap(). The client requests the server allocate this shared memory via coroipcc_zcb_alloc() and free the memory via coroipcc_zcb_free(). Since these operations are expensive, they should be rarely done and zero copy buffering should only be used on often reused buffer areas. To send a request and receive a reply, the API coroipcc_zcb_msg_send_reply_receive() is used.

## 5 coroipcs

Servers link with the coroipcs library and include the coroipcs.h file to access coroipcs services. The coroipcs library includes lifecycle operations, response operations, and integration with third party polling systems.

### 5.1 Lifecycle Operations

The coroipcs system is initialized by coroipcs_init() and exited by coroipcs_exit(). The initialization is defined by the structure coroipcs_init_state shown in Listing 3. This structure includes many user provided function parameters. These routines include scheduling policy, memory mangement, serialization, flow control, security, custom poll handler control, and functions to retrieve operations of the user service.

#### 5.1.1 Scheduling Policy

The coroipcs threads may be scheduled at Posix scheduling policies rather then the default scheduler. The policy parameter to coroipcs_init controls the policy and the sched_param parameter controls the parameters related to the policy.

#### 5.1.2 Memory Management

Many servers provide their own memory allocation. In that case, the internal use of malloc() and free() can be overridden with user defined functions.

#### 5.1.3 Serialization

If the backend function handlers are not thread safe, the user may provide a serialize_lock() function that is executed when the service function callbacks are called and serialize_unlock() function that is executed when the service function callback is done with execution. This acts to serialize input into the service so no extra mutual exclusion is needed. If high concurrency is desired, these functions can be defined to NULL and will not be used. Instead the user should provide finer grained locking within their callbacks.

#### 5.1.4 Flow Control

Two functions are provided to provide flow control into the server handler callbacks determined by the handler_fn_get() callback.

```
typedef int (*coroipcs_init_fn_lvalue) (void *conn);
typedef int (*coroipcs_exit_fn_lvalue) (void *conn);
typedef void (*coroipcs_handler_fn_lvalue) (void *conn, const void *msg);

struct coroipcs_init_state {
  const char *socket_name;
  int sched_policy;
  const struct sched_param *sched_param;
  void *(*malloc) (size_t size);
  void (*free) (void *ptr);
  void (*log_printf) ( const char *format, ...) __attribute__((format(printf, 1, 2)));
  int (*service_available) (unsigned int service);
  int (*private_data_size_get) (unsigned int service);
  int (*security_valid)(int uid, int gid);
  void (*serialize_lock)(void);
  void (*serialize_unlock)(void);
  int (*sending_allowed) (unsigned int service, unsigned int id, const void *msg,
    void *sending_allowed_private_data);
  void (*sending_allowed_release) (void *sending_allowed_private_data);
  void (*poll_accept_add) (int fd);
  void (*poll_dispatch_add) (int fd, void *context);
  void (*poll_dispatch_modify) (int fd, int events);
  void (*poll_dispatch_destroy) (int fd, void *context);
  void (*fatal_error) (const char *error_msg);
  coroipcs_init_fn_lvalue (*init_fn_get) (unsigned int service);
  coroipcs_exit_fn_lvalue (*exit_fn_get) (unsigned int service);
  coroipcs_handler_fn_lvalue (*handler_fn_get) (unsigned int service, unsigned int id);
};
```

Listing 3: The init state structure

The `sending_allowed()` function determines if an IPC message may be delivered to the server. If it returns the value 1, the coroipcs library will deliver the IPC message to the appropriate server handler.

After an IPC message is delivered, the `sending_allowed_release()` callback is executed.

It is often helpful to store some private information for these two functions to share their operating state. A 64 byte parameter `sending_allowed_private_data` is passed to both functions to store this operational state. The use of this private data is optional and invisible to the coroipcs library.

#### 5.1.5 Security

The `security_valid()` function is called by coroipcs when a new IPC connection is made to the system. The uid and gid are passed as parameters to this function. The function should return 1 if the uid or gid are valid users of the coroipcs application, otherwise it should return 0.

#### 5.1.6 Poll Handling

The `poll_dispatch_add()` call is executed when a dispatch routine is required to be added to the poll loop. The `poll_dispatch_modify()` is used modify the events on the existing file descriptor. The `poll_dispatch_destroy()` removes the fd from the polling system.

#### 5.1.7 Function Retrieval

The coroipcs system works by retrieving a function from user defined selectors and executing those functions when the appropriate action is requested by the ipc client library. The `init_fn_get()` function is called to retrieve the initialization function for the service. When the ipc connection disconnects, the `exit_fn_get()` function is called to retrieve the exit function for the ipc connection. Finally `handler_fn_get()` is used to retrieve the appropriate IPC handler.

```
extern void coroipcs_ipc_init (struct coroipcs_init_state *init_state);

extern void
*coroipcs_private_data_get (void *conn);

extern int
coroipcs_response_send (void *conn, const void *msg, size_t mlen);

extern int
coroipcs_response_iov_send (void *conn, const struct iovec *iov, unsigned int iov_len);

extern int
coroipcs_dispatch_send (void *conn, const void *msg, size_t mlen);

extern int
coroipcs_dispatch_iov_send (void *conn, const struct iovec *iov, unsigned int iov_len);

extern void
coroipcs_refcount_inc (void *conn);

extern void
coroipcs_refcount_dec (void *conn);

extern void
coroipcs_ipc_exit (void);

extern int
coroipcs_handler_accept (int fd, int revent, void *context);

extern int coroipcs_handler_dispatch (int fd, int revent, void *context);
```

Listing 4: The coroipcs API

### 5.2 Response Handling

The IPC services on delivery of a message can respond via the APIs shown in Listing 4. More specifically a regular response can be sent via `coroipcs_response_send()` or via iovectors with `coroipcs_response_iov_send()`. To send to the dispatch output channel, `coroipcs_dispatch_send()` can be used and and iovector version is also available with `coroipcs_dispatch_iov_send()`.

### 5.3 Custom Poll Handling

The init functions specified in the structure `coroipcs_init_state` for `poll_accept_add()` and `poll_dispatch_add()` should register callbacks with the poll system which then call the external coroipcs APIs `coroipcs_handler_accept()` and `coroipcs_handler_dispatch()` respectively. The purpose of the external APIs is to translate whatever API the application uses for poll into function calls the coroipcs system can understand.

## 6 Performance

The coroipcs system is designed for high concurrency operation on multiple processors. Each IPC connection is represented in the OS by a separate scheduling entity to allow multi-threaded server designs. As a result, coroipcs should better be able to utilize the operating system scheduling features to achieve higher concurrency then single threaded server applications.

The throughput in megabytes per second for message sizes ranging from 1000 to 500,000 in 1000 byte increments is shown in Figure 2. As can be seen from the Figure 2, newer processor designs have higher total throughput of up to 30 GB/sec for larger message sizes. Older processor designs reach maximum throughput of 6 GB/sec for larger message sizes. The dropoff for very large message sizes on 4 client Nehalem processors is

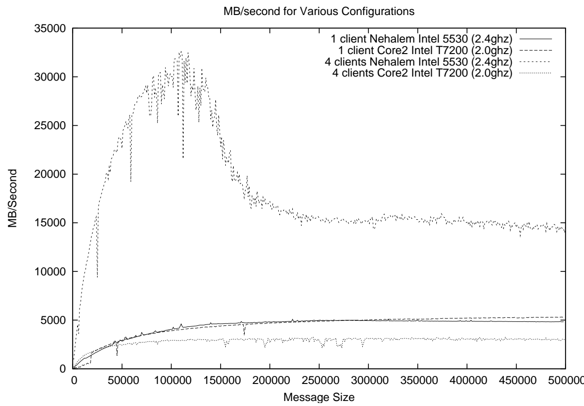



Figure 2: MB/Sec Throughput

unexplained but may be a result of cache behavior of the processor.

Transactions per second is shown in Figure 3. Nahalem with 4 clients in this graph shows very good results of one million transactions per second while a single client shows results of 100,000 transactions per second. As the size of the message increases, more time is spent within the `memcpy()` C library function resulting in lower overall transaction rates.

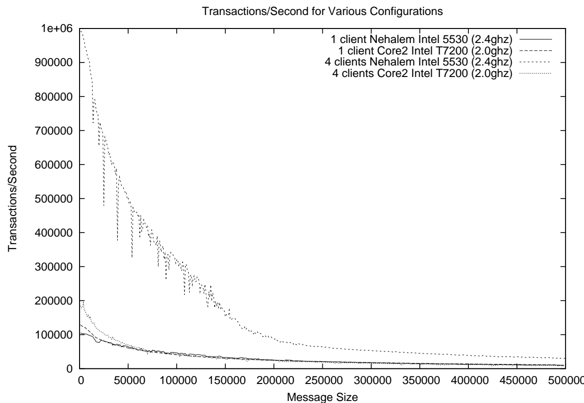



Figure 3: Transactions/Sec Throughput

## 7 Future Work

One area for future development is the tracking and notification of buffer lengths to prevent a blocked client from triggering server memory pressure. There are many choices for how this could be done and remains a further area of investigation.

Currently when a coroipcc request is made, a mutex is taken on the shared memory area responsible for requests and responses. This blocks other requests on the same handle instance from proceeding until a response is made. To improve concurrency, we plan to investigate removal of the mutex requirement by allowing multiple requests and responses to be mapped into the shared memory segment for multithreaded high concurrency applications.

Since our implementation just concluded, we have not had a thourough chance to optimize small message sizes for maximum MB/sec and transactions/sec throughput. We intend to further analyze and characterize the performance of coroipc to find hot spots within the implementation and make improvements where possible.

## 8 Conclusion

The coroipc system is a reusable C library that meets the general needs of many client server applications. It is portable to most Posix platforms, provides a sanitary security model, and is thread safe for both clients and servers. While improvements can be made with performance, the performance of the initial implementation is very good for many workload combinations. Our initial requirements of an IPC system are met satisfactorily and we expect future work to provide improved performance and usability.

# GStreamer on Texas Instruments OMAP35x Processors

Don Darling
*Texas Instruments, Inc.*
ddarling@ti.com

Chase Maupin
*Texas Instruments, Inc.*
chase.maupin@ti.com

Brijesh Singh
*Texas Instruments, Inc.*
bksingh@ti.com

## Abstract

The Texas Instruments (TI) OMAP35x applications processors are targeted for embedded applications that need laptop-like performance with low power requirements. Combined with hardware accelerators for multimedia encoding and decoding, the OMAP35x is ideal for handheld multimedia devices. For OMAP35x processors that have both an ARM®and a digital signal processor (DSP), TI has created a GStreamer plugin that enables the use of the DSP and hardware accelerators for encode and decode operations while leveraging open source elements to provide common functionality such as AVI stream demuxing.

Often in the embedded applications space there are fewer computation and memory resources available than in a typical desktop system. On ARM+DSP systems, the DSP can be used for CPU-intensive tasks such as audio and video decoding to reduce the number of cycles consumed on the ARM processor. Likewise, additional hardware accelerators such as DMA engines can be used to move data without consuming ARM cycles. This leaves the ARM available to handle other operations such as running a web browser or media player, and thus provides a more feature-rich system. This paper covers the design of the TI GStreamer plugin, considerations for using GStreamer in an embedded environment, and the community project model used in ongoing development.

## 1 GStreamer Overview

GStreamer is an open source framework that simplifies the development of multimedia applications, such as media players and capture encoders. It encapsulates existing multimedia software components, such as codecs, filters, and platform-specific I/O operations, by using a standard interface and providing a uniform framework across applications.

The modular nature of GStreamer facilitates the addition of new functionality, transparent inclusion of component advancements and allows for flexibility in application development and testing. Developers can join modular elements together in a pipeline to easily create custom workflows.

GStreamer brings a lot of value-added features to OMAP35x, including audio and video synchronization, interaction with a wide variety of open source plugins (muxers, demuxers, codecs, and filters), and the ability to play multimedia clips such as those available from YouTube. Collaboration with the GStreamer community exposes many opportunities for code reuse, which aids in the stabilization and enrichment of existing components rather than replicating existing functionality. New GStreamer features are continuously being added, and the core libraries are actively supported by participants in the GStreamer community. Additional information about the GStreamer framework is available on the GStreamer project site [3].

## 2 The TI GStreamer Plugin

One benefit of using GStreamer as a multimedia framework is that the core libraries already build and run on ARM Linux. Only a GStreamer plugin is required to enable additional OMAP35x hardware features. The TI GStreamer plugin provides elements for GStreamer pipelines that enable the use of plug-and-play DSP codecs and certain hardware-accelerated operations, such as video frame resizing and accelerated memory copy operations.

In addition to enabling OMAP35x hardware features, the following additional goals needed to be addressed when writing the TI GStreamer plugin:

- The plugin should provide a robust, portable baseline implementation that serves as a stable starting point for customer application development.

- The plugin should be easy to build and install.
- Certain performance requirements need to be met beyond the basic utilization of the DSP and hardware accelerators. More detail on performance considerations will be addressed in section 3.
- The amount of custom TI code should be kept to a minimum by using open source solutions wherever possible.
- There should not be any additional restrictions imposed by the TI GStreamer plugin on the types of pipelines created. For example, the video decode elements should be able to interface with existing video sinks—not just the video sink from our plugin. Likewise, our video sink should also accept buffers from open source ARM video decoders. All elements in the plugin should be interchangeable with ARM-side equivalents when needed.
- The open source community should be able to use the plugin, customize it to meet their needs beyond what is provided in the baseline implementation, and contribute back where it makes sense.

The TI GStreamer plugin provides baseline support for eXpressDSP™ Digital Media (xDM[1]) plug-and-play codecs and a video sink for using video drivers not supported by any open source plugin. Multiple xDM versions are supported, making it easy to migrate between codecs that conform to different versions of the xDM specification.

TI is not supporting the productization of the GStreamer plugin or GStreamer-based solutions. Complete products may require additional development for custom boards, features not specific to TI hardware (i.e., visual effects) or the implementation of applications that provide multimedia functionality through GStreamer. However, many components such as demuxers, media players, and other common features and applications are already available in various open source projects.

## 3 Considerations for Embedded Systems

When working in an embedded system there are usually fewer computation and memory resources available than in the typical desktop system. Following are some of the key resource considerations while implementing the TI GStreamer plugin.

**Limited CPU Resources:**

In an ARM+DSP system, the ARM is sufficient for running Linux, driving the peripherals and perhaps running a simple interface application or browser. However, in CPU-intensive multimedia applications that perform operations on complex media streams, the ARM is simply not powerful enough to do all of the work and still achieve real-time playback or encoding. To meet real-time requirements, the TI GStreamer plugin must utilize the DSP and other hardware accelerators to off-load the work required to process audio and video from the ARM processor.

**Memory Copies are Expensive:**

When processing audio and video it is often necessary to copy data between buffers. For example, when displaying a video frame on a display subsystem such as the frame buffer, it may be necessary to copy data into buffers provided by the device driver. Video frames can be quite large. If the normal system `memcpy` routine is used, it would create a significant load on the ARM processor since real-time playback can require 30 frames to be copied every second. Hardware acceleration for buffer copies must be used to keep the ARM load low enough to perform other tasks such as demuxing a stream or managing a media playback interface.

**Parallelizing I/O Operations:**

In an embedded system the I/O devices available are often slower than those available on a typical desktop system. Audio and video files may be stored on media such as NAND flash or SD/MMC cards. This means that the I/O wait times are longer and have more of an impact on real-time performance. On embedded systems, the I/O operations must be performed while the DSP is performing encode or decode operations to ensure the DSP always has available data.

## 4 Software Stack

Figure 1 depicts what the software stack looks like on an OMAP35x system running a GStreamer-based ap-

[1] TI's xDM specification defines a uniform set of APIs across various multimedia codecs to ease integration and ensure interoperability. xDM is built over TI's eXpress DSP Algorithm Interoperability Standard (also known as xDAIS) specification [7].

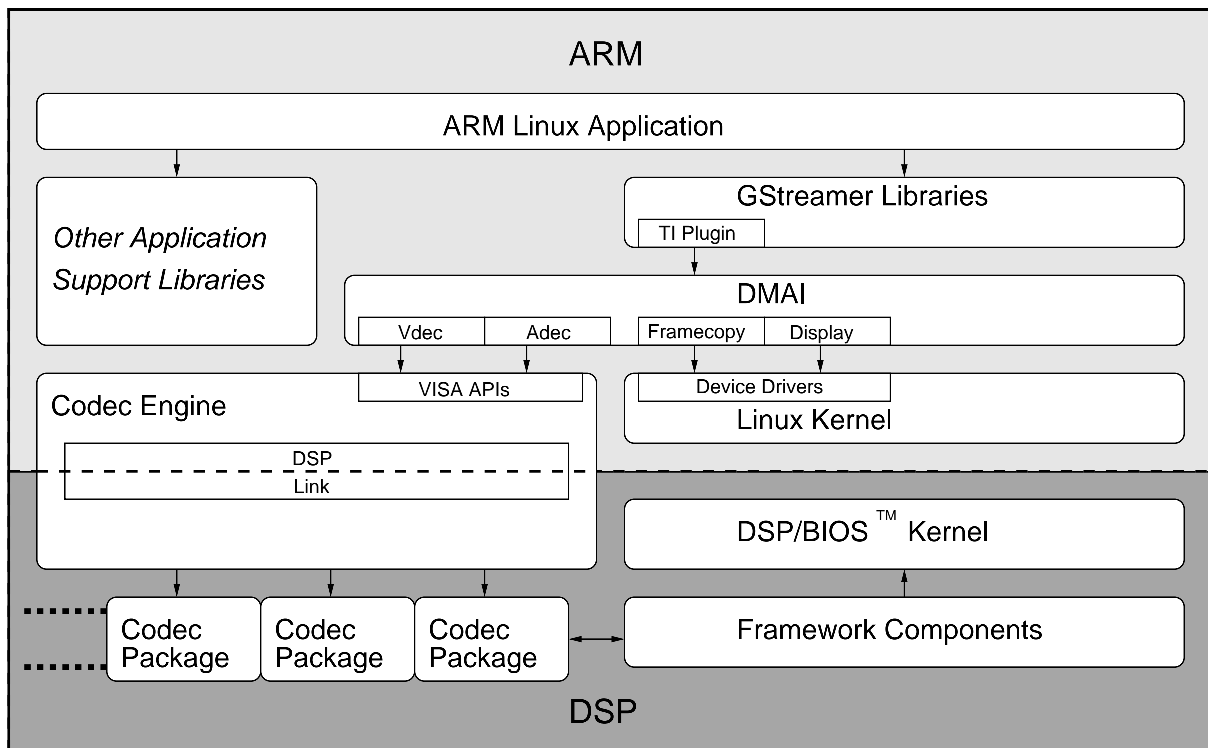



Figure 1: Software Stack for a GStreamer-based application using the TI GStreamer plugin.

plication. At the highest level there will be an ARM Linux application, such as a media player that is using the GStreamer library. At this level, developers familiar with Linux do not need to know a lot about programming for an embedded system. Other than cross-compiling their application, there are not a lot of differences between developing a GStreamer-based application on an OMAP35x and on a desktop system. The GStreamer library loads and interfaces with the TI GStreamer plugin, which handles all the details specific to the entitlement of OMAP35x hardware acceleration and use of the DSP. The core GStreamer library does not need to be aware of anything specific to the OMAP35x.

The TI GStreamer plugin interfaces with OMAP35x hardware using software components from the Digital Video Software Development Kit (DVSDK[2]). DVSDK components are all system tested for interoperability, providing a stable baseline for development. In the DVSDK software model, the DSP is mostly treated as a "black box" for running codecs—all peripherals are controlled using ARM-side Linux device drivers.

As part of the GStreamer framework, the TI GStreamer plugin also gains the ability to interface with many other open source GStreamer plugins that provide features such as:

- Demuxers for AVI, TS, and MP4 containers
- OSS and ALSA audio output
- V4L2 video capture
- ARM codecs including MP3 and AAC decoders

### 4.1 Portability and Reusability through the DaVinci™ Multimedia Application Interface (DMAI)

The most vital DVSDK component used by the TI GStreamer plugin is the DMAI [1], which enables portability to multiple TI platforms and newer DVSDK releases with minimal changes to the plugin code base. The interface with DMAI is also the boundary between the generic ARM Linux components and the DVSDK. DMAI directly and indirectly interfaces with all of the other software components of the DVSDK, providing a clean interface for interacting with hardware accelerators and DSP-side codecs. It should be noted that the

[2] DVSDK release notes and documentation are available from the TI web site [5].

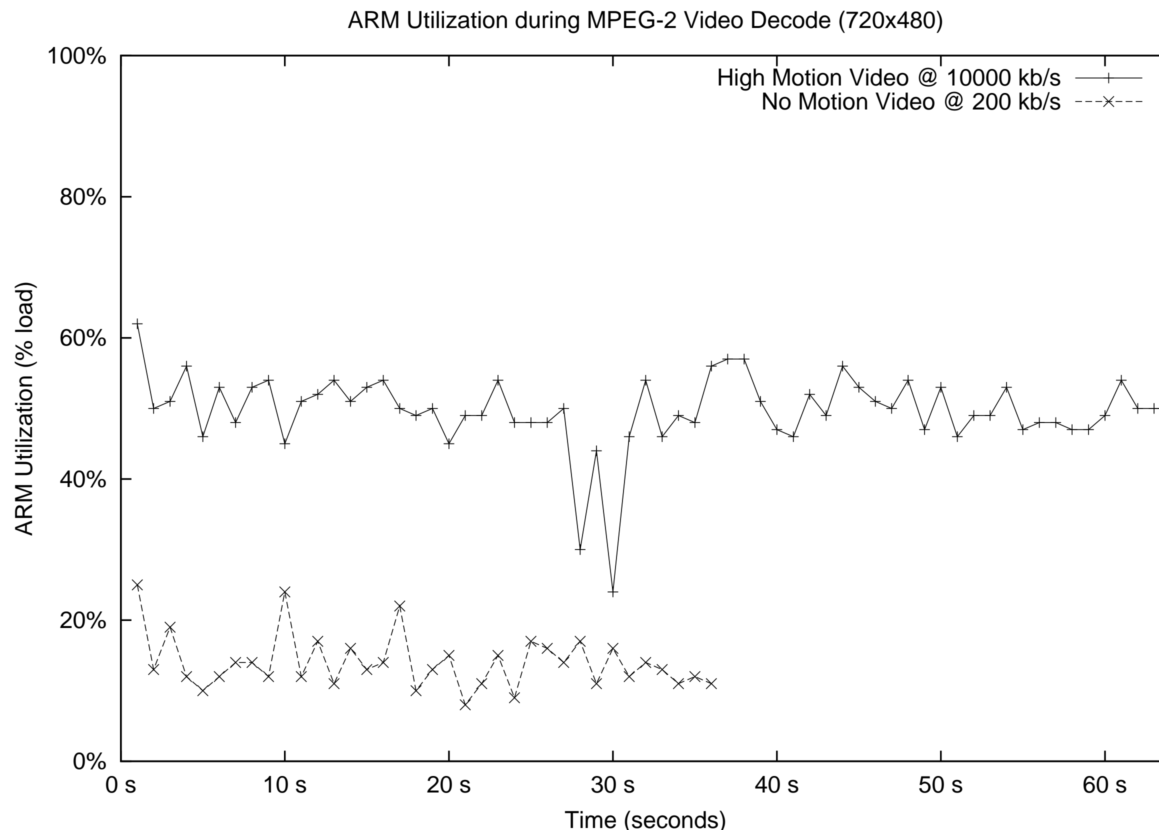



Figure 2: ARM utilization during MPEG-2 video decode.

TI GStreamer plugin also works on other TI platforms through use of the DMAI library. There is a single code base for the TI GStreamer plugin that is shared by all supported platforms.

The DMAI library provides a simple software interface but implements the many details of device driver and codec handshaking under the hood. It also provides a buffer abstraction that allows for the easy transfer of data between codecs, hardware accelerators and device drivers. Hardware acceleration is often provided without requiring developers to understand the platform-specific implementation details. For example, when using DMAI to perform a hardware-accelerated frame copy, DMAI can use a DMA operation on the OMAP35x, but will use the hardware resizer to perform a copy on platforms where a resizer could give better performance.

DMAI interfaces directly with the xDM interfaces of the available codecs and mostly abstracts out the differences between different xDM API versions. Where needed, it also abstracts out differences between device drivers and in some places, differences between kernel versions. For example, DMAI provides a display module that is configurable to use either the frame buffer or V4L2 API. The TI GStreamer plugin does not need any specialized code depending on the type of display it is using. Finally, DMAI aids in the error handling of low-level DVSDK components.

Since platform-specific code is abstracted by the DMAI library, the TI GStreamer plugin is mostly free of platform-specific code, making it extremely portable.

## 5 Performance

The graph in Figure 2 shows the ARM CPU utilization while decoding video files using the OMAP35x MPEG-2 DSP decoder. In this experiment, the decoder is run with two different NTSC-resolution video clips. The first video clip is designed to have zero-motion and a low bitrate to demonstrate the best-case ARM load. The second video clip is designed to have high-motion and a high bitrate to stress the system and demonstrate a worst-case ARM load.

### Creation of MPEG-2 Test Files

No Motion @ 200kb/s:

```
$ gst-launch videotestsrc pattern=9 num-buffers=3600 ! \
      'video/x-raw-yuv, format=(fourcc)I420, width=720, height=480' ! \
      filesink location=sample_m2v.yuv
$ ffmpeg -pix_fmt yuv420p -s 720x480 -i sample_m2v.yuv -vcodec mpeg2video \
      -b 200000 sample_staticimage.m2v
```

High Motion @ 10000kb/s:

```
$ gst-launch videotestsrc pattern=1 num-buffers=3600 ! \
    'video/x-raw-yuv, format=(fourcc)I420, width=720, height=480' ! \
    filesink location=sample_m2v.yuv
$ ffmpeg -pix_fmt yuv420p -s 720x480 -i sample_m2v.yuv -vcodec mpeg2video \
    -b 10000000 sample_snow.m2v
```

### Decode of MPEG-2 Test Files

No Motion @ 200kb/s:

```
$ gst-launch filesrc location=/mnt/sample_staticimage.m2v ! \
    TIViddec2 codecName=mpeg2dec engineName=decode ! fakesink
```

High Motion @ 10000kb/s:

```
$ gst-launch filesrc location=/mnt/sample_snow.m2v ! \
    TIViddec2 codecName=mpeg2dec engineName=decode ! fakesink
```

Figure 3: Steps to create and decode video test files for performance measurements.

Focus is put on MPEG-2 since it has a lower compression ratio than other video codecs. Since the ARM load is directly affected by the rate of data throughput, MPEG-2 is an upper-bound on the ARM load required to feed video data to the DSP codec. The same tests were performed with H.264 and MPEG-4 decoders yielding similar results at the same bitrates. Please note that H.264 and MPEG-4 exhibit better compression and their typical bitrate tends to be lower than MPEG-2.

On average, the ARM is not loaded more than 60 percent while decoding the high-bitrate video clip, and rarely went above 20 percent while decoding the low-bitrate clip. The decoders are allowed to run at maximum speed, and are not slowed down for real-time playback. This explains why the low-motion clip takes less time to decode than the high-motion clip, even though both clips are two minutes in duration. It should be noted that the video clips are read from an SD card, which contributes to part of the ARM load.

Figure 3 shows how GStreamer and FFmpeg[3] are used to create and decode the clips used for ARM load measurements.

## 6 Community Model

The TI GStreamer plugin is an open source project located at `http://gstreamer.ti.com` [4]. The project site provides a collaboration environment that includes:

- Source control using Subversion
- A wiki for documentation

[3] FFmpeg is a command-line utility for recording and converting audio and video streams. More information is available on the FFmpeg web site [2].

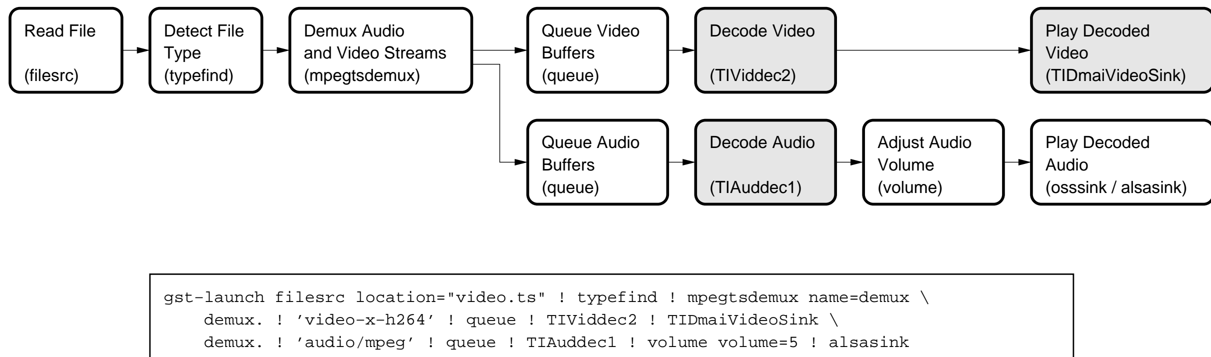



Figure 4: Example GStreamer pipeline. Shaded pipeline elements are provided by the TI GStreamer plugin.

- A package release system
- An issue and feature tracker
- Forums for support and discussion

An IRC channel (`#gst_ti`) is available on `irc.freenode.net` for developers interested in GStreamer on OMAP35x as well as other TI processors.

Anonymous access to the project and Subversion repository is supported. Account registration on the project site is optional but is needed for the submission of bug reports and patches. Developers interested in participating in the project can find answers to frequently asked questions, getting started guides and other project participation guidelines on the project site.

The TI GStreamer plugin project is community supported. TI is committed to enabling the community in their efforts to develop multimedia applications on TI processors using GStreamer. Community members are encouraged to use the forums and IRC channel to ask questions and discuss future development. For developers that want or need more support, a commercial support option is available from RidgeRun[4].

## 7 Plugin Design

Before diving into the design of the TI GStreamer plugin, an overview of the GStreamer pipeline model and how GStreamer plugins integrate into the framework should be discussed first.

[4]More information on RidgeRun is available on the RidgeRun web site [6]. Information on RidgeRun support for GStreamer is located at `http://www.ridgerun.com/products/gstreamer.shtml`.

### 7.1 The GStreamer Pipeline

A typical GStreamer pipeline starts with one or more source elements, uses zero or more filter elements and ends in a sink or multiple sinks. The example pipeline shown in Figure 4 demonstrates the demuxing and playback of a transport stream. An input file is first read using the `filesrc` element, parsed by the `typefind` element to ensure the input file is a transport stream, and then processed by the `mpegtsdemux` element, which demuxes the stream into its audio and video stream components. The video stream is sent through the `TIViddec2` element to decode the video using the DSP on the OMAP35x. Then it is finally sent to the `TIDmaiVideoSink` sink element to display the decoded video on the screen. The audio stream is processed by the `TIAuddec1` element to decode the audio on the DSP and reaches its destination at the `alsasink` or `osssink` element to play the decoded audio, depending on if the system uses an OSS sound driver or an ALSA sound driver.

Note that in the example pipeline, the TI GStreamer plugin is only contributing the `TIViddec2`, `TIAuddec1` and `TIDmaiVideoSink` elements. All other elements in the pipeline come from available open source plugins.

The main GStreamer distribution includes an application called `gst-launch`, which is a simple command line utility that allows you to construct and execute an arbitrary pipeline. It provides a flexible way to test pipelines without having to write entire GStreamer-based applications. The bottom half of Figure 4 shows the `gst-launch` command that would be used to construct and execute the pipeline shown.

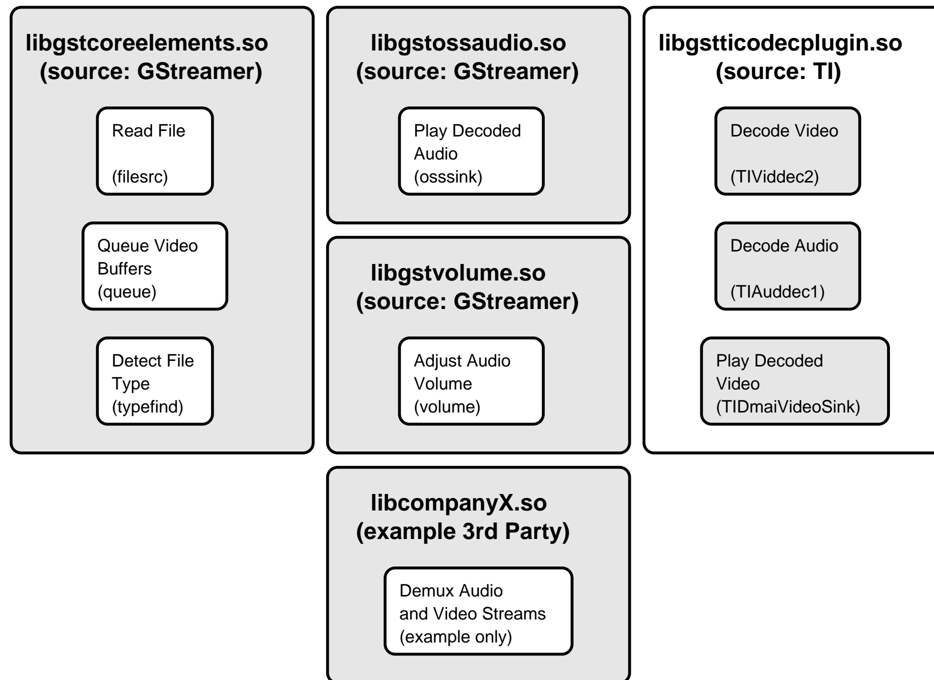



Figure 5: GStreamer pipeline elements and the the Linux shared objects that contain them. Shared object libraries may contain additional elements not shown here.

### 7.2 Shared Object Libraries

GStreamer filter elements are interchangeable, making it easy to perform different operations on a data stream. Further, GStreamer only needs to load into memory the plugins that contain elements for the desired pipeline, saving valuable system resources.

A GStreamer plugin typically maps to one or more shared object libraries on the Linux file system (see Figure 5). A single shared object library contains one or more pipeline elements. When a GStreamer-based application starts, it searches the shared object libraries for available elements. These shared object libraries can come from GStreamer itself or from other parties that provide custom GStreamer elements. The TI GStreamer plugin provides a shared object library that contains all of the pipeline elements that use the DSP and other hardware accelerators on the OMAP35x. These elements can connect and interact with pipeline elements from the main GStreamer base and from other third parties.

### 7.3 Decode Element Design

DSP decode algorithms require input buffers to be located in physically-contiguous memory and to have a full frame available for processing prior to being invoked. Physically-contiguous memory is allocated from memory regions shared by the ARM and DSP, allowing data to be passed between them without additional copy operations. However, these requirements also pose a problem as input buffers to the decode elements can be allocated from regular system memory and in some cases do not hold a complete frame. In order to use the decoder the input data must be prepared first so it is in a form usable by the DSP.

The `TIViddec2` decode element is implemented using two sub-threads (see Figure 6). The queue thread is in charge of preparing the input data for the DSP, and the decode thread invokes the DSP decoder when data is available for processing. The decode thread is a real-time thread to minimize the DSP idle time when there is data available for it to process. The queue thread copies incoming buffers into a physically-contiguous buffer for the DSP decoder. When there is enough data available to satisfy the DSP, the decode thread is signaled and DSP decoder is invoked. Since the code driving the DSP is in a separate thread, the queue thread continues to copy additional buffers into the physically-contiguous buffer while the DSP is running. When the DSP is finished, the decode thread pushes the decoded video frame to the downstream pipeline element.

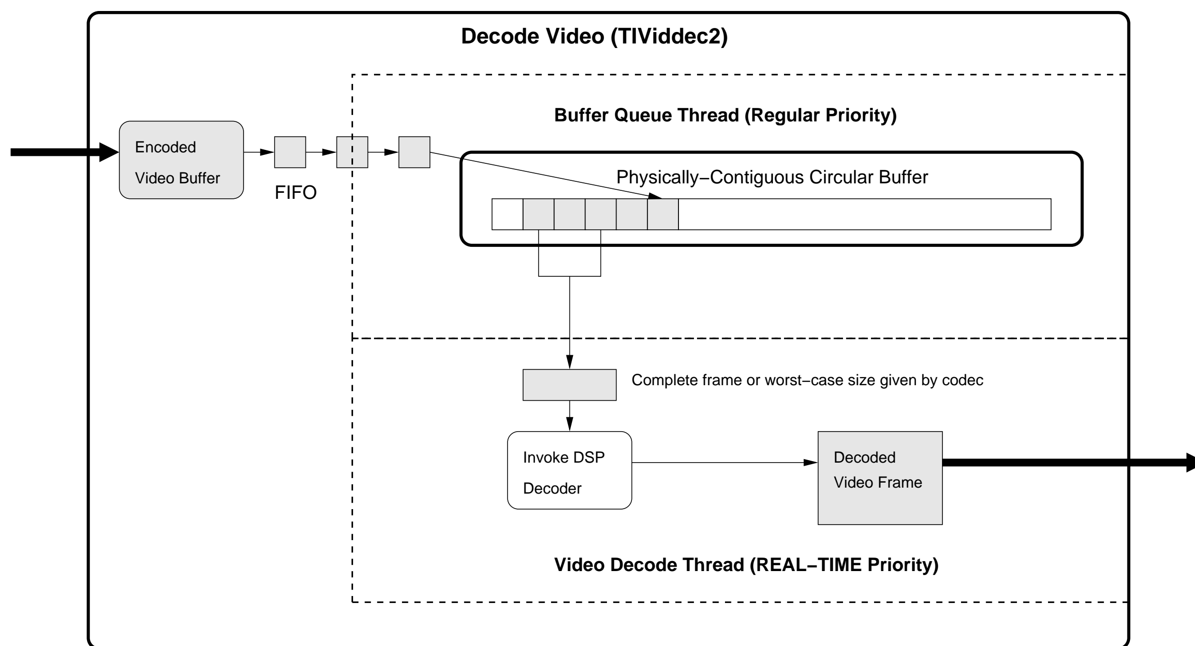



Figure 6: TIViddec2 decode element design.

In some cases it cannot be determined when enough input buffers have been received to guarantee a complete frame is available for calling the DSP decoder. In these cases, the decoder specifies the worst-case amount of data needed before it can be invoked. If more data is passed to the decoder than is actually needed, the decoder returns how much data was consumed so the next time it is invoked the remaining unprocessed data can be given again, along with additional data queued up since the last invocation. To effectively handle this scenario, the physically-contiguous buffer is managed as a circular buffer. This allows the plugin to simply pass the location where the DSP should start processing the next frame and eliminates any need to copy unprocessed data into a new buffer.

All audio, video and imaging decode elements operate in the same manner as the `TIViddec2` video decode element.

## 7.4 Encode Element Design

The design of the `TIVidenc1` encode element is very similar to the `TIViddec2` decode element, but it has one minor difference. When performing an encode operation, there is a potential opportunity for a hardware-accelerated copy of the input buffer when it is known to come from a capture source element. In this case, the input buffer is already physically-contiguous in memory, which allows a hardware-accelerated copy into the element's physically-contiguous buffer. Input buffers received from a capture source also meet the requirements to be passed directly to the DSP. However, a copy is still needed as the capture source has few resources, and there is a risk of starvation if the input buffer is not released as soon as possible.

Although a capture source will typically give a complete frame in each input buffer, the physically-contiguous buffer in the encode element is still managed as a circular buffer. This enables support for encoding a stream from a file or network source where input buffers do not contain full frames. In this case, the encode element operates in an almost identical manner to the decode element.

All video and imaging encode elements operate in the same manner as the `TIVidenc1` video encode element.

## 7.5 Video Sink Design

When the `TIDmaiVideoSink` element receives its first decoded frame, it uses the metadata in the buffer to configure the display device. Several display sink properties are also configurable by GStreamer-based applications to control the selection of frame buffer or V4L2 output, display resolution, video standard and others.

| TI GStreamer Plugin Elements | |
|---|---|
| Element Name | Description |
| TIAuddec1 | Audio decoder for xDM 1.x codecs |
| TIAuddec | Audio decoder for xDM 0.9 codecs |
| TIDmaiVideoSink | Display sink for frame buffer and V4L2 display subsystems |
| TIImgdec1 | Image decoder for xDM 1.x codecs |
| TIImgdec | Image decoder for xDM 0.9 codecs |
| TIImgenc1 | Image encoder for xDM 1.x codecs |
| TIImgenc | Image encoder for xDM 0.9 codecs |
| TIViddec2 | Video decoder for xDM 1.x codecs |
| TIViddec | Video decoder for xDM 0.9 codecs |
| TIVidenc1 | Video encoder for xDM 1.x codecs |
| TIVidenc | Video encoder for xDM 0.9 codecs |

Table 1: TI GStreamer Plugin Elements. DSP codecs currently available for OMAP35x all use the xDM 1.x specification, so a typical GStreamer pipeline on OMAP35x would use `TIViddec2`, `TIAuddec1` and `TIImgdec1` for decode and `TIVidenc1`, `TIImgenc1` for encode.

If no configuration parameters are specified to the `TIDmaiVideoSink` element, it uses reasonable defaults for OMAP35x based on recommendations from the DMAI library. It can optionally calculate the best supported resolution to fit the video clip being played.

Input buffers that come from the `TIViddec2` element can be detected by the video sink, in which case means the input buffers are physically-contiguous and hardware-acceleration is used to copy the input buffers into display buffers. If the input buffers do not come from `TIViddec2`, a regular `memcpy` is used to copy the input buffer.

## 8 Feature Summary and Future Work

A complete list of the elements provided by the TI GStreamer plugin is shown in Table 1. Support for audio encode is still missing but will be addressed in a future release. DSP codecs currently available for OMAP35x all use the xDM 1.x specification, so a typical GStreamer pipeline on OMAP35x would use `TIViddec2`, `TIAuddec1` and `TIImgdec1` for decode and `TIVidenc1`, `TIImgenc1` for encode. Elements that support xDM 0.9-based codecs are listed for completeness. Future work may eliminate the need for GStreamer-based applications to know which xDM version is used by the codecs. The TI GStreamer plugin should be up-to-date with support for new TI processors, updated DVSDK components and updated GStreamer base components. The complete list of planned work is available on the project site.

## References


[1] DaVinci™ Multimedia Application Interface. Project site: `https://gforge.ti.com/gf/project/dmai/`.

[2] FFmpeg. Project site: `http://www.ffmpeg.org/`.

[3] GStreamer Open Source Multimedia Framework. Project site: `http://gstreamer.freedesktop.org/`.

[4] GStreamer on TI DaVinci™ and OMAP™ processors. Project site: `http://gstreamer.ti.com/`.

[5] OMAP3530 Digital Video Software Development Kit (DVSDK) 3.00.00.29 Release Notes. Online documentation: `https://www-a.ti.com/downloads/sds_support/targetcontent/dvsdk/oslinux_%dvsdk/v3_00_3530/exports/omap3530_3_00_00_29_release_notes.pdf`. Free login account required for viewing.

[6] RidgeRun – Embedded Solutions. Company site: `http://www.ridgerun.com/`.

[7] Texas Instruments, Inc. *xDAIS-DM (Digital Media) User Guide*, January 2007. Literature Number: SPRUEC8B.

# From Fast to Predictably Fast

Dominic Duval
*Red Hat Inc.*
dduval@redhat.com

## Abstract

Many software applications used in finance, telecommunications, the military, and other industries have extremely demanding timing requirements. Forcing an application to wait for a few extra milliseconds can cause vast sums of money to be lost on the stock markets, important phone calls to be dropped, or an industrial welding laser to miss its target. Highly specialized realtime operating systems have historically been the only way to guarantee that timing constraints would always be respected.

Several enhancements to the Linux kernel have recently made it possible to achieve predictable, guaranteed response times. The Linux kernel is now, more than ever before, well equipped to compete with other realtime operating systems. However, applications may still need to be modified and adjusted in order to run predictably and fully benefit from these realtime extensions. This paper describes our findings, experiences and best practices in reducing latency in user-space applications. This discussion focuses on how applications can optimize the realtime extensions available in the Linux kernel, but is also relevant to any software developer who may be concerned with application response times.

## 1 Before we start

It is worth emphasizing that no special libraries or application programming interfaces are required under the realtime kernel in order to use the real time capabilities discussed in this document. The `CONFIG_PREEMPT_RT` realtime patch, as opposed to other realtime initiatives in the Linux community, follows standard Posix API's. The realtime patch only affects code in the kernel: user space applications should not notice any difference other than better determinism in how they perform. Consequently, there is no need for applications to invoke special libraries in order to benefit from the realtime kernel.

Some programming techniques, however, are known to create large sources of latencies in applications. This could cause a realtime application to lack the level of determinism that is required from it and to behave unpredictably to some degree. These techniques and habits are precisely the ones we will address in this document and for which alternative methods will be discussed.

## 2 Testing the system for realtime capabilities

Any environment from which realtime capabilities are expected should first make sure that the underlying system meets realtime requirements. This involves making sure that:

- The application may lock memory
- The scheduler API is available and the application is allowed to set the scheduler to a non-default scheduling strategy such as `SCHED_FIFO`
- The kernel was compiled with `CONFIG_PREEMPT_RT=y`
- Both user space and kernel space support robust mutexes
- High Resolution Timers are available
- Clock resolution is high enough to meet the application's requirements

All these tests could be executed by the application itself, in user space. Alternatively, a tool named `rtcheck` is also available for that purpose. `rtcheck` consists of a simple application that automatically conducts the tests listed above and returns 0 if they all succeeded. A non-zero value will be returned if any of the following tests fails:

- Memory Lock. Verify ability to lock 32K of memory in user space and ensure not limits are set by default for memory locking.

- Scheduler. Exercise the scheduler API to determine if it supports real-time; namely setting the scheduler to `SCHED_FIFO`. If it takes this setting, we know we have this capability. We can imply from this test that `SCHED_RR` can also be set.

- `CONFIG_PREEMPT_RT`. Look for the presence of `/proc/loadavgrt`. If found, we can deduce that the system is running with `CONFIG_PREEMPT_RT=y`.

- Robust Mutexes: Do lookups on a few symbols indicative of user space support of robust mutexes and then do test calls of these symbols to confirm kernel support of robust mutexes.

- High Resolution Timers. Nanosleep is timed using `clock_nanosleep()` and `clock_gettime()` calls. The value is compared against a threshold large enough to be infeasible on a system using hrtimers and small enough to be too fine-grained for a system not using hrtimers. The threshold currently being used is 20us.

- Clock Resolution. Using `clock_getres()`, make sure the clock resolution is under 200us.

## 3 Setting Scheduling Policies Right

One of the most basic attribute of a realtime operating system is the ability to run processes and threads with different realtime policies and priorities. This mechanism is strictly enforced by the scheduler of a realtime kernel. Correctly setting priorities for all threads of an application involves first of all evaluating what parts of the application need to behave in a realtime way and which don't have to. Tasks that are best left to regular (i.e., `SCHED_OTHER`) or low priority threads include:

- Interaction with devices for which the speed is considered unpredictable (i.e., storage devices)

- Dynamic memory allocation

- Filesystem operations

- Logging

It does, however, make sense to set a realtime priority for any other task that requires predictability. In this case, the first step involves setting the scheduling policy using the `sched_setscheduler()` system call. We typically need to assign realtime threads the `SCHED_FIFO` or `SCHED_RR` policy here. Threads running with realtime scheduling policies (also called scheduling classes) are fairly different from others running under the regular `SCHED_OTHER` policy:

- They always preempt regular threads (assuming they're ready to run and not blocked).

- They do not expire. There's no such thing as a time slice for real time processes (unless two or more processes rely on `SCHED_RR`)

- They will completely starve other threads of lower priority, realtime or not, if they don't go to sleep.

### 3.1 Setting priorities right

The last parameter in the `sched_setscheduler()` system call consists in a priority ranging from 0 to 99, 99 being the highest possible priority. This value should be the result of careful analysis of your realtime application and how threads interact. Typical priorities look as follows:

| Real Time Priorities | |
|---|---|
| 99 | Watchdog and migration threads |
| 90-98 | High priority realtime application threads |
| 81-89 | IRQ threads |
| 80 | NFS |
| 70-79 | Soft IRQs |
| 2-69 | Regular applications |
| 1 | Low priority kernel threads |

It is worth emphasizing from the example above that interrupts have been assigned priorities (usually around 85) to handle work resulting from the use of a network interface, for instance. One can always reorder those priorities. To make sure network traffic will get prioritized at the driver level, for example, it would be possible to adjust the realtime priority of the corresponding IRQ thread in such a way that it would preempt other high priority realtime threads. This scheme provide complete control over what should be executed first.

Setting the scheduling policy and priority involves invoking the `sched_setscheduler()` call like this:

```
struct sched_param sp;
int policy = SCHED_FIFO
sp.sched_priority = 70;
if (sched_setscheduler(0,policy,&sp)) {
  perror("Could not set policy and priority");
  exit(1);
}
```

Alternatively, a tool such as `rtctl` can be used to set priorities manually at runtime or automatically when the system boots up.

## 4 Avoiding Page Faults

Performance associated to memory access can be affected by two factors: pages of memory being swapped out to disk and memory allocation. By default, Linux does not provide any guarantees about whether memory allocated by an application will remain in physical memory or get swapped out. Memory that ends up on the swap device may eventually need to be copied back to physical memory in order to be accessed, thus causing a major page fault. This is an extremely large source of latency in applications since that delay depends essentially on the speed of the storage device, which is usually a few orders of magnitude slower than physical memory accesses (milliseconds vs nanoseconds).

Likewise, Linux does not guarantee that memory actually gets allocated at all in physical RAM. In fact, memory allocated on the stack or via functions such as `malloc()` is usually not immediately allocated in RAM: pages of physical memory can be allocated on the fly whenever the application writes data to that memory. This mechanism relies on minor page faults. While minor faults can be dealt with relatively quickly, they can also introduce delays in the application.

In order to address these sources of latency we recommend making use of the `mlockall()` system call, which ensures that memory allocated on behalf of a process never causes a page fault. `mlockall()` supports two flags:

- `MCL_CURRENT`: Lock all pages which are currently mapped into the address space of the process.
- `MCL_FUTURE`: Lock all pages which will become mapped into the address space of the process in the future. These could be for instance new pages required by a growing heap and stack as well as new memory mapped files or shared memory regions.

For most applications both flags are needed: this will make sure no present or future allocated memory areas cause any major page faults, i.e.:

```
if (mlockall(MCL_CURRENT|MCL_FUTURE) == -1) {
  perror(Could not lock memory.'');
  exit(1);
}
```

Make sure you have an idea what the memory consumption of your application should look like before you use `mlockall()`. All pages of memory belonging to the process making a call to `mlockall()` will be locked in physical memory. That includes code, data and library contents. Locked memory can be unlocked if needed using the `munlockall()` system call.

For large applications or systems with limited resources, this system call can easily cause memory starvation issues on the entire system. As an alternative to `mlockall()`, applications that require more flexibility may rely on the `mlock()` system call. In this case, specific memory regions may be locked and unlocked later on with `munlock()`.

Furthermore, if the application does not get invoked by the superuser you will need to set the appropriate process capability, i.e., `CAP_IPC_LOCK`. If set, no limits will be placed on the amount of memory the process can lock. Similarly, for unprivileged processes the `RLIMIT_MEMLOCK` limit can be set instead in order to define the maximum amount of memory that can be locked.

You may verify default settings by running the `rtcheck` utility discussed previously in this document or by invoking

```
$ ulimit -l
```

Modifying the `RLIMIT_MEMLOCK` value typically involves assigning the user running the application to the `realtime` group. Limits listed under `/etc/security/limits.d/realtime.conf` will then automatically be applied next time this user logs in. Such a configuration file may look as follows:

```
@realtime   soft  cpu      unlimited
@realtime   -     rtprio   100
@realtime   -     nice     40
@realtime   -     memlock  unlimited
```

Similarly, limits may be set programmatically by the superuser with the `setrlimit()` system call. In the following example we're setting limits for the current process to unlimited:

```
#include <sys/resource.h>
struct rlimit rlim = {RLIM_INFINITY,
                      RLIM_INFINITY};
setrlimit(RLIMIT_MEMLOCK, rlim);
```

### 4.1 Pre-faulting the stack

The stack of a process is another potential source of page faults to be avoided: whenever a local variable gets defined or a function gets executed, the stack grows by a few extra bytes. At some point (i.e., when we run out of physical memory in a stack page) this will trigger a minor page fault and allocate one more page of RAM. This can obviously cause some latency that's not desirable in the context of a realtime application. As a solution, we recommend pre-faulting the stack when the application or the thread get started. This can be done as follows:

```
#define MAX_STACK_SIZE (128*1024)
void prefaulter(void) {
        unsigned char dummy[MAX_STACK_SIZE];
        memset(&dummy, 0, MAX_STACK_SIZE);
}
```

Note that the main challenge here is to determine ahead of time what the maximum stack usage in the application will be. A practical way to determine the stack size is to pre-fault the entire stack to a known value at startup time and determine, when the application gets terminated, which pages of the stack got modified. This iterative technique is not without its limits but the result can at least be used as a starting value in the example listed above.

### 4.2 Working around malloc's unpredictability

glibc's `malloc` function comes with a default set of tunings that works well in the common (non realtime) case, but lacks the predictability expected from realtime applications. Fortunately, a function known as `mallopt()` makes it possible to eliminate many issues associated with `malloc()`.

Calls to `malloc` and `free` are not always necessarily translated to a corresponding call to `sbrk()`. In fact, `malloc()` usually makes use of a set of preallocated subheaps in order to reduce memory fragmentation and speed up memory allocation. This also means that any application calling `free()` will use `sbrk()` to give memory back to the system. We obviously don't want that to happen in the context of a realtime application. The `mallopt()` function, in conjunction with the `M_TRIM_THRESHOLD` flag lets us modify this behavior and completely disable the memory reclaim:

```
if (!mallopt(M_TRIM_THRESHOLD, -1) {
  return 1;
}
```

`malloc` uses `mmap()` by default in order to allocate any block of memory larger than 128k. This can create undesirable situations where a call to `free` will result in the `munmap` system call being invoked, thus giving back locked pages to the kernel. Since we want to keep those locked pages in our address space, we need to disable the use of `mmap()` in `malloc()` context:

```
if (!mallopt(M_MMAP_MAX, 0)) {
  return 1;
}
```

### 4.3 Real time dynamic memory allocator

As we have just seen, allocating memory dynamically is generally considered incompatible with the requirements of a realtime system. Memory locking is widely viewed as the only alternative to this problem.

There are cases, however, where memory locking just is not practical. There are applications for which we just cannot predict the memory footprint. We must thus come up with a solution that makes it possible to rely on a dynamic allocator that offer guaranteed response times. The Two-Level Segregate Fit (TLSF) allocator is a constant cost memory allocator that can be used in these cases. While it cannot avoid major page faults (disabling swap may thus be required), it does provide guarantees about the time required to allocate any amount of memory: TLSF operates with O(1), i.e. constant, cost.

### 4.4 Dynamic library preloading

Linux uses a technique known as "lazy linking" in order to load libraries into memory at execution time. This

```
static bool dumpPageFaults(void) {
  bool l_PageFaultsDetected = false;
  static bool ls_Init = false;
  static struct rusage Previous;
  struct rusage Current;

  getrusage(RUSAGE_SELF, &Current);
  int a_NewMinorPageFaults = Current.ru_minflt - Previous.ru_minflt;
  int a_NewMajorPageFaults = Current.ru_majflt - Previous.ru_majflt;
  Previous.ru_minflt = Current.ru_minflt;
  Previous.ru_majflt = Current.ru_majflt;

  if (ls_Init) {
    if ((a_NewMinorPageFaults > 0) || (a_NewMajorPageFaults > 0)) {
      printf("New minor/major page faults: %d/\d{\n",
              a_NewMinorPageFaults,
              a_NewMajorPageFaults);
      l_PageFaultsDetected = true;
    }
  }
  ls_Init = true;
  return l_PageFaultsDetected;
}
```

Figure 1: An example of how to monitor page faults.

method essentially means that a library to which an executable is linked will not be loaded in RAM unless a call is made to a component of that library. Page faults can therefore be avoided in the common case where no calls are made to some parts of the library and, most importantly, physical memory usage can be minimized.

Applications that link and use libraries will likely cause page faults. As we have seen in the last section, page faults are detrimental to the application's responsiveness and need to be avoided in realtime environments. This means we need a way to load libraries in advance, i.e., whenever the application gets started.

Lazy linking can be controlled by the environment variable `LD_BIND_NOW`. The loader reads this variable in order to determine whether it should load libraries in memory right now (`LD_BIND_NOW`) or when it will actually be invoked (also known as lazy linking). Setting this environment variable can be done on the command line with:

```
$ export LD_BIND_NOW=1
```

Alternatively, lazy linking can be configured on the executable file itself with the `-z` linker option at compile time:

```
$ gcc app.c -o app -Wl,-z,lazy
```

As a result, all dynamic libraries will be automatically loaded when the application gets invoked.

### 4.5 Monitoring page faults

As we have seen in the previous sections, page faults are generally one of the largest sources of latency in realtime applications. Minimizing or eliminating them will help you achieve better latency results. In order to validate the strategies documented in this whitepaper we recommend using the `getrusage()` function defined in `resource.h`:

```
int getrusage(int who, struct rusage
*usage);
```

The first parameter is generally `RUSAGE_SELF`: this will let you access statistics for the current process, which includes all its threads but excludes any child process that may have been created before. The second argument is a reference to a data structure that contains a number of statistics related to the current process:

```
struct rusage {
 [...]
 long   ru_minflt;
 long   ru_majflt;
 [...]
};
```

What we really want to focus on here are the `ru_minflt` and `ru_majflt` fields, which report the number of minor and major faults since the process started, respectively. An increasing number of minor and major faults in a realtime application should be a concern only after the application has been fully initialized: at that point very few operations should generate faults.

As an example, an application can keep track and report page fault statistics by implementing something similar to what's listed in Figure 1 above.

## 5 Efficient Locking

The realtime patch is known to expose lock-related bugs more easily due to the fine grained nature of the realtime kernel. These bugs are not necessarily new in user space applications: they were typically just left unnoticed due to the less dynamic behavior of the standard kernel (this is particularly true for non-SMP systems). Solving locking issues should first involve analyzing which locks are used in the application and ensuring that lock contention won't everbecome a problem as the application scales.

Other steps can also be taken in order to improve lock efficiency in a user space application running on the realtime kernel:

- Never use SysV semaphores in order to protect shared data resources in your application. These mechanisms are not designed and implemented to support priority inheritance, which is a requirement for any realtime system. Pthread mutexes should be used instead.
- Pthread mutexes should be given the `PTHREAD_PRIO_INHERIT` attribute in order to turn priority inheritance on.
- Rely on mutex priority ceilings if priority inheritance can't be used.
- If a counting (i.e., non-binary) semaphore is required in the application, implement it using condition variables.

## 6 Priority Inversion

Extra latency can occur when threads with different priorities share a common resource that's protected by a lock. Priority inversion happens when a high priority thread tries to obtain exclusive access to a resource that's already locked by a lower priority thread. This phenomenon is expected in threaded applications and cannot be avoided in most cases.

A worst scenario (defined as unbounded priority inversion) would consist of three threads A (high priority), B (medium priotiy) and C (low priority), all relying on a common resource. Imagine a situation where thread C starts first and locks the resource. Thread A (which is of higher priority) then wakes up, preempts thread C and tries to access the common resource. Since the resource is already locked, thread A will go to sleep. If thread B wakes up at this point it will delay execution of thread C since it runs with a higher priority. End result is that thread C is delaying thread A (which does not make sense priority-wise) and thread B is not giving thread C a chance to complete its work. This unbounded priority inversion situation will persist as long as thread B keeps executing, which has some obvious latency effects on thread A.

Two mechanisms are available to help with this situation:

- Priority inheritance
- Priority ceilings

### 6.1 Priority Inheritance

With priority inheritance, a low priority thread that holds a mutex can automatically have its priority increased in order to complete its work faster and let a higher priority thread gain access to the lock more quickly. End result is better response time for high priority threads and better predictability.

It is worth noting here that recompiling the application may be required in order to benefit from priority inheritance. Since this feature depends on a synchronization mechanism called *pi futexes*, applications that were compiled under an older glibc release will not get access to the corresponding feature.

### 6.2 Priority Ceilings

The validity of priority inheritance as a way to mitigate problems associated with priority inversion has historically been a debate in the realtime systems community. Another way to address this unbounded priority inversion problem is to design the application in such a way that low priority threads get a temporary boost when they acquire a resource. This makes it impossible for other threads to preempt the one that's currently holding the resource. Moreover, this can minimize the time spent while the resource is locked since the thread that holds it can now run at a higher priority. This requires use of some functions available in the pthread interface:

```
#include <pthread.h>

pthread_mutexattr_setprotocol(&attr,
 PTHREAD_PRIO_PROTECT)
pthread_mutexattr_setprioceiling(&attr,
 PTHREAD_PRIO_PROTECT)
```

## 7 Scheduling

The `sched_yield()` system call has historically been very popular with developers preoccupied with application latency. This function used to make it possible for a process to give up its share of CPU time and let the scheduler determine what should be executed next. The idea was that without calls to `sched_yield()`, the process would consume its CPU time slice and eventually get replaced with another process. This would obviously takes a relatively long period of time to complete and the process would be ready to run again after `sched_yield()` would return.

There are challenges associated with `sched_yield()` usage: this system call gives very little visibility to the scheduler about what the application is expecting to do next. Moreover, POSIX is extremely vague about what should happen with the calling process.

Use of the `sched_yield()` system call is now discouraged under the real time kernel. The new Completely Fair Share (CFS) scheduler was designed in such a way that `sched_yield()` should not be needed in any case. In fact, `sched_yield()` now accomplishes essentially nothing if it's called from a realtime process. Moreover, based on our experience we can conclude that in most cases we can actually re-architect applications that depend on `sched_yield()` in such a way that `pthread_mutex_*` calls or condition variables can replace it. These methods work much more nicely with the realtime kernel, and provide more visibility to the scheduler. It can thus make better decisions.

Under normal circumstances, on a system running the realtime kernel, using `sched_yield()` should not result in better results. For instance, in the case of two CPU-bound programs we extracted the ps output below. We can see that the two processes were allocated the same CPU time even though one of them, `loop_yield`, was making `sched_yield()` calls in every loop:

```
  PID USER PR  NI  S %CPU TIME+   COMMAND
15021 dd   20   0  R 50.1 1:42.33 with_yield
15022 dd   20   0  R 50.1 1:42.96 without_yield
```

There are very rare cases where application developers might actually need true `sched_yield()` capabilities. It is possible, with the realtime kernel, to turn `sched_yield()` into a more aggressive mode that will move the process that makes the call at the very end of the rbtree. This behavior is close to what is available under the standard kernel. With this sysctl turned on (`kernel.sched_compat_yield=1`), a process such as `loop_yield` would end up never getting invoked on the CPU if it was competing against another process that kept the CPU busy with no `sched_yield()` calls:

```
  PID USER PR  NI  S %CPU  COMMAND
15044 dd   20   0  R 99.5  0:25.94 without_yield
15046 dd   20   0  R  0.3  0:00.06 with_yield
```

## 8 Signals

Relying on signals to execute specific code in a realtime application is generally considered a bad idea. The code paths involved in the Linux kernel vary depending on the interface used: it is thus very hard to make this operation deterministic.

A better alternative to signals is to use POSIX Threads to distribute the workload and use condition variables, mutexes or barriers to let them communicate together. This method also provides much greater visibility to the scheduler, which can then make the right decisions to prioritize tasks.

If eliminating signals is not possible at all, we suggest creating one or multiple threads which will be blocking on the `sigwait()` system call. Assuming all other threads in the application have blocked signals, the signal handler thread will be the only one processing the corresponding code. This can lead to better signal response time and will make the entire application more deterministic by eliminating interruptions due to signals.

## 9 ioctls

Some applications make heavy use of the `ioctl()` system call. This is generally used to acquire control data or statistics from hardware devices. One aspect about `ioctl()` is of critical importance for realtime applications: the code executed in the kernel is invoked by default while the Big Kernel Locki (BKL) is held. The BKL is required in a number of different contexts in the kernel. Executing the `ioctl()` system call can end up delaying other threads or processes, and ultimately produce a large source of latency in the application.

A special unlocked version of the `ioctl()` system call may be provided by some drivers. Whenever that is true, any `ioctl()` call will automatically rely on the unlocked version. For that reason, unlocked `ioctl`s are generally considered safe for use in realtime environments. Do keep in mind, however, that the `ioctl` behavior (and the amount of time it takes to return) is ultimately dependent on the driver implementation, where some amount of locking can also happen and where latency may be introduced.

Developers of applications depending on the `ioctl()` system call can address latency issues by first assessing the delay caused by `ioctl()` in the application itself. If that delay is not acceptable, we recommend:

1. Verifying if a version of the driver contains a BKL-free `ioctl` implementation. If so, use it!
2. Wrap any `ioctl()` invocation around a low-priority thread of execution that will not affect the latency-sensitive parts of the application.

## 10 References

Arnaldo Carvalho de Melo. *Earthquaky kernel interfaces.* `http://vger.kernel.org/~acme/unbehaved.txt`. 2008.

Bill O. Gallmeister. *POSIX.4 Programmers Guide - Programming for the Real World.* O'Reilly and Associates, 1995.

*The GNU C Library - Memory Allocation*
`http://www.gnu.org/s/libc/manual/html_node/Memory-Allocation.html`

Philippe Gerum, Karim Yaghmour, Jon Masters, Gilad Ben-Yossef. *Building Embedded Linux Systems.* O'Reilly, 2008.

*Real-Time Linux Wiki*
`http://rt.wiki.kernel.org`

*Red Hat Enterprise Linux Real Time Wiki*
`http://rt.et.redhat.com`

*rtcheck*
`ftp://ftp.redhat.com/pub/redhat/linux/enterprise/5Server/en/RHEMRG/SRPMS/rtcheck-0.7.4-2.el5rt.src.rpm`

*TLSF: Memory Allocator for Real-Time*
`http://rtportal.upv.es/rtmalloc/`

# Combined Tracing of the Kernel and Applications with LTTng

Pierre-Marc Fournier
*École Polytechnique de Montréal*
pierre-marc.fournier@polymtl.ca

Mathieu Desnoyers
*École Polytechnique de Montréal*
mathieu.desnoyers@polymtl.ca

Michel R. Dagenais
*École Polytechnique de Montréal*
michel.dagenais@polymtl.ca

## Abstract

Increasingly complex systems are being developed and put in production. Developers therefore face increasingly complex bugs. Kernel tracing provides an effective way of understanding system behavior and debugging many types of problems in the kernel and in userspace applications. In some cases, tracing events that occur in application code can further help by providing access to application activity unknown to the kernel.

LTTng now provides a way of tracing simultaneously the kernel as well as the applications of a system. The kernel instrumentation and event collection facilities were ported to userspace. This paper describes the architecture of the new LTTng Userspace Tracer and how it can be used in combination with the kernel tracer. Results of some early performance tests are also presented.

## 1 Introduction

Technologies such as multi-core, clusters, and embedded systems are used to build increasingly complex systems, which results in developers facing increasingly complex bugs. These bugs may, for example, occur only in production, disappear when probed, occur rarely, or have a slowdown of the system as the only symptom.These characteristics make traditional debugging tools ineffective against them. New debugging tools are therefore required.

The impact of these tools on system performance must be as small as possible, so they can run on systems in production, whose hardware is chosen to match the production load (and not debugging tools), or on which adding debugging tools may render certain bugs unreproducible.

Kernel tracing is one of these tools. It may be used to understand a great variety of bugs. Quite often, the kernel is aware of all the important activities of an application, because they involve system calls or traps. In certain cases however, kernel tracing is not sufficient. For example, the execution of applications that process a large number of requests or that have a large number of threads may be more difficult to follow from a kernel perspective. For this reason, applications need to be traceable too. It is moreover highly desirable that userspace trace events be correlatable with kernel events during the analysis phase.

LTTng [3], while providing a highly efficient kernel tracer, lacks a userspace tracer of equal performance. In this paper, the LTTng Userspace Tracer, a work in progress to fill this gap, is described. In the next sections, its architecture is presented. Afterwards, performance tests are discussed, followed by proposals for future work.

## 2 Related Work

The classic `strace` tool provides a primitive form of userspace tracing. It reports system calls and signals in a process. Unfortunately, its usage induces a significant performance penalty. It is moreover limited to tracing system calls and signals, both of which are nowadays obtainable at a much lower cost through kernel tracing.

DTrace has statically defined tracing (SDT) that can be used inside userspace applications[6]. This implementation uses special support inside the runtime linker. Upon activation of an instrumentation point, NOP instructions

placed by the linker are replaced by an instruction that provokes a trap. Probes are limited to 4 arguments.

SystemTap has an implementation of SDT[2] that seems to be very similar to that of DTrace.

LTTng has offered several different userspace tracing technologies over the years. The first is called "system call assisted" userspace tracing. It declares two new system calls. The first is used to register an event type; it returns an event ID. The second is used to record an event; it requires an event ID and a payload to be passed as arguments.

The second, called "companion process" userspace tracing, requires no kernel support. Processes write their events in buffers in their own address space. Each thread had a "companion" process, created by the tracing library, that shares the buffers (through a shared memory map). The companion consumes the buffers using a lockless algorithm.

After some refactoring of the LTTng core, these two approaches were dropped. Eventually, a quick replacement was devised, which consists in a simple system call taking a string as argument. Calling it produces an event whose argument is the string. The event always has the same name; an indication of the application generating the event needs to be prepended to the string.

Eventually, the feature was moved from a system call to a file in DebugFS (`/debug/ltt/write_event`). Writing a string to this file generates an event called `userspace_event` whose argument is the string.

Ftrace[7], another kernel tracer, has a similar feature using a file called `trace_marker`.

## 3 Architecture

The LTTng Userspace Tracer (UST) is a port of the LTTng static kernel tracer to userspace. This section describes the architecture of the UST, insisting on the particularities of the userspace implementation. Figure 1 shows an overview of the UST architecture.

Here are some of the important design goals of the UST, that influenced its architecture.

- It should be completely independent from the kernel tracer. Kernel and userspace traces should be correlated at analysis time.
- It should be completely reentrant, supporting multi-threaded applications and tracing of events in signal handlers.
- There should be no system call in the fast path.
- The trace data should never be copied.
- It should be possible to trace code in shared libraries as well as the executable.
- The instrumentation points should support an unlimited number of arguments.[1]
- No special support from the linker or compiler should be required.
- The trace format should be compact.

### 3.1 Tracing Library

Programs that must be traced are linked with the tracing library (`libust`). They must also be linked with the Userspace Read-Copy-Update library (`liburcu`)[4], which is used for lockless manipulation of data structures. They must finally be linked with `libkcompat`[5], a library that provides a userspace version of some APIs available in the Linux kernel (atomic operations, linked list manipulation, kref-style mechanism, and others).

### 3.2 Time

There are no dependencies between the kernel and userspace tracers of LTTng. However, in order to do a combined analysis of a kernel trace and of userspace traces, timestamps of all traces must be coherent (e.g. they must come from the same time source).

An appropriate tracing time source must have a high resolution in addition to being coherent across cores. The cost of accessing this time source must be low in kernel space, but also in userspace (making a system call is too costly). Work is needed in the Linux kernel to make such a time source with all these characteristics available in all architectures.

The UST code currently works only on x86 (32 and 64 bits). Until a suitable time source is provided by the kernel, the TSC is read directly with the `rdtsc` instruction. This is the same time source used by the kernel tracer. It is quick to read and synchronized across cores in most variants of the architecture.

[1]The only constraint is that an event must fit in a sub-buffer.

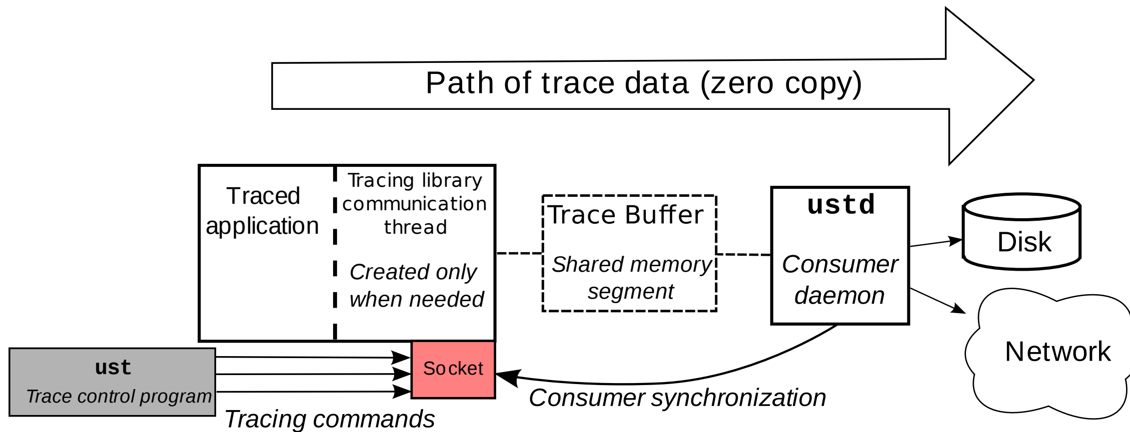



Figure 1: Overview of the LTTng Userspace Tracer architecture.

### 3.3 Instrumentation Points

Instrumentation points consist in ports of the two kernel instrumentation technologies LTTng uses: markers and tracepoints. Their usage is the same as in the kernel.

Inserting a marker is as simple as adding a single line of code at the point where the event must be recorded. Figure 2 shows an example of a marker. Markers include a format string that resembles `printf` format strings; they include the name of each argument, the format of the event in the trace and the type of the variable passed to `trace_mark`.

Tracepoints are designed to be more elegant and provide type checking. An example is shown in Figure 3. They do not include a format string in the call, but necessitate some declarations, typically in separate C and header files. This makes them more suitable for permanent instrumentation. Markers are best suited for quickly adding instrumentation while debugging.

Markers and tracepoints list information about themselves in a special section of the executable or dynamic object. Each library and each executable that contains instrumentation must therefore register its markers/tracepoints section via a call to the tracing library. This is done by invoking a macro that adds a constructor that automatically does this, in one of the source files of every executable and every dynamic object.

Each of the instrumentation points may be enabled or disabled by the user, even while the trace is active. Each time the control flow passes on an instrumentation point, a global variable is tested to verify whether it is enabled.

### 3.4 Buffering Mechanism

The buffering mechanism is a port of the lockless LTTng algorithm. Its design reuses many ideas from the K42 operating system and the Linux kernel Relay[8] system.

Events are written in a circular, per-process buffer, which is divided in sub-buffers. By default, when a sub-buffer is full, it is consumed by a consumer daemon. In another operating mode called *flight recorder*, the circular buffers are constantly overwritten, until the buffers are flushed, either by the user or by a program. This is useful to wait until an infrequent bug occurs in the application.

Each event is associated with a channel. Each process has a distinct buffer for each channel. Having several channels allows to choose the size of sub-buffers per channel. It also allows to keep some events for a longer period of time by putting them in a low-rate buffer when operating in flight recorder mode.

The buffers are allocated inside System V shared memory segments so the consumer daemon can map them in its address space.

Writes to the buffers are done using a lockless algorithm whose correctness was formally verified. It is therefore reentrant and thread- and signal-safe.

### 3.5 Trace Control

There needs to be a way to command an application to start or stop tracing, to enable, disable or list instrumentation points and to control trace parameters such as sub-buffer size and count.

```
trace_mark(main, myevent, "firstarg %d secondarg %s", v, st);
```

Figure 2: Example of a marker. The first argument is the channel and the second is the event name. The third is the format string of the event arguments while the last are the event arguments. In the format string, each argument name is followed by its format.

```
trace_main_myevent(v, st);
```

Figure 3: Example of a tracepoint.

A helper application called `ust` is used for this purpose. It communicates with the traceable application through a Unix socket. Communications are handled by a special thread in the traced process.

In order for the UST to be as minimally invasive as possible, this thread is not launched automatically when the application starts. Instead, the tracing library constructor registers a signal handler for a particular signal. When that signal is received, a listener thread is started. This thread creates a named socket in a predefined directory. The name of the socket is the process ID.

For now, the SIGIO signal is used. Although this signal is used for other purposes on occasion, the siginfo_t structure allows to determine whether the signal was sent by a process or the kernel.

### 3.6 Data Collection

A single process collects trace data for all processes being traced on the system. This process is called `ustd`. It opens a named socket, called `ustd` and located in the same directory as the applications' sockets. Through it, `ustd` can be commanded to collect the trace data of a certain buffer of a given PID.

Upon receiving this command, `ustd` creates a new thread that connects to the socket of the tracing process, first sending it the SIGIO signal if the socket is not yet available. It then requests the shared memory segment IDs for the buffers and maps them.

Still using the socket of the traced application, this consumer thread sends a command requesting access to the next sub-buffer. When the next unconsumed sub-buffer is full, a reply is sent, and the consumer thread writes its data to the trace file, reading from the shared memory segment. Because the memory area passed to `write()` is in the shared memory segment, no copying in RAM occurs.

### 3.7 Early and Late Tracing

A few complicating factors must be taken into account when tracing very early or late in the program lifespan.

#### 3.7.1 Tracing from program start

Sometimes, it is important to trace the program from its beginning. One can try to start the program, and then enable tracing. But chances are by the time the SIGIO signal is sent and received, and the command to start tracing is sent through the socket, some events will have been lost. In some cases, the program may have already ended.

Therefore the UST has a special mechanism for tracing from the beginning of the program execution. To trace a program from its beginning, the user can run the program with two environment variables defined. These variables are parsed by the tracing library constructor. Defining both these variables guarantees that by the time the program enters its `main()` function, tracing will have started.

**`UST_TRACE=1`** Automatically activate tracing on program start.

**`UST_AUTOPROBE=1`** Automatically enable all instrumentation points.

#### 3.7.2 Tracing until the end of the program

Things are also slightly more complicated when tracing near the end of a program. The program can crash and

be unable to notify `ustd` that its last sub-buffer should be consumed. Worse, it may end before `ustd` is able to map its buffers. In the former case, the end of the trace will be lost. In the latter, the full trace is lost, since the kernel deallocates shared memory segments when their last user disconnects from them. The following describes how the UST deals with these issues.

When a program crashes, its socket connections are closed by the kernel. `ustd` can detect this and run a crash recovery procedure on the buffer. The recovery procedure identifies which sub-buffers contain data that is not yet consumed, and how much data can be recovered in each one of them. This data is appended to the trace file. The procedure guarantees that all the events up to the last that is recovered are valid and that none was skipped (provided there are no lost events in the buffer due to overflow). It is possible to determine what data in each sub-buffer is valid, because some counters used in the atomic algorithm are mapped along with the buffer in the shared memory segment.

When the program lifetime is too short for `ustd` to have time to map its memory, a different problem is encountered. Although the UST does not yet support this case, it is planned to use a destructor to handle this case. If the destructor of the trace library detects that a trace is being recorded and that its buffers have not yet been mapped, it will extend the life of the process slightly to give time to `ustd` to map them.

## 4 Trace Analysis

LTTV[1], the LTTng Viewer, is a graphical trace viewer for LTTng traces. LTTV provides a number of graphical, statistical and text-based views for traces. Furthermore, it has the ability to display concurrently the events of several traces that were recorded simultaneously. This is useful for viewing traces recorded in virtual machines at the same time as a trace of the host system.

This feature can also be used to display a kernel trace at the same time as userspace traces. In the event list, the events of all the traces are then interleaved. This allows to get a better grasp of problems that involve both the userspace and kernel side. The usage of a precise and common time source ensures events in the list are correctly ordered even if they are produced on different cores or on different sides of the kernel/userspace border.

## 5 Performance

This section presents some early performance measurements for the UST, as well as a comparison with the performance of DTrace SDT for an equivalent tracing task.

The tests were run cache-hot on a dual quad-core Xeon 2 GHz with 8 GB of RAM. DTrace was run under OpenSolaris. The test consisted in running 60 times the command `find /usr -regex '.*a'`. This regular expression was chosen arbitrarily to provoke `malloc`/`free` activity.

The calls to `malloc` and `free` made by `find` were instrumented. This was done by intercepting the calls to them using a shared library loaded with `LD_PRELOAD`. The intercepting functions contained the actual instrumentation points and called the real version of the function. The `malloc`/`free` interception was active for all tests, even when not tracing. The `malloc`/`free` arguments and return values were recorded by the instrumentation.

Event counts vary between DTrace and UST tests because the `/usr` directory contained more files in the Linux system (for UST tests) than in the OpenSolaris system (used for DTrace tests).

The DTrace performance (Table 1) was first measured with tracing disabled. Then, it was measured with tracing enabled, with two different scripts. One (*printing probe*) printed the function name (`malloc` or `free`), its arguments and its return value. The output was redirected to a file. The other (*simple probe*), only counted the number of events. Its aim was to verify how much time is due to the actual printing operation. The cost per event was obtained by taking the time in excess of the time with tracing disabled and dividing it by the number of events.

The UST performance (Table 2) was measured first with the instrumentation not compiled in and then compiled in. The difference between these two measures was not significant. In fact, in these tests, the execution time diminished when compiling in the instrumentation. With probes connected but tracing not active, the execution time was slightly higher. In this mode of operation, a function call is made upon hitting an instrumentation point, but the function returns almost immediately, after finding out tracing is disabled. Finally, with tracing

| Test | Exec. time | Nb. of events | Cost / event |
|---|---|---|---|
| Not tracing | 53.29 s | – | – |
| Tracing, simple probe | 251.81 s | 44,085,780 | 4.5 $\mu$s |
| Tracing, printing probe | 274.51 s | 44,085,780 | 5.0 $\mu$s |

Table 1: DTrace results.

| Test | Exec. time | Nb. of events | Cost / event |
|---|---|---|---|
| Not tracing, instrumentation not compiled in | 92.61 s | – | – |
| Not tracing, instrumentation compiled in | 92.18 s | 145,168,560 | $\approx 0$ |
| Not tracing, probes connected | 99.25 s | 145,168,560 | 46 ns |
| Tracing | 193.94 s | 145,168,560 | 698 ns |

Table 2: LTTng Userspace Tracer results.

enabled, a cost per event of 698 ns was obtained. The cost per event was calculated by taking the time in excess of the time with instrumentation not compiled in and dividing it by the number of events.

The LTTng UST had a cost per event more than 7 times lower than DTrace. This difference is explained by the method used by each tracer to record events. While DTrace executes a trap at each event, the UST writes the event in a buffer in the program memory, saving a round-trip to the kernel.

The UST has a low per event cost, while having no apparent impact while disabled. This makes it particularly useful in production systems, and other systems where affecting performance as little as possible is critical. Its compact trace format further limits its impact by limiting the disk and network usage.

As the UST becomes more mature, it is likely that new optimizations will result in an even lower cost per event. The Future Work sections mentions a few possibilities to this effect.

## 6 Future Work

The current per-process buffers were a simple first step for a port. However, this approach has an important limitation. It induces cacheline bouncing on multi-threaded applications. Using per-thread buffers would fix this problem.

In the kernel, the most optimized variant of the markers uses *immediate values*, a technique that modifies an instruction at the instrumentation point site when enabling or disabling markers. This code modification consists in changing the immediate value in a *load immediate* instruction. This instruction is immediately followed by a test of the register in which the value was loaded. Depending on the result of the test, the event is recorded or not. Although this approach is faster than the current test of a global variable, is much more architecture-dependant.

`UST_AUTOPROBE` should allow the specification of a list or pattern of markers. Its current limitation of activating all of them at once may cause a performance penalty that is higher than necessary on programs where markers encountered extremely often are compiled in but not needed for the specific problem being debugged.

Complex programs that necessitate userspace tracing are often written in high-level languages. Therefore the UST should be available to these languages. For example, a Java API using the JNI to interface the C API would be straightforward to implement.

Work is currently in progress to enhance the daemon so it can send traces over a network. This is particularly useful on special purpose systems with little or no disk space available.

## References

[1] LTTV. `http://lttng.org`.

[2] Systemtap static probes.
`https://fedoraproject.org/wiki/Features/SystemtapStaticProbes`.

[3] Mathieu Desnoyers and Michel R. Dagenais. The LTTng tracer: A low impact performance and

behavior monitor for GNU/Linux. In *Linux Symposium*, Ottawa, Ontario, Canada, June 2006.

[4] Mathieu Desnoyers and Paul E. McKenney. Userspace Read-Copy-Update Library. `http://ltt.polymtl.ca/cgi-bin/gitweb.cgi?p=userspace-rcu.git`.

[5] Pierre-Marc Fournier and Jan Blunck. libkcompat. `http://git.dorsal.polymtl.ca/?p=libkcompat.git`.

[6] Frank Hofmann. The DTrace backend on Solaris for x86/x64. `http://opensolaris.org/os/project/czosug/events_archive/czosug2_dtrace_%x86.pdf`.

[7] Steve Rostedt. ftrace. `http://lwn.net/Articles/290277/`.

[8] Karim Yaghmour, R Wisniewski, R Moore, and M Dagenais. relayfs: An efficient unified approach for transmitting data from kernel to user space. In *Linux Symposium*, Ottawa, Ontario, Canada, 2003.

# Twenty Years Later: Still Improving the Correctness of an NFS Server

Robert Gardner
*Hewlett Packard*
rob.gardner@hp.com

Scott D'Angelo
*Hewlett Packard*
scott.dangelo@hp.com

Matt Sears
*Hewlett Packard*
matt.sears@hp.com

## Abstract

The NFS reply cache, also known as the Duplicate Request Cache, was first described over twenty years ago [Juszczak] as a way to help a server give correct responses to certain types of replayed operations. Some operations, called idempotent, can be safely repeated and will do no harm. Other operations, called non-idempotent, can only succeed once [Callaghan]. For example, a request to read a certain block of a file will produce the same result each time. But an operation such as rename will succeed the first time, but a subsequent retry will result in an error being reported to the client. The reply cache keeps track of responses to recently performed non-idempotent transactions, and in case of a replay, the cached response is sent instead of attempting to perform the operation again. In addition to avoiding these client-visible errors, performance is also improved by avoiding unnecessary work.

The trouble begins when the size of the cache is inadequate to deal with the rate of incoming transactions. Now the mechanism breaks down, and replayed requests may result in duplicate work being done and erroneous results generated. Even modest workloads can result in an enormous rate of non-idempotent requests which would necessitate enlarging the reply cache to unacceptable levels. Heavy workloads can cause network congestion and delays that can foil attempts to cache enough transactions to maintain correctness. Simply increasing the cache size, even by large factors, isn't effective.

We address these problems by making the cache smarter instead of larger. First, we add the concept of protecting a cache entry, which temporarily makes it exempt from the usual replacement process. Next, we add some heuristics that grant or revoke the protection of a cache entry. Finally we eliminate automatic expiration of cache entries. Taken all together, this scheme drastically reduces the number of errors reported by clients on a large network.

## 1 Traditional Design

Linux drew strongly from its predecessors in its implementation of the NFS reply cache, and despite occasional rumors of being rewritten [Kirch], it has changed very little since its inception. In addition to the actual reply data, each cache entry contains other essential information about one NFS transaction, including the client's IP address, the transaction ID (XID), NFS procedure number, timestamp, etc. As entries are created, they are placed onto hash chains indexed by the XID to enable faster searching. When a new request is received, the cache is searched. If a match is found (a hit) then the cached reply is sent back to the client. Otherwise, a new entry is made, replacing an existing entry via a least-recently-used policy. An entry "expires" after two minutes, which excludes it from searches, even if other cache entries have not replaced it. This is to avoid issues with transaction ID re-use [Werme]. The cache entries are also kept on an additional linked list that is ordered by time of use. When an entry is created or touched, it is moved to the head of this list. This makes the least recently used entry instantly accessible when replacement is invoked.

The size of the NFS reply cache is critically important, since it directly affects residency time of a cache entry. If entries are replaced too quickly, then a replayed transaction will not be found in the cache and a client visible error will result. We call this a *critical reply cache miss*. A worse consequence of a critical reply cache miss is the "lost write due to replayed truncate" problem [Sun] which can cause data loss.

The design and sizing of the reply cache dates back to 1989 [Juszczak] when processing power and network bandwidth were rather limited, which in turn constrained the rate of incoming NFS requests. The power of a modern server makes configurations possible which were probably not envisioned twenty years ago. Nowadays it is not uncommon for a server to handle requests

from hundreds, or even thousands of clients simultaneously, so it is not a huge leap to realize that the reply cache must be sized appropriately for the expected load.

The number of entries in the reply cache has been subject to many changes over the years, with various implementations (i.e., BSD, DEC Unix, HP-UX, Solaris, etc) employing widely differing sizes, and the Linux code settling on 1024 entries. The paucity of data to justify any particular choice suggests that some guesswork was involved in each implementation. A clever scheme to dynamically resize the reply cache based on demand was described in [Banks], and this work will probably be incorporated into the Linux kernel in the very near future. Our experiments with a reply cache 16 times larger than normal (16384) showed cache entries surviving for mere seconds. With typical RPC replay timers being multiples of *minutes* [Eisler], we were convinced to look at possibilities that did not involve enlarging the cache.

In addition to the sizing dilemma, the cache replacement algorithm is rather unintelligent. The simple least-recently-used policy doesn't take into account any statistical data available. For instance, busy clients may consume many cache entries, but this should not have adverse effects on less busy clients. Network congestion can cause replies to be lost and wreak havoc on the simplistic cache replacement algorithm. Since there is a practical limit to how much memory should be devoted to the cache, a solution more appropriate for today's enterprise environments is needed.

## 2 Is this a problem worth solving?

A human operator may not notice, may not care, or very likely will not know how to interpret the symptom of a critical reply cache miss. All that will be seen by a user is a puzzling failure of a `mkdir` command, for instance, when in fact the directory was created successfully. A user faced with this scenario is likely to pretend the whole thing was a dream, and move on. Consequently, problem reports are rare, and unlikely to identify the true source of the problem. This line of reasoning might explain why we haven't seen many efforts to correct the problem.

Now envision a different type of scenario. There's a file server with hundreds or even thousands of active clients. The clients are running applications that are updating databases, or something equally critical. Perhaps client applications exit prematurely when they see unexpected failures of simple file operations. In these scenarios, critical reply cache misses are likely to be noticed, since applications are much less forgiving than human operators when faced with unexpected results. At best, this will result in customer complaints, and at worst, there may be data corruption.

New file serving protocols, such as NFSv4.1 [Noveck] [Shepler], do not suffer from this problem. But history has shown that emerging technologies do not instantly displace old ones, and NFSv3 will likely be around for many more years.

## 3 New Paradigms, New Problems

A High Availability NFS server coupled with a clustered filesystem [CITI] [Bhide] creates new problems for the NFS reply cache. Although high availability and cluster issues are not the focus of this paper, the myriad problems that they expose provided much of the impetus to design new methods. These new ideas are completely applicable to a standalone server.

Transient service disruptions, such as failover events, often exacerbate networking back-off mechanisms and can cause extended replay delays. In practice, a failover may require several seconds or more to complete, but may incite transaction replay delays of a minute or more. Failover events are hot spots of trouble for the reply cache because of the greatly increased probability of a lost reply and a subsequent replayed transaction.

For example, when a failover event is initiated for administrative reasons, i.e. maintenance, load balancing, etc., the contents of the reply cache from the failing server must be transported to the takeover server. Various schemes for doing this have been suggested [Bhide] and the straightforward method we use is to simply have the failing server write out the contents of the reply cache to some network accessible location, and then have the takeover server read it back. This approach immediately caused a new problem, as the flood of new cache entries from the failing server displaced more recent cache entries that were already resident on the takeover server. New logic had to be created to deal with these competing sets of cache entries.

## 4 Evolution and Implementation of a New Approach

The cache entry competition caused increased client-visible errors right after a failover event. As a first step, *acquired* cache entries should not displace existing entries on the takeover server, but rather should be merged with them as space allows. Since clients of the failing server are victims of a transient service disruption, they are more likely to retry requests, and thus their reply cache entries are more important than existing ones. This led to the first major change to the reply cache logic. A `protected_until` field was added to the cache entry structure, signifying that this entry is exempt from replacement and reuse until a certain time in the future. This gave the acquired entries a survival advantage over normal entries, and greatly eased the problem of failover related client request errors.

Once this protected status was available for cache entries, other opportunities arose to implement policies that could be applied to grant or revoke this special status, and thus help in other problematic scenarios. For instance, when a cache hit is found, the lookup code checks to see if it's for a transaction that is still being processed by the filesystem. If so, we have no answer to give the client, so `RC_DROPIT` is returned, which has the effect of sending no reply whatsoever. It's not clear why the client sent a duplicate request. Perhaps the original request was delayed due to a media problem, disruption in connection to a SAN or RAID, filesystem repair, etc. In any case, no response has yet been sent for the original request, and now the code is dropping the reply to the duplicate request. It seems like a sure bet that the client is going to try the request again! As such, the cache entry is a perfect candidate for "protected" status.

A similar piece of logic in the lookup code discards the reply for a request received too soon after the previous identical request. Again, it's not clear what could cause this to happen, but if it does, we know that the client may not have received a response to its previous request, and it won't receive a response to the current one either. Likewise, this cache entry ought to be protected since it is very likely that the client will try again soon. These hints were very successful in reducing critical reply cache misses.

The two minute cache entry expiration came under scrutiny next. If extreme conditions could cause replays to be delayed for long periods, then the two minute expiration time would foil any attempt to do anything more sophisticated. The best explanation available for the existence of the expiration period is that it prevents false positive cache hits when a client reuses XIDs without cycling through the entire 32-bit space available [Werme]. This may happen, for instance, when a client reboots or when multiple NFS clients originate from the same IP address [Oracle]. Though XID reuse could well be considered a client bug, with the fix belonging in the client code, a responsible server should attempt to handle this better. Our reply cache code stores a simple checksum of some of the data payload along with the traditional cache keys and requires a match of the checksum before declaring a cache hit. This substantially reduces the probability of a false positive cache hit. Given this new logic, the two minute expiration logic was eliminated entirely, thus paving the way for cache entries to have extended lives in the system.

Following the successes of the previous hints, an exploration was made of other available information that could be used to predict which cache entries are likely to be needed in the future. One predictor seemed to be very powerful: a client running a single threaded NFS application generally does not issue a new request until it receives a reply to the last one. This means that a *replayed transaction is very likely to be the last one received from that client*. This suggests that the cache hit rate can be greatly increased by simply remembering each client's most recent transaction. This rule for single threaded clients only seems to fail when the client application writes lots of data, since *write* operations may be reordered, delayed, or grouped by the client.

Storage is allocated to keep essential information about the most recently used transaction for a fixed number of clients. This new structure is called the Most Recently Used (MRU) list. It is indexed using a standard hashing function of the client's IP address and keeps track of the XID of the most recent transaction, the time that the transaction occurred, and a pointer to the actual reply cache entry for the transaction.

When a cache entry is being made, the MRU list is updated with the information describing the new transaction. The reply cache entry itself is marked with a protect time in the future using the previously described `protected_until` cache entry field. If there is an existing *most recently used* cache entry associated with the client, its protection time is cancelled, thus making it eligible for replacement.

## 5 Test Environment and Results

During the course of rigorous product testing, our servers are routinely subjected to extremely high loads and unusual configurations. It was in this harsh environment that many subtle problems came to light. Most of the problems can be boiled down to critical reply cache misses, and so for this paper, test data was generated using a relatively small network and artificially injecting transaction reply losses into the traffic. These losses caused the clients to retry after 60 seconds. Ten clients were doing nothing but generating a non-idempotent workload on the server, while two other clients ran the well known test suite, Connectathon [CTHON]. Connectathon encompasses a large variety of tests, each designed to exercise some particular aspect of functionality over NFS, but only the "special" tests were run since those emphasize operations that are sensitive to reply cache behavior. This setup generated roughly 450 non-idempotent transactions *per second*. Linux kernel version 2.6.29 was used, changing only the reply cache size from its original value of 1024.

Critical reply cache misses are reduced to *zero* for single threaded NFS clients, which also brings the number of client visible errors on unlink, rename, etc. to zero. The graph (Figure 1) shows the number of critical reply cache misses for several easily observed non-idempotent operations using a variety of reply cache sizes (1024, 4096, and 16384) on the stock Linux kernel. Increasing the cache size to 16384 makes essentially no difference in the number of client visible errors. The cache size must be increased to over 27,000 entries before a significant improvement in behavior is seen. Also shown is the result for a reply cache modified with our MRU logic. Note that this modified reply cache only had 128 entries.

Do be aware that although the Connectathon test suite is not sensitive to the subtle corrupting effects of replayed *write* operations, the MRU list has the same power to deal with these as it does with other replayed non-idempotent operations.

## 6 Potential problems and opportunities for further development

The MRU list only keeps track of one transaction per client. If there is more than one thread on a single client that is making requests, the scheme breaks down. If one thread gets stuck waiting for a lost reply, the other is not impeded and keeps sending requests. This could result in multiple outstanding RPC transactions, of which only one will be recorded by the MRU list. Ignoring multiple applications running on a client, this primarily affects user space NFS client implementations, such as Oracle's Direct NFS [Oracle], and clients that connect from a private network using network address translation [NAT]. The latter includes some configurations of virtual machines [Xen] [VMware] [VirtualBox].

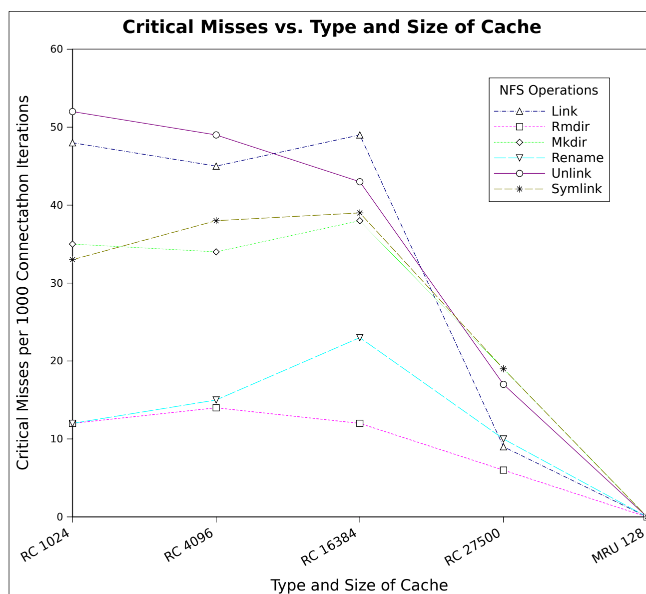



Figure 1: Results

Is it possible somehow to distinguish between two parallel streams of requests from the same client? Parallel userland NFS clients (such as Oracle DNFS) must each maintain their own TCP connections, each with a unique source port. Keeping an MRU entry for each unique (IP address, source port) pair might solve the problem for this particular case. Parallel userland NFS clients also probably generate different continuous streams of XIDs, so it may be workable to detect this and remember the last XID for each stream. It also may be possible to distinguish different client threads through their credentials (i.e., different UIDs).

When the MRU fails to provide a matching cache entry, there are some hints we can use to attempt detection. XIDs from each client are *often* monotonically increasing with respect to the client's host byte order. If the XID of a new request is numerically less than that of the most recently used one, that indicates that it might be a replay. Though information beyond the single most

recently used transaction is not saved, this situation can be flagged and a log may be kept of how often it occurs. Similarly, if a transaction arrives with an XID that matches the XID in the MRU list, but misses in the cache, then it must be because the protection time given to the cache entry wasn't long enough, and it got replaced by a newer entry. Once again, this situation can also be recorded for later analysis. Although the RPC specification [RFC1831] explicitly prohibits treating the XID as a sequence number as we do here, our use of it in this way is only a heuristic for failure detection and has no effect on the semantics of the NFS server.

Another problem is that the MRU list must keep track of at least one datum for each client. If the number of clients exceeds the size of the MRU list, we're back at square one. So another opportunity for improvement is a dynamically expanding MRU list, or at least a dynamically sizable list that an administrator could configure.

Finally, along with the addition of new data structures and code comes complexity, and subsequently the potential for bugs and performance loss. This is especially important since the reply cache can already be somewhat of a bottleneck, and at least one good effort has been made to remedy this [Banks]. The actual performance implications of the MRU list have not been thoroughly analyzed and there is the potential that improving correctness in this fashion hurts performance in an area where it cannot be afforded. For instance, the MRU list search algorithm is simplistic. A hash of the client's IP address is used as an index into the list, but if there is a collision, then a linear search algorithm takes over. Clearly this could be made better with a more sophisticated hashing scheme, but this tradeoff was made after brief analysis showed that the linear search has to traverse more than a few entries only when the MRU list is nearly full.

## 7 Conclusions

The current NFS reply cache implementation is not sufficient for today's enterprise environments. The fixed size cache is not large enough and there may not be a practical way to make it large enough to deal with heavy workloads without client visible errors. The logic of the existing solution only takes into account the age of a cache entry when deciding on an entry to replace. There is abundant information available that could help to make more intelligent replacement decisions, but none of it is utilized. Our contribution is to make use of some of this data, and make better decisions about which cache entries to keep and which to throw away. By making the reply cache algorithm *smarter* instead of simply larger, we have minimized the likelihood of errors in both the clustered/HA environments as well as the single server node environments.

## 8 Acknowledgements

Thanks to all the diligent and critical reviewers, especially Chet Juszczak, J. Bruce Fields, Greg Banks, and Stuart Friedberg. We would also like to thank Hewlett-Packard and the Storage Works Division for allowing us the opportunity to explore, develop and improve on the existing code base. These improvements are not a theoretical novelty done for research purposes, but rather are a genuine attempt to solve real world problems, and code implementing these ideas are being shipped as part of the HP StorageWorks Scalable NAS File Serving Software product.

## References

[Juszczak] Juszczak, C., *Improving the Performance and Correctness of an NFS Server*, USENIX Conference Proceedings, Winter, 1989

[Callaghan] Callaghan, B., NFS Illustrated, ISBN 0-201-32570-5

[Kirch] Kirch, O., *Why NFS Sucks*, Linux Symposium, 2006, Ottawa

[Werme] Werme, R., *RPC XID Issues*, Connectathon 1996 Talks, `http://www.connectathon.org/talks96/werme1.pdf`

[Sun] Sun Microsystems, *NFS: Network File System Version 3 Protocol Specification*, 1994

[Banks] Banks, G., *Making the Linux NFS Server Suck Faster*, Presented at linux.conf.au, 2007

[Eisler] Eisler, M., *NFS over TCP, Again*, March 1, 2006, Connectathon Talks, `http://www.connectathon.org/talks06/eisler.pdf`

[Noveck] Noveck, D., *NFSv4.1 Current Status*, Feb. 5, 2007, Connectathon Talks, `http://www.connectathon.org/talks07/NFSv41update.pdf`

[Shepler] Shepler, S., Eisler, M., Noveck, D., *NFS Version 4 Minor Version 1*, IETF Draft, Dec. 15, 2008, `http://www.ietf.org/internet-drafts/draft-ietf-nfsv4-minorversion1-29.txt`

[CITI] Center for Information Technology Integration, *Linux NFS Requirements for Cluster File System and Multi-protocol Servers*, Feb. 2008, `http://www.citi.umich.edu/projects/cluster_nfsv4/google+citi-SoW-redact.pdf`

[Bhide] Bhide, A., Elnozahy, E., Morgan, S., *A Highly Available NFS Server*, Proceedings of the Winter 1991 USENIX Conference

[CTHON] *Connectathon Test Suite*, `http://www.connectathon.org/nfstests.html`

[Oracle] *Oracle Database 11g Direct NFS Client*, Oracle White Paper, July 2007

[NAT] *The IP Network Address Translator (NAT)*, Network Working Group, May, 1994, `http://www.ietf.org/rfc/rfc1631.txt`

[RFC1831] *Remote Procedure Call Protocol Specification Version 2*, Internet Engineering Task Force, `http://www.ietf.org/rfc/rfc1831.txt`

[Xen] Barham, P., B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield (2003). *Xen and the art of virtualization*, In SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles, New York, NY, USA, pp. 164-177. ACM Press.

[VMware] `http://vmware.com/`

[VirtualBox] Sun Microsystems, *Optimizing the Desktop using Sun VirtualBox*, `http://www.virtualbox.org/`

[Sandberg] Sandberg, R., D. Goldberg, S. Kleiman, D. Walsh, B. Lyon, *Design and Implementation of the Sun Network Filesystem*, USENIX Conference Proceedings, USENIX Association, Berkeley, CA, Summer 1985.

[Pawlowski] Pawlowski, B., C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz (1994). *NFS Version 3 Design and Implementation*, In Proceedings of the Summer 1994 USENIX Technical Conference, pp. 137-152.

[RFC1094] Network Filesystem Specification, Version 2, RFC 1094, Sun Microsystems, Inc., March 1989, `http://www.ietf.org/rfc/rfc1094.txt`

[RFC1813] Network Filesystem Specification, Version 3, RFC 1813, IETF, June 1995, `http://www.ietf.org/rfc/rfc1813.txt`

# Memory Migration on Next-Touch

Brice Goglin
*INRIA, LaBRI, University of Bordeaux*
Brice.Goglin@inria.fr

Nathalie Furmento
*CNRS, LaBRI, University of Bordeaux*
nathalie.furmento@labri.fr

## Abstract

NUMA abilities such as explicit migration of memory buffers enable flexible placement of data buffers at runtime near the tasks that actually access them. The `move_pages` system call may be invoked manually but it achieves limited throughput and implies a strong collaboration of the application. Indeed, the location of threads and their memory access patterns must be carefully known so as to decide when migrating the right memory buffer on time.

We present the implementation of a *Next-Touch* memory placement policy so as to enable automatic dynamic migration of pages when they are actually accessed by a task. We introduce a new PTE flag setup by `madvise`, and the corresponding *Copy-on-Touch* codepath in the page-fault handler which allocates the new page near the accessing task. We then look at the performance and overheads of this model and compare it to using the `move_pages` system call.

## 1 Introduction

The democratization of *Non-Uniform Memory Access* (NUMA), from ITANIUM based platforms, to AMD HYPERTRANSPORT architecture and INTEL's new QUICKPATH interconnect, raises the need to carefully place data buffers near the tasks that access them [1]. Indeed, when local data access is significantly faster than remote access, data locality becomes a critical criterion for scheduling tasks. And the idea of migrating data buffers together with their accessing tasks has to be considered.

In the last decade, LINUX slowly learnt how to manage NUMA requirements. It first gained NUMA-aware allocation facilities in the early 2.6 kernels, either automatically on first touch, or thanks to the `mbind` and `set_mempolicy` system calls. It then acquired memory migration abilities a couple years ago with the addition of `migrate_pages` and `move_pages`. These features enable the manual adaptation of the data distribution across memory nodes to the current task locations. However, dynamic applications with migrating threads may require the corresponding data buffers to migrate automatically as well.

Indeed, threads are a convenient way to program modern highly-parallel hosts. Parallel programming languages such as OPENMP [6] try to ease the mapping of parallel algorithms onto the architecture. The quality of the thread scheduling has a strong impact on the overall application performance because of affinities. This issue now becomes critical due to the variable memory access latencies and bandwidths that NUMA architectures exhibit. We thus propose in this article an implementation of the *Next-Touch* policy which provides applications with a convenient way to have memory buffers dynamically follow their accessing tasks.

The rest of this paper is organized as follows. In Section 2, we provide background information about multithreaded applications requirements and LINUX abilities over NUMA architectures. Section 3 explains our implementation of the *Next-Touch* policy in the LINUX kernel. Experiments shown in Section 4 emphasize the performance advantages of our approach. Before concluding, both our design and implementation are discussed in Section 5.

## 2 Multithreading on NUMA Architectures

We briefly introduce in this section the requirements in term of memory affinity in multithreaded applications, the available migration strategies in LINUX, and our motivations to implement a *Next-Touch* strategy.

### 2.1 Dynamic Multithreading Requirements

Memory access requirements of multithreaded applications obviously depend a lot of on thread access patterns, but also on the way the application parallelism is mapped onto threads. Indeed, a application binding one thread per core and allocating its dataset nearby will get satisfying performance. If some threads need to exchange some data, the application has to either migrate the thread near their new target buffers, or migrate these buffers. As long as the application manipulates threads directly (for instance through the pthread interface) and knows their memory access patterns, manipulating memory buffers manually is possible.

The situation becomes far more complex when parallel programming languages are involved. OPENMP appears nowadays as a very easy way to exploit multicore architectures. Indeed, it enables easy thread-based parallelization of sequential applications by adding pragmas in the source code. The democratization of this approach in high-performance computing raises two problem. First, OPENMP does not provide the compiler or runtime-system with any information about memory access patterns. Second, parallel sections may be nested and thus cause dramatic load imbalance in case of irregular applications such as adaptive mesh refinement. Indeed, one of the OPENMP threads may open a new parallel section if it has to much work to do compared to its teammates.

Such nested parallelism causes the operating system or the runtime OPENMP system to load-balance newly created threads across all cores in the machine. Each migrated thread may thus have to migrate its own dataset so as to reduce distant memory accesses and avoid congestion in the memory interconnect [7]. However, having a precise knowledge of which buffer to migrate is often hard. And the absence of memory-affinity-related pragmas in parallel languages does not help. Moreover predicting or detecting each thread migration is also difficult (unless the scheduler is embedded in the application). Migrating memory buffers on time when threads are migrated is thus a challenge.

### 2.2 NUMA Management APIs in LINUX

NUMA-awareness was added to LINUX during the development of 2.6 kernel. Most features are made available to user-space applications thanks for instance to the libnuma interface [3]. It provides applications with memory placement primitives such as `set_mempolicy` and `mbind` that insure buffers are allocated on the right NUMA node(s). Such static policies are indeed useful when the application knows where the accessing threads run: if a thread is bound to a NUMA node, its dataset may be bound there as well.

When the thread location is unknown, a commonly-used approach is *First-Touch*. It relies on the operating system laziness when it comes to actually allocating physical pages. Many OPENMP applications thus add an initialization round where each thread touches all the pages of its own dataset so that they are actually allocated on its local NUMA node. However, as soon as some threads have to migrate (for instance because of load-balancing in irregular parallelism), the *First-Touch* approach cannot help anymore since pages have already been touched and allocated during the initialization round.

Memory migration is thus required as a way to have datasets migrate with their accessing tasks. LINUX earned migration primitives such as `move_pages` and `migrate_pages` a couple of years ago [4]. The latter migrates an entire process address space onto some NUMA node(s). It was designed together with *Cpusets* as a way for administrators to partition machines. And therefore it is not relevant for multithreaded applications where only part of the address space may migrate.

The `move_pages` system call is the only way to explicitly migrate a buffer within an application[1]. However, it is a synchronous function that must be invoked by user-space, for instance when a thread is migrated or when it starts working on a new dataset. It requires a strong cooperation between the application and the runtime system when some threads decide to migrate (assuming they even properly know their thread access patterns).

### 2.3 The Need for Dynamic Migration Primitives

The democratization of dynamic parallelism such as adaptive mesh refinement, especially thanks to nested parallel sections in OPENMP, goes beyond what LINUX memory management interface targets nowadays. Both

[1]The `mbind` system call with the `MPOL_MF_MOVE` flag is actually somehow equivalent to `move_pages`.

the operating system and user-space applications or run-time systems may have to migrate threads for load-balancing reasons. It thus becomes important to have an easy way to migrate the corresponding buffers at the same time. However, the *First-Touch* approach is not applicable except during the initialization phase. And explicit migration requires the precise knowledge of when each thread is migrated and of their memory access patterns. It is difficult to achieve because parallel languages such as OPENMP do not provide the required annotations, and also because user-space has no way to specify task-memory affinity to the LINUX kernel.

Actually, it is not clear that such affinities belong in the kernel. People have been working on user-level thread scheduling as a way to get highly-configurable scheduling algorithms as well as reduced management costs. This model enables the addition of affinities between threads and/or data and the design of custom schedulers dedicated to some classes of applications [9]. One may think that migrating data buffers may thus have to be managed in user-space as well. However, it brings back the issue of parallel languages not providing the required annotations to explicit task/data affinities. And still, requiring a precise knowledge of these affinities is a very hard work for the developer, for instance because some buffers may be accessed by multiple threads with different read/write patterns. From the user-level developer point of view, it is much more work than just trying to load-balance threads across all cores of the machine.

We thus envision the addition in the LINUX kernel of a new mechanism for managing the requirements of such multithreaded applications. The idea behind the *Next-Touch* policy is to have data buffers automatically migrate near their accessing tasks when touched. As many other LINUX features such as page allocation or *Copy-on-Write*, this policy relies on the operating system laziness since migration only occurs when it is actually needed. The application thus just has to mark buffers as *Migrate-on-Next-Touch* when it knows that thread migration is expected in the near future: for instance when the application begins a new phase with different memory access patterns, or when it enters a new OPENMP parallel section. Such events are indeed very common in multithreaded applications, and may be easily located by their developer.

As a result, as soon as a thread touches a marked buffer that is not allocated on its local memory node, it is automatically migrated. The scheduler may then freely migrate threads to accommodate load-balancing to dynamic/nested parallelism without having to care about data affinities. This model dramatically reduces the work for the developer since there is no need to know where buffers are allocated, when each thread is actually migrated, and which buffers are manipulated by each thread.

Some proprietary operating systems such as SOLARIS actually implement such a policy and it has been proven to significantly help high-performance computing [5]. We detail in the next section our design and implementation of a similar solution in the LINUX kernel.

## 3 Implementation of the Next-Touch Policy

We now explain why a *Next-Touch* policy requires kernel support and how we implemented it.

### 3.1 Why a User-Space Implementation is a Bad Idea

Implementing a *Next-Touch* policy is possible in user-space thanks to user-directed memory protection and segmentation fault management. This strategy has been used to implement distributed shared memory across machines and may also be used to detect next touches. Indeed, the `mprotect` system call may be used to prevent application accesses to any memory zone and cause segmentation faults that a custom signal handler will catch. This handler then just needs to migrate the corresponding buffer and set the default protection back. This strategy is however hard to implement safely in multi-threaded environment and obviously exhibits an important overhead. For instance, it has been shown to increase the performance of a Jacobi Solver on NUMA machine much less than a native *Next-Touch* approach under SOLARIS [8].

One unexpected drawback of this approach is actually the limited performance of the `move_pages` system call. Indeed, aside from having to call `mprotect` twice (hence flushing the TLBs) and handle the segmentation fault signal, migrating pages has a very large initialization overhead. One could think that this *Next-Touch* approach could then be used only for large buffers, but the asymptotic throughput of `move_pages` is actually low as well[2].

[2] Even after `move_pages` was fixed in 2.6.29 to have a linear complexity as shown in `http://lkml.org/lkml/2008/10/11/117`.

Still, one advantage of a user-space implementation is that migrating at the user level lets the user application manage buffer granularity. The signal handler may thus migrate a single page or a very large buffer depending on the application datasets and wishes. However, again, it requires the application to know the memory access patterns of its threads. Also not that many applications actually rely on very large granularity. And it is not clear that migrating a large buffer at once will always be faster than migrating multiple pages independently in the kernel.

One way to observe the relative slowness of a user-space implementation is to compare it with a *Copy-on-Write* across different NUMA nodes. The pseudo-code below is indeed able to copy-on-write pages from NUMA node #0 to #1 at more than 800 MB/s on a quad-socket Barcelona machine. However, as of 2.6.29, `move_pages` cannot migrate the same pages at more than 600 MB/s. Actually, *Next-Touch* may be seen as *Copy-and-Free-on-Read-or-Write*. We therefore feel that LINUX should be able to provide a *Next-Touch* policy with a similar implementation and performance as *Copy-on-Write*.

```
buffer = mmap(NULL, LENGTH, ...,
              MAP_PRIVATE, ...);
mbind(buffer, LENGTH, <node #0>);
/* prefault on node #0 */
memset(buffer, 0, LENGTH);
if (!fork()) {
  mbind(buffer, LENGTH, <node #1>);
  sched_set_affinity(<node #1>);
  /* copy-on-write on node #1 */
  for(i=0; i<LENGTH; i+= PAGE_SIZE)
    buffer[i] = 0;
}
```

### 3.2 Page-Faulting on Next-Touch

LINUX implements *Copy-on-write* by removing the write-access bit from the PTEs (*Page Table Entry*) so that any write generates a page-fault. The page-fault handler verifies in the VMA flags (*Virtual Memory Area*) that this write-access is actually valid from the application point of view. If so, it copies the page instead of killing the process with a segmentation fault. The strategy thus relies on the difference between the high-level VMA flags (defined/visible at the application level) and the low-level PTE flags (defined by the kernel and used by the processor).

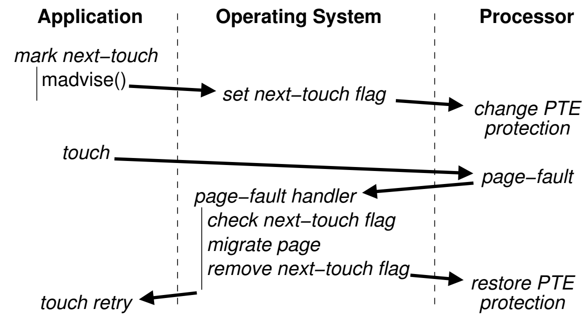



Figure 1: Description of the implementation of the *Next-touch* policy using `madvise` and a dedicated flag in the page-table entry (PTE).

We implemented the *Next-Touch* policy in a similar manner, i.e. by removing read and write-access permissions from the PTEs so that a page-fault occurs on next touch, as depicted in Figure 1. However, the implementation is harder than *Copy-on-Write* since there cannot be any high-level VMA flag for *Next-Touch*. Indeed, the migrate-on-next-touch status is only temporary. It must be cleared when the touch occurs. And a VMA might have been only partially touched, causing only some pages to have been migrated yet while some other are still marked.

For this reason, we implemented the *Next-Touch* policy by only modifying PTEs: When the *Next-Touch* flag is added, read and write access is disabled. When the page-fault occurs, the flag is removed and regular permissions are re-enabled.

The application interface to enable the *Next-Touch* policy relies on a new `madvise` behavior which is implemented in the kernel by `madvise_migratenexttouch()` and in the end by `set_migratenexttouch_pte_range()`. The whole kernel implementation is quickly summarized in Figure 2.

### 3.3 Migrating in the Page-Fault Handler

Once read/write access has been disabled for some pages, the page-fault handler has to be able to actually detect whether a fault was caused by the *Next-Touch* policy. Comparing VMA and PTE flags cannot help here, but our new *migrate-on-next-touch* PTE flag was

```
/***** in mm/madvise.c *****/
static void
set_migratenexttouch_pte_range(mm, vma, pmd, addr, end)
{
  ...
  if (pte_present(oldpte)) {
    pte_t ptent = ptep_modify_prot_start(mm, addr, pte);
    ptent = pte_modify(ptent, vm_get_page_prot(0)); /* no access rights granted */
    ptent = pte_mkmigratenexttouch(ptent);
    ptep_modify_prot_commit(mm, addr, pte, ptent);
  }
  ...
}
static long
madvise_vma(vma, prev, start, end, behavior)
{
  ...
  case MADV_MIGRATENEXTTOUCH:
    error = madvise_migratenexttouch(vma, prev, start, end);
    break;
  ...
}

/***** in mm/memory.c *****/
static int do_migrateontouch_page(mm, vma, address, page_table, pmd, ptl, orig_pte)
{
  ...
  /* if page already local, no need to migrate */
  if (page_to_nid(old_page) == numa_node_id())
    goto reuse;
  ...
  /* similar to do_wp_page() and clear the migrate-on-next-touch PTE flag */
  ...
}
static inline int handle_pte_fault(mm, vma, address, pte, pmd, int write_access)
{
  /* handle !pte_present */
  ...
  /* handle migrate-on-next-touch */
  if (pte_migratenexttouch(entry))
    return do_migrateontouch_page(mm, vma, address, pte, pmd, ptl, entry);
  /* handle copy-on-write */
  ...
}
```

Figure 2: Summary of the implementation of the *Next-Touch* policy through a new `madvise` behavior, a new PTE flag, and the corresponding page-fault handling code which mimics a copy-on-write.

designed specifically for this. The detection must occur after having taken care of non-present pages (since only pages that are present in physical memory may need migration). Migration on next touch should however be handled before looking at *Copy-on-Write* so that the latter does not have to check our new PTE flag which disables write-access in a similar way. The way `handle_pte_fault()` manages these cases and the invocation of our new `do_migrateontouch_page()` function is summarized in Figure 2.

Migrating the page in the handler is the key to performance here. We chose to target the performance critical case, which is private anonymous mappings. Fortunately, the `madvise` system call is only an advise given by the application to the kernel. It thus does not definitely enforces that any kind of memory mapping should actually be migrated on next touch. We discuss this design choice further in Section 5.

Migrating private anonymous mapping is actually very simple since there is no need to handle shared pages properly. The final code is therefore very similar to the *Copy-on-Write* code (in `do_wp_page()`). The old page is copied into a new page that was allocated on the local node. Then the old page is released.

## 4 Performance Evaluation

We present in this section a performance evaluation of our *Next-Touch* policy implementation in the LINUX kernel. The experimentation platform is a quad-socket quad-core OPTERON Barcelona (2347HE, 1.9 GHz) machine. It runs 2.6.27 with the `move_pages` performance-fix patches and our *Next-Touch* patches.

### 4.1 Migration Throughput

Figure 3 presents a comparison of the throughput of various data migration strategies. It first shows that the existing migration system calls have a very large initial overhead and a limited asymptotic throughput (800 MB/s for `migrate_pages` and 600 MB/s for `move_pages`).

Our *Next-Touch* implementation shows a very small base initialization overhead. Its asymptotic throughput (800 MB/s) is reached with very small buffers. It shows

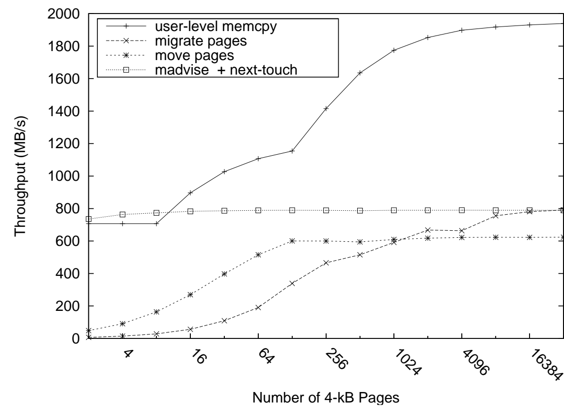



Figure 3: Migration and memory copy throughput comparison between NUMA nodes #0 and #1.

that only our *Next-Touch* implementation is able to migrate small buffers efficiently.

Overall, migrating pages appears to be much slower than copying data manually with `memcpy` both for small and large buffers. It explains why some people even consider copying data between different buffers and modifying the application pointers instead of actually using migration. We measured that the kernel copies data during migration at a 1 GB/s throughput. It is much slower than user-space copies due to less MMX/SSE-like optimizations. But the actual slowness of migration is also related to the management overheads that we detail in the next section.

### 4.2 Understanding the Overheads

We measured that setting a memory range as migrate-on-next-touch with madvise only costs about 600 ns plus 40 ns per page. Then the actual migration on next-touch costs about 5.2 $\mu$s per page. Since the actual memory copy throughput in the kernel is about 1 GB/s, it shows that only 1 $\mu$s (20 %) is needed to manage each page (handle the page-fault and then do the actual update of kernel structures such as the PTE).

On the other hand, `move_pages` shows a 72 $\mu$s base overhead and then requires 2.5 $\mu$s to handle each page migration (before copying at 1 GB/s rate). We found out that it is possible to reduce it down to 1.4 $\mu$s/page[3] but it remains much higher than the 1 $\mu$s management

[3] `move_pages` performance might be further improved in upcoming kernels, as explained at `http://lkml.org/lkml/2009/4/15/75`.

cost in the *Next-Touch* fault handler. The reason is that move_pages works on pages instead of PTEs. It is able to migrate many different types of pages, including shared mappings. Therefore, it has to isolate pages from concurrent process access during migration. Once the data has been copied, it also has to update the PTEs of all involved processes (not only the migrating one).

The initialization code of move_pages therefore drains pages out of the per-CPU *pagevec* lists so that they may actually be isolated from migration once they are in the main LRU list. This invocation of lru_add_drain_all() schedules a deferred work on each processor. It is actually responsible for the whole 72 $\mu$s base overhead of move_pages. However we do not understand why this cost appears to be linear with the number of processors in the machine (about 6 $\mu$s plus 4 $\mu$s per processor on our machine) while we expected this parallel operation to scale with satisfying performance under normal load. Further optimization may be needed here.

This result raises the question of whether all the move_pages complexity is actually required in most cases. Indeed, migrating shared mapping pages may not be interesting for many applications. It may then be interesting to look at a new migration primitive that would ignore complex cases such as shared mappings. We will discuss this idea further in the next sections.

# 5 Discussion

We discuss in this section several questions that have to be raised when designing and implementing a *Next-Touch* policy.

## 5.1 User Interface

Our implementation expects applications to mark contiguous areas as migrate-on-next-touch using the madvise system call and the new MADV_MIGRATENEXTTOUCH behavior. Some other interfaces such as set_mempolicy or mprotect could be considered but they work on VMAs and rely on setting static flags in the kernel structures. However, the *Next-Touch* policy has no reason to be defined on a per-VMA basis, and it must only be set temporarily since the status disappears after the actual touch. Moreover, the madvise interface only gives *Hints* to the kernel. It is thus possible to ignore it under some special circumstances such as non-migratable pages. For the record, SOLARIS uses madvise with the behavior MADV_ACCESS_LWP meaning that the next lightweight process will access the memory range heavily. The implications are therefore even less strict than ours since the SOLARIS kernel could even try to optimize some internal structures that are not directly related to memory migration.

Our implementation is page-based while a user-space implementation may migrate large buffers at once. Adding granularity information to a kernel implementation would require a new interface. VMAs cannot be used to do so since they may be merged/split by the kernel during many system calls such as mprotect. Actually, applications may enforce the migration of a whole segment on our page-based implementation by touching all pages. The overhead of this strategy is linear. And migrating pages enables more laziness and thus may reduce the actual copy overhead in some cases. Indeed, as the madvise overhead is small, applications may mark very large buffers as *Next-Touch* even if there is a chance that some pages are never actually touched, and thus never migrated for real.

Another question that needs to be raised is whether read and write accesses must be distinguished. Our implementation migrates pages in both cases, but some applications may actually want to migrate only in case of a write-access. Indeed, for instance, a single-producer-many-consumers model may need a privileged write access. Implementing a MADV_MIGRATEONNEXTWRITE might thus be interesting as well. Its implementation would be even closer to the existing *Copy-on-Write* code.

Finally, it is not clear whether applications may sometimes need to query the migrate-on-next-touch status of a segment, or even clear it. No such usage looks obvious in the context of high-performance computing. Clearing would be easy to implement using another madvise behavior. However, retrieving the status of pages looks harder. The move_pages system call is able to retrieve the location of a set of pages. We could imagine marking its return values with a special bit if a new flag MPOL_MF_GETNEXTTOUCH was given.

### 5.2 Implementation Details

The main drawback of our experimental implementation is that it only works with private anonymous mappings. Migration of file-backed mappings is not supported so far because it does not seems to be widely used in high-performance computing. However, we do not see any reason to not implement it. We do not support the migration of shared mappings either, because it is harder to implement since it requires to update all address spaces pointing to the migrated pages. It may actually be one of the reason why `move_pages` is slower than our approach.

Our implementation enforces the copy of the touched page into a new one even in case of a read touch. If a private page is still used by 2 processes because none of them modified it yet, the *Next-Touch* always causes its duplication as *Copy-on-Write* does. Both processes then keep their own private copy of the original page. This strategy does not break the semantics of memory mappings but it may slightly increase the memory consumption. Indeed, pages that are marked as migrate-on-next-touch may actually be duplicated earlier than with a regular *Copy-on-write* model. Our feeling is that this implementation has the advantage of not migrating pages that are used by other processes. It is not clear to us that some process should have a privileged access to a shared page regardless of the other processes using it. Our early duplication of pages on next-touch causes each process to keep their own pages locally as they wish. Moreover, since setting the *Next-Touch* policy is only a hint, the idea of ignoring it for shared pages has to be considered anyway. This idea goes with our proposal for a new migration primitive as explained in Section 4.2.

Another point that might need to be discussed is whether the *Next-Touch* flag should be stored in PTEs or in pages. One advantage of switching to page flags would be that they are more room for additional flags than in PTEs. However using page flags would also imply that shared pages are migrated as `move_pages` does. However, as explained above, it is not clear that it is the desired behavior. The idea of using PTE flags has the advantage of keeping the page-fault handler very similar to the *Copy-on-Write* handler (`do_wp_page()`). Merging our implementation into a more generic *Copy-on-Write* handler might even be possible.

Our implementation as well as the *Copy-on-Write* handler uses `alloc_page_vma()` to allocate the new page. The default behavior is thus to allocate a local page, except if the application sets a NUMA binding policy on the virtual region. It may result in funny situations where *Next-Touch* pages get migrated to another NUMA node than the one touching them. It is not clear whether this should be handled automatically by the kernel, since a valid application should have canceled the NUMA binding policy before enabling the *Next-Touch* policy. However, our `do_migrateontouch_page()` only checks whether the old page is already local. It might have to be changed into checking whether the old page matches the memory allocation policy.

The last question that may have to be raised is when the *Next-Touch* status of a page should be cleared. It looks obvious that calling `move_pages` should cancel pending migration on next touch since the application is trying to enforce the actual location of pages synchronously. However, it is less obvious for cases where the allocation policy is modified with `set_mempolicy` or `mbind`. Meanwhile, PTE modifications (for instance in case of `mprotect`) probably needs to maintain the *Next-Touch* flag. It raises again the question of adding a `madvise` behavior for cancelling a pending migration on next touch.

## 6 Conclusions

As NUMA architectures are becoming mainstream thanks to the spreading of HyperTransport and QuickPath technologies, affinities between tasks and data becomes a critical criteria for scheduling decisions. Dynamic applications such as adaptive mesh refinement with OpenMP threads have complex and irregular access patterns. The ideal thread and data distribution across the machine may thus evolve during the execution. Migration of data buffers therefore becomes a convenient way to dynamically maintain locality.

The Linux kernel has gained NUMA abilities during the 2.6 development but we explained that the existing primitives are mostly designed for static application behaviors. Dynamic parallelism requires more complex capabilities so as to take care of affinities between threads and data buffers dynamically and automatically. Requiring the application to pass the whole knowledge of memory access patterns down to the thread scheduler would lead to way too much development overhead.

The *Next-Touch* policy is a convenient way to implement the automatic migration of data buffers near their

accessing tasks. We feel that it may easily be applied to multithreaded applications by locating the application phases, where the thread-data affinities may change, or when a new OPENMP parallel section begins. We presented an implementation of this policy in LINUX and showed that it provides interesting performance improvements thanks to minimal initialization overhead, page-based granularity, and satisfying asymptotic throughput. Applying this strategy to high-performance computing application is under work and shows interesting result such as 100 % speedup on a OPENMP-threaded LU factorization.

Several key points have been discussed regarding the actual user interface that should be offered to applications and its internal implementation in the kernel. We also detailed the overheads of the existing `move_pages` system call and of our implementation. The former is still under optimization, but it still exhibits a large initialization cost due to its ability to handle complex cases. We thus raised the idea of adding a new migration primitive with improved performance thanks to relaxed guaranties and a more simple interface. Other ideas could be studied, such as offloading page copies during migration on DMA engine hardware [2]. We hope that these results will attract developers into working in this area.

## References

[1] Timothy Brecht. On the Importance of Parallel Application Placement in NUMA Multiprocessors. In *Proceedings of the Fourth Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, San Diego, CA, September 1993.

[2] Andrew Grover and Christopher Leech. Accelerating Network Receive Processing (Intel I/O Acceleration Technology). In *Proceedings of the Linux Symposium (OLS2005)*, pages 281–288, Ottawa, Canada, July 2005.

[3] Andreas Kleen. A NUMA API for LINUX, April 2005. Novell, Technical Linux Whitepaper.

[4] Christoph Lameter. Local and Remote Memory: Memory in a Linux/NUMA System. In *Linux Symposium (OLS2006)*, Ottawa, Canada, July 2006.

[5] Henrik Löf and Sverker Holmgren. affinity-on-next-touch: Increasing the Performance of an Industrial PDE Solver on a cc-NUMA System. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 387–392, Cambridge, MA, November 2005.

[6] OpenMP: Simple, Portable, Scalable SMP Programming. `http://openmp.org`.

[7] Nathan Robertson and Alistair Rendell. OpenMP and NUMA Architectures I: Investigating Memory Placement on the SGI Origin 3000. In Springer Verlag, editor, *Proceedings of the 3rd International Conference on Computational Science*, volume 2660 of *Lecture Notes in Computer Science*, pages 648–656, 2003.

[8] Christian Terboven, Dieter an Mey, Dirk Schmidl, Henry Jin, and Thomas Reichstein. Data and Thread Affinity in OpenMP Programs. In *Proceedings of the 2008 workshop on Memory access on future processors (MAW '08)*, pages 377–384, New York, NY, 2008. ACM.

[9] Samuel Thibault. A Flexible Thread Scheduler for Hierarchical Multiprocessor Machines. In *Proceedings of the Second International Workshop on Operating Systems, Programming Environments and Management Tools for High-Performance Computing on Clusters (COSET-2)*, Cambridge, MA, June 2005.

# Non Privileged User Package Management: Use Cases, Issues, Proposed Solutions

François-Denis Gonthier
Kryptiva, Inc.
fdgonthier@kryptiva.com

Steven Pigeon
École de Technologie Supérieure
Département de Génie Logiciel et
des Technologies de l'Information
spigeon@etsmtl.ca

## Abstract

The package manager and the associated repositories play a central role in the usability and stability of user environments in GNU/Linux distributions. However, the current package management paradigm puts the control of the system exclusively in the hands of the system administrator, a root-like user. The non privileged user must rely on the administrator to install the packages he needs, while having to deal with delays or even refusal. We think that *non privileged package management* is the solution to the users' woes. We show that not only non privileged package management has realistic use cases, but also that it is quite feasible. We examine several possible existing solutions and show how they cannot be satisfactory for the deployment of unprivileged user package management. Finally, we analyse the dpkg package manager and show how it can be extended to include safe, consistent, non privileged user package management. Amongst results, we present the conflict resolution rules to include multiple databases, to ensure system consistence and proper dependency management. We also present how to modify user environment initialization to include alternate install locations. We show, finally, the feasibility and usefulness of unprivileged user package management and how small the changes to be made to a package manager such as dpkg are.

## 1 Introduction

Non privileged user package management is not considered as an important use case in package management. Package management focuses mainly on security and system stability, relying on a centralized model where control lies in the hands of the administrator(s). This model, essentially the only one used in Linux distributions, relies on the implicit assumption that users cannot manage their environment in any meaningful way, except for minor tweaks and configurations, and that decisions regarding packages can only be taken by administrators. We argue that while this model has proved itself effective, there is room for the users to manage their environments beyond mere tweaks.

The problem of non privileged user package management does not present itself when the user is the owner and administrator of the machine but poses itself clearly when the user is but one of many users of shared workstations or of a multi-user server for which he does not possess administrative privileges. In this case, the user must ask the administrator for the packages he needs, and his request may very likely be denied, either because the package is against the local policies, because the package conflicts with other packages on the system, because it would affect adversely the other users, because it represents a security risk, or even because the administrator decides that the benefits to the user from adding the requested packages are not worth the effort. Whatever the reasons, the net result is that the user does not get the needed packages and is left at his own devices.

Being left at his own devices, the user will simply resort to installing the needed software from tarballs, downloaded from some location without authentication. Launching the `configure` script, by specifying install location (usually through the `--prefix` switch), building the software using the created Makefile, and modifying his environment variables, he will eventually succeed in locally installing the software.

The tarball approach suffers from a number of important drawbacks, despite being the standard for develop-

ment builds. First, one must deal manually with the missing dependencies of the software being compiled and installed. As most software do not quite fail building but merely disable features when dependencies are missing, it is every difficult for novice users—and even more advanced ones—to configure software correctly. Eventually, after fetching and building all needed dependencies, the user manages to compile, install, and to get the software to run correctly, but he still faces the onus of making updates by himself, going through the tarballing all over again at each new release.

It is our opinion that users should be able to locally install software through the much simpler process of using the distribution-specific package manager, thus obtaining all the benefits of centralized, trusted, and simple to access distribution-specific repositories—as exist for all major distributions—that provide packages with all their dependencies. In addition to the simplified installation process, the user should get updates automatically through the distribution's update manager, without so much intervention as a simple confirmation.

However, non privileged user package management is seemingly a complex problem, and one can oppose it several objections (that we discuss in Section 2.2) of which security is the most obvious and the most serious. Indeed, one cannot grant any user the right to install any package as it may affect adversely other users or even render the system inoperable. Delegating package management via a simple mechanism like the sudoers list both is dangerous and insufficient.

The correct solution is therefore, in our opinion, to use *fully relocatable packages* and to allow *non privileged users* to perform private installs in their home directories (or some other accessible location) through the package manager. This implies that there is a local package database for each user, and that the package manager must maintain coherence between the user's and the system's databases, being fully capable of detecting and resolving package conflicts.

In this paper, we study the problem of non privileged user package management. We outline the problem, as we see it, and present venues for solutions. The paper is organised as follows. Section 2 presents the proposed use cases as well as possible objections to the proposed model. Section 3 reviews the existing strategies to delegate package management as well as existing package managers, and discusses their shortcomings. In Section 4 we outline our proposed solution using dpkg and apt as example implementations. Finally, we conclude in Section 5.

## 2 Non Privileged User Managed Packages

In this section, we propose use cases for non privileged user package management and discuss possible objections to the extension of classical package management to include non privileged user operations.

But before continuing, let us present the definitions of the terms we will be using in the remainder of this text and that should not be ambiguous to the reader.

The *administrator* is a special user with access to all of the system's files and programs. Amongst other things, an administrator is a user that can manage packages on a system. The administrator is a user with root privileges. The *user*, on the other hand, is a user with no administrative rights. His privileges are limited to his personal files, programs made available to him, and he cannot install packages on the system.

In a virtualization context, the *host* is the machine or system that provides CPU and shared resources to one or more guests operating systems. The host has complete control over the guests it runs. A *guest* is a virtualized instance of an operating system running on a host. The guest can use the resources provided by the host, but in isolation from the other guests.

The *package manager* is the software suite that takes over the operations of installing, maintaining, or removing pre-compiled software packages in Linux distributions. Examples of packages managers are the Red Hat Package Manager (RPM) and Debian's dpkg. Selectable pre-compiled software packages are stored in one or many shared *repositories*.

A *maintainer script* is a script that is called during package maintenance whether installation, upgrade or removal. Maintainer scripts usually perform complex tasks such as creating users and groups, generating or removing application-specific configuration files, generate SSL or SSH keys, etc.

*Repositories* are storage locations containing a typically large number of software packages, which can be retrieved by the package manager. Repositories contain a certain amount of meta-data about the packages they contain. *Trusted* repositories contain digitally signed packages and are deemed safe (malware free).

### 2.1 Use Cases

The most difficult part, it seems, is to justify non privileged user package management as a valid and important use case for Linux-based computer users. While it may not seems *a priori* as an important use case, the proliferation of shared Linux workstations at work, in schools, and at home, warrants the question to be explored seriously. Indeed, how do we bring a superior user experience, better customization, and higher usability to users sharing computers, as in, for example, a computer science class or similar environments while minimizing the effort from the system's administrators?

Non privileged user package management may be the solutions to a number of administrative woes. Consider:

1. **Reduced workload for administrators**. Administrators would not be pressed to install user-requested packages. The current installation process asks for the administrator to personally intervene to install the packages, but only after having investigated whether or not the packages threaten the system in some way, and after deciding whether the user's benefit is worth the effort of installation.
2. **Reduced delays for users**. Users in large machine parks would not need to wait for their requests for given packages to be processed and possibly denied by the administrators. Users could install packages in their environment right away, to no detriment to other users.
3. **User Empowerment**. Users would gain direct control over their work environment without impeding on the other users. Users would be able to configure and streamline their environments for maximum usability and productivity, having all the application *they* need, rather than subset of software pre-established by the administrators.

What we advocate is, in essence, a shift away from the current centralized, somewhat totalitarian, management model to a distributed, delegated management model where users can setup their own environments, subject to the system's policies, while minimizing impacts onto other users.

As of today, adept users circumvent the impossibility of installing their own packages through the package manager by making local installs from tarballs. As already described, this is a tedious and error-prone process. Additionally, packages installed from tarballs are not recognized by the system's package manager and therefore are not automatically upgraded. Ideally, user-installed package should provide the same facilities than system packages, that is, minimizing installation effort, reducing considerably the risk of setup and configuration problems, while increasing maintainability through periodic and automatic package upgrades.

### 2.2 Objections

Even though "user empowerment" is a nice idea, one may object to non privileged user package management by raising a number of objections. A possible list of such objections could be as follows:

1. **Delegation using sudoers**. One could allow users to access the package manager without really giving them root privileges by adding them and the specific command to the sudoers list. However, allowing users to use the package manager to install global packages is tantamount to giving them root access as they are free to install whatever package they want, including broken or malicious packages. Even well intentioned, they can install software that affects adversely the other users and they can make the system inoperable as a whole. This would be prevented by user-managed local installs, since unprivileged users are granted permission to install packages from a possibly limited set of packages (defined by administrator policies) and only in their own user environments; and thus installed software runs with the users' privilege levels. Decisions on how to resolve packages conflicts are made by the user—but never to the detriment of the system. We will this discuss this issue in detail in Section 4.5.
2. **"Root" packages**. Packages containing software that must be run with privileges level higher than a normal user, such as kernel modules, services using protected resources like port numbers under 1024, etc., clearly cannot be left to user management. Therefore, there must be configurable policies built into the package management system to prevent users from installing such packages, and this issue cannot be removed by non privileged user package management. However, critical packages can be tagged in the repository as such and are therefore prevented from being installed by a non privileged user.
3. **Security of repositories**. Trust management for packages is a major issue. One cannot allow the

addition of arbitrary repositories, nor allow the installation of packages of unknown origin. Indeed, if one allows user-managed packages, does it imply that one also must allow user-managed repositories as well? If so, it means that the repositories allowed must be trusted repositories (for example, distribution-specific canonical repositories and their trusted mirrors) to prevent users from adding potentially harmful packages from arbitrary repositories. A system-level policy must be used to allow or disallow users from installing packages from unknown repositories or from a local directory.

4. **Redundancy and Disk Space**. A given package may be installed by more than one user, resulting in multiple copies of the package's files (including configuration) that must be maintained. Moreover, if the user can access his account from machines with different architectures (via NIS and home directories over NFS, for example), the package may be installed for each architecture, if available. Unless users lock the package version (for reasons their own), these multiple copies should be updated correctly whenever system-wide updates are launched, resulting in extra computational cost. This is mitigated by using group-level installs (which we will discuss later on in Section 4.2) and by the fact that the number of user-managed packages on a given system is expected to remain relatively modest compared to the total number of packages installed on the system. Moreover, disk space quotas would sufficient to prevent users from using an inordinate amount of disk space, at the detriment of other users.
5. **Users ignore policies**. The administrator may want to prevent users from installing software such as games, BitTorrent software, etc. As simply informing the users of one's wishes is not sufficient to prevent them from installing forbidden software, the policies must be enforced in the package manager system itself. Policies define what is an acceptable package (and its provenance), and where it can be installed. We discuss policies in sect 4.6.
6. **Users don't know what they're doing**. While it may be true that not every user is familiar with the intricate details of package management, and that users may not understand the impacts of installing a given package, the package management system must prevent them from causing damage to the system and their own environments by applying its rules of conflict resolution.
7. **Packages and Repositories must be modified**. It is true that a major reworking of repositories, packages, and software they contain is needed to make user-managed packaging systems possible. Every package has to be tagged with the specific set of privileges required for its installation and use, but more importantly, has to be made completely *relocatable* so that user installs can be completed. Relocation means that maintainer scripts and meta-data, other than system-provided, must be rewritten in order to accept arbitrary locations for the packages—it may even imply change in the software itself so it can adapt to new locations. It also implies that the user environment setup scripts must be modified to ensure that correct environment variables, paths and priorities for packages are set.
8. **Package Managers must be modified**. The package management software must be modified as well to take into account the new meta-data found in repositories and packages. More importantly, package management software must be modified to manipulate multiple package databases and deal with package conflicts between the non privileged users installs and the system's packages, ensuring consistency of the system as a whole. In particular, conflict resolution rules must be extended to include several databases. Far from being impossible, we show that, indeed, conflict resolution rules may be extended to include several package databases.

Most of the preceding objections can be lifted, either totally or at least greatly mitigated. For example, one could use the potential explosion in needed disk space as an argument against user-managed packages, but in reality, this is not a problem given that there are other facilities within the operating system to limit a user's disk space usage (which would already be in use in a multi-user system), and that, for all intent, the cost of the disk space itself is negligible.

The only serious objection to user-managed packages is the amount of work needed to convert repositories and modify package management software. What would be an argument against the modification of packages is but an argument about workload, not about the philosophy of non privileged user package management itself. The amount of work needed to modify the repositories for added security and to allow truly relocatable package is not small, but may not be as important as first thought. We discuss the necessary changes to packages in Section 4.4 and 4.7.

As for package management software, we show in this paper that the changes are likely minimal and that conflict resolution rules may be extended to non privileged user package management as well, as we will show in Section 4.5.

## 3 Existing Solutions

While it is our opinion that no exact solution to our problem already exists, in this section, we consider the different solution venues. First, we discuss VServer, a system-level virtualization kernel modification that allows one to create distinct virtual copies of the kernel on a single machine. We discuss PackageKit, a package management API and GUI. We then discuss widely used package managers such RPM (Red Hat) and dpkg (Debian). For each, we explain why they are not exactly solutions to the problem we are interested in, as stated in Section 2.

**VServer** is a modification to the Linux kernel to allow system-level virtualization, enabling the computer to run several guest virtual instances of the same host kernel. This means that one can setup several isolated instances of the same Linux distribution, or even different distributions provided they use the same kernel. Each instance can be used by different owners, each enjoying root privileges but unable to influence other instances. VServer is therefore used to share the same hardware between users with different needs, as each user can install his own customized environment.

Delegating system management using virtualization is a too heavy-handed answer to our problem. Using VServer, the host system's administrator relinquish full control to his users (the administrators of the guests) as to what is installed in their instances. The administrator can still control which repositories the users can use, but this means reducing the customizability of the guest systems. Obviously, the VServer kernel extension provides no solutions for redundancy, as each guest has its own instances of files. However, redundancy can be limited somehow by hard-linking files across guests, but this also limits the freedom of the users to choose their packages, while adding the possibility of conflicts. Redundancy is also mitigated as, very often, but not always, VServer is used to create several server-type guest environments, which are, almost by definition, much lighter—in terms of the number of packages installed—than desktop environments.

Since a VServer guest installation is a complete Linux installation, it gives its guest administrators full control on which packages are installed in the guest system (provided they are so allowed by the host administrator). Inside the guest system, the problem of allowing users to install packages is still complete. Users having access to the guest system but that are not administrator for that guest cannot install packages, and the guest administrator cannot delegate this right to his users in a way that does not raise the objections stated earlier. The multiplication of guest installations may also mean that the host administrator has an increased workload as he must now provide not only for the host, but also for the various guests' environments and users. Since all guests could potentially be very different distributions (but sharing the same kernel) the lack of uniformization in each environment clearly does not simplify the host's administrative work.

**PackageKit**'s primary goal is to standardize package management across distributions. PackageKit abstracts the complexities of the various existing package managers by offering a consistent interface across the various distributions that already use it. It is composed of a privileged dæmon, `packagekitd`, several distribution-specific back-ends and GUIs. The front-ends communicate with the dæmon using the D-BUS desktop integration protocol, which in turn, delegates the actual package management to the distribution-specific back-end. By design, PackageKit deals with some of the objections we raised to the traditional package management strategy.

The use of a privileged dæmon means that users can be granted or denied its use in a secure fashion using privileges set by the PackageKit administrators. However, PackageKit is still limited by the inherent (in)capability of the underlying package managers as it uses them as back-ends to complete its tasks. Improper configuration of PackageKit opens the door to the same kinds of problems encountered when using delegation through sudoers. If we suppose that user requests can be filtered, the filters become a critical element of the package management system. Malicious—or simply unskilled—users could otherwise install packages that results in system-wide damage. Such policy filters are not implemented in PackageKit as of now; however, due to PackageKit's design, there should be no major obstacle to adding this feature [1, 2].

This being said, it is worthwhile to note that Pack-

ageKit's design is not fundamentally incompatible with non privileged package management, even though it currently offers no direct support for it. It would be indeed possible to do so, since a contribution to PackageKit allows the local install of specific audio/video codecs to a user's directory.

**RPM**, the Red Hat Package Manager—one of the two major package managers along with Debian's dpkg—implements package relocation in two ways, although the original intend was not to allow regular users to perform private installs [3]. The first type of relocation, using the `--prefix` switch, causes all of the package's contents to be installed in the specified directory, including configurations files that would normally be installed in locations such as `/etc/`. The `--prefix` option provides support to the maintainer scripts through the environment variable `RPM_INSTALL_PREFIX` which contains the path to the alternate location. Maintainer scripts can use the variable to detect relocation and use the new install location.

The second method, path replacement, is a simple technique that allows to install a subset of package's files in alternate locations and uses the `--relocate` switch to do so. For example, specifying `--relocate=/etc=$HOME/etc` would cause the package manager to substitute `/etc` for `$HOME/etc` wherever it occurs and, accordingly, all files that would have been installed under `/etc/` are now occupying the same relative location under the specified alternate location. Used without care, the `--relocate` option can lead to non-functional packages, as it offers no means to signal relocation to maintainer scripts.

RPM allows for relocating the package database, either by using the `--dbpath` switch, which specifies an explicit location, or by using the `--root` switch, which specifies a new root directory under which the relative location of files is preserved. However, relocating the database may necessitate the use of `--nodeps` and `--noscripts`, two switches bypassing important features of the package manager. Using `--nodeps` will cause the package manager to skip dependency checks as the package manager will refuse to install packages that have missing dependencies, either because they are not installed or because relocation prevents the package manager from detecting them correctly. The `--noscripts` may be necessary for non relocatable packages as their post-install scripts are not using relative, relocatable, paths.

While this gives users the opportunity to install packages in their own directories and manage their own databases without root privileges, the RPM approach has a number of serious drawbacks. First, relocation using path replacement may cause the installed software to fail as it may not be able to find the files it needs. Were the packages relocatable, they would be able to look for the configuration files in the new locations rather than in the default locations; something that is, as of now, provided neither by the RPM package manager nor by the packages themselves. Second, relocating a RPM package will likely cause it to be unmaintainable. RPM keeps track of packages that are installed, and where, but if the maintainer scripts are not able to manage relocation, fully automatic package updates will not be possible, even with user package databases. Relocation can also break dependencies when other packages depending on the relocated packages are installed as the installed dependencies are detected, but default locations still assumed. Third, and lastly, RPM is unable to maintain the consistency between the system's database and the user databases because it merely allows relocation of the database; not multiple databases, thus potentially causing the user packages to break whenever system-side dependencies are modified. The user must, each time, manually reconfigure his packages to adapt for system-side changes, a time consuming and error-prone process.

RPM offers enough options to allow non privileged users to manage their own databases and packages, but as we explained, packages cannot be maintained automatically and may be installed in a non-functional state that may require manual effort to reconfigure properly—if possible at all. In addition, none of the user's environment variables will be updated correctly, requiring further manual intervention.

**Dpkg**, Debian's package manager simply does not allow non privileged users to maintain private databases. Dpkg has no facilities to allow relocation, save for installation inside "chrooted" environments. Maintaining a local database and relocated packages will cause dpkg to use the `chroot` system call before launching maintainer scripts. However, `chroot` is a privileged command unavailable to normal users. Additionally, dpkg's maintainer scripts are often complex, and invariably assume `/` as the path prefix.

While adding relocation capabilities to dpkg was discussed [4], there are no serious plans to implement the

feature, as deemed an unimportant border case [5].

So, in essence, neither PackageKit nor VServer can be used to allow non privileged user managed package. PackageKit offers an abstraction to package management in order to standardize the package management API, relying on distribution-specific package management software to perform the actual management. PackageKit may be promising because it may allow the addition of policies to package management, but still lacks the possibility to relocate packages as it is fundamentally limited by the underlying package manager. VServer, on the other hand, only serves to create a nested version of the problem where the management of guest installs is delegated to their respective administrators (which are themselves subordinated to the host's administrator) and where the users of each guest are as normal users on a normal distribution, that is, unprivileged and unable to manage their own packages.

The principal package managers' limitation is that they expect their databases to be in a unique preassigned location and cannot deal with multiple, possibly conflicting, databases. Therefore, to minimize risks for the system, the databases are set to be accessed with root privileges only. Package managers may offer means to relocate the database, but they are limited to a simple relocation. If a user uses these features to create a database within his home directory, he could, theoretically, manage his own packages. But we saw that doing so forfeits most of the package manager's advantages such as automatic updates, conflict and dependency resolution, and automatic configuration.

Parts of the problems with relocation and automatic package configuration lie with the packages themselves; as their install scripts are unable to provide for arbitrary relocation of the files they contain, and even worse, not all software is capable of being relocated. This means that the users must perform the necessary configurations by hand, if allowed by the software at all.

In the next sections, we outline what we think would constitute a viable solution addressing all the objectives (and possible objections) stated until now, in particular package relocation, conflict and dependency resolution using multiple databases.

## 4 Proposed Solution

The minimal changes made to a package manager to accommodate unprivileged user package management necessarily depends strongly on the package manager one wants to modify. The first important modification is to allow the package manager to use user-specific databases in addition to the system database. The second is how the packages themselves are modified to allow fully relocatable installs. The third important modification concerns the users' run-time environment that must be set up correctly so that the users' packages are correctly configured. All modifications must be minimal, and, if possible, hidden to the user. Furthermore, a good solution would also be compatible with the File Hierarchy Standard (FHS) [6].

To outline our proposed solution, we will use dpkg as an example, especially that dpkg does not currently allow local installs at all. We will present how to extend dpkg to include unprivileged user package management.

### 4.1 Package Databases

The package manager must be extended to account for user-managed package databases. The user must be able to create a database for himself without further privileges than he already has. The creation of such a database would be automatic whenever the user invokes `apt-get install` or `dpkg -i` without root privileges. The location of the user's package database would be hidden in a dot file, or even a file within a dot directory. The database would maintain the list of packages installed by the user as well as their locations. Note that now, default location takes quite a different meaning. It can mean the usual FHS or a relocated file hierarchy relative to the user's home directory, depending on the privileges used during installation.

Without any special rights, the user could now (possibly with the help of some desktop applet) perform automatic and periodic updates of his installed packages. As packages are updated by either the user or by the system's administrators, dependencies conflicts must be resolved in an intelligent fashion. We will discuss conflict resolution at length in Section 4.5.

### 4.2 Managing Local Databases

The default user database location should be sufficient in most cases, but it could be explicitly relocated. By default, privileged user would use the system default database, location, and install paths, which are relative

to /. For an unprivileged user, the hierarchy would be relative to his $HOME. In addition to installed packages and their location, the users' databases must include a list of package-provided setup scripts that allow each package to configure its environment properly.

The package manager must now support at least a few new commands which we will outline. Let those new commands be accessed through the added program dpkg-env. Note that in addition to dpkg-env, one would modify dpkg itself (and any front-ends, such as apt) and the necessary modifications will be made clear in the following paragraphs.

Through dpkg-env, the administrators and users will be able to perform local database management and environment setup. The first important set of commands would allow the addition, management, and removal of local user (and group) databases. For example, a user can invoke dpkg-env --new without privileges to create his own local package database. If it already exists, the command succeeds. One could also invoke dpkg-env --new --user username with sufficient privileges to create a local database for user username (it would also be automatically created for the user invoking non privileged package management for the first time). Invoking dpkg-env --new --group groupname with sufficient privileges would create a group-specific database located in /var/lib/dpkg/groups/ or, if specified otherwise, in some other location. The packages themselves would be installed in /var/lib/dpkg/groups/$GROUP. The important nuance with group installs is that all users that are member of this group will inherit the group's packages and environment at the login. A corresponding --remove command would destroy a database after launching deinstallation of all related packages, preventing unusable packages as well as loss of disk space.

### 4.3 Configuring the environment

The next important command relates to the user's environment setup. For example, dpkg-env --setup would scan the system database, the groups' databases and finally the user database to set up correctly the environment for every installed package. The system database would provide only default location installation, so all packages in the system database can be swiftly dealt with by using the default environment settings (as it is currently done on Debian and related distributions). For groups and users, the procedure is similar. The database is scanned and the package-specific environment setup scripts are called. If a package offers (or needs) no such script, it inherits default group or user location through $PATH and $LD_LIBRARY_PATH. If a package does offer such a script, it is executed and the changes made to the environment are propagated to the session.

The problem with dpkg-env --setup is that it must be called just after a user logs in and before he begins his session, so that he can use all the available packages correctly, including packages such as window managers or interface extensions, for example. Currently, it can be done in three different ways: using PAM, using shell-specific profile configuration files, or using .xsession.

The Pluggable Authentication Module (PAM) is the standard Linux user authentication mechanism. It includes modules that are capable of launching actions when the user logs in, before shell or session scripts are executed. However, there are currently no simple way of modifying the environment variables from PAM. PAM uses the pam_env module that allows the user to add variables to his environment through the user's .pam_environment file. However, the file only contains a static list of environment variables and their values, and as such, cannot be used to run the package-specific scripts needed to configure the environment properly. The possibility of calling scripts from PAM might be an interesting addition to the session initialization process.

Using shell profile scripts is not universal. The scripts are recognized and executed by Bourne compatible shells (sh, bash, ksh, zsh, etc.) when they are invoked. The .profile is ignored until a shell is launched, and so is useless if the user logs in via a graphical interface that does not invoke a shell. This can be solved by using the system-wide X session configuration file. In both cases, it merely suffice to append a call to dpkg-env --setup to the files, and the program is run using the user's credentials with no need for special privileges.

### 4.4 Package-Specific Environment Setup Scripts

To support complete relocalisation, it is likely that the package will need, in addition to a field that tags it as

relocatable or not, a script that prepares its environment once installed so that it runs properly. If the package requires nothing more than the default (relocated) locations, the script is not required, as `dpkg-env` would already provide the correct values through `$PATH` and other environment variables. The packages might, however, require a special setup.

While executable programs need little more than modifying `$PATH` to be available, it is not so with all packages. Native libraries can be found with `$LD_LIBRARY_PATH` setup so that the proper precedences are respected. Manual packages can already install manpages in alternate location, provided that `$MANPATH` is set or `--manpath` specified. Desktop elements, such as icons and menu items can be installed and found through various environment variables that hopefully complies with the Base Directory Specification, already in use in major distributions [7, 8].

Configuration files are often expected to be found in the `/etc/` directory. Programs using absolute path to the configuration files will need to be modified to access them through environment variables and relative paths. Dynamically updated data, such as logs, are usually found in `/var/`. Just as with configuration file, should this directory be relocated, the packages must access it through the environment set up by `dpkg-env`. However, care must be taken to ensure that the relocated directory grants write access only to users that are entitled to use the package.

Finally, static data such as images, sound effects, etc, are usually installed in `/usr/share` and do not require much caring for. They can be relocated without harm provided that the access rights are set up properly for their intended users and that proper environment variables are set so that applications can find them.

### 4.5 Conflict Resolution

If both users and administrator install packages independently, conflict is bound to happen sooner or later. By conflict, we mean whenever the situation comes up where the user's environment is affected by a change in the system's environment or when the user installs conflicting packages. Rules must be applied to decide what will be the course of actions should conflict arise. In the next few paragraphs, we will present conflictual situations and how to resolve them, while ensuring the system's integrity as a whole, possibly to the inconvenience of the user or group. The main conflict resolution rules would be as follows:

1. If a package $x$ is user-installed and subsequently system-installed, the package manager should proceed to uninstall the package from the user's packages while leaving his configuration files untouched. Same would occurs if the same package $x$ is updated system-side and now meets the user's version.
2. The package $x$ is installed system-side, but a user has a different version $x'$ that depends on user-installed package $y$. If $x$ and $y$ are incompatible, accept or deny the removal of $x'$ and $y$ from the user's packages and system-side installation of $x$ and $y$.
3. If a system-side package $x$ is installed, and that $x$ is user-installed and is depended upon by a user-installed package $y$, remove both $x$ and $y$ from the user's packages and install $y$ server-side.
4. If a system-side package $x$ is installed or upgraded and the users has a package $y$ that depends on a different version of package $x$, $x'$, also user-installed, and that $y$ is incompatible with $x$, accept the removal of $x'$ and $y$ from the user's packages and proceed with the system-side installation of $x$, or fail the install or upgrade of $x$.
5. If a system-side package $x$ is installed or upgraded while the user has package $x'$ depended upon by user-side package $y$, and $y$ is incompatible with $x$, accept the removal of $y$ from the user's packages, or fail the install or upgrade of $x$.
6. If a user-side package $x$ is upgraded and an older user-side package $x'$ is installed, proceed with the install of $x$.
7. If a user-side package $x$ is upgraded while depended upon by user-side package $y$, but $y$ (and newer versions of $y$) is incompatible with the new version of $x$, confirm the upgrade of $x$ and the removal of $y$, or fail the upgrade of $x$.
8. If a user-side package $x$ is upgraded while depended upon user-side $y$, and the new version of $x$ is incompatible with the current version of $y$ but a newer version of $y$ that is compatible with the new version of $x$ exist, upgrade both packages after confirmation.
9. If a user tries to install a root package $x$ his request is denied and the package manager bails out.

In the previous rules, the same applies when 'user' is changed for 'group'.

Unsurprisingly, the rules are reminiscent of the rules already existing in current package managers, except

that they are extended to include both system-side and user/group-side packages. As with system-side only package management, the administrator is proposed choices and must accept or refuse changes to the packages based on information provided by the package management engine. The difference is that now, the administrator can decide to forgo upgrade of a package because it breaks a user-side install, and if he does break user-side packages, he does so knowingly.

## 4.6 Policies

Currently, neither dpkg nor apt provide for policies. As discussed earlier in Section 2.2, it may be wise to prevent users from adding repositories and install certain types of packages, regardless of their origin. Preventing users from adding repositories greatly reduces security risks, as one cannot ascertain the trustworthiness of these new repositories. The same applies when the user tries to install a package from one of his directories.

If adding repositories is allowed, the data must be stored in a configuration file in the user's home directory, preventing effects on the system as a whole. Adding a repository asks for authentication, and this can be done using a white/black list system, where trusted repositories are listed. Such lists would be distribution-specific.

Filtering by package type (either categories, sections or tags), or even by repository, may help the administrators to ensure efficient usage of facilities. Software likely to disrupt work environment, such as games, BitTorrent software, resource-hungry applications, or software with unwanted licenses, may be blocked by administrator fiat. The granularity of the policies could be very coarse (repository level), coarse (package class level, tag sets), or even fine (a specific package with a specific version).

This limited type of policy management can be implemented by adding a separate module to offer the administrator the possibility of editing policies. Categories would include policies for all unprivileged users, for specific unprivileged users, for all groups, for specific groups. The package manager would simply access the policies database to determine whether or not a given operation can proceed. We think that the modifications to the package manager are minimal since it suffice to verify if the operation can be performed by checking against policies.

In Section 2.2, we also noted that the disk space usage problem can be mitigated (or even eliminated) by the standard quota facilities, and so we believe that it is unnecessary to modify the package manager to track a user's disk space usage. It would however need to test if sufficient space is left to perform the desired operations.

## 4.7 Modifications to existing Package Management Software

Let us pursue with dpkg as an example package manager and present the modifications needed to make unprivileged user package management possible.

The first step would be to modify the tools used to produce packages. In the case of debhelper (the main tool used for the creation of packages for dpkg, which provides a set of scripts to handle various repetitive tasks), for example, most of the modifications are contained within the default behavior of environment setup scripts. For the vast majority of packages, it will suffice for the setup script to add the package's file locations to `$PATH` and `$LD_LIBRARY_PATH` through a helper script API that provides `sh`-like functions to ensure correctness and avoid multiplicities. One would then simply rebuild the package with the added setup script, provided that the software contained in the package is relocation-aware.

The dpkg command line needs to be extended to support multiple database paths. By default, dpkg would be able to locate the system and the invoking user databases. How operations are performed would depend on the access rights provided on invocation: root access would affect the system database while unprivileged invocation would affect the user's database. Of course, it must be possible to specify explicitly operation mode regardless of current privilege level.

The most important internal change to dpkg will be to include install location to the package name and version to the database. Multiple versions of the same package can exist simultaneously in different locations and dpkg must be able to manage them correctly and independently.

**Conflict resolution** must obey the resolution rules presented in Section 4.5, with the effect that system-side packages have higher priority than user or group-installed packages, as the goal is to keep the system consistent, even if it means breaking a user's environment.

Therefore, to apply system-side conflict resolution rules as stated, dpkg must access and read *all* databases. By doing so, conflicts can be resolved, correct updates of users environments can be performed, and ensure the system is left in a stable configuration, even if it means that some users may have some of their packages upgraded or removed. Ideally, package removal due to conflicts must be notified to users. To apply user-side conflict resolution, only the system and the invoking user's database must be read and verified.

**Maintainer scripts** that allow correct package relocation can be instructed of the new location via an environment variable or a direct argument; in either case dpkg must be modified to provide the correct value to the maintainer scripts for install, updates or removals. The current version of dpkg uses the `chroot()` system call before calling a maintainer script to provide it with the correct relative location and while this feature is useful by itself, it would be activated only through the explicit use of the `--root` command line option.

**Apt** would need to behave correctly when called by a non privileged user. This means it would access the calling user's database or bail out elegantly if this is not possible. This is needed to ensure that invoking `apt-get install` functions correctly, installing packages in the user's environment. Apt must locate automatically the relevant databases, searching into user-configured or standard locations. User-configured locations could reside in `$HOME/.apt.conf`, which would be read each time apt is invoked. A personal `source.list` must also be locally stored and checked by apt to allow a user to maintain his personal repository list (while subject to system's policies). This file should use the same syntax as the global `source.list` file.

To test for a relocatable package, it suffice to add a boolean field that indicates whether or not the package is relocatable, allowing apt to bail out gracefully if a non-relocatable package is installed with insufficient privileges. Since in Debian-like repositories meta-information about packages can be copied in the repository index, it would be easy to make apt warn the user about the non-relocatable nature of a package even before downloading. This functionality must therefore be included in GUI-based front-ends like Synaptic to help users manage their packages.

Policies for repositories and packages available to the user also have to be included at this level. There already is a configuration file, `/etc/apt/preferences`, that enables the system administrator to pin particular packages to inhibit changes. This configuration file could be extended with similar syntax to include information about the prioritization and exclusion of repositories. For example, fields such as `Rep` and `Rep-Priority`.

The package manager (and its front-ends) must also be able to prevent the installation of a package with untrusted origins. A user should not be able to install a package, should the administrators decide so, from a non authenticated location such as one of his directory. In most distributions, the repositories already contains the quasi-totality of safe packages, so the need to use untrusted packages is quite lowered—the only possible exception would be development tarballs corresponding to packages too recent to have made their way to the trusted repository.

## 5 Conclusion

In this paper, we have presented the problem of non privileged user package management. In Section 2 we presented the proposed use cases and possible objections. The use case we stressed most is when the user shares workstations in a work or school environment and he does not have sufficient privileges to install packages he needs and therefore has to rely entirely on the system's administrators, which may lead to unacceptable delays or even plain refusal. We presented, in Section 3, the existing solutions to the problems and why they were not quite satisfactory. In particular, we discussed the current package manager and how they fail at providing all the tools necessary to make non privileged user package managing possible.

Finally, in Section 4 we outlined the various modification to be brought to package managers to make non privileged user package management possible, in particular conflict resolution rules and how dpkg, taken as an example of package manager, should be modified to accommodate our proposal. We discussed policies that would be needed to ensure the system's stability as a whole. We also showed that setting up the user environments at login would be rather simple despite the lack of standard initialisation procedure across Linux distribution. Finally, we think that we made clear that not only non privileged user package management would be beneficent to users, it is also quite possible to implement

using relatively little effort compared to what one might have thought initially. Having shown the feasibility of non privileged user package management, the next logical step is to proceed to implementation, which is the object of future work.

## References

[1] *The PackageKit Package Management Front-End*, http://www.packagekit.org

[2] *The PackageKit Source Repository*, http://www.packagekit.org/pk-faq.html

[3] Edward C. Bailey, *Taking the Red Hat Package Manager to the Limit*, Red Hat Inc, 2000. Chapter 15, http://rpm.org/max-rpm/ch-rpm-reloc.html

[4] Kenneth Arnold, *Relocatable Package Development proposal*, Dpkg's Wiki post, http://www.dpkg.org/dpkg/RelocatablePackages

[5] Daniel Burrows, *Re: Non-privileged user package management*, Gmane forum message, http://permalink.gmane.org/gmane.linux.debian.apt.devel/15021

[6] Rusty Russell, Daniel Quinlan, Christopher Yeoh (eds), *The File Hierarchy Standard, V2.3*, http://www.pathname.com/fhs/pub/fhs-2.3.html

[7] Waldo Bastian, Francois Gouget, Alex Graveley, George Lebl, Havoc Pennington, Heinrich Wendel, *Desktop Menu Specification, V1.0*, http://standards.freedesktop.org/menu-spec/menu-spec-1.0.html

[8] Waldo Bastian, *Base Directory Specification, V0.6*, http://standards.freedesktop.org/basedir-spec/basedir-spec-0.6.html

# GeoDNS—Geographically-aware, protocol-agnostic load balancing at the DNS level

John Hawley
*Linux Foundation / Kernel.org*
warthog9@eaglescrag.net

## Abstract

The Open Source community has grown from a series of small projects hosted from anywhere that was convenient, to a globally distributed set of mirrors providing content to every region of the planet. While there has been an outpouring of support in providing mirrors to every possible facet of the Open Source community by independent entities, a problem has arisen in how to efficiently load-balance across this global infrastructure, not only for the benefit of the mirrors, but for the users as well. There are a multitude of solutions currently available to try and cope with the issues of load-balancing including physical load-balancers like squid, protocol-specific redirection like mod_geoip in Apache, and full-scale commercial content distribution like Akamai. For most situations, the commercial solutions are far outside of reach and may necessitate the removal of existing infrastructure. Things like squid require, for practicality, all of the machines to be in a single location or for a single location to provide the sum total of the bandwidth available. Lastly, the protocol-specific solutions like mod_geoip work well for their own protocol, but leave other services like rsync, ftp, git, and svn to fend for themselves—assuming that the protocol even supports redirection. Most do not.

GeoDNS is the idea of taking an incoming DNS request, doing the geographic look-up at the request time, and returning different results based on the incoming IP address. This particular approach, taken by several DNS servers including bind-geodns, powerdns, and tinydns (with patches), allows geographically diverse mirroring infrastructures like `Kernel.org`, Wikipedia, and many other sites to direct users seamlessly to an appropriate server. This helps distribute the system loads across the entirety of their mirroring infrastructure. This protocol-agnostic approach is more universal and simpler for end users to handle by making seemingly hard choices transparent to them.

## 1 Internet—20th Century tech solving 21st Century problems

The Internet has existed for four decades. It has evolved, grown, changed, and adapted from its humble beginnings as Arpanet to what it looks like today, with TCP/IP, IPv4, and IPv6. Though it has changed radically from where it started, the Internet's fundamental building blocks, TCP/IP and DNS, have changed very little and they continue to serve the Internet well. Though while they are the bedrock of the Internet, and are serving the needs of billions, small changes can be made to them to greatly extend their usefulness and continue to serve the Internet for several more decades.

### 1.1 International Growth

The Internet started as a small research project out of the Advanced Research Projects Agency (ARPA) of the United States Department of Defense. It initially spanned two nodes on the California Seaboard, and has grown to include on the order of 1.596 billion users[1] worldwide. The United States boasted, for a number of years after the inception of the Internet, the largest amount of capacity to host sites and projects connected to the Internet. This meant that a great deal of the content of the Internet was accessed by going to a single point, likely in the US, where the servers would be the best connected and provide the most good to all.

However, this single-point-of-connectivity model was unsustainable with the amount of growth outside of the United States. This growth far outstripped the capacity of intercontinental links, including satellite relays and undersea cables, as well as having significant cost and performance considerations. A growing pressure externally from a growing Internet population drove the need

[1] `http://www.internetworldstats.com/stats.htm`

for content to move from a centralized single provider to a more distributed localized content distribution, with mirrors of data forming all over the world and content distribution companies such as Akamai[2] stepping up to help fill this demand.

### 1.2 Mirrors, Mirrors everywhere, but not a drop to drink

As with many solutions to a problem, you may solve what is immediately needed, but it in turn creates a new problem that needs to be solved. This new-found infrastructure for mirroring and content distribution provided one such situation. Originally the Internet was intended to provide a good means of dealing with content as it existed in its centralized locations. When you went in search of content based on a URL, it was expected that the look-up would only involve a single unique system, whether this was a individual server or a group of them within close proximity to each other, providing the content. However with both the explosion of the Internet and the subsequent need for mirrors to become more localized to the users seeking the data, the need for changing this philosophy became apparent.

A simple approach initially was, and predominantly still is, employed by providing a user with a listing of the mirrors available, usually grouped by the server's country of origin. While this works, it makes the user interfaces difficult and requires that the user make an educated guess as to which mirror is likely closest to them and will provide the fastest service. These decisions can be quite difficult; it has been found for many Canadian users that it is faster to go to a mirror based in the United States as opposed to a Canadian one due to the existing backbone and routing infrastructure present in much of Canada.

Additionally, load-balancers are available to help act as a director to content. Typically, however, load-balancers act as a man in the middle (so to speak), providing a single point of entry and exit for a cluster of machines. This cluster typically resides in a single geographic location, and the load-balancer itself is a limiting factor in how much content can be distributed from these machines, as the load-balancer acts as a bottleneck to the cluster. This works particularly well for single sites, but it does not work when the machines that need to be load-balanced are geographically disjoint, though this provides high-availability with its load-balancing.

To help both high-availability and physically disjoint systems, things like round-robining in DNS can be used. This helps—in particular this alleviates a need for a user to make an explicit choice in server; however, it has a downside in that it is highly dependent on the implementation of the DNS look-up engine at the client. This dependency, unfortunately, is known to have some serious flaws in certain implementations—in extreme cases, even going so far as to sort the list of returned IP addresses in numeric order and to always use the lowest numerical address, instead of randomly cycling the list. Despite these issues and dependencies, there are a number of implementations that do the correct procedure and this particular approach does indeed help with load distribution.

With the issues inherent in round-robin DNS and the lack of geographic diversity in normal load-balancers, an additional approach is protocol-specific extensions. Protocols like HTTP have the ability to respond to clients with a redirect, pointing the client to a new server to acquire the content it is seeking. While this works, each implementation is inherently tied to a specific protocol and requires both client- and server-side support. If the option of redirection is not already available in a protocol, adding it would be difficult and would leave many clients unable to take advantage of the new feature. So while HTTP supports this option of redirection, many other protocols do not. This includes ftp and rsync, which are both heavily used in content distribution.

None of these solutions solves the problem of geographically diverse load-distribution universally; at best, these solve the problem for a particular niche. A global distribution system made up of independent entities needs to be transparent to the end users, as it's difficult for them to make good decisions as they lack information needed to make them. It needs to be versatile, able to expand and contract, resilient to changes, and most of all be protocol-agnostic so that existing and any future protocols can be easily or trivially supported.

## 2 GeoDNS—Knight in Dingy Armor

GeoDNS is an attempt to solve the shortcomings inherent in the predominant and existing load-balancing

[2] `http://www.akami.com`

schemes. It targets this by making a small, server-side-only change to a DNS server to allow it to respond to requests in a slightly different way, depending on the origin of the DNS request. This particular approach solves many of the issues inherent in round-robin DNS, centralized load-balancers, and protocol-specific redirection; however, it comes with its own set of quirks and issues that are equally inherent in its implementation.

### 2.1 Basic ideas

GeoDNS itself is a rather innocuous change to the way DNS handles requests, but gives the basic ability to do wide-scale, simple load-balancing without the need for changes to clients or custom protocols. DNS is a fundamental building block of the Internet. Every client that is attached to this global network already has a the ability to make a DNS query, converting a textual string like `example.com` into a numeric address, `208.77.188.166`. GeoDNS, like DNS views, differs slightly from a normal DNS query in that the response is altered based on additional criteria.

In the case of DNS views, it checks the incoming IP address, looking for matches in a range, and returns a different address based on each defined range. This is typically done to deal with the proliferation of Network Address Translation (NAT), where the IP address externally may be a routable IP, but is inaccessible to internal NATed machines. The DNS server returns the routable IP externally, and a non-routable IP internally based on the view criteria.

Strictly speaking, returning different data based to any query, according to RFC, would be the result of an overlapping tree and thus a "non-fatal error." However, since a client is unlikely to ever get into a situation where it would get multiple incompatible responses to an individual query, this doesn't actually cause any unexpected behavior to the client.

It does, however, break the idea of transparency across the Internet, where everything on the Internet is globally viewable and a DNS query will return the same result set no matter where you are. But this is no more broken than the idea of NAT, which currently has a huge proliferation and is arguably the reason that the IPv4 address space has not yet been completely exhausted. However, while this lack of transparency and consistency is not ideal from a DNS perspective, it is an effective means of solving a distribution problem and should be used on an as-needed basis and not be considered a standard practice for all queries.

### 2.2 GeoDNS: DNS View with a twist

GeoDNS comes into play to solve the same fundamental problem that a DNS view was introduced for: to return a more local resource for usage. Though where a DNS view is more likely to be solving the problem of routable vs. non-routable IP addresses, GeoDNS is more targeted at giving a user a more appropriate resource based on physical location.

GeoDNS functions very similar to a DNS view, whether it's implemented as one or not. The requesting client's IP is checked against a database, which determines a general geographic location for the requesting IP. Using this geographic identifier, a response is generated that is specific for that location, and sent back to the requesting client.

### 2.3 So it seems to work, but who uses it?

GeoDNS has a limited set of interested parties, which is one of the reasons that GeoDNS has not become standard in most DNS servers. Most users have their machines in a single location, on a single subnet, and not scattered across the globe. There are, however, installations that are making use of GeoDNS successfully.

For instance, this paper is focused on `kernel.org` and its usage of GeoDNS and that it works for the specific setup that `kernel.org` has. `Kernel.org` makes use of a modified set of patches that originate from caraytech. This patch set has been updated a couple of times, with the last being in 2008 by `kernel.org`.[3] However, despite `kernel.org`'s popularity within the Linux community, it is a relatively small distribution system, with only 4 locations in three countries serving the worldwide populace. Larger installations of GeoDNS servers that affect far more users exist. In particular, the largest known user of GeoDNS is Wikipedia.

Wikipedia and Wikimedia, for instance, moved to using a combination of Bind and PowerDNS[4] for their DNS

[3] It should be noted that this patch series makes use of MaxMind's GeoCountry look-up database.

[4] PowerDNS first introduced its geobackend in 2004.

servers in 2005.[5] Bind handles the primary zones themselves, while PowerDNS with the geobackend powers the distribution of users to a local Wikipedia server. A more complete description and an implementation that mirrors the Wikimedia setup can be found on the Blitzed IRC (Internet Relay Chat) network.[6]

A similar system using pgeodns (perl geo DNS server) serves `cpan.org` and `perl.org`, and has done so since 2001. CPAN is the primary locus of the Perl module development community. The pgeodns server is specifically attached to `search.cpan.org` to give to give a user a transparent local mirror for searching the CPAN archive, thus distributing the search load across multiple machines worldwide.

While `kernel.org`, Wikimedia, and CPAN are making use of the GeoDNS servers to act as load balancers, there are companies such as Server4Sale[7] that use a modified tinydns server, and are using it as a protection mechanism for their network and clients. Specifically, Server4Sale has made it available under the name Veridns, with the specific intention of using it to help deal with the problems of Denial Of Service (DOS) and Distributed Denial of Service (DDOS) attacks.

There are a number of other, primarily open source communities using GeoDNS-based services, from Apache[8], to Mozilla[9,10] to solve the same fundamental mirroring and load-balancing problem that many worldwide distribution systems are facing: how to get the data that users are seeking to them using the fastest possible system, and to load-balance these users across the world.

### 2.4 Where's the catch?

The art of attaching a geographic location to an IP does not come without some downsides. The databases are guaranteed to be inaccurate, require constant updates due to a shifting reality, and currently they are anything but future-proof.

Each database—whether it's being maintained by a corporate entity like MaxMind, which has a vested interest in its accuracy, or being kept up by a small open source project—faces the same challenge: the Internet is constantly shifting, and no one is required to publish geographically accurate information. So the collective maintainers are forced to go and mine public sources like Regional Internet Registries like RIPE, ARIN, and APNIC for information, but at best that gets them a broad brush-stroke of information. It does not, however, get the more subtle differences or weirdness that are sometimes used on the Internet, such as a US-based company using a subset of its IP addresses at a European facility. To become more accurate, further mining based on more data is needed, but even the best available databases are only claiming 99.8% accuracy,[11] which means that out of a potential 4.2 billion addresses, roughly 8.5 million addresses are incorrect.

These databases, while already trying to play catch-up with the ever shifting sands of the Internet, are facing another daunting and more complicated task ahead of them: creating and maintaining an additional database mapping IPv6 addresses to geographic locations. Currently all of the available databases only support IPv4, the dominant address scheme for the Internet since its first proposal in RFC 791 in September of 1981. IPv4 is running out of usable address space. (Though it has been predicted repeatedly that it should be exhausted already, usage of NATed networks has kept IPv4 dominant.) It will eventually run out of address space, and the Internet will be forced to make the long, arduous, and painful move over to IPv6—and in the process, go from a possible 4.2 billion addresses to 340 undecillion addresses. Currently most users of IPv6 are going through various tunnels and gateways, which will mask an address's actual location. Couple that with a low penetration of IPv6, and there is no current incentive to solve this particular problem by the open source community, and especially not by a for-profit entity.

So while things like GeoIP are working in their current form, they are not a perfect solution: the databases are inherently inaccurate and they currently lack support for

[5] `http://en.wikipedia.org/wiki/PowerDNS#PowerDNS_and_Wikimedia`

[6] `http://blitzed.org/DNS_balancing`

[7] `http://pub.mud.ro/wiki/Geoipdns`

[8] `http://mail-archives.apache.org/mod_mbox/www-infrastructure-dev/200904.mbox/<4239a4320904011536x5726d0eraae7065967ac1b77@mail.gmail.com>`

[9] `http://blog.mozilla.org/mrz/2008/05/28/geo-dns-getting-the-bits-closer-to-you/`

[10] `http://blog.mozilla.org/mrz/2008/06/11/geodns-one-week-later/`

[11] MaxMind's commercially available GeoIP Country Database claims an accuracy of 99.8% – `http://www.maxmind.com/app/country`

the next generation of the Internet. Despite these problems, GeoIP is quite useful. Even a database that is 80% accurate will work for a majority of the world's population as they would expect.

## 3 What to expect from GeoDNS

GeoDNS on the whole is an incredibly powerful and useful tool for administrators and world wide distributors. It can transparently deal with simple distribution across the entire globe and provide a simple interface for users to work with and use.

### 3.1 Production Zone Implementation

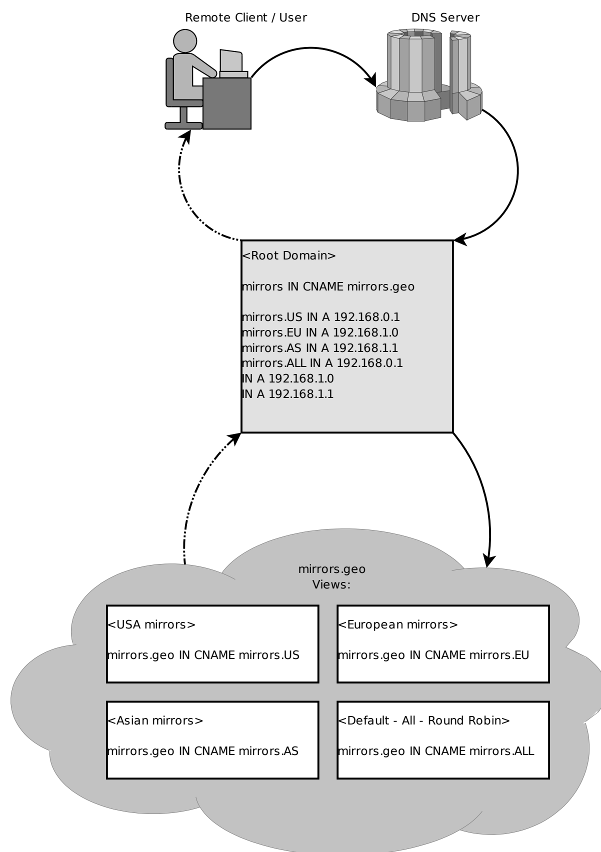



Figure 1: Suggested Flow request for a GeoDNS DNS request

GeoDNS is nothing more then a DNS answering system, and as such has a huge amount of flexibility and infrastructure that can take advantage of it. However, there are some implementation details that should be taken into account.

Since a GeoDNS-based server does not respond in the same manner as a normal DNS server, special care should be taken in choosing slaves. Slave DNS servers for your zone are going to need to be running the same GeoDNS server that your master is running, assuming that your GeoDNS server implementation supports replication at all. The best option would be to run both the master and the slaves. While this is a cumbersome proposition to some, it does solve the problem of compatibility and simplifies the fact that that zone is unexpected for most DNS servers.

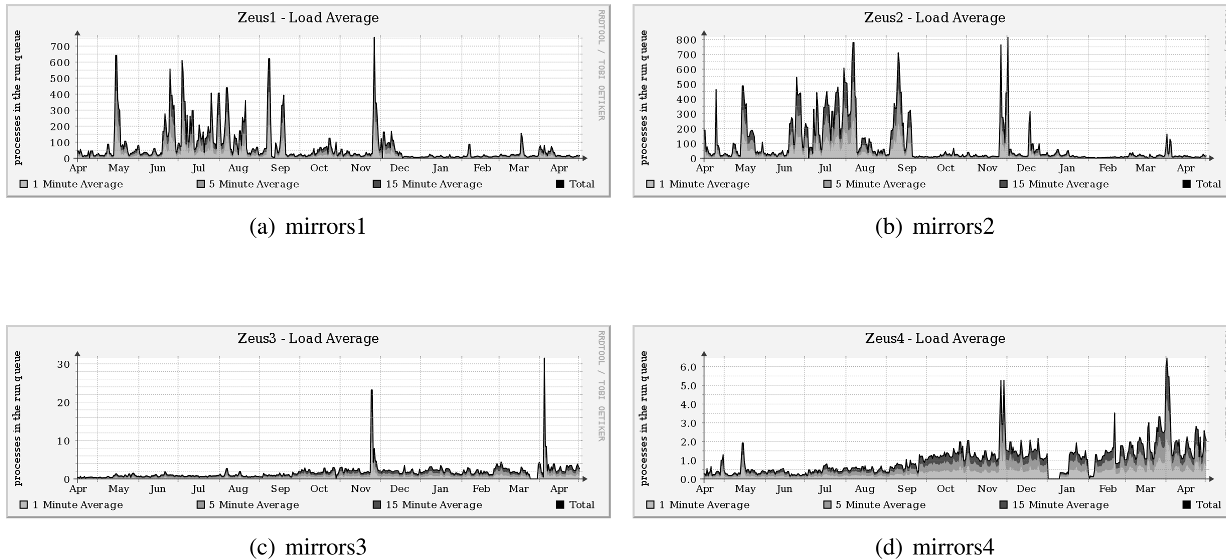
In the case of Bind + GeoIP, it has some simplifications that make it appealing. For instance it leverages the existing view infrastructure, giving it the full ability to replicate each view independently[12] and all the other advantages of views. PowerDNS with the geobackend unfortunately does not have the ability to act as the master or slave for DNS purposes,[13] however, the basic implementation of the zone structures are similar.

A basic and scalable configuration is two-fold. There will be two different servers, one authoritative for the actual IP addresses that you are using, and one that is authoritative for your GeoIP zone. While Bind can handle these both in a single instance, common pieces that would be available in all zones (like the non-geoip hosts) would need to be copied multiple times and maintained independently. This copy-and-paste structure can lead to errors and make it difficult to maintain. For convention, `example.com` will be the primary static zone with all the IP addresses in it, while `geo.example.com` will be the zone handling all of the geographic look-ups.

The request for a domain that will be served via GeoDNS should come in to a common disambiguation point, something like `mirrors.example.com`, which should be present in the primary static zone. This would be a CNAME that that would point to `mirrors.geo.example.com`, which is in the geographically aware zone. When the DNS client requests the next hop for `mirrors.geo.example.com`, GeoIP look-up would occur on the server. It would match the incoming request's IP address and select the appropriate zone's view to return. The response, as all

[12] How Can I Make A Server A Slave For Both An Internal And An External View At The Same Time? When I Tried, Both Views On The Slave Were Transferred From The Same View On The Master. `http://www.isc.org/node/282`

[13] `http://doc.powerdns.com/geo.html`




(a) mirrors1

(b) mirrors2

(c) mirrors3

(d) mirrors4

Figure 2: Load graphs of `mirrors.kernel.org`

entries in the `geo.example.com` zone, should likely be CNAMEs that point back to the primary zone of `example.com`. This final CNAME look-up would point to an IP address, or a round robin of addresses to further add distribution. Figure 1 illustrates the basic request and response structure that this would entail.

Breaking the zones up into GeoIP vs. non-GeoIP zones has several advantages. It allows you to trivially migrate between GeoDNS solutions, as the zone handling those look-ups is independent. Master and slave relationships are not limited to other GeoDNS servers in the non-GeoIP address space, meaning you can use any combination of DNS servers. Breaking this relationship up also means that requests are only being performed for only those items that require it, like a mirroring infrastructure as opposed to the mail (MX) records for a domain.

### 3.2 GeoDNS: Case study—kernel.org

To give some basis for a real-world example, `kernel.org` is a worldwide distribution system with equipment in three countries and five separate data centers. Its content includes the primary download location for the Linux kernel source code, and it acts as a tier-1 mirror for many of the distributions based on Linux. It boasts a user community that spans the globe, with users accessing its content on every continent, including Antarctica. With such a large user base, and a historical issue of only having servers based in the United States, `kernel.org` was looking to simplify the problem of mirror distribution without causing undue confusion to its user base.

A simple solution was first implemented using a country- or region-specific domain. New equipment was brought online in Europe, the Netherlands, and Sweden respectively, so `mirrors.eu.kernel.org` and `www.eu.kernel.org` were set up. This solved the basic problem of getting user-recognizable connectivity to the European mirrors. However, because `mirrors.kernel.org` and `www.kernel.org` have been more or less ingrained in the mindsets of users, the machines did not see quite the pick-up in usage we had hoped for, and it was clear that users in Europe were still coming back to the US mirrors to get their content.

Several different solutions were considered to more transparently direct users to a closer mirror. `Kernel.org` had a simple set of requirements:

1. The need to deal with a geographically diverse set of machines. This generally rules out things like normal load balancers.

2. Be protocol agnostic, as `kernel.org` serves data over ftp, HTTP, rsync, and git. It could not be subject to protocol-specific load balancing, so this rules out things like mod_geoip for Apache.

3. Be transparent to end users. Many of `kernel.org`'s users are automated scripts and programs, so having a disambiguation page to direct users to the closest mirror is not a solution that would work.

A DNS server based on the BGP (Border Gateway Protocol) as its determination mechanism was considered. BGP is, however, notoriously difficult to gain access to with the need to acquire an autonomous system (AS) ID from ICANN. Coupled with the difficulty in getting an AS, many larger backbone providers are uninterested in peering and sharing their routing table information with smaller entities sometimes having requirements to control upwards of seven hundred unique and routable IP addresses. Once peered, the problem becomes one of completeness of the routing table provided, with many instances of partial views of the entire routing table which would be difficult to make good decisions based on. This particular approach was abandoned for these reasons.

A GeoIP approach was then investigated, with specific emphasis on GeoIP coupled to a DNS server. It was decided to work with the Bind-based patches and to go with an organization similar to what is described in Section 3.1. The entire setup was flipped over on September 19th, 2008. Figure 2 shows the loads of the `kernel.org` machines before and after the switch to using GeoDNS. Prior to the middle of September, the two US machines, `mirrors1` and `mirrors2`, had a high and periodic load with long and consistently high loads. This is particularly evident in the summer months of 2008, with loads consistently above 50 and easily peaking above 200 in instances. It is also clear that during this same timeframe, the loads on `mirrors3` and `mirrors4`, the European machines, was virtually nonexistent. The loads were barely even high enough to register on the graphs. After September 19th, the graphs take on a much different outlook, and in fact the change was noticeable within hours of the DNS change.

Saying that the GeoDNS server's activation was a success would be an understatement. After the September 19th switchover, the graphs for the two US-based machines drop dramatically. Over the course of 7 months, they've maintained a relatively consistent load well below 100, with occasional spikes above 100. The loads on the European based machines have risen as a result of the higher traffic. Though the change is a noticeable, the machines are still coping with the new load without issue. So, in this specific circumstance, GeoDNS has been an unequivocal success and has helped to extend and better the services that `kernel.org` offers.

# Porting to Linux the Right Way

Migrating data between kernel and user space

Neil Horman
*Red Hat*
nhorman@redhat.com

## Abstract

Linux has grown to be a major development platform over the last decade, often becoming the primary target for many new applications and appliances. Of course, businesses always wanting to stay current; the rate at which software has been ported to Linux has also gone on the rise. Often this is a trivial matter, especially in environments in which the development model is similar (AIX to Linux, Solaris to Linux, even Windows to Linux). However, there are environments (particularly in the embedded space) in which porting often becomes difficult. A stronger coupling of application and driver, coupled with a "just get it working fast" mentality, invariably leads to substandard porting efforts which result in products with degraded performance that leave developers and consumers alike with a bad taste in their mouths. This paper seeks to ease some of that porting effort by focusing on what has been one of the most often mis-ported areas of code: the user space / kernel space boundary, specifically the movement of data between these domains. This paper will discuss in general terms: the common monolithic application model most often associated with embedded systems development; its refactoring when porting to Linux; the modeling and description of data that must be passed between the refactored components when porting to Linux; and the selection of an appropriate mechanism for moving that data back and forth between user and kernel space. In so doing, the reader will be exposed to several mechanisms which can be leveraged to achieve a superiorly ported software product that provides both a better customer experience and a greater confidence in Linux as a future development platform.

## 1 Introduction

Linux has seen an almost meteoric rise in popularity over the past several years. Rather by definition, this increase in popularity has drawn developers to consider Linux as a target platform in many market segments, from servers to appliances, to small embedded solutions. Of course, along with the interest in new development on Linux comes the desire to bring older software to Linux, in an effort to leverage already existing products in potential new market spaces at reduced costs. This paper focuses on some of the pitfalls that befall developers seeking to port applications to Linux from alternative operating systems. In particular, it seeks to address the difficulties faced by embedded developers seeking to formalize the user space / kernel space split in applications which formerly were more tightly coupled in that respect. It seeks to do this by providing an overview of how the user / kernel space boundary is defined, and how to move data safely and efficiently between the two domains.

## 2 Defining the user / kernel space boundary

Prior to discussing the mintuae of moving data back and forth across the user / kernel space boundary, it is useful to better understand what the user / kernel space boundary is. Nominally, when discussing this particular separation, many people encounter this high level diagram:

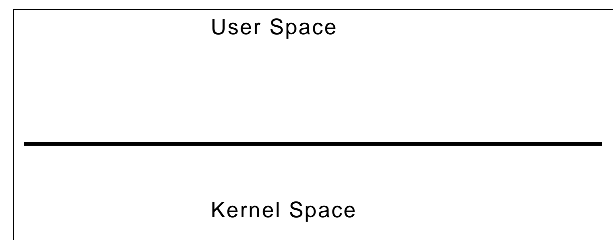



Figure 1: The most seen & worst overview of user / kernel separation

This is the sort of visual aid that is understood best by people who already understand how this separation

works, and consequently have no need for a visual aid. For everyone else, however, this is a rather lackluster description of how the two execution domains are separated. The separation of user and kernel space can be best described as having the following characteristics:

- A separation of accessible address spaces, except where granted by appropriate system calls.
- A mechanism which allows access to the functionality of kernel space code, preferably without granting the user space context direct access to the memory holding the code for that functionality.

Different architectures provide various mechanisms for enforcing the above points to various degrees, but the end result remains the same: the user / kernel space boundary enforces the isolation of an application from operating system services, forcing applications to make use of those services only in the ways defined by the operating system. This provides both stability and consistency of use.

## 3 Mapping strongly coupled applications to user / kernel space

Often, especially in the embedded space, the operating system is treated as a convenience library, rather than a environment to target. The application is paramount, enjoying complete or near complete control over the system's resources. In such designs, the software architecture can often be characterized as such:

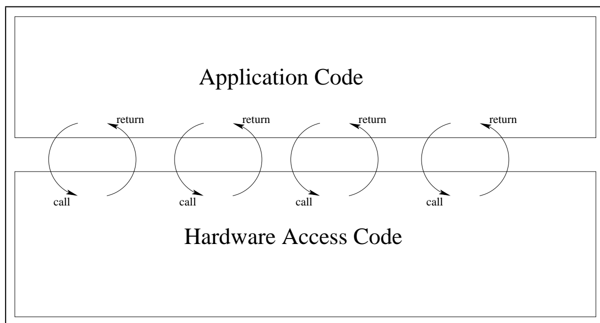



Figure 2: A common design approach to a tightly coupled embedded application

Strongly coupled applications, to whatever degree they manage to separate true application code from hardware-specific access code, interface the two with an API that can be difficult to separate. The API is often characterized by some of the following aspects:

- Customized for use by an application;
- Assumes shared memory space with application;
- Assumes common system behavioral characteristics on both sides of the API.

While some designs attempt to adhere to some level of compliance to a standard API, many do not, and the resulting temptation to speed the design cycle by taking various shortcuts invariably leads to porting difficulties when an operating system more strictly enforces the use of a pre-defined API. This leads to the inevitable question: How does one convert a system with application code and hardware access code that is tightly coupled into a system that is capable of operating in separate memory contexts using an enforced set of access methods? The answer is less difficult than many think. Generally speaking, one can accomplish this goal by:

1. Selecting/creating an API to use as a user / kernel shuttle;
2. Modeling the data that needs to cross that boundary;
3. Selecting the appropriate kernel APIs to implement that transport.

Note this will not provide the most efficient interface possible, but it will provide the most stable interface possible for the system being ported, which is really the goal here. With the time saved on avoidance of future bugs, one can focus on efficiency improvements later on.

### 3.1 Selecting/creating an API to use as a user / kernel shuttle

The first step in adapting an application from an strongly coupled environment is to select a good location within the system to segment it. There are many factors which can affect this decision, and will vary largely dependent on the system in question. Systems which honor their defined APIs will be more conducive to separation

than those which do not. "Thinner" APIs will make for quicker work than those with larger sets of functions. There are many facets to selecting an appropriate API for splitting an application during a porting effort, but the items below should be focused on first and foremost:

1. **Cleanliness.** Above all else, a candidate API should provide a boundary through which data can pass only in narrowly defined channels, and at approved times. APIs which provide or use information that exists external to the API implementation should not be considered unless such uses may be corrected. For instance, the use of a global variable or resource within an API's implementation is bad, simply because external access to such a variable will no longer be possible from outside the API after the port.

2. **Complexity.** The simpler an API is, the better, as migrating its implementation to one which exists across memory domains will become much easier. APIs which embody a significant amount of stateful information are difficult to manage, as that state may potentially need to be tracked, replicated, and kept in sync in both kernel and user space. Conversely, APIs which are simple, pass less data, and have fewer return codes will make for an easier split.

3. **Functional Requirements.** Be careful to closely examine an API's implementation when selecting it for use as the point where an application is separated from its hardware-dependent components. Sections within an API (and its dependencies) may require behaviors or resources that may not be available in either kernel or user space without additional work. Selecting an API that leaves a bit of code in the application that blocks while waiting for an interrupt to fire will require additional retrofitting to function properly after the port. Likewise, consider an API implementation which creates a thread that spins, polling for data. Moving it into kernel space will result in horrible inefficiencies resulting from differing scheduling behavior down in the kernel under various conditions.

4. **Size.** The more functions an API implements, the more difficult it will be to port. Quite simply, there is a quanta of work to be done for each function call implemented in an API; therefore there is a corresponding increase in the porting effort when a larger API is selected for this split. Hence, while not always possible, "thinner" APIs typically result in easier porting efforts.

5. **Volume.** Keep in mind how heavily a given API is used by an application, and under what circumstances. APIs containing very few and simple function calls are tempting to select as a candidates for a split point, but if the application must call functions in this API thousands of times to complete a given task, there will be a performance impact. Bear in mind that transitions between user and kernel space have a cost in terms of pipeline flushes, TLB and L2 cache flushes, etc. (which varies from architecture to architecture). The number of times you need to make that transition will have a significant impact on your system's performance.

While clearly one will have to make compromises when selecting a point at which to separate a software system, the above are the most important aspects to keep in mind. Neglecting any one of these aspects, while perhaps further minimizing your time to complete a port, will result in a system with stability and performance which will be at best an approximation of the original system on the original OS.

### 3.2 Modeling data transfers across the user / kernel space boundary

Having selected a point at which to segment the software system into a user space component and kernel space component, a developer must now re-implement the internals of that API such that data is transparently sent from user space to kernel space and back at appropriate times, and with previously provided guarantees on the data's integrity, format, etc. In most use cases, understanding the data transfers profile is fairly straightforward, but modeling data transfers in some cases can be non-obvious. All data transfers can be described in three aspects: Timing, Quantization, and Access.

#### 3.2.1 Timing

The timing of a data transfer here refers to when data transferred across an API is acted upon by either a user of the data or the implementation of the API.

1. **Asynchronous application to driver.** These are message-based transactions. Data is accumulated into a discrete bundle of arbitrary size and passed to the driver. The return code to the submission of this transaction typically provides status of the submission effort, but not the result of the action which the data submitted embodies. Any action which the driver may take on the submitted data is deferred to an arbitrary later time, and results of those operations may or may not reported back to the application via a separate data transfer.

2. **Asynchronous driver to application.** These are the inverse of the the previous transfers. They send similarly formatted messages, only in the reverse direction. Messages generated by the driver code are queued for reception by the application through various APIs. It is interesting to note that the various APIs available for these transactions have various restrictions which do not exist in their counterparts. Those limitations will be noted in the next section. Such transactions may be driven by the previous transaction type, or may be generated independently by any number of conditions or events.

3. **Synchronous.** These are transfers which are, as the name implies, synchronous. Data passed via this profile is handed to the driver, operated on, and released at the conclusion of the API call. Return codes typically embody the result of whatever operations the driver performed on the passed data.

### 3.2.2 Quantization

The quantization of a data transfer refers to how the data is viewed by the API user or the API implementation. Some APIs handle data transfers as distinct units, atomic in their transfers, while others treat data as an arbitrarily sized sequence of bytes, re-segmenting the data in whatever manner is convenient.

1. **Stream data.** Stream data is unbounded. While it may contain a begin, end or other control marker, it does not normally distinguish data record boundaries. A transfer of data may contain an arbitrary number of bytes; users of the API or the implementation are not guaranteed to receive any particular amount of data during any typical transfer. Nominally, however, stream-type data is guaranteed to maintain its order (all bytes transferred in a stream are handled in the same order they are sent).

2. **Packet data.** Packet data is bounded and quantized. While the size may not remain constant from one API call to the next, packet-style data is always treated as an atomic unit. Packet data may or may not guarantee ordering of data.

### 3.2.3 Access

Access describes how an API makes data transfers available to either its implementation or its users. These methods are well known and well understood by all developers, but it is useful to provide a reference here so that the different methods are kept in mind when modeling your data transfers and selecting an API to use during your porting effort.

1. **Value.** Data is passed into the API and a copy is made for use internal to the API. Changes to the data made internal to the API are not visible to the user of the API.

2. **Reference.** Data itself is not passed into the API directly, but rather by a memory reference to the data's location. The user of the API and its implementation share access to the data and may need to co-ordinate that access to avoid corruption.

Using these three aspects in all their combinations, it is possible to completely model how data is moved between a user of API and its implementation. Once that is determined, the task of selecting an pre-defined kernel interface API to act as a transport for splitting a legacy software project into kernel and user space segments becomes much easier.

## 3.3 Selecting the appropriate kernel APIs for data transport

Now that we have described how data can be passed through APIs in general terms, we can describe the various available kernel interface in those same terms so that we can select an appropriate interface to better adapt to our existing model to optimize our porting effort. There

are several kernel interfaces to choose from, each of which offers a different set of data transfer characteristics. Each kernel interface is documented here. There are only three major interfaces from kernel space to user space: the character driver interface, the socket interface, and the signal interface. Each provides an API that allows a developer to implement various different data transfer models. Note that there are other APIs which allow for various types of data transfer (the file system interface, the block driver interface, etc.); however, the three aforementioned interfaces cover completely the types of data transfer listed in Section 3.2. While the other interfaces provide excellent mechanisms to design various types of systems, the above listed interfaces provide generic data transfer resources that allow for any application to be relatively easily split between user and kernel space.

#### 3.3.1 Character Driver Interface

The character driver interface is seemly the de-facto interface that developers select—a transport API when segmenting a monolithic application during a porting effort. It offers several data transfer models, and as such is reasonably flexible, is fairly well understood when coming from other operating environments, and is fairly easy to implement. The user space API consists of the following elements:

1. **open**
   **int open(const char *pathname, int flags)**
   This is the well known and understood call to establish access to a filesystem object. Nominally, this call is used to open a regular file. However, under the Linux design model, this function can also be used to obtain access to a device through a device special file (created with the `mknod` utility or through the `udev` system). Special device files contain a major and minor number, which are used to resolve which device driver will handle operations issued to the device. Open accepts a path (nominally for the porting purposes here) to a device special file, and a set of flags which define various characteristics of the communication to the device (read/write permissions, auto close properties, etc.). It returns a descriptor to the opening process; this descriptor is used in later calls below.

2. **read**
   **ssize_t read(int fd, void *buf, size_t count)**
   The read call provides a synchronous, stream-oriented byte sequence passed by value to the user. Kernel modules that implement the character driver interface are notified immediately when a user space process issues a read request, and is required to return either an error code, or a data buffer (which will be copied into the process context) of a size equal to or less than the size passed in during the request.

3. **write**
   **ssize_t write(int fd, const void *buf, size_t count)**
   The write call provides the converse operation of the read call. It similarly provides an synchronous interface for passing stream-oriented byte sequences by value. The difference of course is that write calls allow a user process to pass data from a process to a kernel module. Unlike **read**, however, a return code is the only thing returned to the user. The conditions for success or failure in this call are dependent on the kernel module implementing the driver the data is passed to, as it what is done with the data when it arrives at the driver.

4. **ioctl**
   **int ioctl(int d, int request, ...)**
   `Ioctl` is the Swiss army knife of the character driver interface. It allows an arbitrary-length list of data items to be passed by value down to the driver associated with the descriptor passed in as the first argument. The flexibility and open format in the amount of data that can be transferred in this API call makes this a very attractive interface to implement the segmentation of application during porting, but it should in fact be used sparingly. Because the remaining arguments on the function are variable, only weak or non-existent type checking abilities ensue; the only way to properly interpret the arguments is via the request argument, which shares name space with every other driver layer between the application and the driver itself. While this call can be very useful, it can also introduce subtle and difficult to detect errors, and as such should be used carefully.

5. **mmap**
   **void *mmap(void *start, size_t length, int prot, int flags, inf fd, off_t offset)**

The `mmap` call is far more useful than developers normally give it credit for. The `mmap` call allows a user-space process to specify an address space 'hint' along with an open file descriptor and offset from that descriptor. In response, the object associated with that descriptor will map the specified length of data into the requested address range. This allows the `mmap` call to be categorized as an asynchronous kernel-to-user-space data transfer, passing packet data by reference. This call is nominally associated with regular files, in which the files contained data is mapped into process address space, allowing for direct access. What is not nominally recognized, however, is that `mmap` can be used equally well for any arbitrary device. The offset argument is passed directly to a driver, and used as an arbitrary handle, in which the driver passes back whatever data is required. For example, a character driver can be implemented to pass a stream of handles to a process via the `read` call, which can then be used in a sequence of `mmap` calls to access other out-of-band data. This provides a more efficient transfer mechanism for large volumes of data by only passing handles by value (which are much smaller than the requisite data they reference).

6. **munmap**
   **int munmap(void *start, sizei_t length)**
   This is of course the inverse of the previous call. It allows a block of data previously passed by reference to be released by an application. The memory referenced becomes inaccessible to the process, and the driver is informed of the release operation so that any needed clean up can be performed.

7. **close**
   **int close(int fd)**
   This ends a connection to a device driver/kernel module, and provides the driver the opportunity to clean up any remaining resources associated with that process connection.

The kernel space API consists of a registration and de-registration function, and several ancillary functions which map one-to-one to the user space API, each called directly in response to a user space call of the same type:

1. **register_chrdev**
   **int register_chrdev(unsigned int major, const char *name, const struct file_operations *fops)**
   This is the major kernel hook which allows you to register a special character device file to the kernel. The major parameter allows you to specify a major value that is matched against any special character mode device file opened in user space. A file opened containing a matching major number will be directed to this module via the function pointers passed in to the registration routine via the `file_operations` structure.

2. **unregister_chrdev**
   **void unregister_chrdev(unsigned int major, const char *name)**
   This is of course the converse of the above function. It allows kernel code to disconnect an association between a driver module and a major device number.

### 3.3.2 Socket Interface

The socket interface is far less often considered for use as an API with which to shuttle data between kernel and user space, but it should be. Highly flexible, and fairly well understood from an application standpoint, the socket API allows a developer to move data between user and kernel space in a variety of ways and formats, using both custom built protocols, and (perhaps more notably) using already existing protocols. While not often suggested, the infrastructure to create and manage sockets with the kernel has been in place for some time, making it possible to write a kernel module which can simply open a TCP, UDP, or other protocol socket, and accept incoming data from user space by sending to the opened port via the `localhost` address. Likewise, the Netlink protocol family has existed for some time (arguably in relative obscurity), for the sole purpose of connecting user space processes with kernel space services. Netlink also provides the added capability of dynamic sub-protocol registration (via the generic Netlink infrastructure), which allows for dynamic service discovery, making porting efforts even easier.

The API for working with sockets in user space is very mature and well known. While this list is not exhaustive, it enumerates the core functionality of the API:

1. **socket**
   **int socket(int domain, int type, int protocol)**

The `socket` call, like the `open` call, allocates a communication channel to a peer. Depending on the use of the subsequent calls, the peer could be a remote host, another process on the local host, or a service in the kernel. The `domain` argument specifies the address family (which specifies the format in which the peer to which you will connect is addressed). The type generally specifies whether the connection to the peer will be passing stream-oriented data or packet-oriented data, and the protocol specifies the transport layer protocol that the connection will use. While it is possible to create your own protocol (which this call can then provide access to user space to), given that the effort here is to simply migrate data between user and kernel space, it is far more efficient to simply use IPv4 (The `AF_INET` family), IPv6 (the `AF_INET6` family), or the netlink protocol (`AF_NETLINK`). Any one of these protocols will allow a user-space application to take full advantage of all the features of this API for the purpose of data transfer to kernel space.

The socket call either returns a negative error message, or a positive value that represents a handle to use in subsequent socket API calls.

2. **bind**
   **int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)**
   The `bind` call operation is somewhat specific to the address family, type, and protocol specified in the `socket` call. Generally speaking, the `bind` call associates a socket with an input filter. The format of the filter is specific to the protocol selected. For example, the `AF_INET` family allows you to bind your socket to an input address and port, so that you will only receive frames on a certain interface. `AF_NETLINK`, in contrast, allows you to specify a bitmask of multicast groups that you might receive frames on, in addition to messages directed at your specific process ID. This filter is passed in via the `addr` pointer.

3. **connect**
   **int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen)**
   The `connect` call associates a socket with a peer address. Like `bind` associates a socket with a local endpoint, `connect` establishes the peer that the socket communicates with. This operation is specific to the address family and protocol which were specified in the call to `socket` above. Some protocols, like TCP, communicate with the remote endpoint to establish a connection, while others simply record the address of the remote endpoint.

4. **send/sendmsg/sendto**
   **ssize_t send(int s, const void *buf, size_t len, int flags)**
   **ssize_t sendto(int s, const void *buf, size_t len, int flags, const struct sockaddr *to, socklen_t tolen)**
   **ssize_t sendmsg(int s, const struct msghdr *msg, int flags)**
   These functions provide various methods for the asynchronous transfer of data from user space to kernel space, passed by value, in either a stream- or packet-oriented format. There are three variants of the `send` routine, as different connections find different implementations more useful than others. Note that the `send` routine omits a remote address, which means the remote peer was specified with a prior call to `connect`. Conversely, the `sendto` operation allows for data to be sent on an unconnected socket, with each call specifying the recipient address (allowing one socket to communicate with multiple peers). `Sendmsg` is a variant of `sendto`, but uses a `msghdr` structure, which allows for a series of non-contiguous data pointers to be sent at once. Interestingly, all three calls may be used to send either packet data or stream data. The ability to re-segment data to split or merge data as it was sent from user space is encoded in the specific protocol as selected in the `socket` call.

5. **recv/recvmsg/recvfrom**
   **ssize_t recv(int s, void *buf, size_t len, int flags)**
   **ssize_t recvfrom(int s, void *buf, sizei_t len, int flags, struct sockaddr *from, socklen_t *fromlen)**
   **ssize_t recvmsg(int s, struct msghdr *msg, int flags)**
   These functions are the counterparts of the above `send` routines. They allow an application to poll a socket to see if any data is available from the peer(s) which the socket might be communicating with. Data transfer is asynchronous with its arrival at the protocol implementation in the kernel, is passed by value, and can be either stream- or packet-oriented. Overall, the operation is identical

to the `send` counterparts.

6. **setsockopt/getsockopt int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen);**
   **int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen);**

   These two calls allow for a user application to fine tune the operation of the selected communication protocol. As noted a socket can have behavior tuned at various levels (generic socket, protocol-specific, transport-specific, etc). If you are writing a custom protocol, these settings can adjust anything the developer would like (the level parameter name-space is global, but by defining a new level, the `optname` parameter becomes unique, so any number of options may be defined, unlike the `ioctl` call). Any amount of data may be passed by value via the `optval` pointer, but its interface is limited and inefficient.

7. **close int close(int fd)**
   Like the `close` call in the character driver, this call simply disconnects the descriptor in the user space application from its peer.

The kernel interface operates much like the character device kernel interface, with some enhanced abilities for fine tuning. There are two main registration and deregistration functions which allow one to add both an address family and a protocol, allowing for a larger set of communication methods. Nominally, however, the use of sockets as a data transfer mechanism doesn't require the creation of a new protocol. Developers looking to port applications to Linux and split their software into a kernel and a user space component are highly encouraged, for the sake of simplicity, to simply utilize an exiting protocol for communication, such as UDP, TCP, or netlink.

### 3.3.3 Signal Interface

The signal interface rounds out our kernel / user space data transfer methods. The signal interface provides a data transfer model that provides one thing that the other interfaces do not. The other interfaces may provide data asynchronously or synchronously, but all require polling to retrieve that data (via the `read` or `recvmsg` family of calls). The `signal` interface provides true asynchronous data delivery, requiring no additional action to receive data beyond registering a reception function. Also, the signal interface is normally not used alone, but rather in conjunction with one of the other interfaces to provide a complete data transfer system when splitting a legacy application between user space and kernel space.

The signal interface is enumerated below:

1. **signal/sigaction**
   **sighandler_t signal(int signum, sighandler_t handler)**
   **int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)**
   The `sigaction` and `signal` calls both associate a signal value with an action, as defined by the handler pointer. The handler pointer contains a function pointer which is defined within the application, which is called each time a signal is delivered to the application process. The two calls are effectively identical; the `sigaction` call simply provides a few more options for fine-tuning signal delivery behavior. The `signum` parameter enumerates the signal value with which the action is associated. Some signal values are predefined (SIGINT, SIGSTOP, SIGKILL, etc.), and may have actions which are pre-defined. Other signals are generic and can have actions which are specific to the application being run (SIGUSR, etc.). Some signals have at-most-once semantics (multiple signal deliveries result in only one call to the action handler between certain delimiting events. Others ranges can queue their events creating a call-per-event model.

2. **sigpending**
   **int sigpending(sigset_t *set)**
   This call allows a user to determine which signals may be pending for delivery to the application. During periods when the application may block the delivery of signals, this calls provides a window to see what the application may be missing, so to speak.

3. **sigsuspend**
   **int sigsuspend(const sigset_t *mask)**
   The `sigsuspend` call provides an interface for the user-space application to temporarily block signals from being delivered.

4. **kill**
   **int kill(pid_t pid, int sig)** This call completes the signal API. It allows a user-space process to deliver a signal to another process, as identified by the `pid` parameter. The kernel also contains a variant of this call which allows kernel code communicate via the signal API to an application.

## 4 Conclusions

Legacy applications offer a mature stable code base which should not be discarded lightly. Most, if not all, applications can be ported to Linux with a minimum of effort, if proper care is taken to both segment the application properly to a kernel and a user space component and appropriate APIs are used to efficiently and correctly transfer data between the two.

# Tracing the HA Cluster of Guests with VESPER

(Virtual Embraced Space ProbER)

Sungho Kim
*Hitachi, Ltd., Systems Development Lab*
sungho.kim.zd@hitachi.com

Satoru Moriya
*Hitachi, Ltd., Systems Development Lab*
satoru.moriya.br@hitachi.com

Satoshi Oshima
*Hitachi, Ltd., Systems Development Lab*
satoshi.oshima.fk@hitachi.com

## Abstract

Recently, many tracing infrastructures, like kprobes, tracepoints, ftrace, etc. have been merged into the mainline kernel. They seem useful to tell what is going on inside the kernel in the physical machine. So, it is natural that we tend to question if can we use them to trace virtual machines.

In this paper, we introduce VESPER, the framework to trace guest kernel states from the host utilizing in-tree tracing stuff in the just same manner as host kernel tracing. In particular, the mechanism of injecting probes to guest and splicing guest tracing reports onto host to alleviate data copy overhead will be focused upon. To verify the efficiency of VESPER, we take HA cluster with guests on in-tree hypervisors, KVM, for test cases. By combining tracepoints with kprobes to monitor guests, VESPER shows the improvement on fail-over response latency caused by application-bound as well as system-wide failure, against conventional heartbeat.

## 1 Introduction

Tracing issues have recently been focused on Linux kernel community. Kernel Markers and trace points have already merged into the mainline kernel and various Ftrace engines have been introduced at the LKML [1]. Those tracing facilities can provide lightweight mechanism to understand the behavior of the kernel compared to dynamic tracing facilities such as Kprobes which utilize `breakpoint`. It seems to be reasonable to use the tracing facilities to debug the kernel. However, we have been thinking about other use cases for the facilities than the debugging. Our suggestion is applying the kernel tracing to the decision maker of fail-over or migration in the clustering of virtual machines in enterprise server systems requiring efficient resource utilization and system dependability. In a certain system, fail-over response latency is the bottleneck for the high availability of service, which comes from the polling-way of heartbeating between cluster nodes. Even in a virtualized environment, a cluster manager such as Heartbeat [2][3] software delivers messages between cluster nodes to check their health through network periodically (known as heartbeat). If any node fails to reply in a certain time, the manager will assign the service which the faulty node was providing to another node. This amount of time before switching to another node is called the deadtime, key to ascertaining node death in Heartbeat. With this approach, however, the manager cannot immediately determine faults, nor get detailed information about faulty nodes; this results in fail-over response latency. But what if the tracing technology is applied to the heartbeat? This allows the cluster manager to have:

- Prompt notification when corresponding events happen around a probe.
- Dynamic probe insertion to guarantee service availability while inserting a probe when using Kprobes.
- Arbitrary probe insertion to any process address to hold its versatile probing capability when using tracepoints and etc.

Utilizing this featured technology to examine the health of a virtual cluster member machine could lead to faster and more efficient evaluation criteria for system switching or migration than a simple, periodic message de-

livery mechanism. Therefore, we have proposed VESPER (Virtual Embraced Space Prober)[4] which gathers guest information effectively in a virtualized environment, taking advantage of the full features of the tracing facilities. VESPER simply transfers probes generated in the host to the targeted guest. The transferred probes do all the necessary probing work themselves in the guest, and then VESPER simultaneously obtains the result of probes through shared memory built across the host and guest. However, VESPER was only available for the paravirtualized guests on Xen [5] because it utilized the Xenbus for the communication between the host and the guest. To make VESPER more available for the various virtualization technology we adopt Virtio [6] proposed as the standard of virtual I/O mechanism in the Linux community. In this paper, we will describe how to port VESPER to Virtio and how to inject the probes into hardware-virtualized guest on KVM [7], which is a in-tree hypervisor supporting Virtio.

Section 2 briefly describes the mechanism of QEMU [8] and KVM using Virtio to implement virtual I/O devices. Section 3 presents the brief description of VESPER architecture and implementation of VESPER as a virtual I/O device in QEMU, while Section 4 discusses trace issues using VESPER. Finally, we conclude this paper in Section 5.

## 2 Overview of Virtio in QEMU

Before getting into the porting issue, we briefly describe how QEMU and KVM works for virtual I/O devices (called virtio devised hereafter) such as `virtio_blk` and `virtio_net`.

In Virtio, all virtio devices are treated as pci devices under `virtio_pci` in the guest. So, virtio devices need to invoke pci emulator in QEMU. Fortunately, `virtio_init_pci` in `qemu/hw/virtio.c` in QEMU does necessary works for them. In the process of initializing virtual machine hardware, virtio devices invoke `virtio_init_pci` to register them as pci devices in QEMU. Then `virtio_init_pci` allocates VirtQueue, the control block for the shared memory between the host and the guest in Virtio, for the devices and invokes `pci_register_io_region` to store the configuration information of the devices just like native pci devices. The different thing from the native PCI devices is to register callback functions to each configuration area other than specific data. The callback functions are the key components to share virtio devices between the host and the guest. Especially in KVM, I/O accessing of the guest to those io regions occurs `VM_EXIT` to switch cpu context from the guest to the host and QEMU at last. As a matter of fact, `virtqueue_ops.kick` of `virtio_pci`, the communication method in Virtio, in the guest invokes `iowrite16`. In the cpu context switch between the host and guest in KVM, callback functions registered in the specific area invoke service routines for each devices. Accessing other features data of the device follows the same processing below.

1. Register Device in QEMU.

   Register the device in the PCI bus in QEMU.

2. Register IO Region in QEMU.

   Map the allocated IO Region with proper callbacks.

3. Register VirtQueue in QEMU.

   Allocate VirtQueue to handle the shared memory prepared by the guest in QEMU.

4. Retrieve Device Configuration.

   Registered virtio device driver probed in the guest, the driver executes I/O access to retrieve the device features.

5. Set `virtqueue`.

   Virtio device driver in the guest allocates `virtqueue` and `vring` memory, the control block for `vring` and the shared memory between the host and the guest, respectively. Then it executes I/O access to the IO Region in QEMU and sets vring physical address of the guest. In this procedures, the host and the guest finally establish the shared memory with `vring`.

6. Send Request.

   Virtio device driver in the guest sends requests written on the `vring` by I/O accessing to the IO Region.

7. Send Response in QEMU.

   Callbacks in the IO Region do serve and send back the result by inject interrupt to the guest via KVM.

In the next section, we describe the detailed implementation of the VESPER for the Virtio in QEMU and KVM.

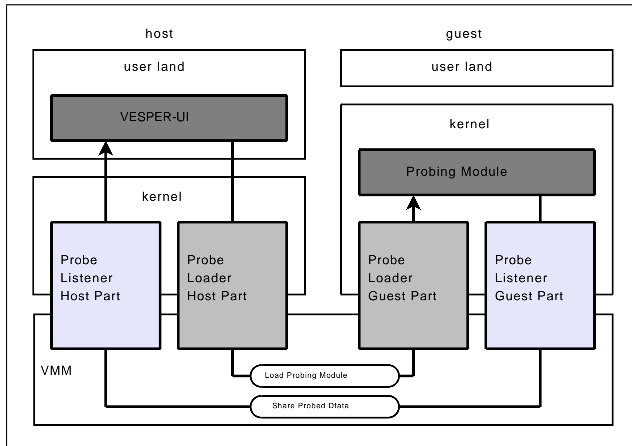



Figure 1: Architecture of VESPER

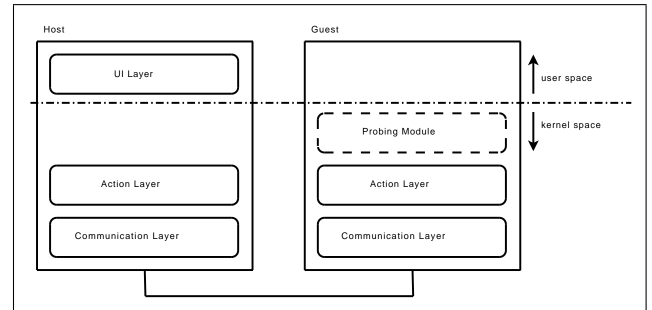



Figure 2: Layer of VESPER

# 3 Implementation of VESPER

For probing the guest, VESPER uses the tracing facilities to hook into the guest Linux kernel, and uses `relayfs` to record probed data in the probe handler of the tracing facilities. In this section, we take a brief look at the VESPER architecture and its semantics. Then we present the implementation of Virtio support in VESPER.

## 3.1 VESPER Architecture

As mentioned before, VESPER uses the tracing facilities to hook into the guest kernel. However, because they are the probing interface for the local system, they cannot implant probes into the remote system directly. Besides, in the typical use case of them, the probe handler in them comes in the form of a kernel module. Therefore, in using them to hook into the guest kernel, VESPER should be able to load probing kernel modules, on which the handlers to probe are implemented, from the host to the guest. This loading capability of VESPER is implemented as split drivers and named *Probe Loader*.

In addition, in VESPER, the probing modules use relay buffers to record data in the probe handlers. At this point, probing modules are in the guest; thus, VESPER needs to transfer the buffer data from the guest to the host. This relayed data transfer capability is also implemented as split drivers and named *Probe Listener*.

Figure 1 is the block diagram of the VESPER component.

As just described, VESPER contains two pairs of split drivers. These drivers are implemented for each VMM because they strongly depend on the underlying VMM. So, we divide VESPER into three layers (shown in Figure 2): UI Layer, Action Layer, and Communication Layer, in order to localize VMM-dependent code. This structure lets VESPER run on Xen and KVM by replacing only the VMM architecture-dependent layer—the Communication Layer.

## 3.2 VESPER Semantics

In order to implant probes into the guest, VESPER should load probing modules from the host to the guest. And then, VESPER sends probed data from the guest to the host.

Figure 3 illustrates the semantics overview of VESPER.

### 3.2.1 Module Loading

The first step to probe the guest kernel is to load the probing module onto the guest.

**0.** Make Module

First of all, one should make a probing module which uses the tracing facilities and relayfs.

**A1.** Module Load Command

Execute the probing module insertion via interfaces provided by the probe loader. One can also specify module parameters, if needed.

**A2.** Obtain Module Information

On the host-side Action Layer of the probe loader, from user space, VESPER obtains the module information to insert such as the module's name, its

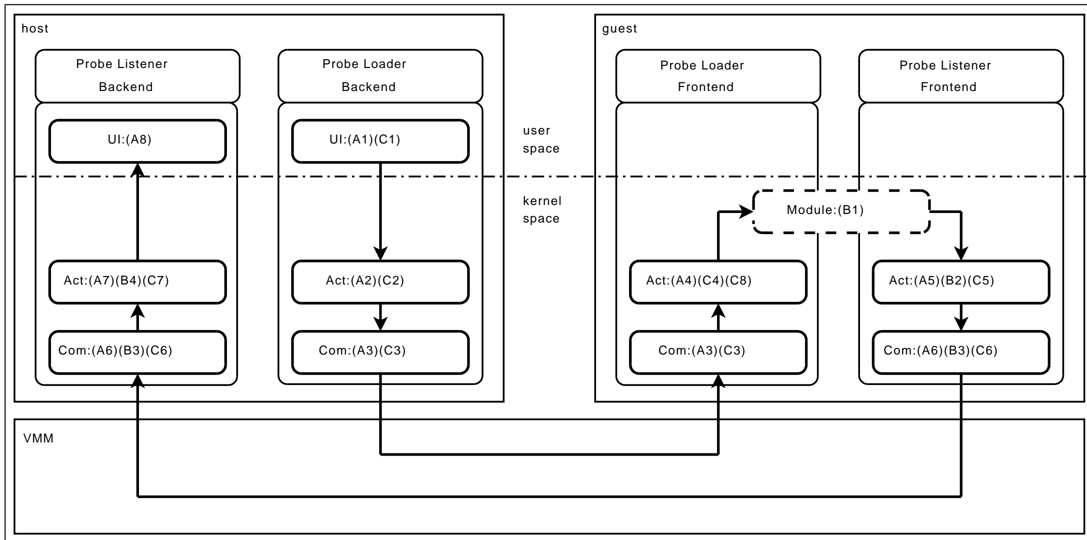



Figure 3: Process Flow of VESPER.

size, and its address with others related to module parameters, if any.

**A3.** Send/Receive Request

Through the interface provided by the VMM or Virtio, the probe loader transfers the probing module's information between the host and guest.

**A4.** Load Module

In the Action Layer, the probe loader on the guest side loads the module without userspace help.

**A5.** Share Relay Buffer

In the Action Layer, the guest's probe listener gets relayfs buffer information such as the read index, buffer ID, etc., from the probe modules; it then exports the buffer to the host.

**A6.** Send/Receive Buffer Information

Communication layer of guest probe listener transfer the shared buffer information to host via VMM interface, if any, or Virtio, and then, host probe listener receives it.

**A7.** Setup Relayfs Structure

The Action Layer of the host's probe listener builds the relayfs structure based on the information received from the guest.

**A8.** Analyze/Offer Probed Data

One can read probed data through the UI Layer of the probe listener.

### 3.2.2 Probing

After finishing the procedures in Section 3.2.1, the host and guest probe listeners share relay buffers of the probing module. Consequently, it is not necessary to transfer all the recorded data from guest to host, but it is necessary to transfer index information about shared relay buffers, where the data is, to get the start index for the actual access to relay buffers by host.

**B1.** Gather Guest Kernel Data

Once loaded, the probing module puts data into the relay buffer in the probe handler.

**B2.** Get Index Information

When change occurs in the relay sub-buffer, the action layer of the guest probe listener gets the index information and creates a message to notify host.

**B3.** Send/Receive Message

Through the communication layer, the host probe listener is notified of the index data from the guest.

**B4.** Update Index

The action layer function of the host probe listener updates the index of relayfs in the host, based on the received message.

#### 3.2.3 Module Unloading

Basically, unloading a probing module below is similar to loading a module. The significant difference is that it takes two steps in the unloading module process, because before removing the relay buffer in the handler in the guest, the exported user interface for the buffer in host should be dropped.

**C1.** Module Unload Command

**C2.** Obtain Module Information

**C3.** Send/Receive Request

**C4.** Unload Module (step1)

**C5.** Stop Sharing Relay Buffer

**C6.** Send/Receive Buffer Information

**C7.** Destroy Relayfs Buffer

**C8.** Unload Module (step2)

In the next section, we describe the detailed implementation of the probe loader and listener.

### 3.3 Implementation of VESPER supporting Virtio

As previously described, VESPER consists of two components named probe loader and probe listener. Each component is split into three parts, UI Layer, Action Layer, and Communication Layer, to confine the dependency on the underlying VMM. In this section, we present the implementation of VESPER from the viewpoint of the Communication Layer supporting Virtio.

#### 3.3.1 Probe Loader

In order to implant the probing module from the host into the guest memory space, VESPER needs to be able to transfer the module image somehow. Usually, the guest sends request. However, in this case, the host should send the module image and request to implant it to the guest memory. Therefore, the loader utilizes the event mechanism of QEMU. Like `virtio_console`, the loader attaches the watcher, to poll UI layer, to QEMU. When the module image is written to UI layer, QEMU sends the event notification to the loader. Then the loader dumps the image to `vring`, established between the host and the guest, and injects interrupt to the guest.

In writing the image to the `vring`, `struct iovec` and `struct scatterlist` translation is needed to use Virtio facilities. In QEMU, `VirtQueueElement` and `virtqueue_push` in `qemu/hw/virtio.c` helpers are prepared for the needs. On the other hand, the loader in the guest uses `sg_set_buf` and `sg_init_one` to prepare the memory where the host writes the image. To load the module into the kernel, the name of module and the size of module are needed in invoking `load_module` in the kernel. So, the header of the request, `struct virtio_loader_inhdr` is attached to inform the guest of the related information of the module. From the viewpoint of the guest, the guest is not able to find out how much memory is needed to accomplish the the request. So, the guest prepares only one page for the request and notify the memory size per request in the header of `struct virtio_loader_outhdr`. Then the host sets the left length of the module onto the header as well. Figure 4 depicts the details.

#### 3.3.2 Probe Listener

The Probe Listener should retrieve the probed data from the guest and make it available to user-space applications in the host. Probing modules use relayfs to record the probed data which describes the behavior of the guest kernel around the probe points.

Relayfs in the guest tends to allocate its buffers by pages, and the listener sets their physical address to `vring` to share them with the host. So, the listener in QEMU can see the all data on the buffers. However, those data should be copied to the relayfs in the host kernel to keep user interface to the relayfs for user application utilizing the probed data on the host. In addition, even QEMU and the relayfs can share the buffers, they should share the control information such as the read index and padding value to access proper data as well. Our design decision to tackle these problems is :

1. QEMU `mmaps` the buffers pages notified by `vring` into the action layer of the listener in the host. The action layer creates relayfs rchan with the buffers.

```
typedef struct VirtIOLoader
{
    VirtIODevice vdev;
    VirtQueue *vq;
};

typedef struct VirtIOLoaderReq
{
    VirtIOVsp *dev;
    VirtQueueElement elem;
    struct virtio_loader_inhdr *in;
    struct virtio_loader_outhdr *out;
};

struct virtio_loader_outhdr
{
    uint32_t type;
    uint8_t max_size_per_req;
    uint8_t status;
};

struct virtio_loader_inhdr
{
    uint32_t type;
    uint8_t mod_name[];
    uint64_t mod_size;
    uint64_t left_len;
};
```

Figure 4: Prototype of the device of ProbeLoader

2. Every time the relayfs in the guest updates the read index, the guest will notify the index via Virtio. QEMU will access the action layer in the host to inform it. Then the action layer updates its index of the rchan.

As a result, Probe Listener consists of two components, which are a buffer share function and an index update function, and it is implemented as split drivers, just like Probe Loader.

#### 3.3.3 Buffer Share Function

As mentioned before, because the probing module records probed data into relay buffers, Probe Listener shares them between the host and the guest. Sharing the buffers is implemented by using Virtio like the module transfer function in Probe Loader. Similarly, information about which buffers need to be shared is provided from the guest to the host by using Virtio.

Once the relay buffers are exported by the guest and their information is received by the host, the relayfs structure is built on the host to provide probed data in the buffers to user-space applications. At this time, Probe Listener in the host does not call `relay_open` to create a `rchan` structure, which is a control structure of relayfs. This is because Probe Listener does not need to newly allocate the pages for the relay buffers, but should just map the pages exported by the guest through QEMU. Therefore, Probe Listener sets up the `rchan` structure manually. After rchan is set up, the interface to read this relay buffers is created on `/sys/kernel/debug/vesper/domid/modname/` like other subsystems which use relayfs. User applications can read this interface directly.

Finally, to stop sharing the buffer, the probe listener executes the above process in reverse. Removing the relay structure is done at first, and then exporting relay buffers is stopped.

#### 3.3.4 Index Update Function

When the probe listener shares the relay buffers between the host and the guest, it must synchronize some buffer information such as the read index between both rchan structures. If it does not, the user application on the host cannot read the probed data correctly. Probe Listener uses Virtio for the information transfer. The guest's probe listener creates a message including the information, pushes it to `vring`, and then notifies the host's probe listener in QEMU. The host's probe listener gets the message from `vring` and accesses the action layer in the host to update its own relay buffer information with the message.

Ideally, probe listener should update that information immediately whenever the guest rchan is changed. However, message passing by `vring` is too expensive to update each time due to the intervention of the interrupt mechanism to notify host of the existence of pending messages. Hence, Probe Listener updates the buffer information when switching to sub-buffer occurs. In doing so, probe listener updates the buffer control information, and the user application can get the latest data probed from the guest.

Figure 5 depicts the definition of the device for Probe Listener.

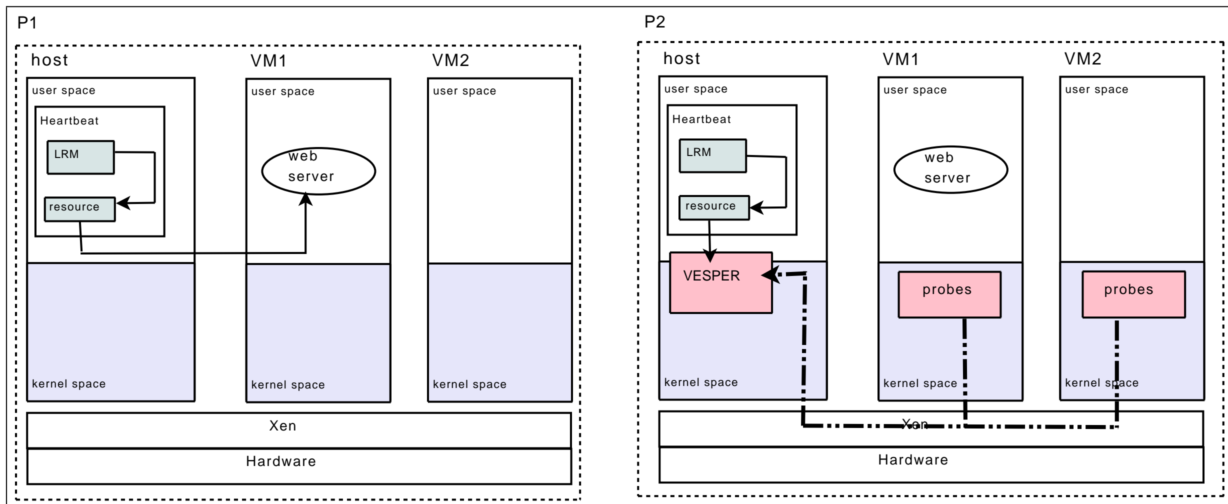



Figure 6: Test environment to demonstrate the usability of VESPER.

## 4 Tracing and evaluation

We have two different tracing facilities. one is for static tracing technology, tracepoints representatively and the other is for dynamic tracing technology, Kprobes. each technology has advantages and disadvantages compared to each other. The static tracing tends to show lower overhead than the dynamic tracing. However, the static tracing requires the kernel recompiling. So, we choose the proper tracing technology case by case. In our test environment, Figure 6, we use trace points to monitor system-wide figures like memory pressures, scheduling and etc. On the other hand, we use Kprobes to monitor application-specific figures like "signal" to the specific application.

In the environment, we prepare two physical machines. We set up two guests as resources managed by the LRM (Local Resource Manager) of Heartbeat on each physical machine. On each physical machine, the webserver is on the one guest, we say VM1; it is actively performing web service. On the other hand, the webserver in the other guest, we say VM2, is inactive.

When something wrong happens to VM1, Heartbeat lets VM2 take over all of roles which VM1 was performing. However, one physical machine, P1, has Heartbeat's LRM without involvement of VESPER to show how Heartbeat works in the usual way. However, the other physical machine, P2, has LRM cooperating with VESPER. To evaluate VESPER, we insert Kprobes around `send_sigal` and trace points probe around `out_of_memory`. Then, we send `SIGTERM` to the webserver at first. Kprobes thus inserted will put clues like the callstack to the `SIGTERM` on the relayfs. Then, we measure the recovery time on P1 and P2. We can easily expect that P2 will recognize what happened to VM1 of P2 as soon as the webserver is gone, because of the prompt notification done by VESPER. For the next, we execute a test program allocating a huge memory to occur `OOM`. Also, we measure the recovery time on P1 and P2. In this case, P1 did not recognize `OOM` because the test program is killed.

Through the experiment, we could verify better performance on response latency with VESPER.

However, special care should be taken with two issues regarding probes. One is what probe points are suitable for proper monitoring. If the targeted, necessary probe points miss, no more improvement over usual Heartbeat can be expected. Actually, the problem on where probe points should be inserted seems very tricky to handle, because highly experienced developers or system administrators on kernel context and applications running on the server are required to select optimal probe points. The other is about overhead produced by the execution of probes especially in dynamic tracing tracing technology. One should adjust the overhead according to the required service performance. Both issues exclude each other. More probes inserted to hit fine-grained events cause more overhead in probing, obviously. Some mechanism to help one select optimal probes could be needed. Some suggestions for these issues will be mentioned as future works of VESPER in the next section.

```
typedef struct VirtIOListener
{
    VirtIODevice vdev;
    VirtQueue *vq;
};

typedef struct VirtIOListenerReq
{
    VirtIOVsp *dev;
    VirtQueueElement elem;
    struct virtio_listener_inhdr *in;
    struct virtio_listener_outhdr *out;
};

struct virtio_listener_outhdr
{
    uint32_t type;
    unit8_t guestid;
    target_phys_addr_t buf_addr;
    uint8_t mod_name[];
    uint64_t buf_size;
};

struct virtio_listener_inhdr
{
    uint8_t status;
};
```

Figure 5: Prototype of the device of ProbeListener

## 5 Conclusion and future works

In this paper, we expanded VESPER to Virtio on KVM as a framework to insert probes in virtualized environments and discussed what topics VESPER can solve in clustering computing. After that, we described the design and the implementation of VESPER. Then we suggested a test bed to show the performance improvement on fail-over response latency. Finally we discussed some considerations on places and overhead of probing. To address these considerations, we have some plans for future VESPER developments.

### 5.1 Probing aid subsystem

For the ease use of cluster manager or other applications, we plan to develop a probing aid subsystem. Probing points could be classified into several groups based on their functionality. The subsystem thus can pre-define several groups of probe points and abstract them to its clients or application—like memory group, network group, block-io group, etc. The clients just select one of groups, and the subsystem will generate all needed probes relayed to VESPER. Also, fine-grained selection from several groups will be supported by the subsystem.

### 5.2 Precaution capability on collapse of the host

If the host collapsed, all services running on the guests would be lost. It is obviously a big problem. Therefore, VESPER should probe the host simultaneously to check whether the host is in good condition. When VESPER catches a sign of the host's collapse, the cluster manager notified by VESPER could take necessary action, such as live migration to other host.

## 6 Acknowledgements

We would like to thank our colleagues for reviewing this paper.

## 7 Legal Statements

Linux is a registered trademark of Linus Torvalds. Other company, product, and service names may be trademarks or service marks of others.

## References

[1] LKML, http://lkml.org

[2] Alan Robertson, "Linux-HA Heartbeat Design," In *Proceedings of the 4th International Linux Showcase and Conference*, 2000.

[3] Heartbeat, http://linux-ha.org.

[4] VESPER, http://vesper.sourceforge.net/.

[5] The Xen vitual machine monitor, http://www.cl.cam.ac.uk/Research/SRG/netos/xen/.

[6] Virtio, http://lwn.net/Articles/239238.

[7] KVM, http://kvm.qumranet.com/.

[8] QEMU, http://www.qemu.org

# Hardware Breakpoint (or watchpoint) usage in Linux Kernel

Prasad Krishnan
*IBM Linux Technology Centre*
`prasad@linux.vnet.ibm.com`

## Abstract

The Hardware Breakpoint (also known as watchpoint or debug) registers, hitherto was a frugally used resource in the Linux Kernel (ptrace and in-kernel debuggers being the users), with little co-operation between the users. The role of these debug registers is best exemplified in a) Nailing down the cause of memory corruption, which be tricky considering that the symptoms manifest long after the actual problem has occurred and have serious consequences—the worst being a system/application crash. b) Gain better knowledge of data access patterns to hint the compiler to generate better performing code. These debug registers can trigger exceptions upon `events` (memory read/write/execute accesses) performed on `monitored` address locations to aid diagnosis of memory corruption and generation of profile data.

This paper will introduce the new generic interfaces and the underlying features of the abstraction layer for HW Breakpoint registers in Linux Kernel. The audience will also be introduced to some of the design challenges in developing interfaces over a highly diverse resource such as HW Breakpoint registers, along with a note on the future enhancements to the infrastructure.

## 1 Introduction

Hardware Breakpoint interfaces introduced to the Linux kernel provide an elegant mechanism to monitor memory access or instruction executions. Such monitoring is very vital when debugging the system for data corruption. It can also be done to with a view to understand memory access patterns and fine-tune the system for optimal performance.

The hardware breakpoint registers in several processors provide a mechanism to interrupt the programmed execution path to notify the user through of a hardware breakpoint exception. We shall examine the details of such an operation in the subsequent sections.

Possibly, the biggest convenience of using the hardware debug registers is that it causes no alteration in the normal execution of the kernel or user-space when unused, and has no run-time impact. The most notable limitation of this facility is the fewer number of debug registers on most processors.

## 2 Hardware Breakpoint basics

A hardware breakpoint register's primary (and only) task is to raise an exception when the monitored location is accessed. However these registers are processor specific and their diversity manifests in several forms—layout of the registers, modes of triggering the breakpoint exception (such as exception being triggered either before or after the memory access operation) and types of memory accesses that can be monitored by the processor (such as read, write or execute).

### 2.1 Hardware breakpoint basics—An overview of x86, PPC64 and S390

Table 1 provides a quick overview of the breakpoint feature in various processors and compares them against each other [1, 2].

## 3 Design Overview of Hardware Breakpoint infrastructure

### 3.1 Register allocation for kernel and user-space requests

While debug registers would treat every breakpoint address in the same way, there is a fundamental difference in the way kernel and user-space breakpoint

| Features / Processor | Register Name | Number of Breakpoints | Data(D) / Instructions(I) | Breakpoint lengths (length in Bytes) |
|---|---|---|---|---|
| x86/x86_64 | Debug register (DR) | 4 | D / I | 1, 2, 4 and 8 (x86_64 only) |
| PPC64 | Data(Instruction) Address Breakpoint register (DABR / IABR) | 1 | D / I (on selected processors only) | 8 |
| S390 | Program Event Recording (PER) | 1 | D / I | Varied length. Can monitor range of addresses |

Table 1: Processor Support Matrix

requests are effected. A user-space breakpoint belonging to one thread (and hence stored in `struct thread_struct`) will be active only on one processor at any given point of time. The kernel-space breakpoints, on the other hand should remain active on all processors of the system to remain effective since each of them can potentially run kernel code any time. This necessitates the propagation of kernel-space requests for (un)registration to all processors and is done through inter-processor interrupts (`IPI`). The per-thread user-space breakpoint takes effect only just before the thread is scheduled. This means that a system at run-time can have as many breakpoint requests as the number of threads running and the number of free (i.e., not in use by kernel) breakpoint registers put together (number of threads x number of available breakpoint registers) since they can be active simultaneously without interfering with each other.

On architectures (such as x86) containing more than one debug register per processor, the infrastructure arbitrates between requests from multiple sources. To achieve this, the implementation submitted to the Linux community (refer [3]) makes certain assumptions about the nature of requests for breakpoint registers from user-space through ptrace syscall, and simplifies the design based on them.

The register allocation is done on a first-come, first-serve basis with the kernel-space requests being accommodated starting from the highest numbered debug register growing towards the lowest; while user-space requests are granted debug registers starting from the lowest numbered register. Thus in case of x86, the infrastructure begins looking out for free registers beginning from DR0 while for kernel-space requests it will begin with DR3 thus reducing the scope for conflict of requests.

In order to avoid fragmentation of debug registers upon an unregistration operation, all kernel-space breakpoints are "compacted" by shifting the debug register values by one-level although this is not possible for user-space requests as it would break the semantics of existing ptrace implementation. This implies that even if a user-thread downgraded its usage of breakpoints from `n` to `n - 1`, the breakpoint infrastructure will continue to reserve `n` debug registers. A solution for this has been proposed in Section 8.1.

### 3.2 Register Bookkeeping

Accounting of free and used debug registers is essential for effective arbitration of requests, and allows multiple users to exist concurrently. Debug register bookkeeping is done with the help of following variables and structures.

`hbp_kernel[]` – An array containing the list of kernel-space breakpoint structures

`this_hbp_kernel[]` – A per-cpu copy of `hbp_kernel` maintained to mitigate the problem discussed in Section 7.2.

`hbp_kernel_pos` – Variable denoting the next available debug register number past the last kernel breakpoint. It is equal to `HBP_NUM` at the time of initialisation.

`hbp_user_refcount[]` – An array containing refcount of threads using a given debug register number. Thus a value `x` in any element of index `n` will indicate that there are `x` number of threads in user-space that currently use `n` number of breakpoints, and so on.

A system can accommodate new requests for breakpoints as long as the kernel-space breakpoints and those

of any given thread (after accounting for the new request) in the system can be fit into the available debug registers. In essence,

```
Debug registers >= Kernel Breakpoints
+ Max(Breakpoints in use by any given
thread)
```

### 3.3 Optimisation through lazy debug register switching

The removal of user-space breakpoint, happens not immediately when it is context switched-out of the processor but only upon scheduling another thread that uses the debug register in what we term as *lazy debug register switching*. It is a minor optimisation that reduces the overhead associated with storing/restoring breakpoints associated with each thread during context switch between various threads or processes. A thread that uses debug registers is flagged with `TIF_DEBUG` in the `flag` member in `struct thread_info`, and such threads are usually sparse in the system. If we must clear the user-space requests from the debug registers at the time of context-switch (in `__switch_to()` itself), it could be done either

- unconditionally on all debug registers not used by the kernel or
- only if the thread exiting the CPU had `TIF_DEBUG` flag set (which is false for a majority of the threads in the system).

In both the cases, we would add a constant overhead to the context-switching code irrespective of any thread using the debug register.

## 4 The Hardware Breakpoint interface

### 4.1 Hardware Breakpoint registration

The interfaces for hardware breakpoint registration for kernel and user space addresses have signatures as noted in Figure 2.

A call to register a breakpoint is accompanied by a pointer to the breakpoint structure populated with certain attributes of which some are architecture-specific.

```
int register_kernel_hw_breakpoint(struct hw_breakpoint *bp);
int register_user_hw_breakpoint(struct task_struct *tsk,
                                struct hw_breakpoint *bp);
```

Figure 2: Hardware Breakpoint interfaces for registration of kernel and user space addresses

```
struct hw_breakpoint {
        void (*triggered)(struct hw_breakpoint *,
                          struct pt_regs *);
        struct arch_hw_breakpoint info;
};
```

Figure 3: Hardware Breakpoint structure

The generic breakpoint structure in the Linux kernel of -tip git tree presently looks as seen in Figure 3.

The `triggered` points to the call-back routine to be invoked from the exception context, while `info` contains architecture-specific attributes such as breakpoint length, type and address.

A breakpoint register request through these interfaces does not guarantee the allocation of a debug register and it is important to check its return value to determine success.

Unavailability of free hardware breakpoint registers can be most common reason since hardware breakpoint registers are a scarce resource on most processors. The return code in this case is `-ENOSPC`.

The breakpoint request can be treated as invalid if one of the following is true.

- Unsupported breakpoint length
- Unaligned addresses
- Incorrect specification of monitored variable name
- Limitations of register allocation mechanism

#### 4.1.1 Unsupported breakpoint length

While the breakpoint register can usually store one address, the processor can be configured to monitor accesses for a range of addresses (using the stored address

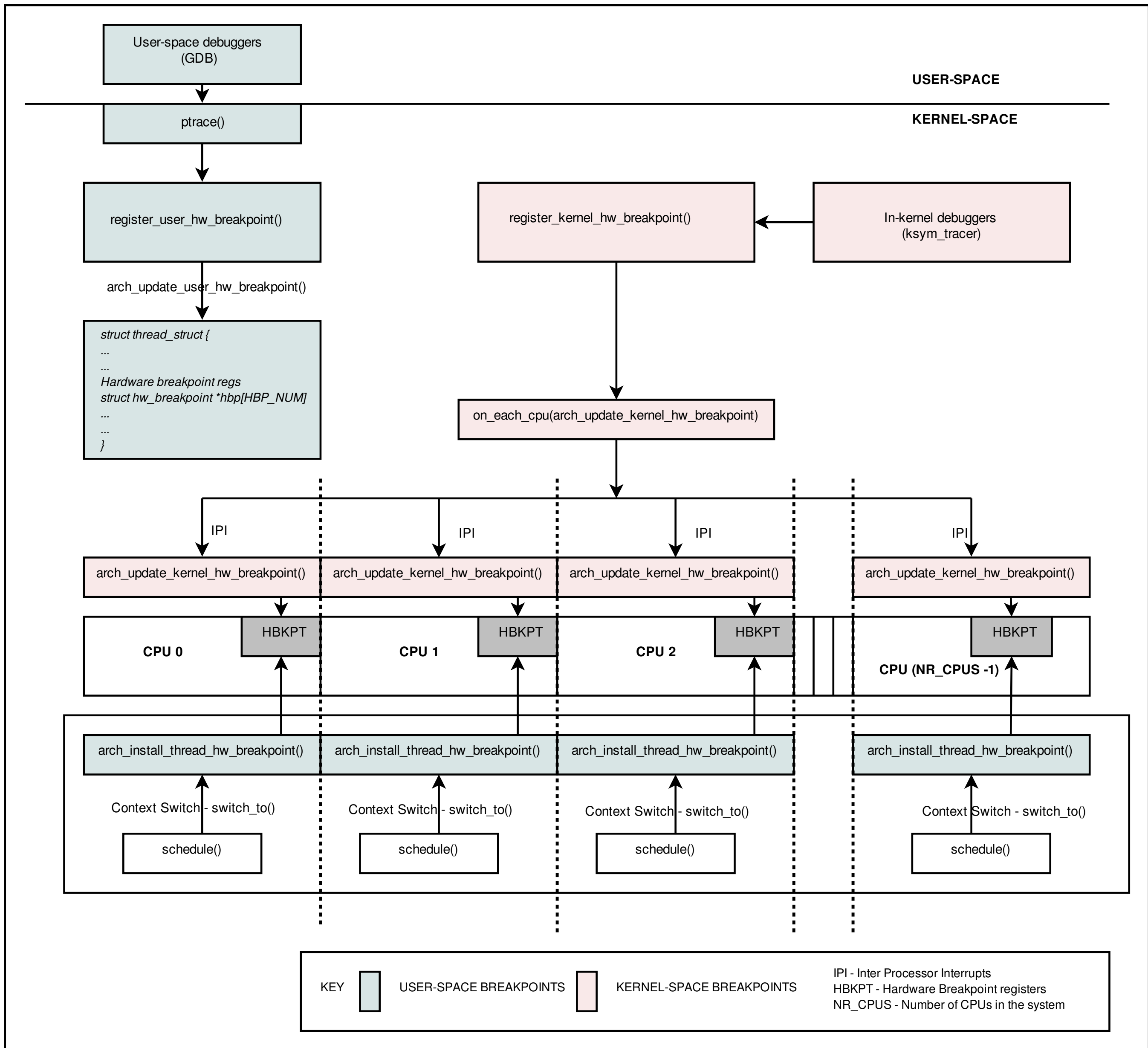



Figure 1: This figure illustrates the handling of requests from kernel and user-space by the breakpoint infrastructure

as a base). For instance, in certain x86_64 processor types, up to four different byte ranges of addresses can be monitored depending upon the configuration. They are byte length of 1, 2, 4, and 8. However on PPC64 architectures, this is always a constant of 8 bytes. Thus a given breakpoint request can be treated as valid or otherwise depending upon the host processor. The arch-specific structure is designed to contain only those fields that are essential for proper initiation of a breakpoint request and all constant values are hard-wired inside the architecture code itself.

### 4.1.2 Unaligned addresses

Certain processors have register layouts that impose alignment requirements on the breakpoint address. The alignment requirements are in consonance with the breakpoint lengths supported on these processors. For instance, in x86 processors the supported lengths as we know are 1, 2, 4, and 8 bytes which in turn dictates that the addresses must be aligned to 0, 1, 3, and 7 bytes.

#### 4.1.3 Incorrect specification of monitored variable name

The breakpoint interface is designed to accept kernel symbol names directly as input for the location to be monitored by the breakpoint registers. Invalid values can be the result of incorrect symbol name. Since user-space symbols cannot be resolved to their addresses in the kernel, their breakpoint requests would fail if accompanied by a symbol name. As a means to resolve a conflict, that may arise when incoherent kernel symbol and address are mentioned, the address is considered valid and the supersedes the kernel symbol name.

#### 4.1.4 Limitations of register allocation mechanism

The register allocation mechanism (as discussed in the Design overview section above) may also result in failure of registration due to lack of debug registers despite availability of a different numbered physical register. This is identified as a limitation of the present debug register allocation scheme, and virtualisation of debug registers is planned as a solution for the same.

At the end of a successful registration request the user can assume that the request for breakpoints are effected by storing kernel-space request on all CPUs and user-space requests only when the process is scheduled.

### 4.2 Hardware Breakpoint handler execution

Almost all of the hardware breakpoint exception handling code is in architecture specific code. This is due to the fact that each architecture handles the breakpoint exception in its handler code differently.

However a few operations are common to the handlers designed for x86 and PPC64. The primary objective of the exception handler is to trigger the function registered as a callback during breakpoint registration, which requires access to the correct instance of `struct hw_breakpoint` that was provided to the breakpoint interface during registration. The correct breakpoint structure has to be deciphered from a set of user-space and kernel-space breakpoint requests.

#### 4.2.1 Identification of stray exceptions

But before that, the handler execution code must be resilient to recognise stray exceptions and ignore them. Such stray exceptions can be the result of one of causes detailed below.

Memory access operations on addresses that are outside the monitored variable's address range but within the breakpoint length. For instance, on PPC64 processors the `DABR` always monitors for memory access operations (as specified in the last two bits of DABR) in the double word (8 bytes) starting from the address in the register. However the user's request would be limited to only a given kernel variable (whose size is smaller than a double-word). Hence any accesses in the memory region adjacent to the monitored variable falling within the breakpoint length's scope causes the breakpoint exception to trigger.

Lazy debug register switching causes stale data to be present in debug registers (as discussed above in Section 3.3) and can give rise to spurious exceptions. This typically happens when a process accesses memory locations that were monitored previously by a different process but are not reset due to lazy switching.

#### 4.2.2 Identification of breakpoint structure for invocation of callback function

The user-space breakpoint requests are thread-specific and so, stored in the `struct thread_struct`, while kernel-space breakpoints being universal are stored in global kernel data structures, namely `hbp_kernel` as noted above in Section 3.2.

On x86 processors, which provide four debug registers it is more challenging to identify the corresponding breakpoint structure, when compared to architectures that allow only one breakpoint at any point in time. Upon encountering a breakpoint exception, the bit settings in the status register for debugging `DR6` is looked upon. Based on the bits that are set, the appropriate breakpoint address register (DR0-DR3) is understood to have been the cause for the exception. Depending upon whether the register was used by the kernel or user-space the breakpoint structure is retrieved from either the kernel's global data structure or the process' instance of the per-thread structure respectively.

```
void unregister_kernel_hw_breakpoint(struct hw_breakpoint *);
void unregister_user_hw_breakpoint(struct hw_breakpoint *,
                                  struct task_struct *);
```

Figure 4: Hardware Breakpoint interfaces for unregistration of kernel and user space addresses

Using such architecture-specific methods to identify the appropriate breakpoint structure, the user-defined callback function is invoked.

This will be followed by post processing, which may include single-stepping of the causative instruction in architectures where the breakpoint exception is taken when the impending instruction will cause the memory operation monitored by the debug register.

Since the breakpoint handler is invoked through a notifier call chain, the return code is used to decide if the remaining handlers have to be invoked further. Detection of multiple causes for the exception will then be required to choose the appropriate return code and will form part of the post processing code.

### 4.3 Hardware Breakpoint unregistration

Hardware breakpoint unregistration is done by invoking the appropriate kernel or user interface with a pointer to the instance of breakpoint structure. An invocation to the interface always results in successful removal of the breakpoint and hence doesn't return any value to indicate success or failure. The interfaces are as shown in 4.

#### 4.3.1 Need for per-cpu kernel breakpoint structures

It is much safer and easier to remove user-space breakpoints, compared to kernel-space requests (refer to 7 section for a related issue). It requires updating of the appropriate bookkeeping counters and per-thread data structures containing breakpoint information (apart from clearing the physical debug registers). While processing user-space unregistration requests, if the breakpoint removal causes the any member of `hbp_user_refcount[]` to turn into zero (i.e., result in a state where there are no threads using the debug register corresponding to the array index of the member that turned zero), it indicates the availability of one new free debug register since the last user of that debug register has released the resource.

Kernel-space breakpoints are loaded onto all debug registers to the obvious fact that the kernel-code may be executed on any and all processors at any given point of time unlike the thread-specific breakpoints which run only on one processor at any given instant.

Thus a removal request for kernel-space breakpoints should be propagated to all processors (in the same fashion as a registration request) through inter-processor interrupts. The process of unregistration is complete only when the callbacks through the IPI in each of the CPU returns.

Sample output from ksym tracer

```
# tracer: ksym_tracer
#
#      TASK-PID     CPU#      Symbol        Type    Function
#         |           |         |             |       |
bash          30897 3   pid_max             RW .do_proc_dointvec_minmax_conv+0x78/0x10c
bash          30897 3   pid_max             RW .do_proc_dointvec_minmax_conv+0xa0/0x10c
bash          30897 3   pid_max             RW .alloc_pid+0x8c/0x4a4
bash          30897 1   pid_max             RW .alloc_pid+0x8c/0x4a4
```

Figure 5: Sample output from ksym tracer collected when tracing pid_max kernel variable for read and write operations

## 5 Beyond debugging of memory corruption—Ftrace, memory access tracing and data profiling

The ksym tracer is a plugin to the ftrace framework that allows a user to quickly trace a kernel variable for certain memory access operations and collect information about the initiator of the access.

It provides an easy-to-use interface to the user to accept the kernel variable and a set of memory operations for which the variable will be monitored. While, it is currently restricted to trace only in-kernel global variables, the ksym_tracer's parser can be extended to accept module variables and kernel-space addresses as input.

These traces can help in profiling memory access operations over data locations such as read-mostly or write-mostly.

| Operation / Machine | register_kernel | | unregister_kernel | |
|---|---|---|---|---|
| | System A | System B | System A | System B |
| Trial 1 | 5066 | 5770 | 244 | 24 |
| Trial 2 | 5319 | 6279 | 204 | 21 |
| Trial 3 | 5309 | 6193 | 228 | 20 |
| Trial 4 | 6068 | 6092 | 206 | 18 |

Table 2: Time taken for (un)register_kernel operation in micro-seconds

| Operation / Machine | Breakpoint handler | |
|---|---|---|
| | System A | System B |
| Trial 1 | 2230 | 4677 |
| Trial 2 | 1980 | 4255 |
| Trial 3 | 1805 | 4224 |
| Trial 4 | 1644 | 4035 |

Table 3: Time taken for breakpoint handler with a dummy callback function (in nano-seconds)

# 6 Overhead measurements of triggering breakpoints

Readings of the following measurements have been tabulated.

- Table 2 – Contains overhead measurements for register and unregister requests on two systems.
- Table 3 – Average time taken for the breakpoint handler execution with a dummy trigger in four different trials on two systems.

The trials were conducted on two machines, System A and B whose specifications are as below.

System A – 24 CPU x86_64 machine Intel(R) Xeon(R) MP 4000 MHz

System B – 2 CPU i386 Intel(R) Pentium(R) 4 CPU 3.20GHz

These systems, chosen for tests are sufficiently diverse in the number of CPUs in them to expose the overhead caused by of IPIs in the `(un)register_kernel_hw_breakpoint()` operations. The readings were taken without any true workload on the systems.

While the overhead for unregister operations is greater in System A (with many CPUs), interestingly this behaviour does not manifest during the register operations (Refer to Table 2).

# 7 Challenges

Among the the goals set during the design of the hardware breakpoint infrastructure, a few to mention are:

- provide a generic interface that abstracts out the arch-specific variations in breakpoint facility and allowing the end-user to harness this facility through a consistent interface
- provide a well-defined breakpoint execution handler behaviour despite the nuances in such as trigger-before-execute and trigger-after-execute (which are dependant on the type of breakpoint and the host processor)
- balance between the the need for a uniform behaviour and exploitation of unique processor features

The implementation of such goals gave rise to challenges, some of which are discussed here.

## 7.1 Ptrace integration

The user-space has been the most common user of hardware breakpoints through the ptrace system call. Ptrace interface's ability to read or write from/into any physical register has been exploited to enable breakpoints for user-space addresses. While it required little or no knowledge about the host architecture's debug registers, it remained the responsibility of the application invoking ptrace (such as GNU Debugger GDB) to be a knowledgeable user and activate/disable them through appropriate control information.

For instance, on x86 processors containing multiple debug registers and dedicated control and status registers (unlike in PPC64 where the control and debug address registers are composite), operations such as read and write become non-trivial—i.e., every request for a new breakpoint must require one write operation on the debug address register (DR0 - DR3) and one for the control register.

Since ptrace is exposed to the user-space as a system call it is important to preserve its error return behaviour. Achieving this becomes complicated because of the fact that ptrace and its user in the user-space assumes exclusive availability of the debug registers and are ignorant

of any kernel space users. Hence, the number of available registers may be lesser than the ptrace user's assumption and may result in failure of request when not expected.

On architectures like x86 where the status of multiple breakpoint requests can be modified through one ptrace call (using a write operation on debug control register DR7), care is taken to avoid a partially fulfilled request to prevent the debug registers from gaining a set of values that is different from the ptrace's requested values and its past state. Consider a case where, among the four debug registers, one was active and the remaining three were disabled in the initial state. If the new request through ptrace was to de-activate the single active breakpoint and enable the rest of them, then we do not effect the breakpoint unregistration first but begin with the registration requests and this is done for a reason.

Supposing that one of the breakpoint register operation fails (due to one of the reasons noted above in Section 4.1) and if it was preceded by the unregister operation the result of the ptrace call is still considered a failure. The state of the debug registers must now be restored to its previous one which implies that the breakpoint unregistration operation must be reversed. Under certain conditions this may not be possible leaving the debug registers with an altogether new set of values.

Thus all breakpoint disable requests in ptrace for x86 is processed only after successful registration requests if any. This prevents a window of opportunity for debug register grabbing by other requests thereafter leading to a problem as described above.

### 7.2 Synchronised removal of kernel breakpoints

A kernel breakpoint unregistration request would require updating of the global kernel breakpoint structure and debug registers of all CPUs in the system (similar to the process of registration). However every processor is susceptible to receive a breakpoint exception from the breakpoint that is pending removal although the related global data structures may be cleared by then causing indeterminate behaviour.

This potential issue was circumvented by storing a per-cpu copy of the global kernel breakpoint structures which would be updated in the context of IPI processing. It enables every processor to continue to receive and handle exceptions through its own copy of the breakpoint data until removed. Although this generates multiple copies of the same global data, it is much preferred over the alternatives such as global disabling of breakpoints (through IPIs) before every unregister operation, due to the overhead associated with processing the IPIs (Refer Table 2 for data containing turnaround time for register/unregister operations).

## 8 Future enhancements

Enhanced abstraction of the interface to include definitions of attributes that are common to several architectures (such as read/write breakpoint types), widening the support for more processors, improvements to the capabilities, interface and output of `ksym_tracer`; creation of more end-users to support the breakpoint infrastructure such as "perfcounters" and SystemTap in innovative ways are just a few enhancements contemplated at the moment for this feature.

Virtualised debug registers was a feature in one of the versions of the patchset submitted to the Linux community but was eventually dropped in favour of a simplified approach to register allocation. The details of the feature and benefits are detailed below.

### 8.1 Virtualisation of Debug registers

In processors having multiple registers such as x86, requests for breakpoint from ptrace are targeted for specific numbered debug register and is not a generic request. While this mechanism works well in the absence of any register allocation mechanism and when requests from user-space have exclusive access to the debug registers, their inter-operability with other users is affected.

The hardware breakpoint infrastructure discussed here, mitigates this problem to a certain extent by using the fact that requests from ptrace tend to grow *upwards*—i.e., starting from the lower numbered register to the higher ones.

A true solution to this problem lies in creating a thin layer that maps the physical debug registers to those requested by ptrace and allow the any free debug register to be allocated irrespective of the requested register number. The ptrace request can continue to access through the `virtual debug register` thus allocated.

### 8.2 Prioritisation of breakpoint requests

Allow the user to specify the priority for breakpoint requests to be handled. If a breakpoint request with a higher priority arrives, the existing breakpoint yields the debug register to accommodate the former. An accompaniment to this feature would be the callback routines that are invoked whenever a breakpoint request is preempted or regains the debug registers on the processor. This is done at the time of every new registration to balance the requests and accommodate requests based on their priorities.

This feature was a part of the original patchset but was subsequently removed based on community feedback [4].

## 9 Conclusions

The Hardware Breakpoint infrastructure and the associated consumers of the infrastructure such as `ksym_tracer` makes available a hitherto scarcely used hardware resource to good use in newer ways such as profiling and tracing apart from their vital roles in debugging. The overhead in taking a breakpoint, as our results in Section 6 show are tolerable even in production environments and if any would be the result of the user-defined callback function. It is hoped that when the patches head into the mainline kernel, a wider user-feedback and testing will help evolve the infrastructure into a more powerful and robust one than the proposed.

## 10 Acknowledgements

The author wishes to thank his team at Linux Technology Centre, IBM and the management for their encouragement and support during the creation of the hardware breakpoint patchset and the paper.

The profound work done by Alan Stern, whose patchset and ideas were the foundation for the present code in -tip tree, and an earlier patchset from Prasanna S Panchamukhi need a mention of thanks from the author.

The design of this feature is heavily influenced by suggestions from Ingo Molnar and code was vetted by Ananth N Mavinakayanahalli, Frederic Weisbecker and Maneesh Soni; also benefiting from the in-depth review of the patches by Alan Stern. The author gratefully acknowledges their contribution.

Special thanks to Balbir Singh for initiating the author into the creation of this paper and being a great source of encouragement throughout.

The author wishes to thank Naren A Devaiah and the IBM management who generously provided an opportunity to work on this feature and paper, without which its presentation at the Linux Symposium 2009 wouldn't have been possible.

## 11 Legal Statements

© International Business Machines Corporation 2007. Permission to redistribute in accordance with Linux Symposium submission guidelines is granted; all other rights reserved.

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

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

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

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

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

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

## References

[1] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2008. `www.intel.com/Assets/PDF/manual/253669.pdf`.

[2] International Business Machines Corporation. *Power ISA™ Version 2.05*, 2007. `http://www.power.org/resources/reading/PowerISA_V2.05.pdf`.

[3] K. Prasad. Hardware breakpoint interfaces, June 2009. `http://lkml.org/lkml/2009/6/1/282`.

[4] K. Prasad. Introducing generic hardware breakpoint handler interfaces, March 2009. `http://lkml.org/lkml/2009/3/10/183`.

# Shoot first and stop the OS noise

Dealing with microsecond latency requirements

Christopher Lameter
*Linux Foundation*
cl@linux-foundation.org

## Abstract

Latency requirements for Linux software can be extreme. One example is the financial industry: Whoever can create an order first in response to a change in market conditions has an advantage. In the high performance computing area a huge number of calculations must occur consistently with low latencies on large number of processors in order to make it possible for rendezvous to occur with sufficient frequency. Games are another case where low latency is important. In games it is a matter of survival. Whoever shoots first will win.

An operating system causes some interference with user space processing through scheduling, interrupts, timers, and other events. The application code sees execution being delayed for no discernible reason and a variance in execution time due to cache pollution by the operating system. Low latency applications are impacted in a significant way by OS noise.

We will investigate issues in software and hardware for low latency applications and show how the OS noise has been increasing in recent kernel versions.

## 1 Introduction

Operating system noise is something of a mystery to most user space programmers. The expectation by those writing the application is that the operating system is simply letting the application run. Users see the main function of the Operating System to provide resources for the program to run effectively. In the case of OS noise the Operating System itself becomes a problem because the OS is interfering with the application by making uses of the processor for maintenance tasks or operating system threads that also have to run on the processor. Running OS code may impact the application which will not perform as expected. The execution times of critical code segments in the application may vary a lot without any discernible reason. One hears complaints from users that the OS should just get out of the way. The problem becomes more severe as the number processors increases and as interconnects become faster.[1] The timing requirements for critical sections move from milliseconds to microseconds. Modern processors can perform a significant amount of work in a microsecond provided that there are no latencies associated with the data needed for processing. Therefore, processor caches have a significant effect on the latencies of critical code segments. The OS code causes disturbances in the processor caches that has an effect long after the application continues execution.

The problem was first noticed first in High Performance Computing.[2] HPC applications typically go through a highly parallel calculation phase and then a rendezvous phase in which the results of the calculations are exchanged. It was noted that the calculation phase was rarely performing within the boundaries expected. The problem became worse as the number of processor increased. The investigation found that the rendezvous phase will be delayed if any one of the processes is held up due to OS interference (like for example a timer interrupt). The more processors exist the more likely the chance that OS interference will cause a delay. This is especially severe on Linux due to the staggering of the timer interrupts over all processors in the system. As a result the timer interrupts will not run all at the same time (which would potentially overload the interconnect between the processors). The more processors a system has the shorter the period that no timer interrupt is running on any processor and the more likely that the

[1] Infiniband hardware can f.e. perform transfers between machine in 1-3 microseconds!

[2] See especially Petrini, 2003

rendezvous phases are delayed due to a single processor being hit with OS interference. This can lead to severe performance regressions so that some vendors have started to modify the scheduler to have special synchronized periods dedicated to OS processing in order to be able to execute concurrently on all processors without OS interference during other times.[3]

In the financial industry we see a arms race to lower latencies.[4]

Latencies for the exchange of financial data and for trading used to be measured in milliseconds but that has now come to focus on microseconds. Whoever can react in the fastest way to changing market conditions may take advantage of a favorable trade opportunity. The latency requirements in the financial sector focus more on networking and on the need of fast processing of huge and complex sets of data. The classic decision support system (DSS) paradigm is taken to extremes there. Stopping the noise results in concrete market advantages.

A similar move is seen in the gaming industry. There also a growing focus on smaller and smaller intervals for critical processing develops. If interactive computer games are played over the Internet then the focus is mostly on millisecond latencies since the WAN links do not allow smaller latencies. This limits the richness of interactivity of computer games. Recently there has been an increased move towards putting interactive games on LANs where players are in local proximity (LAN parties). Gaming software can exchange large sets of information in sub millisecond time frames in such configurations. There it is likely that we will also facing issues with microsecond latency demands in the future and therefore OS noise is also becoming factor. OS noise there can determine whoever will be able to shoot first. One side effect of latency in shooter games is that the bullet of the slower machine seems to hit the target (since the slower machine was not able to acquire the updated position of the enemy) but the enemy takes no damage since the person has already moved on in the game servers reckoning and the game server determined that the shot missed the target. The enemy is hit, it dies a horrible death on the screen of the shooter and then suddenly continues running down the corridor.[5]

[3]See Beckman 2008, 5. Tsafir 2008, section 4. Petrini 2003, 10

[4]F.e. 29West–a major player for middleware in the financial area–recently announced a vision for zero latency.

[5]See http://en.wikipedia.org/wiki/Multiplayer

## 2 What is Operating System Noise

What exactly is Operating System noise? The common definition in use is any disturbance caused by the OS making use of a processor. I would like to extend that definition to cover everything not under the control of the application that has a negative impact on performance and latencies observed by code running in user mode. This goes beyond the strict notion of OS noise and more towards a notion of general noise (maybe better called *system noise*) that impact on an applications performance. Noise is not only the result of interruption of code execution (be it the periodic timer interrupt, device interrupts, software interrupts, faults and so on) but also memory subsystem disturbances due to the Operating Systems putting pressure on the cache subsystems of the processor which causes application cache line refetches.[6]

The use of on chip resources of a CPU by an OS or another application are important. Resources commonly available are the processor caches, the TLB entries, page tables and various register copies. Contention on all these levels can reduce the performance of the application. If the OS scans through a large list of objects in a regular way[7] then a large number of TLB entries may be evicted that have to be re fetched from memory later. If memory becomes scarce and the OS evicts pages from memory then the eviction may have a significant latency effect since the evicted pages will have to be re-read from secondary storage (such as a hard disk) when it is needed again.

Processors are also not isolated from each others. The notion of "CPU" that the Linux OS has is basically a hardware managed execution context. These can share caches with other "CPUs" on various levels. If cache sharing occurs between multiple of the CPUs then a process on another processor can cause cache lines of the application to be evicted, can use TLB lines and other processor resources that cause latencies for the application. The most significant effects occurs if multiple execution context share all resources of the processor like for example in hyper-threading. Operating system schedulers (like the Linux scheduler) currently only have simple handling of these dependencies and rely

[6]According to Beckman cache line eviction is the major effect increasing the latency of critical sections. But that may depend on the type of load running.

[7]Like for example done by the SLAB allocator in Linux

mainly on heuristics. This leads to a situation in which increasing load customarily leads to a general slowdown of all processes running on the machine once all cpus are actively processing data.

# 3 Latency Overview

In order to talk in a meaningful way about latencies it is important to know what these time frames represent in reality. One thing that is often forgotten is that telecommunication or general signal latencies are limited by the speed of light (300.000km/sec). The relativistic limits become significant when signals have to run over long distances. Whenever signals must travel across a WAN link latencies in the range of milliseconds become unavoidable. Signals travel over fiber optic or copper links at the speed of around 200.000km/sec. Since the earth has a circumference of 40.000km: A signal that is supposed to reach any point on the earth (the earth is round so we can reach any point within 20.000km) must account for a minimal latency of 1/10th of a second.

Here is a list of latencies and how they apply to networking and OS events. Each latency includes the notion of a distance that a signal can have traveled in that latency period:

## 3.1 1 second

- Time needed for light to reach the moon.

## 3.2 100 milliseconds

- A signal can reach all of the earths surface.
- Minimum human reaction speed.
- Timer interrupt interval for Linux systems configured with 100 HZ.
- Half of the TCP retry interval and SYNACK interval
- Typical Internet latency for high speed consumer grade links

## 3.3 10 milliseconds

- 2000km distance. Reach surrounding metropolitan areas.
- Timer interrupt interval for systems configured with 1000 HZ.
- Major fault (page needs to be read in from disk).
- Rescheduling to a different process on the same CPU.

## 3.4 1 millisecond

- 200km distance. Reaches systems in your city.
- Sound travels 34 centimeters. A signal from a speaker reaches your ear.
- Average seek time of a hard disk.
- Camera shutter speed.

## 3.5 100 microseconds

- 20km. Signal confined to local LAN or building.
- Maximum tolerable interrupt hold off.
- Best Ethernet ping pong times on 1G between neighboring systems.

## 3.6 10 microseconds

- 2km. Signal confined to local LAN.
- Minor page fault (Copy on write after fork).
- Duration of time interrupt.
- Duration of typical hardware interrupt.
- Typical IRQ hold off period if kernel disables interrupts.
- Duration of a system call.
- Context switch.
- Relativistic time distortion in GPS systems that needs to be compensated for.

### 3.7 1 microsecond

- 200m. Local LAN.
- Resolution of *gettimeofday()* system call.
- Duration of a vsyscall
- PTE miss and reloading of TLB
- Start of hardware interrupt processing

### 3.8 100 nanoseconds

- 20m. Within your room.
- Cache miss. Time needed to fetch data from memory.
- TLB miss.

Signalling latencies are currently a major restriction for building large supercomputers. The latency of memory subsystems can only be reduced if the subsystems are packed in a dense way. If the memory is over 20m away from the processor (even less in reality) then the time it takes the signal to travel across the wire will take up a major portion of the latency. It really does not matter how fast the memory is if its physically too far away.

Similarly one has to be careful of offers of "faster" DSL lines or network connections. "Faster" not mean that the speed of the data going across the wire is increased. It means that the number of bytes that can be transmitted at one time is increased. "Faster" DSL means a higher capacity link not that there will be any real increase in transmission speed. For gamers the distance to the game server is really important. If you live far away from the population centers then one is usually at a severe disadvantage due to signal run latencies. Others will shoot first and you will shoot and hit them but they wont get killed. The only solution for gamers to be able to shoot first is to figure out where the game serves are located and move nearer to them.

I keep getting questions about how to make things faster in terms of latency but the laws of physics are squarely in the way here. There is no solution in sight. Maybe someone can rework physics to show us how its done? Then maybe we can quantum tunnel signals, use warp drive bend time and space to fix this issue. If we can do that then we can likely also do all the other nice stuff that shows up in space fiction movies.

## 4 Characteristics of Noise

There are a couple of way to characterize noise. Noise can be seen as an interruption of the execution of the application. Noise in this form is a simple DoC (Denial of CPU) by the OS and can be measured by repeatedly taking time stamps. If the difference is higher than usual then some outside influence interrupted the process and stole processing time. It is typical to set some limitation of a boundary over which a delay is long enough to be considered a notable noise event. The important characteristics of noise that emerge from this method are the noise duration and their frequency.

Noise also has an influence on the execution speed of code through additional cache misses, TLB misses and internal processing within the processor. It is far more difficult to measure these effects. If the execution speed of a given segment of code is known then the deviation can be determined by also measuring the execution speed of a segment. However, that is only possible for code segments that can be executed repeatedly with the same data. The noise can then be quantified in the percentage of slowdown in the code segment due to noise interference.[8]

Noise interacts with the application in various ways. There are applications that are sensitive to certain types or noise and can tolerate others. Typically one assumes a linear correlation of the noise due to application performance. However the noise can resonate with processing intervals of the application which can then lead to a butterfly effect that amplifies the delays in the applications. The intervals of communication of the application are of importance. If the application does not frequently exchange information with other processes then the impact of fine grained noise (as usually presented by an OS) is minimal. However, if information exchanges occur frequently then the final grained noise can affect the critical communication paths and significant effects can develop.[9]

Some researchers have found that noise below the 1 microsecond boundary usually does not cause significant harm.[10] In the following surveys we will adopt 1 microsecond as a boundary for an OS event that is considered significant for our investigations.

[8] The measuring points will add additional latencies and cause more disturbance of processor resources in addition to the OS noise

[9] Petrini, 2003

[10] Tsafir 2008, under section 3.2 *Granularity*

### 4.1 Sources of noise

- The Linux scheduler is a prime cause for OS noise. Even if a process runs on an otherwise idle system: The scheduler will reschedule another process on any processor at least once a second (involuntary context switches). These context switches can be avoided by setting a real time priority (SCHED_RR or SCHED_FIFO). But real time priorities still do not stop the OS from processing maintenance code on the same processor. Especially the scheduler softirq will still be executed from the timer interrupt to keep statistics and check if other processes should be run on the processors. The scheduler is currently not designed to leave a running process alone.

- The Linux timer interrupt occurs with HZ frequency every second. Typically kernels are run with 1000 HZ meaning that a noise event occurs every millisecond. The timer interrupt in turn may run various regular maintenance tasks that increase the length of the events impacting the application.

  The kernel has an option to enable a tickless system (CONFIG_NO_HZ) but the tick is only switched off if a processor is idle. A busy processor will invariably get hit by the timer tick. The description of CONFIG_NO_HZ as enabling a "tickless system" is a bit misleading.

  There are other options for how to schedule OS maintenance events. Solaris only has a timer interrupt on processor 0. All other processors are left alone. The scheduler executing on processor 0 schedules the processes running on the other processors. On Solaris it is important to run processes on other processors in order to reduce OS noise.[11]

- Cache disturbances

  If multiple cpus (hardware execution context) share the same caches then another executing process on other cpus has access to processor resources necessary for execution. This is particularly significant if the processor supports hyper threading. All caches are shared then. L2 and L3 caches are also frequently shared between multiple processors.

- TLB miss

  TLB misses occur when the cache of virtual to physical mappings of the processors get exhausted. This is common if a threads memory accesses are sparse or are randomly covering large memory areas. Pointer chasing is a typical application that creates TLB misses. If the working set of a process becomes larger than the TLB coverage then it is possible that every memory access requires a new TLB fetch. If the TLB use of an application is high then OS processing may cause key TLB entries to be evicted.

  TLB resources are typically shared between multiple cpus meaning that the full TLB coverage is not available for single processor.

- Major page fault

  Major page faults involve bringing in a page from secondary storage (usually a hard disk). These are also a major causes of latency. Major faults are avoided by read ahead functionality of the file system. If the system detects linear reads from a disk device then multiple of pages are read in anticipating future faults. If read ahead has been performed for a page then only a minor fault will be generated.

  Major faults can accumulate if the OS starts to evict pages from memory that have been rarely used. If the pages that are missing from the process are relatively sparsely spread over large areas of secondary storage then the read ahead logic will be ineffective and each page fault may cause long latencies. From the application perspective these are not discernible from OS noise. The application accesses a memory location which results in an unexpected major delay.

- Minor page fault

  A minor fault is making a page visible to a process that has never accessed it before. If another process or read ahead has already brought a page into memory then a minor fault involves settings up the page tables so that the page becomes visible in the address space of a process.

  A minor fault can also occur when writing to the memory of a page. In that case we may have to copy the page (Copy-On-Write = COW) or update page dirty statistics.

- System threads

  The Linux kernel itself creates threads that are used for scheduling background write out, event handling and so on. File systems and other kernel

[11] Radojkovic 2007, 5

subsystems create their own background processes. These are usually fixed to a specific processor. The way to keep these quiet is not do perform actions that require background activities on a processor. The activities of these threads will shut down after some idle time of the subsystems.

It is fairly typical for these threads to only cause minor delays. However, the scheduler has to perform a context switch to the threads and back. The overhead increases significantly if large lists have to be processed (LRU expiration of inodes, slab object expiration).

- User space background daemons

  The user space background daemons are mostly created during boot up and have various administrative functions. It is possible to bind these processes to specific processors through the *taskset* tool. These background daemons can cause particularly long hold offs. Notorious examples are logging daemons that can issue fsync() system calls to force the log messages onto disk which may cause long delays due to synchronous write outs to disk.

## 5 Utilities to measure noise

I found no tools to measure noise under Linux so I created a series of test programs available at http://gentwo.org/ll. A small introduction to some of their features.

### 5.1 Low latency library

The low latency library (*ll*) contains basic function to obtain time stamps in a variety of magnitudes in an inexpensive way via the time stamp counts. Logic is included to determine the processors characteristics and the cache layouts from user space. These are basic necessities for measuring intervals in an accurate way with the least impact on a user space program and for tuning a user space application to the cache size or number of cores available.

### 5.2 latencytest

*latencytest* is a tool that continually retrieves the value of the time stamp counter and compares with the last time stamp obtained before. If a certain threshold (default 1 microsecond) is reached then the event is registered as a noise event. Latencytest produces a histogram and prints out statistics regarding the intervals observed. Latencytest monitors various scheduler statistics about itself and will note if the scheduler moves the process to another processor or performs a context switch away from the process.

Latencytest is a test load that can be run while other system activity is going on or with special scheduler parameters to see how the scheduler would change the treatment of a process.

The code used to determine how scheduling can affect the test load can be used as sample code to instrument a user space program. Typically these would be used to determine characteristics of key critical code segments.

### 5.3 latencystat

*latencystat* is able to display latency statistics of any process in the system via the */proc/<pid>/schedstat* information. It is comparable to *vmstat* and displays continually how long a process ran without another process having been scheduled and determines the average wait time from the point that a process became runnable until the scheduler gave the processor to the process.

### 5.4 trashcache

*trashcache* is a program that runs forever and does random memory accesses in order to trash the processor caches. Running *trashcache* on a sibling CPU can be used to gauge the impact of CPU cache thrashing on a user space process.

### 5.5 OS diagnostics

The operating system itself has counters for scheduler events that one can query f.e. via the *getrusage()* system call. Example code can be found in the source code for the *latencytest* tool.

### 5.6 udpping

*udpping* is another tool to measure OS noise by sending network packets back and forth between two systems. The UDP ping pong is the fastest time to communicate between two hosts using the IP stack. Noise shows up as variances of transfer times between both system.

## 6 Some OS noise measurements under Linux

The tools above can be used to measure the OS noise characteristics under Linux. Here we measured a completely idle system and see what the effect of the OS has on a simple test load (latencytest tool). It is important to note that this is the best scenario that can ever happen for the given kernel version. There will be numerous additional disturbances through cross-cache effects, system and user space daemons and so on if the system would be running a realistic load. What we measure here is a best case scenario. Everything else is guaranteed to be worse than what we measure.[12]

First a test running *latencytest* for various kernel versions. The tests are run for 10 seconds each and we record events longer than 1 microsecond. Most of the events recorded are timer interrupts of a duration longer than 1 microsecond. Timer interrupts may occur that are less of one microsecond in duration but these low latencies are only reached during favorable conditions when not much work is to be done from the timer interrupt and if the queue of functions to call is small. The test load does not have a large cache footprint (fits nicely into the L1 cache) meaning that most of the cache lines used for the timer interrupt will remain in memory. The processor in use here is a Penryn, dual quad core (Xeon X5460) at 3.16Ghz.

| Version | Test 1 | Test 2 | Test 3 | Sum |
|---|---|---|---|---|
| 2.6.22 | 383 | 540 | 667 | 1590 |
| 2.6.23 | 2738 | 2019 | 2303 | 7060 |
| 2.6.24 | 2503 | 573 | 583 | 3659 |
| 2.6.25 | 302 | 359 | 241 | 902 |
| 2.6.26 | 2503 | 2501 | 2503 | 7507 |
| 2.6.27 | 2502 | 2503 | 2478 | 7483 |
| 2.6.28 | 2502 | 2504 | 2502 | 7508 |
| 2.6.29 | 2502 | 2490 | 2503 | 7495 |
| 2.6.30 | 2504 | 2503 | 2502 | 7509 |

Table 1: Latency events >1 microseconds for a number of kernel versions

The number of noise events was initially quite low. With 2.6.23 (which introduced a new scheduler) we see a significantly higher number of noise events. Things improved with 2.6.24. In 2.6.25 we had a significant reduction of the OS noise to the smallest value seen. However, that was lost in 2.6.26.

[12]For a worse case run a latencytest during a kernel compile

The tests were run on a "tickless" system (CONFIG_NO_HZ is set) because the description for a "tickless" system given was that the timer interrupt only occurs as necessary. However, as seen here: The timer interrupt seems to occur regularly.

Each test run 10 seconds and in those 10 seconds 2500 time 1 HZ intervals occur since the kernel was configured with 250 HZ. Therefore what we see here are must be timer interrupts causing noise. The timer interrupt in 2.6.22 and 2.6.25 ran less than 1 microseconds otherwise the latencytest tool would have registered them.

The next test shows the average length of the noise events registered.

| Version | Test 1 | Test 2 | Test 3 | Average |
|---|---|---|---|---|
| 2.6.22 | 2.55 | 2.61 | 1.92 | 2.36 |
| 2.6.23 | 1.33 | 1.38 | 1.34 | 1.35 |
| 2.6.24 | 1.97 | 1.86 | 1.87 | 1.90 |
| 2.6.25 | 2.09 | 2.29 | 2.09 | 2.16 |
| 2.6.26 | 1.49 | 1.22 | 1.22 | 1.31 |
| 2.6.27 | 1.67 | 1.28 | 1.18 | 1.38 |
| 2.6.28 | 1.27 | 1.21 | 1.14 | 1.21 |
| 2.6.29 | 1.44 | 1.33 | 1.54 | 1.44 |
| 2.6.30 | 2.06 | 1.49 | 1.24 | 1.60 |

Table 2: Duration of Latency events >1 microseconds for a number of kernel versions

The tests shows that the time spend in the timer interrupts gradually increases. Interestingly 2.6.22 and 2.6.25 have much longer noise durations. The long durations may be a consequence of the OS batching multiple events in fewer timer events. The remaining timer events have less processing to do and therefore their processing time may under the 1 microsecond boundary. A significant portion of timer events in 2.6.22 and 2.6.25 took less than 1 microsecond.

We see the effect of kernel bloat in 2.6.28, 2.6.29 and 2.6.30. The average time spend in the timer interrupt gradually increases. This causes more and more regressions for latency sensitive applications. The question is what is worse: Batching events to have fewer noise above 1 microsecond of longer duration or having more events with a smaller duration.

The above results suggests a simple way to reduce the frequency of OS noise in the Linux kernel: Reduce the frequency of the timer interrupts. In the following measurements we let *latencytest* run for 60 seconds

and measure the number of noise events: Once with SCHED_OTHER which allows the scheduler to schedule other processes on the processor (although there is nothing else running on the system). And the second time with a real time priority SCHED_FIFO which does not allow the scheduler to take away the processor and give it to another process (but the kernel can still execute any of its threads and thereby create OS noise).

| HZ | Events | Duration | CSw | RT events | RT dur. |
|---|---|---|---|---|---|
| 18 | 1088 | 2.76 | 76 | 893 | 2.41 |
| 60 | 6042 | 2.62 | 95 | 6013 | 2.43 |
| 100 | 6012 | 2.47 | 103 | 6011 | 2.60 |
| 250 | 15024 | 2.52 | 94 | 15012 | 2.56 |
| 300 | 18022 | 2.16 | 65 | 18021 | 2.31 |
| 1000 | 60047 | 2.10 | 63 | 60043 | 2.20 |
| 4000 | 240139 | 2.00 | 61 | 240145 | 2.12 |

Table 3: Kernel Latency events depending on the timer interrupt frequency

The kernel supports timer interrupt frequencies (HZ) from 16 HZ to 4000 HZ. The arch specific configuration on x86 only allowed for 100-1000 HZ. A patch was used to extend the range of timer interrupt rates.

The effect of RT priorities is a bit disappointing. RT priorities do not significantly reduce the OS noise. RT scheduling prohibits context switches but these have a minor impact here. Pretty worrying is that in ranges higher than 250 HZ the overhead for RT scheduling increases and the timer interrupts become longer for RT scheduling despite the additional context switches that occur for SCHED_OTHER.

The duration of the timer events slightly decreases as the number of timer interrupts per second is increased. However, the change in duration does not seem to be that significant. This suggests that it may be best to reduce HZ as much as possible especially since high resolution timers are used in various places in the kernel now were accurate reaction to timeouts is important.

Since we saw that CONFIG_NO_HZ does not eliminate the HZ frequency interrupts while a process is executing it is interesting to see how a kernel would behave without CONFIG_NO_HZ. Surprisingly OS noise is significantly reduced by switching CONFIG_NO_HZ off. The number of involuntary context switches is reduced. Average durations are significantly reduced. The number of events over 1 microseconds drops by half for 1000

| HZ | Events | Duration | CSw |
|---|---|---|---|
| 18 | 1084 | 7.37 | 62 |
| 60 | 6016 | 2.02 | 63 |
| 100 | 6037 | 1.46 | 98 |
| 250 | 15016 | 1.87 | 61 |
| 300 | 18018 | 1.40 | 62 |
| 1000 | 34597 | 1.30 | 79 |
| 4000 | 151744 | 1.25 | 92 |

Table 4: Kernel Latency events for a system with ticks during 60 seconds

HZ and 4000 HZ. Switching off idle processor seems to be a very scheduler intensive activity. It is good for power consumption but it does not reduce the system noise as one would have expected.

## 7 Conclusion

The noise is there in the Linux kernel and it is gradually increasing as the kernel gets bloated with new features. I think it is necessary to keep an eye on the latencies created by the OS since we are seeing regressions when using newer kernels for latency sensitive applications.

In order to reduce the noise created in the Linux kernel we need to go beyond the real time scheduling policies (SCHED_RR and SCHED_FIFO). The following measures may be useful:

- Not running a timer interrupt if not necessary. Could we have a true tickless system? Currently Linux claims to be tickless but the truth is that a tick still is used when a process is running. A tick makes sense if multiple processes are contending for time on the same processor. If there is no other process at the same or higher priority contending then there is no need for a timer interrupt until the processor voluntarily gives up the time slice or until another process is created that can contend for the processor. We already have high res timers. Is it not possible to calculate how long a process is allowed to run and have the scheduler processing only occur when we reach that point?

  If multiple processors are contending for a processor and we assign a time slice to a processor then there still is no reason to run an timer interrupt before the end of that time slice. The OS needs to have a concept of an on demand timer interrupt that is only enabled on request.

- The scheduler needs to be more aware of the cache relationships between multiple "CPU"s that the OS knows about. The chance is good that threads of the same process will share data and therefore it is essential that the scheduler put threads of the same process on cpus that share CPU caches. If a process is running on one CPU and is marked as latency sensitive (using SCHED_RR and SCHED_FIFO) then scheduling on a sibling needs to be avoided as much as possible to leave the CPU cache undisturbed.

- It may be useful to make processor 0 a special processor that is used for system tasks. It could have a special role that the scheduler is aware of (comparable to Solaris). Processor 0 is already special because it is running a timer interrupt that is tasked with keeping system time. Therefore the noise created by processor 0 is already increased. Non latency sensitive tasks could be scheduled on processor 0 to keep needless noise away from the other cores. High priority tasks can then be scheduled on the other processors as needed whereas lower priority user space tasks (such as the regular daemons) could be mostly scheduled on processor 0.

- Processor 0 could take over tasks from other processors (like scheduling for idle processors). If a processor is busy and no CPU specific events are scheduled on a processor then processor 0 could take over managing the run queue and interrupting the target CPU as the need arises.

## 8 References

Beckman, P., Iskra, K., Yoshii, K., and Coghlan, S. 2006. "Operating system issues for petascale systems." *SIGOPS Oper. Syst. Rev.* 40, 2 (Apr. 2006), 29–33.

Beckman, P., Iskra, K., Yoshii, K., Coghlan, S., and Nataraj, A. 2008. "Benchmarking the effects of operating system interference on extreme-scale parallel machines." *Cluster Computing* 11, 1 (Mar. 2008), 3–16.

Ferreira, K.B., Bridges, P., and Brightwell, R. 2008. "Characterizing application sensitivity to OS interference using kernel-level noise injection." In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (Austin, Texas, November 15–21, 2008).

Van Hensbergen, E. 2006. "P.R.O.S.E.: partitioned reliable operating system environment." *SIGOPS Oper. Syst. Rev.* 40, 2 (Apr. 2006), 12–15.

E. V. Hensbergen. "The effect of virtualization on OS interference." In *Proceedings of the 1st Annual Workshop on Operating System Interference in High Performance Applications*, August 2005.

Petrini, F., Kerbyson, D.J., and Pakin, S. 2003. "The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q." In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing* (November 15–21, 2003). Conference on High Performance Networking and Computing. IEEE Computer Society, Washington, DC, 55.

Radojkovic, P., Cakarevic, V., Verdu Pajuelo, A., Gioiosa, R., Cazorla, F.J., Nemirovsky, M., and Valero, M. 2008. "Measuring Operating System Overhead on CMT Processors." In *Proceedings of the 2008 20th international Symposium on Computer Architecture and High Performance Computing* (October 29—November 01, 2008).

Tsafrir, D., Etsion, Y., Feitelson, D.G., and Kirkpatrick, S. 2005. "System noise, OS clock ticks, and fine-grained parallel applications." In *Proceedings of the 19th Annual international Conference on Supercomputing* (Cambridge, Massachusetts, June 20–22, 2005).

# Tuning 10Gb network cards on Linux

A basic introduction to concepts used to tune fast network cards

Breno Henrique Leitao
*IBM*
leitao@linux.vnet.ibm.com

## Abstract

The growth of Ethernet from 10 Mbit/s to 10 Gbit/s has surpassed the growth of microprocessor performance in mainstream servers and computers. A fundamental obstacle to improving network performance is that servers were designed for computing rather than input and output.

The processing of TCP/IP over Ethernet is traditionally accomplished by a software running on the central processor of the server. As network connections scale beyond Gigabit Ethernet speeds, the CPU becomes burdened with the large amount of TCP/IP protocol processing required by the current network speed. Reassembling out-of-order packets, handling the protocols headers, resource-intensive memory copies, and interrupts put a tremendous load on the system CPU, resulting in a CPU doing almost I/O traffic instead of running applications.

During the network evolution, each generation of new cards presents a lot of features that increase the performance of the network, and when not properly configured, can harm the overall system and network performance.

Linux, on the other side, is an operating system that runs from embedded system to super computers, and its default configuration is not tuned to run 10 Gbit/s network cards on wire speed, and it possibly will limit the total available throughput artificially. Hence, some modifications in the system is required to achieve good performance. Most of these performance modifications are easy to do, and doesn't require a deep knowledge of the kernel code in order to see the results.

This paper will describe most of the basic settings that should be set in a Linux environment in order to get the maximum speed when using fast network adapters. Since this is a vast topic, this paper will focus on the basic concepts.

## 1 Terminology

This section will cover basic terminology used in the article. For other terminologies, RFC2647[1] is a good place to start looking for.

### Throughput

The term throughput basically has the same meaning as data transfer rate or digital bandwidth consumption, and denotes the achieved average useful bit rate in a computer network over a physical communication link. The throughput is basically measured with little overhead, and for reference, it is measured below the network layer and above the physical layer. In this way, throughput includes some protocol overhead and retransmissions.

When talking about network terminology, it is worth to remember that one Kbit/s means 1,000 bit/s and not 1,024 bit/s.

### Round trip time

Round trip time (RTT) is the total amount of time that a packet takes to reach the target destination and get back to the source address.

### Bandwidth delay product

Bandwidth Delay Product (BDP) is an approximation for the amount of data that can be in flow in the network during a time slice. Its formula is simply a product of the link bandwidth and the Round Trif Time. To compute the BDP, it is required to know the speed of the slowest link in the path and the Round Trip Time (RTT) for the same path, where the bandwidth of a link is expressed in Gbit/s and the round-trip delay is typically between 1

msec and 100 msec, which can be measured using `ping` or `traceroute`.[1]

In TCP/IP, the BDP is very important to tune the buffers in the receive and sender side. Both side need to have an available buffer bigger than the BDP in order to allow the maximum available throughput, otherwise a packet overflow can happen because of out of free space.

## 1.1 Jumbo Frames

Jumbo frames are Ethernet frames that can carry more than the standard 1500 bytes of payload. It was defined that jumbo frames can carry up to 9000 bytes, but some devices can support up to 16128 bytes per frame. In this case, these super big packets are referenced as Super Jumbo Frames. The size of the frame is directly related to how much information is transmitted in a slot,[2] and is one of the major factors to increase the performance on the network, once the amount of overhead is smaller.

Almost all 10 Gbit/s switches support jumbo frames, and new switches supports super jumbo frames. Thus, checking if the network switches support these frames is a requirement before tuning this feature on.

Frame size is directly related to the interface Maximum Transfer Unit (MTU), and it is a specific adapter configuration that must be set for each node in a network. Moreover, all interfaces in the same network should have the same MTU in work to communicate properly, a different MTU on a specific node could cause awful issues.

Setting the MTU size uses the following command `ifconfig <interface> mtu <size>`.

It is important to note that setting the MTU to a high number, does not mean that all your traffic would use jumbo packets. For example, a normal ssh session is almost insensible to a MTU change, once almost all SSH packets are sent using small frames.

Once you the jumbo frames are enabled on an interface, the software application should start using it, otherwise the performance will not be as good as expected.

[1] Although there are better ways to measure the packet RTT, these tools are enough for the purpose of this paper.

[2] As referenced on 802.3.

It is also observed that TCP responsiveness is improved by larger MTUs. Jumbo frames accelerate the congestion window increase by a factor of six compared to the standard MTU. Jumbo frames not only reduce I/O overhead on end-hosts, they also improve the responsiveness of TCP.

## 1.2 Multi streams

The network throughput is heavily dependent on the type of traffic that is flowing in the wire. An important point is the number of streams, also viewed as a socket, opened during a time slice. It involves creating and opening more than one socket and parallelizing the data transfer among these sockets. The link usage is better depending on how many streams are flowing. Even if the system is well tuned, it is not easy to achieve the wire speed using just a single stream.

The rule of thumb says that multi stream applications are preferred instead of an application that has just a connection and try to send all the data through it. Actually a lot of software, as application servers, allows setting the number of opened sockets, mainly when the application is trying to connect to a database.

As practical case, the Netperf tool provides a more accurate result when using multi stream mode with 8 actives streams.

## 1.3 Transmission queue size

The transmission queue is the buffer that holds packets that is scheduled to be sent to the card. Tuning the size of this buffer is necessary in order to avoid that packet descriptors are lost because of no available space in the memory.

Depending on the type of the network, the default 1000 packets value could not be enough and should be raised. Actually, a good number is around 3000 depending of the network characteristics.

## 1.4 SMP IRQ affinity

The main way that a CPU and a I/O device communicates is through interrupts. Interrupt means that when a device wants attention, it raises an interruption, and the CPU handles it, going to the device and checking

```
              CPU0   CPU1      CPU2     CPU3
  16:          832   1848      1483     2288  XICS  Level  IPI
  17:            0      0         0        0  XICS  Level  hvc_console
  18:            0      0         0        0  XICS  Level  RAS_EPOW
  53:           53      0   1006570        0  XICS  Level  eth0-TxRx-0
  54:           28      0         0   159907  XICS  Level  eth0-TxRx-1
  55:      2105871      0         0        0  XICS  Level  eth0-TxRx-2
  56:      1421839      0         0        0  XICS  Level  eth0-TxRx-3
  57:           13    888         0        0  XICS  Level  eth0-tx-0
  58:           13      0       888        0  XICS  Level  eth0-tx-1
  59:           13      0         0      888  XICS  Level  eth0-tx-2
  60:            4      0         0        0  XICS  Level  eth0:lsc
 256:            1      0         0        0  XICS  Level  ehea_neq
 261:            0      0         0        0  XICS  Level  eth1-aff
 262:          119   5290         0        0  XICS  Level  eth1-queue0
 273:         7506      0         0     1341  XICS  Level  ipr
```

Figure 1: Which CPU is handling which device IRQ

what are the devices needs. On an SMP system, the specific CPU handling your interruption is very important for performance.

Network devices usually has from one to a few interrupt line to communicate with the CPU. On multiple CPU system, each CPU call handles an interruption request. Usually the round robin algorithm is used to choose the CPU that will handle a specific interruption for a specific card. In order to check which CPU handled the device interruption, the pseudo-file `/proc/interrupts` will display all the interrupts lines, and which CPU handled the interruptions generated for each interrupt line. Figure 1 is an example of the content of `/proc/interrupts` pseudo-file.

In order to achieve the best performance, it is recommended that all the interruptions generated by a device queue is handled by the same CPU, instead of IRQ balancing. Although it is not expected, round robin IRQ distribution is not good for performance because when the interruption go to another fresh CPU, the new CPU probably will not have the interrupt handler function in the cache, and a long time will be required to get the properly interrupt handler from the main memory and run it. On the other hand, if the same CPU handles the same IRQ almost all the time, the IRQ handler function will unlikely leave the CPU cache, boosting the kernel performance.

In this manner, no IRQ balancing should be done for networking device interrupts, once it destroys performance. It is also important that TX affinity matches RX affinity, so the TX completion runs on the same CPU as RX. Don't mapping RX completion to RX completion causes some performance penalty dues cache misses.

Almost all current distros comes with a daemon that balance the IRQs, which is very undesirable, and should be disable. In order to disable the `irqbalance` tool, the following command should stop it.

```
# service irqbalance stop
```

If disabling this daemon is a very drastic change, then it is possible to disable `irqbalance` only for those CPUs that has an IRQ bound, and leave the IRQ balance enabled on all other CPUs. This can be done by editing the file `/etc/sysconfig/irqbalance`, and changing the `IRQBALANCE_BANNED_CPUS` option. So, all the CPUs that are bound to an interface queue, must be set as banned in the `irqbalance` configuration file.

Once the `irqbalance` daemon is disabled or configured, the next step is to configure which CPU will handle each device interruption.

In order to bind an interrupt line to a CPU, kernel 2.4 or superior provides scheme on the `/proc` interface which was created to allow the user to choose which

group of processors will handle a specific interrupt line. This configuration lives at `/proc/<IRQ number> /smp_affinity`, and can be changed any time on-the-fly. The content of the `smp_affinity` is a hexadecimal value that represents a group of CPU, as for example, `ff` representing all eight CPUs available. So, each field in the bit mask corresponds to a processor, and to manipulate it properly, a binary to hexadecimal transformation will be required.

Each digit in `ff` represents a group of four CPUs, with the rightmost group being the least significant. The letter `f` is the hexadecimal representation for the decimal number 15 and represent `1111` in binary, and each of the places in the binary representation corresponds to a CPU.

| CPU | Binary | Hex |
|---|---|---|
| 0 | 0001 | 0x1 |
| 1 | 0010 | 0x2 |
| 2 | 0100 | 0x4 |
| 3 | 1000 | 0x8 |

By combining these bit patterns (basically, just adding the Hex values), it is possible to a group of processors. For example, a representation of a group of two CPUs, for instance CPU0 (0x1) and CPU2 (0x4), is the sum of both individually, which means 0x1 + 0x4 = 0x5.

### 1.5 Taskset affinity

On a multi stream setup, it is possible to see some tasks that are transmitting and receiving the packets, and depending on the schedule policy, the task can migrate from one CPU to other from time to time. Since task migration is a huge penalty for performance, it is advised that a task is bound to one or few CPUs, as described above for IRQ. Since disabling the receive and sent task from being floating in all the CPUs, the cache misses rate is decreased, and the performance improved.

In order to bind a task to a CPU, the command `taskset` should be used as follows.

```
$ taskset -p 0x1 4767
pid 4767's current affinity mask: 1
pid 4767's new affinity mask: 1
```

### 1.6 Interrupt coalescence

Most modern NICs provide interruption moderation or interruption coalescing mechanisms to reduce the number of IRQs generated when receiving frames. As Ethernet frames arrive, the NIC saves them in memory, but the NIC will wait a receive delay before generating an interruption to indicate that one or more frames have been received. This moderation reduces the number of context switches made by the kernel to service the interruptions, but adds extra latency to frame reception.

Using interruption coalescence doesn't allow the system to suffer from IRQ storms generated during high traffic load, improving CPU efficiency if properly tuned for specific network traffic.

Enabling interruption coalescence could be done using the tool `ethtool` together with parameter `-C`. There are some modules that do not honor the `ethtool` method of changing the coalescence setting and implements its own method. In this case, where this option is specific to the device driver, using `modinfo` to discover the module parameter that enables it is the best option.

#### NAPI

"New API" (NAPI) is a feature that solves the same issue that Interrupt coalescence solved without hurting latency too much. NAPI was created in the linux kernel aiming to improve the performance of high-speed networking, and avoid interrupt storms. NAPI basically creates a mixture of interrupts and polling, called "adaptive interrupt coalescing." It means that in high traffic the device interruptions are disabled and packets are collected by polling, decreasing the system load. When the traffic is not high, then the interrupt scheme takes place, avoiding long latencies because of the pooling scheme. Also, NAPI-compliant drivers are able to drop packets in NIC (even before it reaches the kernel) when necessary, improving the system overall performance.

NAPI was first incorporated in the 2.6 kernel and was also backported to the 2.4.20 kernel. In general, running an NAPI enabled driver is a plus to get good performance numbers.

## 2 Offload features

Originally TCP was designed for unreliable low speed networks, but the networks changed a lot, link speed be-

comes more aggressive and quality of service is now a strict requirement for a lot of applications. Although these huge changes is still happening, the TCP protocol is almost the same as it was designed. As the link speed grows, more CPU cycle is required to be able to handle all the traffic. Even the more current CPUs waste a considerable amount of cycle in order to process all the CPU communication.

A generally accepted rule of thumb[3] is that one hertz of CPU is required to send or receive one bit of TCP/IP[2]. For example a five gigabit per second of traffic in a network requires around five GHz of CPU for handling this traffic. This implies that two entire cores of a 2.5 GHz multi-core processor will be required to handle the TCP/IP processing associated with five gigabit per second of TCP/IP traffic. Since Ethernet is bidirectional, it means that it is possible to send and receive 10 Gbit/s. Following the 1 Hz per bit rule, it would need around eight 2.5 GHz cores to drive a 10 Gbit/s Ethernet network link.

Looking into this scenario, the network manufacturers started to offload a lot of repetitive tasks to the network card itself. It leaves the CPU not so busy executing the usual networking tasks, as computing the checksum and copying memory around.

The manufactures classify offload features in two types, those that are stateful and those that are stateless, where a state references the TCP state. This section will cover only the stateless features, and the stateful feature will be briefly described in Section 5.8.

Almost all offload features are configured using the `ethtool` tool. In order to check which features are set, run `ethtool` using the `-k` parameter, as follows:

```
# ethtool -k eth2

Offload parameters for eth2:
rx-checksumming: off
tx-checksumming: off
scatter-gather: off
tcp segmentation offload: off
udp fragmentation offload: off
generic segmentation offload: off
```

[3]This general rule of thumb was first stated by PC Magazine around 1995, and is still used as an approximation nowadays.

In order to enable or disable any feature, the parameter `-K` should be used with `on` to enable the feature, and `off` to disable it. The following example enables TX checksum feature for interface `eth2`.

```
# ethtool -K eth2 tx on
```

### 2.1 RX Checksum

The TCP RX checksum offload option enables the network adapter to compute the TCP checksum when a packet is received, and only send it to the kernel if the checksum is correct. This feature saves the host CPU from having to compute the checksum, once the card guaranteed that the checksum is correct.

Enabling RX checksum offload can be done using `ethtool -K ethX rx on`[4]

Note that if receive checksum offload is disabled, then it is not possible to enable large receive offload (LRO).

### 2.2 TX Checksum

TX Checksum works almost as RX checksum, but it asks the card to compute the segment checksum before sending it. When enabling this option, the kernel just fill a random value in the checksum field of the TCP header and trust that the network adapter will fill it correctly, before putting the packet into the wire.

In order to enable TX checksum offload, the command `ethtool -K ethX tx on` should be run.

When the TX and RX offload are enabled, the amount of CPU saved depends on the packet size. Small packets have little or no savings with this option, while large packets have larger savings. On the PCI-X gigabit adapters, it is possible to save around five percent in the CPU utilization when using a 1500 MTU. On the other hand, when using a 9000 MTU the savings is approximately around 15%.

It is important to note that disabling transmission checksum offload also disables scatter and gather offload since they are dependent.

[4]Replace ethX to the target interface

### 2.3 Scatter and Gather

Scatter and Gather, also known as Vectored I/O, is a concept that was primarily used in hard disks[3]. It basically enhance large I/O request performance, if supported by the hardware. Scatter reading is the ability to deliver data blocks stored at consecutive hardware address to non-consecutive memory addresses. Gather writing is the ability to deliver blocks of data stored at non-consecutive memory addresses to consecutively addressed hardware blocks.

One of the constraints that happens to DMA is that the physical memory buffer should be contiguous in order to receive the data. On the other hand, a device that supports scatter and gather capability allows the kernel to allocate smaller buffers at various memory locations for DMA. Allocating smaller buffers is much easier and faster than finding for a huge buffer to place the packet.

When scatter and Gather is enable, it is also possible to do a concept called *page flip*. This basically allows the transport and other headers to be separated from the payload. Splitting header from payload is useful for copy avoidance because a virtual memory system may map the payload to an possible application buffer, only manipulating the virtual memory page to point to the payload, instead of copying the payload from one place to another.

The advantage of Scatter and Gather is to reduce overhead allocating memory and copying data, also as having a better memory footprint.

In order to enable Scatter and gather, the command `ethtool -K ethX sg on` should be run.

### 2.4 TCP Segmentation Offload

TCP segmentation offload (TSO), also called Large Segment Offload (LSO), is feature used to reduce the CPU overhead when running TCP/IP. TSO is a network card feature designed to break down large groups of data sent over a network into smaller segments that pass through all the network elements between the source and destination. This type of offload relies on the network interface controller to segment the data and then add the TCP, IP and data link layer protocol headers to each segment.

The performance improvement comes from the fact that the upper layers deliver a huge packet, as 64K, to the card and the card splits the this packet in small frames which honor the MTU size.

Some studies suggests that enabling TSO saves around 10% in the CPU when using a 1500 MTU.

In order too enable TCP segmentation offload, the command `ethtool -K ethX tso on` should be run.

### 2.5 Large Receive Offload

Large Receive Offload (LRO) is a technique that increases inbound throughput of high-bandwidth network connections by reducing CPU overhead. It works by aggregating, in the card, multiple incoming packets from a single stream into a larger buffer before they are passed to the network stack, thus reducing the number of packets that have to be processed, and all headers overhead. This concept is basically the opposite of TSO. LRO combines multiple Ethernet frames into a single receive, thereby decreasing CPU utilization when the system is receiving multiple small packets.

There are two types of LRO, one that are just a change in the network structure that is usually enabled by default in new drivers. This one aggregates frames in the device driver rather than in the card itself. The other one is a hardware feature that aggregate the frames into the card itself and provides much better results. The last one is specific for each device driver and it is usually enabled using a module parameter, as `lro_enable` for s2io driver.

### 2.6 Generic Segmentation Offload

After TSO was implemented, It was observed that a lot of the savings in TSO come from traversing the kernel networking stack once instead than many times for each packet. Since this concept was not dependent of the hardware support, the Generic Segmentation Offload (GSO) was implemented to postpone the segmentation as late as possible. Once this is a general concept, it can be applied to other protocols such as version 6 of TCP, UDP, or even DCCP.

GSO like TSO is only effective if the MTU is significantly less than the maximum value of 64K.

In order to enable generic segmentation offload, the command `ethtool -K ethX gso on` should be run.

# 3 Kernel settings

The Linux kernel and the distributions that package it typically provides very conservative defaults to certain network kernel settings that affect networking parameters. These settings can be set via the `/proc` filesystem or using the `sysctl` command. Using `sysctl` is usually better, as it reads the contents of `/etc/sysctl.conf` or any other selected chosen config script, which allows to keep the setting even after the machine restart. Hereafter only the modifications using the sysctl will be covered.

In order to change a simple configuration, a command as `sysctl -w net.core.rmem_max=16777216` should be run. In this case, the parameter `-w` want to set the value `16777216` into the variable `net.core.rmem_max`. Note that this is a temporary setting, and it will be lost after the system is restart. However most of the configuration should hold after a system restart, and in this case a modification in the file `/etc/sysctl.conf` will be necessary.

Also, it is possible to create a file with a lot of configuration, which can be called using the command that follows.

```
# sysctl -p /script/path
```

It's also possible to see all the parameters related to network using the following parameter.

```
# sysctl -a | grep net
```

On other hand, checking just a option can be done as follows.

```
# sysctl net.ipv4.tcp_window_scaling
```

## 3.1 TCP Windows Scaling

The TCP/IP protocol has a header field called *window*. This field specifies how much data the system which sent the packet is willing and able to receive from the other end. In other words, it is the buffer space required at sender and receiver to save the unacknowledged data that TCP must handle in order to keep the pipeline full.

In this way, the throughput of a communication is limited by two windows: congestion window and receive window. The first one tries not to exceed the capacity of the network (congestion control) and the second one tries not to exceed the capacity of the receiver to process data (flow control). The receiver may be overwhelmed by data if for example it is very busy (such as a Web server). As an example, each TCP segment contains the current value of the receive window. So, if the sender receives an *ACK* which acknowledge byte 4000 and specifies a receive window of 10000 (bytes), the sender will not send packets after byte 14000, even if the congestion window allows it.

Since TCP/IP was designed in the very far past, the window field is only 16 bits wide, therefore the maximum window size that can be used is 64KB. As it's known, 64KB is not even close to what is required by 10Gbit/s networks. The solution found by the protocol engineer was windows scaling describer in RFC 1323, where it creates a new TCP option field which left-shift the current window value to be able to represent a much larger value. This option defines an implicit scale factor, which is used to multiply the window size value found in a regular TCP header to obtain the true window size.

Most kernels enables this option by default, as could be seen running the command `sysctl net.ipv4.tcp_window_scaling`.

## 3.2 TCP Timestamp

TCP timestamp is a feature also described by RFC 1323 that allow a more precise round trip time measurement. Although timestamp is a nice feature, it includes an eight bytes to the TCP header, and this overhead affects the throughput and CPU usage. In order to reach the wire speed, it is advised to disable the timestamp feature, setting running `sysctl -w net.ipv4.tcp_timestamps=0`

## 3.3 TCP fin timeout setting

When TCP is finishing a connection, it gets into a state called **FIN-WAIT-2**, which still wasting some memory resources. If the client side doesn't finish the connection, the system needs to wait a timeout to close it by itself. On Linux, it is possible to determine the time that must elapse before TCP/IP can release the resources for a closed connection and reuse them. By reducing the value of this entry, TCP/IP can release closed connections faster, making more resources available for new connections. So, when running on a server that has a big

amount of closed socket, this adjust saves some system resources. In order to adjust this value, the parameter `net.ipv4.tcp_fin_timeout` should be changed to a smaller number than the default. Around 15 and 30 seconds seem to be a good value for moderate servers.

### 3.4 TCP SACK and Nagle

TCP Sack is a TCP/IP feature that means TCP Selective Acknowledgement and was described by RFC 2018, It was aimed for networks that have big windows and usually lose packets. It consists in sending explicit **ACK** for the segments of the stream has arrived correctly, in a non-contiguous manner, instead of the traditional TCP/IP algorithm that implements cumulative acknowledgement, which just acknowledges contiguous segments.

As most of the 10 Gbit/s network are reliable and usually losing packets is not the common case, disabling SACK will improve the network throughput. It is important to note that if the network is unreliable, and usually lose packets, then this option should be turned on.

In order to disable it, set `0` into the `net.ipv4.tcp_sack` parameter.

On the other side, Nagle algorithm should be turned on. Nagle is a TCP/IP feature described by **RFC 896** and works by grouping small outgoing messages into a bigger segment, increasing bandwidth and also latency. Usually Nagle algorithm is enabled on standard sockets, and to disable it, the socket should set `TCP_NODELAY` in its option.

### 3.5 Memory options

Linux supports global setting to limit the amount of system memory that can be used by any one TCP connection. It also supports separate per connection send and receive buffer limits that can be tuned by the system administrator.

After kernel 2.6.17, buffers are calculated automatically and usually works very well for the general case. Therefore, unless very high RTT, loss or performance requirement is present, buffer settings may not need to be tuned. If kernel memory auto tuning is not present (Linux 2.4 before 2.4.27 or Linux 2.6 before 2.6.7), then replacing the kernel is highly recommended.

In order to ensure that the auto tuning is present and it is properly set, the parameter `net.ipv4.tcp_moderate_rcvbuf` should be set to 1.

Once the auto tuning feature is on, the limits of memory used by the auto tuner should be adjusted, since most of the network memory options have the value that need to be set. The value is an array of 3 values that represents the minimum, the initial and maximum buffer size as exemplified above. These values are used to set the bounds on auto tuning and balance memory usage. Note that these are controls on the actual memory usage (not just TCP window size) and include memory used by the socket data structures as well as memory wasted by short packets in large buffers.

```
net.ipv4.tcp_rmem = 4096 87380 3526656
```

There are basically three settings that define how TCP memory is managed that need a special care and will be described hereafter. They are `net.ipv4.tcp_rmem`, which define the size (in bytes) of receive buffer used by TCP sockets, `net.ipv4.tcp_wmem` which define the amount of memory (in bytes) reserved for send buffers, and the `net.ipv4.tcp_mem`, which define the total TCP buffer-space allocatable in units of page (usually 4k on x86 and 64k on PPC). So, tuning these settings depends on the type of the Ethernet traffic and the amount of memory that the server has.

All of those value should be changed so that the maximum size is bigger than the BDP, otherwise packets can be dropped because of buffer overflow. Also, setting `net.ipv4.tcp_mem` to be twice the delay bandwidth delay product of the path is a good idea.

Secondarily, there are a lot of configurations that need to be tuned depending on the how network is working, and if the performance requirement was met. The important ones are `net.core.rmem_max`, `net.core.wmem_max`, `net.core.rmem_default`, `net.core.wmem_default`, and `net.core.optmem_max`. The kernel file `Documentation/network/ip-sysctl.txt` has a description for almost all of these parameters and should be consulted whenever necessary.

Another useful setting that deserves a check is `net.core.netdev_max_backlog`. This parameter defines the maximum number of packets queued on the

input side, when the interface receives packets faster than kernel can process them, which is usual on 10 Gbit/s network interface. This option is opposite for the `txqueuelen`, that define the length of the queue in the transmission side. As the value for the maximum backlog depends on the speed of the system, it should be fine tuned. Usually a value near 300000 packets is a good start point.

## 4 Bus

One important aspect of tuning a system is assuring that the system is able to support the desired needs, as throughput and latency. In order to ensure that a system is able to run on the desired speed, a brief overview of common the bus subsystem will be described. Since 10 Gbit/s network cards are only available on PCI extended (PCI-X) and PCI Express (PCI-E), this article will focus on these buses.

PCI is the most common bus for network cards, and it is very known for its inefficiency when transferring small bursts of data across the PCI bus to the network interface card, although its efficiency improves as the data burst size increases. When using the standard TCP protocol, it is usual to transfer a large number of small packets as acknowledgement and as these are typically built on the host CPU and transmitted across the PCI bus and out the network physical interface, this impacts the host overall throughput.

In order to compare what is required to run a 10 Gbit/s card on full speed, some bus performance will be displayed. A typical PCI bus running at 66 MHz, has an effective bandwidth of around 350 MByte/s.[5] The more recent PCI-X bus runs at 133 MHz and has an effective bus bandwidth of around 800 MByte/s. In order to fill a 10 Gigabit Ethernet link running in full-duplex, the order of 1.25GByte/s of bandwidth is required in each direction, or a total of 2.5 GByte/s, more than three times the PCI-X implementations, and nearly seven times the bandwidth of a legacy PCI bus. So, in order to get the best performance of the card, a PCI-E network card is highly recommended.

Also, usually most of the network traffic is written or read from the disk. Since the disk controller usually shares the same bus with network cards, a special consideration should be taken in order to avoid bus stress.

[5] PCI Encoding overhead is two bits in 10.

### 4.1 PCI Express bandwidth

PCI Express version 1.0 raw signaling is 250 MByte/s lane, while PCI Express version 2.0 doubles the bus standard's bandwidth from 250 MByte/s to 500MByte/s, meaning a x32 connector can transfer data at up to 16 GByte/s

PCI Express version 3.0 that is already in development and is scheduled to be released in 2010, will be able to deliver 1 GByte/s per lane.

### 4.2 Message-Signalled Interrupt

One important PCI-E advantage is that interrupts are transferred in-line instead of out-of-band. This feature is called Message-signalled Interrupt (MSI). MSI enables a better interrupt handling since it allows multiple queueable interrupts.

Using MSI can lower interrupt latency, by giving every kind of interruption its own handler. When the kernel receives a message, it will directly call the interrupt handler for that service routine associated with the address. For example, there are many types of interrupts, one for link status change, one for packet transmitted status, one for packet received, etc. On the legacy interrupt system, the kernel is required to read a card register in order to discover why the card is interrupting the CPU, and to call the proper interrupt handler. This long path causes a delay that doesn't happen when the system is using MSI interrupts.

MSI-X is an extension to MSI to enable support for more vectors and other advantages. MSI can support at most 32 vectors while MSI-X can up to 2048. Also, when using MSI, each interrupt must go to the same address and the data written to that address are consecutive. On MSI-X, it allows each interrupt to be separately targeted to individual processors, causing a great benefit, as a high cache hit rate, and improving the quality of service.

In order to ensure that MSI or MSI-X are enabled on a system, the `lspci` command should be used to display the PCI device capabilities. The capabilities that describe these features and should be enabled are `Message Signalled Interrupts` or `MSI-X` as described by Figure 2.

```
# lspci | grep "10 Giga"
0002:01:00.0 Ethernet controller: Intel Corporation 82598EB 10 Gigabit AF
                                                Network Connection (rev 01)

# lspci -vv -s 0002:01:00.0  | grep Message
Capabilities: [50] Message Signalled Interrupts: 64bit+ Queue=0/0 Enable+
```

Figure 2: lspci output

### 4.3 Memory bus

Memory bus might not be as fast as required to run a full duplex 10 Gbit/s network traffic. One important point is ensure that the system memory bandwidth is able to support the traffic requirement. In order to prove that the memory bandwidth is enough for the desired throughput, a practical test using Netperf can be run. This test displays the maximum throughput that the system memory can support.

Running a Netperf test against the *localhost* is enough to discover the maximum performance the machine can support. The test is simple, and consist of starting the `netserver` application on the machine and running the test, binding the client and the server in the same processor, as shown by Figure 3. If the shown bandwidth is more than the required throughput, then the memory bus is enough to allow a high load traffic. This example uses the `-c` and `-C` options to enable CPU utilization reporting and shows the asymmetry in CPU loading.

```
# netperf -T0,0 -C -c
             Utilization
             Send     Recv
...  Throughput  local    remote  ...
     10^6bits/s  % S      % S

      13813.70   88.85    88.85
```

Figure 3: Test for memory bandwidth

### 4.4 TCP Congestion protocols

It is a very important function of TCP to properly match the transmission rate of the sender and receiver over the network condition[4]. It is important for the transmission to run at high enough rate in order to achieve good performance and also to protect against congestion and packet losses. Congestion occurs when the traffic offered to a communication network exceeds its available transmission capacity.

This is implemented in TCP using a concept called windows. The window size advertised by the receiver tells the sender how much data, starting from the current position in the TCP data byte stream can be sent without waiting for further acknowledgements. As data is sent by the sender and then acknowledged by the receiver, the window slides forward to cover more data in the byte stream. This is usually called sliding window.

Some usual congestion avoidance algorithm has a property to handle with the window size called "additive increase and multiplicative decrease" (AIMD). This property is proven to not be adequate for exploring large bandwidth delay product network, once general TCP performance is dependent of the Congestion control algorithm[5]. And the congestion control algorithm is basically dependent of network bandwidth, round trip time and packet loss rate. Therefore, depending on the characteristics of your network, an algorithm fits better than another.

On Linux it is possible to change the congestion avoidance algorithm on the fly, without even restarting your connections. To do so, a file called `/proc/sys/net/ipv4/tcp_available_congestion_control` lists all the algorithms available to the system, as follows:

```
# cat /proc/sys/net/ipv4/tcp_available_
congestion_control
cubic reno
```

Also, it is possible to see what algorithm the system is currently using, looking into file `/proc/sys/net/ipv4/tcp_congestion_control`. To set a new algorithm, just echo one of those available algorithm name into this file, as follows:

```
# echo cubic > /proc/sys/net/ipv4/tcp
_congestion_control
```

This article will outline some of the most common TCP/IP collision avoidance algorithms.

### 4.5 RENO

TCP Reno was created to improve and old algorithm called Tahoe[6]. Its major change was the way in which it reacts when detect a loss through duplicate acknowledgements. The idea is that the only way for a loss to be detected via a timeout and not via the receipt of a dupack is when the flow of packets and `acks` has completely stopped. Reno is reactive, rather than proactive, in this respect. Reno needs to create losses to find the available bandwidth of the connection.

Reno is probably the most common algorithm and was used as default by Linux until kernel 2.6.19.

When using Reno on fast networks, it can eventually underutilize the bandwidth and cause some variation in the bandwidth usage from time to time[7]. For example, in slow start, window increases exponentially, but may not be enough for this scenario. On a scenario when using 10Gbit/s network with a 200ms RTT, 1460B payload and assuming no loss, and initial time to fill pipe is around 18 round trips, which result in a delay of 3.6 seconds. If one packet is lost during this phase, this time can be much higher.

Also, to sustain high data rates, low loss probabilities are required. If the connection loose one packet then the algorithm gets into AIMD, which can cause severe cut to the window size, degrading the general network performance.

On the other hand, Reno in general is more TCP-friendly than FAST and CUBIC.

### 4.6 CUBIC

CUBIC is an evolution of BIC. BIC is algorithm that has a pretty good scalability, fairness, and stability during the current high speed environments, but the BIC's growth function can still be too aggressive for TCP, especially under short RTT or low speed networks, so CUBIC was created in order to simply its window control and improve its TCP-friendliness. Also, CUBIC implements an optimized congestion control algorithm for high speed networks with high latency

In general, CUBIC is on the best algorithm to get the best throughput and TCP-fairness. That is why it is implemented and used by default in Linux kernels since version 2.6.19.

### 4.7 FAST

FAST (FAST AQM Scalable TCP) was an algorithm created with high-speed and long-distance links in mind.

A FAST TCP flow seeks to maintain a constant number of packets in queues throughout the network. FAST is a delay-based algorithm that try to maintain a constant number of packets in queue, and a constant window size, avoiding the oscillations inherent in loss-based algorithms, as Reno. Also, it also detects congestion earlier than loss-based algorithms. Based on it, if it shares a network with loss-based "protocol," the FAST algorithms tend to be less aggressive. In this way, FAST has the better Friendliness than CUBIC and Reno.

## 5 Benchmark Tools

There are vast number of tools that can be used to benchmark the network performance. In this section, only `Netperf`, `Mpstat` and `Pktgen` will be covered. These are tools that generate specifics kind of traffic and then shows how fast the network transmitted it.

### 5.1 Netperf

Netperf[6] is a tool to measure various aspects of networking performance. It is primarily focus is on data transfer using either TCP or UDP. Netperf has also a vast number of tests like stream of data and file transfer among others.

[6] `http://www.netperf.org`

```
# netperf -t TCP_STREAM -H 192.168.200.2 -l 10
TCP STREAM TEST from 0.0.0.0 port 0 AF_INET to 192.168.200.2 port 0 AF_INET
Recv   Send    Send
Socket Socket  Message  Elapsed
Size   Size    Size     Time     Throughput
bytes  bytes   bytes    secs.    10^6bits/sec

 87380  16384  16384    10.00    7859.85
```

Figure 4: Netperf running for 10 seconds

```
# netperf -H 192.168.200.2  -t TCP_RR -l 10
TCP REQUEST/RESPONSE TEST from 0.0.0.0 port 0 AF_INET to
                                                192.168.200.2 port 0 AF_INET
Local /Remote
Socket Size   Request  Resp.   Elapsed  Trans.
Send   Recv   Size     Size    Time     Rate
bytes  Bytes  bytes    bytes   secs.    per sec

16384  87380  1        1       10.00    42393.23
```

Figure 5: Netperf running with TCP_RR

It is easy to benchmark with Netperf. An installation of the tool is required in the client and server. On the server side, a daemon called `netserver` should be started before the tests begin, so that the Netperf client could connect to it. Netperf usually is listening on port 12865.

In order to ensure that the Netperf setup is working properly, a simple test as `netperf -H <hostname>` is enough to prove that the communication is good between the client and the server.

Netperf tests are easier when considering to use the set of scripts files provided with the Netperf distribution. These scripts are usually located at `/usr/share/doc/netperf/examples`. These scripts has a set of features and configuration that is used on those test cases. Since these scripts are heavily configurable, it is not required to read the entire Netperf manual in order to run more complex tests.

Netperf supports a lot of tests cases, but in this paper only TCP/UDP streams and transaction will be covered.

**Streams**

On streams test, Netperf supports a vast number of testcases, but only three are widely used, they are `TCP_STREAM`, `TCP_MAERTS`[7] and `UDP_STREAM`. The difference between `TCP_STREAM` and `TCP_MAERTS`, is that on first, the traffic flows from the client to the server, and on `TCP_MAERTS`, the traffic flows from the server to the client. Thus running both `TCP_STREAM` and `TCP_MAERTS` in parallel generate a full-duplex test.

Figure 4 shows a simple example of `TCP_STREAM` test case running against IP `192.168.200.2` for ten seconds.

**Transactions**

Transaction is another area to investigate and tune in order to improve the quality of the network, and to do so, Netperf provides a testcase to measure request/response benchmark. A transaction is defined as a single reply for a single request. In this case, request/response performance is quoted as “transactions/s” for a given request and response size.

[7] MAERTS is an anagram of STREAM

```
#!/bin/bash

NUMBER=8
TMPFILE=`mktemp`
PORT=12895

for i in `seq $NUMBER`
do
    netperf -H $PEER -p $PORT -t TCP_MAERTS -P 0  -c -l $DURATION
            -- -m 32K -M 32K -s 256K -S 256K >> $TMPFILE &
    netperf -H $PEER -p $PORT -t TCP_STREAM -P 0  -c -l $DURATION
            -- -m 32K -M 32K -s 256K -S 256K >> $TMPFILE &
done

sleep $DURATION

echo -n "Total result: "

cat $TMPFILE |  awk '{sum += $5} END{print sum}'
```

Figure 6: Script to run Netperf using multistream

In order to measure transactions performance, the test type should be `TCP_RR` or `UDP_RR`, as shown by Figure 5.

In order to do some complex benchmark using transaction, a script called `tcp_rr_script` is also available and can be configured to accomplish the user's needed.

Since Netperf uses common sockets to transmit the data, it is possible to set the socket parameters used by a test. Also, it is possible to configure some details of the specific test case that is being used. To do so, the parameter `-- -h` should be appended in the end of the Netperf line, as `netperf -t TCP_RR -- -h` for the request performance test case.

### 5.2 Netperf multi stream

Netperf supports multi stream in version four only, which may not be available to some distro distro. Netperf version two is the mostly distributed and used version. The script described in Figure 6 simulates a multi stream Netperf, and is widely used in the network performance community.

### 5.3 Pktgen

Pktgen is a high performance testing tool, which is included in the Linux kernel as a module. Pktgen is currently one of the best tools available to test the transmission flow of network device driver. It can also be used to generate ordinary packets to test other network devices. Especially of interest is the use of pktgen to test routers or bridges which often also use the Linux network stack. Pktgen can generate high bandwidth traffic specially because it is a kernel space application. Hence it is useful to guarantee that the network will work properly on high load traffic.

It is important to note that pktgen is not a Netperf replacement, pktgen cannot execute any TCP/IP test. It is just used to help Netperf in some specific transmission test cases.

Pktgen can generate highly traffic rates due a very nice kernel trick. The trick is based on cloning ready-to-transmit packets and transmitting the original packet and the cloned ones, avoiding the packet construction phase. This idea basically bypass almost all the protocol kernel layers, generating a high bandwidth traffic without burdening the CPU.

```
Params: count 10000000  min_pkt_size: 60  max_pkt_size: 60
     frags: 0  delay: 0  clone_skb: 1000000  ifname: eth3
     flows: 0 flowlen: 0
     dst_min: 10.1.1.2  dst_max:
     src_min:   src_max:
     src_mac: 00:C0:DD:12:05:4D  dst_mac: 00:C0:DD:12:05:75
     udp_src_min: 9  udp_src_max: 9  udp_dst_min: 9
     udp_dst_max:
     src_mac_count: 0  dst_mac_count: 0
     Flags:
Current:
     pkts-sofar: 10000000  errors: 16723
     started: 1240429745811343us
              stopped: 1240429768195855us idle: 1954836us
     seq_num: 10000011  cur_dst_mac_offset: 0
              cur_src_mac_offset: 0
     cur_saddr: 0xa010101  cur_daddr: 0xa010102
     cur_udp_dst: 9  cur_udp_src: 9
     flows: 0
Result: OK: 22384512(c20429676+d1954836) usec,
10000000 (60byte,0frags)
  446737pps 214Mb/sec (214433760bps) errors: 16723
```

Figure 7: pktgen output example

Figure 7 shows a typical example of the output after running a test with pktgen.

It is easy but not trivial to run a pktgen example and the steps will not be covered in this article. For more information, the file `Documentation/networking/pktgen.txt` at the kernel code is a good source.

### 5.4 Mpstat

Mpstat is a tool provided by `sysstat`[8] package that is probably the most used tool to verify the processors load during a network performance test. Mpstat monitors SMP CPUs usage and it is a very helpful tool to discover if a CPU is overloaded and which process is burdening each CPU.

Mpstat can also display statistics with the amount of IRQs that were raised in a time frame and which CPU handled them, as it is shown on Figure 8. This is a very useful test to guarantee that the IRQ affinity, discussed earlier, is working properly. In order to discover which IRQ number represents which device line, the file `/proc/interrupts` contains the map between the IRQ number and the device that holds the IRQ line.

### 5.5 Other technologies

This section intends to briefly describe some other technologies that help to get a better network utilization.

### 5.6 I/OAT

Intel I/OAT is a feature that improves network application responsiveness by moving network data more efficiently through Dual-Core and Quad-Core Intel Xeon processor-based servers when using Intel network cards. This feature improves the overall network speed.

In order to enable the Intel I/OAT network accelerations the driver named `ioatdma` should be loaded in the kernel.

[8] `http://pagesperso-orange.fr/sebastien.godard/`

```
12:12:25  CPU  18/s  59/s  182/s  216/s  338/s  471/s  BAD/s
12:12:25    0  0.09  0.88   0.01   0.00   0.00   0.00   2.23
12:12:25    1  0.13  0.88   0.00   0.00   0.00   0.00   0.00
12:12:25    2  0.15  0.88   0.01   0.00   0.00   0.00   0.00
12:12:25    3  0.19  0.88   0.00   0.00   0.00   0.00   0.00
12:12:25    4  0.10  0.88   0.01   0.00   0.00   0.00   0.00
12:12:25    5  0.13  0.88   0.00   0.00   0.00   0.00   0.00
12:12:25    6  0.08  0.88   0.01   0.00   0.00   0.00   0.00
12:12:25    7  0.20  0.88   0.00   0.00   0.00   0.00   0.00
```

Figure 8: Mpstat output

```
modprobe ioatdma
```

It is not recommended to remove the `ioatdma` module after it was loaded, once TCP holds a reference to the `ioatdma` driver when offloading receive traffic.

### 5.7 Remote DMA

Remote Direct Memory Access (RDMA) is an extension of DMA where it allows data to move, bypassing the kernel, from the memory of one computer into another computer's memory. Using RDMA permits high-throughput, low-latency networking, which is a great improvement in many areas. The performance is visible mainly when the communication happens between two near computers, where the RTT is small.

For security reasons, it is undesirable to allow the remote card to read or write arbitrary memory on the server. So RDMA scheme prevents any unauthorized memory accesses. In this case, the remote card is only allowed to read and write into buffers that the receiver has explicitly identified to the NIC as valid RDMA targets. This process is called *registration*.

On Linux there is a project called `OpenRDMA` that provides an implementation of RDMA service layers. RDMA layer is already implemented by common applications such as NFS. NFS over RDMA is already being used and the performance was proved to be much better than the standard NFS.

### 5.8 TCP Offload Engine

TCP Offload Engine (TOE) is a feature that most network cards support in order to offload the TCP engine to the card. It means that the card has the TCP states for an established connection. TOE usually help the kernel, doing trivial steps in the card as fragmentation, acknowledges, etc.

This feature usually improve the CPU usage and reduces the PCI traffic, but creates issues that are hard to manage, mainly when talking about security aspects and proprietary implementations. Based on these facts, Linux doesn't support TOE by default, and a vendor patch should be applied in the network driver in order to enable this feature.

## 6 References

[1] D. Newman, *RFC2647—Benchmarking Terminology for Firewall Performance*, `http://www.faqs.org/rfcs/rfc2647.html`

[2] Annie P. Foong, Thomas R. Huff, Herbert H. Hum, Jaidev P. Patwardhan, Greg J. Regnier, *TCP Performance Re-Visited* `http://www.cs.duke.edu/~jaidev/papers/ispass03.pdf`

[3] Amber D. Huffman, Knut S. Grimsurd, *Method and apparatus for reducing the disk drive data transfer interrupt service latency penaltyr* US Patent 6640274

[4] Van Jacobson, Michael J. Karels, *Congestion Avoidance and Control*, `http://ee.lbl.gov/papers/congavoid.pdf`

[5] Chunmei Liu, Eytan Modiano, *On the performance of additive increase multiplicative decrease (AIMD) protocols in hybrid space-terrestrial networks*, `http://portal.acm.org/citation.cfm?id=1071427`

[6] Kevin Fall, Sally Floyd, *Comparisons of Tahoe, Reno, and Sack TCP* `http://www.icir.org/floyd/papers/sacks.pdf`

[7] Jeonghoon Mo, Richard J. La, Venkat Anantharam, and Jean Walrand *Analysis and Comparison of TCP Reno and Vegas*, `http://netlab.caltech.edu/FAST/references/Mo_comparisonwithTCPReno.pdf`

# A day in the life of a Linux kernel hacker...

Why who does what when and how!

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

## Abstract

The Linux kernel is a huge project with contributors spanning the globe. Its usefulness and other advantages continue to draw new users on a daily basis. But some users will discover problems with the code, and others will eventually find a need to add their own features to Linux. Whether you are a user in need of support or a developer trying to enhance the kernel, it is good to know something about who is in the community and how they work together.

This topic will introduce the newcomer to some of the characters in the Linux community and some of the roles they play. It will highlight some of the tasks Linux hackers perform on a day-to-day basis, and give a general overview of how work gets done within the community.

## 1 Introduction

Have you ever wondered how Linux kernel hackers spend their days? I am sure that many people have a picture in their mind: a pale-faced, shaggy, sandal-clad man basking in the glow of his LCD in a corner of his mother's basement while C code for drivers, schedulers, and memory allocators oozes from his fingertips. That image is, of course, not entirely inaccurate. However, there is much more to the community than that stereotype. Not only is there a wide diversity amongst the participants, there are far more roles to play than merely that of sunlight-deprived developer!

### 1.1 Why is this interesting?

You might ask, "So what?" Many might be satisfied to use Linux (either directly or indirectly) without knowing any details about how it came to be or how it continues to evolve. But one must realize that the Linux kernel is a huge software project with literally millions of lines of code, thousands of adjunct developers, hundreds of regular developers, and dozens of core developers. The Linux kernel is a study in management of complex projects, and there are many lessons to be drawn from observing how Linux is developed.

Beyond one's own intellectual expansion, there are more practical reasons why one might want to understand how Linux is developed. A direct user of Linux (or any other system) is bound to encounter a problem eventually, and that user will probably want to see that problem fixed. Also, many systems are now developed using Linux as a component. Developers working on such systems will want to understand how the community works, not only to help themselves get problems fixed, but also to understand how to get their own code incorporated into Linux in order to reap the benefits of community maintenance. Finally, one may wish to become an active member of the Linux community. In fact there are many reasons for joining the community. These include "scratching your own itch," making the world a better place, or simply building your own public profile.

### 1.2 What is so different?

Of course, lots of software is developed behind closed doors in any number of companies around the world. Surely there are any number of people who know how to develop software? That is true, but there are some key differences between those practices and how software is developed in the open source community. Some important differences exist in terms of the role of profit, the hierarchy of authority, and how technical decisions are made.

One of the more obvious points about the open source community is the role of financial profit. Such profit is not necessarily the main motivator for participation in

the community. Many people participate solely as hobbyists, while others have specific needs and participate in order to "scratch their own itch." Still others honestly believe that they are making the world a better place. The presence of such players in the community creates a far different dynamic than one finds in a traditional closed software shop.

Another poignant difference between traditional software development and the open source community is the lack of central authority figures. Even Linus Torvalds himself has no inherent authority beyond his own skill and participation. If Linus were to make irrational decisions or were simply to lose interest in the Linux kernel project, the community would simply reform around one or more other leaders. The need to recruit and retain contributors through the merit of the leadership is another key difference between traditional software development and the open source community.

A final difference worth noting is the notion of meritocracy. In a traditional software development shop, a project of any considerable size will quickly be subdivided into component parts and each part will be assigned to an individual or team. Those teams will typically toil in isolation until they have something working, then they will submit that to the final product with little or no external review. In the open source community, open review is part of the process both during development and before the final merge. In many cases, alternative implementations compete with one another and the community chooses between them based on technical merit and individual needs. Such "wasted" effort would not be tolerated in most traditional software development shops, but the open source community is big enough to afford it and is stronger for it.

### 1.3 Let's explore!

Hopefully the point has been made that the open source community in general and the Linux community in particular is worth studying. Below is a discussion of what types of people are involved in that community and what roles they play. Also discussed are some of the tools they use and how they spend their time. Finally, some time is spent discussing the actual processes used for development and how they interact with one another. By the end, the reader should have a good idea of who in the community does what, when they do it, why they do so, and how.

## 2 Why who does what...

People from all over the world become involved in the Linux community for any number of reasons. These people apply their diversity of talents to a number of different roles within the community, many of which are not directly related to writing software. A number of tools facilitate this cooperation. An overview of these topics will guide the reader's understanding of the community.

### 2.1 Motivations

It is probably impossible to enumerate every possible motivation for becoming a Linux contributor. Still, most motivations fall into a few loosely defined categories. These motivations run the gamut from commercialism to volunteerism, and span from self-interest to altruism.

The most well-known reason and most commonly cited one for getting involved with Linux is "scratching an itch." Nearly everyone needs software for something nowadays, and many people and organizations need software either that is unavailable or that those people and organizations cannot afford to obtain from a vendor. In many such situations, this software is developed and deployed internally by those organizations.

The nature of software is such that once it exists, the cost of duplicating and distributing that software is negligible. This is especially true when that distribution is done electronically and done by others at their own expense. Such distribution also allows those with similar software needs to find and support each other, sharing resources to develop and improve software for the widest possible audience. As the audience widens, so increases the potential benefits of such sharing. Since the kernel is the central component of a Linux-based system, "scratch your own itch" contributors are quite common in the Linux community.

A large number of Linux contributors are working on commercial software development projects that use the Linux kernel. Those employed by distribution vendors such as Red Hat are obvious examples of this. But there are any number of smaller software vendors developing embedded systems or other specialized products and making contributions to the kernel. These contributions are often small bug fixes or specialized device drivers,

but can be more generic components such as filesystems, compression algorithms, networking subsystems, etc. Just like "scratch your own itch" contributors, commercial developers recognize the value of community software development and maintenance of kernel components.

A few other motivations are commonly found within the community. Some number of community contributors make their living doing contract development work. This typically involves short-term work on behalf of product-focused companies that need specific features developed for the Linux kernel. Some other contributors are fortunate enough to be sponsored to do Linux kernel work under their own direction due to the good graces of some company or other benefactor. Finally, some number of contributors work on Linux because they believe they are making the world a better place or for some other similarly altruistic reason.[1]

## 2.2 Roles

In a community-based software development project, it is important to recognize that not everyone is writing code for the project. While this is certainly an important and necessary skill, it is generally insufficient for a successful project. People are needed to test the software and to report bugs, to write and review the code, to manage the code, and to document and write about the code. Each of these roles plays an important part within the community.

### 2.2.1 Bug Reporter

One of the most important roles in the community is the bug reporter. While many people will run new kernels and many of those will experience one problem or another, few will bother to report those problems and fewer still will not only provide useful information but also remain engaged long enough to find fixes. Those individuals are invaluable in the creation and maintenance of high quality software.

[1]In any case, the author suspects that the sum of these groups is dwarfed by the number of commercial and "scratch your own itch" contributors.

### 2.2.2 Tester

In a sense, every user is also a tester. But in reality most users exercise little more than the core functionality of a given piece of software. True testing requires repetition, documentation, and skill as well as the dedication to apply those resources. Testers not only find problems but also assist in analysis by finding the boundaries of the problems they identify. Testers are highly valued members of the community.

### 2.2.3 Coder

The coder is the most celebrated member of the community. The coder tackles the problem of producing source code changes to fix a problem or implement a new feature. Coders provide the raw material for the Linux kernel. Obviously, without coders the project would not exist.

### 2.2.4 Reviewer

An often overlooked person in the community is the reviewer. When the coder does his job, he posts his product (i.e., a patch) to a mailing list. The reviewer evaluates the change, comments upon its form and impact, and often makes suggestions for changes or refinements. The reviewer has one of the most important roles in ensuring initial code quality.

### 2.2.5 Maintainer

The maintainers are the "old men"[2] in the community. The maintainers are responsible for taking what the coders produce, deciding when the reviewers have added enough value, and merging the results into the upstream kernel tree. They also communicate with testers, bug reporters, and others to ensure that code quality is maintained at a good level and that good processes are being followed. Maintainers perform a role similar to a manager or team leader in a traditional software development shop.

[2]Of course, they are not all old and not all men...

### 2.2.6 Technical Writer

A technical writer is one who produces documentation of technical details surrounding the kernel. This primarily includes documenting both those APIs for internal use within the kernel and those for communicating with userland programs. Without such documentation, it would be difficult to sustain the development community.

### 2.2.7 Journalist

An important aspect of maintaining a community is keeping contributors aware of what else is happening within that community. The Linux kernel is a huge project, and it can be difficult to know what new developments are in progress and what old components are being revamped or removed. Following all of the relevant mailing lists is a daunting task by itself, much less if one is trying to write code or run tests. The journalists in the community keep everyone abreast of what is happening now and what is coming next.

## 2.3 Tools

It should not be surprising that a software development community uses a number of software tools to keep itself running smoothly. Still, it is worthwhile to enumerate some of them and discuss how they are used. Important tools include both those for communications and those specific to software development.

### 2.3.1 Email

Email is the single most important tool within the community. The Linux kernel community is diverse and spread across the globe. It is generally difficult to assemble people in one place or even to gather for a conveniently timed teleconference. Consequently discussions are held on mailing lists. This has the added benefits of documenting and archiving such discussions as well as generally keeping such discussions short and direct.

### 2.3.2 Bugzilla, etc.

Bugzilla and other bug-tracking tools are often used for their intended purpose. The kernel has its own Bugzilla instance,[3] but the bug trackers of distributions and other kernel-related projects are often used as well. Such tools help to organize bug report information and to segregate one bug's information from reports of other bugs, as well as from other discussions that would otherwise clutter a mailing list.

### 2.3.3 IRC

IRC is commonly used by active kernel contributors for real-time communications. Chat bridges the gap between email and telephony, allowing precise communications in a timely and direct fashion.

### 2.3.4 Wikis

A wiki is the documentary analog of open source development. As such, it fits nicely with the general mindset of the Linux kernel development community. Many parts of the project use wikis to document designs, user interfaces, API changes, and other information pertinent to users and/or other developers.

### 2.3.5 Code Analyzers

Policing large bodies of code can be daunting, and line-by-line analysis of code for trivial or subtle coding errors can be tedious and error prone. Fortunately, many such problems can be identified algorithmically. The kernel includes `checkpatch.pl` which can be used to find many simple problems, especially those relating to coding style. Tools such as Sparse[4] are often used for deeper code analysis.

### 2.3.6 Git

No discussion of tools used in the Linux kernel community would be complete without mentioning `git`, the primary revision control tool used within the community. Unlike traditional revision control tools, `git` uses

[3] `http://bugzilla.kernel.org`

[4] `http://www.kernel.org/pub/software/devel/sparse/`

a distributed development model. Among other ramifications, this means that every copy of the `git` repository can operate independently. Further, formerly independent repositories can be merged at any time, allowing for development efforts to proceed according to their own schedules without requiring lots of work to resynchronize with the upstream kernel tree. `Git` is probably the single most important tool in use by the community today.

### 2.4 Patches

An important point must be made regarding patches. A patch is a unit of change to source code.[5] Typically patches are sent in email for review and then later applied to a tree in `git` to form the basis of future development. Patches contain limited context used to identify affected pieces of source code files. This helps to make them resilient against unrelated changes in the same files.

The great thing about a patch is that it focuses attention just on those pieces of code that are being changed. This allows for direct review of a proposed change without lots of effort in locating or identifying that change. The alternative[6] is to pass around changed versions of complete files. These files are awkward to handle and their use makes identifying changes difficult and error prone. The use of the simple patch is an elegant enabling technology for distributed development on a large scale.

## 3 When and How…

Now that we have identified the motivations of the players, the roles they play, and the tools they use, we should look at the processes used to coordinate their efforts. We will discuss how development needs are identified, what process is used to vet patches, and the route patches follow on their way to the "official" Linux kernel.

[5]While the word "patch" can connote something shoddy or temporary to a native English speaker, in the context of Linux kernel development it has no such connotations. Instead, the term patch derives from the name of the non-interactive editor used to apply the changes to the source code.

[6]The author's experience suggests that this code review alternative is used all too often in the traditional software development world.

### 3.1 Identify a Need

Perhaps it goes without saying, but the first step in any development process is to identify a need. In many cases, the need will originate with a bug report. This might be in Bugzilla or another bug tracker, or it might come via email or IRC. In other cases, the need will derive from an external project requirement such as the need for a driver for a new device or a new networking subsystem for a certain application. In other cases the "need" is because some other operating system has a feature that is deemed desirable for Linux. Finally, in many cases the development need originates with someone saying "wouldn't it be cool if…?" In any case, once a need is identified, the next step is to write some code and post a patch.

### 3.2 Development Cycle

Many people find the Linux development process to be daunting. In reality it is quite simple, although personalities can make the process a bit humbling. The basic process begins with creating and posting a patch to an appropriate mailing list. With any luck this provokes someone to review the patch and make appropriate comments. Or, reviewers may simply indicate their approval with an "ACK."[7] In many cases it will be necessary to make revisions to the patch and post a new version to the same mailing list. This process should be repeated until the patch is accepted by the maintainer.

Often new contributors are either intimidated by the above process or they simply do not believe it to be the best use of their time. The temptation is to develop in solitude until the developer is completely confident in the soundness of a patch. Do not make this mistake! Inevitably someone will find legitimate problems with any significant patch series. Trying to avoid the review-revise-repost cycle will only waste a developer's time and create frustration between the developer and the community when the patch is finally posted for review.

### 3.3 Path Through the Trees

The first stop for an accepted patch is in a maintainer's tree. A variety of maintainers' trees exist for subsystems

[7]*ACK* is short for *acknowledged*.

like networking and SCSI, features like SELinux and realtime, and architectures like ARM, MIPS, SPARC, etc. Maintainers' trees are usually limited to usage by interested parties such as developers and users with specific needs or interests.

To expand test coverage and community exposure, other trees aggregate input from the various maintainers' trees. In the past this role was primarily filled by Andrew Morton's `-mm` tree, but more recently the `linux-next` tree has become more popular. The `linux-next` tree pulls the current versions of many (probably most) maintainers' trees to create a preview of what will soon be available in the "official" Linux kernel.

Periodically Linus will pronounce a kernel ready for release. Prior to that time, maintainers will have been staging changes for the next Linux release and making them available through the `linux-next` tree. After the release, Linus spends two weeks merging the patches the maintainers have been staging. Between the end of that period and the next release, only necessary bug fixes are merged into the "official" tree by Linus. Any changes that are not necessary bug fixes are again staged by the maintainers for the following release. This period lasts several weeks as the kernel is exposed to more testing and as bugs are uncovered and fixed. After 2–3 months, Linus will pronounce the kernel ready for another release, and then the cycle begins again.

## 4 Conclusion

The reader has been provided with an overview of how the Linux kernel is developed. We have discussed why the contributors are involved, and what jobs they perform. We discussed many of the tools the community uses to manage itself, and specifically discussed the importance of the patch as a unit of work.

With all that background information, we went on to discuss the development cycle for the kernel. We touched on how development needs are identified, and discussed how patches are proposed, reviewed, and accepted into the kernel. Finally we discussed the various trees a patch has to traverse before making its way to Linus.

The author hopes this information has been useful. The community needs to sustain itself with contributors. A variety of roles need to be filled—no special training is required. The author hopes that the reader will be inspired to find a way to join us! A well informed and active community continues to be the force behind the continued success of Linux and the open source community in general.

# Transcendent Memory and Linux

Dan Magenheimer, Chris Mason, Dave McCracken, Kurt Hackel
*Oracle Corp.*
first.last@oracle.com

## Abstract

Managing a fixed amount of memory (RAM) optimally is a long-solved problem in the Linux kernel. Managing RAM optimally in a virtual environment, however, is still a challenging problem because: (1) each guest kernel focuses solely on optimizing its entire fixed RAM allocation oblivious to the needs of other guests, and (2) very little information is exposed to the virtual machine manager (VMM) to enable it to decide if one guest needs RAM more than another guest. Mechanisms such as *ballooning* and hot-plug memory (Schopp, OLS'2006) allow RAM to be taken from a *selfish* guest and given to a *needy* guest, but these have significant known issues and, in any case, don't solve the hard problem: Which guests are selfish and which are needy? IBM's Collaborative Memory Management (Schwidefsky, OLS'2006) attempts to collect information from each guest and provide it to the VMM, but was deemed far too complex and attempts to upstream it have been mostly stymied.

Transcendent Memory (*tmem* for short) is a new approach to optimize RAM utilization in a virtual environment. Underutilized RAM from each guest, plus RAM unassigned to any guest (*fallow* memory), is collected into a central pool. Indirect access to that RAM is then provided by the VMM through a carefully crafted, page-copy-based interface. Linux kernel changes are required but are relatively small and not only provide valuable information to the VMM, but also furnish additional "magic" memory to the kernel, provide performance benefits in the form of reduced I/O, and mitigate some of the issues that arise from ballooning/hotplug.

## 1 Introduction

RAM is cheap. So, if a Linux system is running a workload that sometimes runs out of memory, common wisdom says to add more RAM. As a result, in any given system at any given time, a large percentage of RAM is sitting unused or *idle*. But this RAM is not really empty; Linux—and any modern operating system—uses otherwise idle RAM as a *page cache*, to store pages from disk that might be used at some point in the future. But the choice of which pages to retain in the page cache is a guess as to what the future holds—a guess which is often wrong. So even though this RAM is holding real data from the disk, much of it is essentially still idle. But that's OK; in a physical system, there's nothing else to do with that memory anyway, so if the guess is wrong, no big loss, and if the guess is right, the data need not be read from the disk, saving an I/O.

The whole point of virtualization is to improve utilization of resources. The CPUs and I/O bandwidth on many physical servers are lightly utilized and so virtualization promises to consolidate these physical servers as virtual servers on the same physical machine, to better utilize these precious CPUs and I/O devices. Statistically, these virtual servers rarely all simultaneously assert demand for the same resources, so the physical resources can be *multiplexed*, thus allowing even more virtual machines to share the same physical machine.

But what about RAM? RAM is harder to statistically multiplex and so is becoming a bottleneck in many virtualized systems. One solution is always to just add more RAM, but as CPUs and I/O devices are more efficiently utilized, RAM is becoming a significant percentage of the cost of a data center, both at time-of-purchase and as a sink for energy. As a result, RAM is increasingly *not* cheap, and so we would like to improve the utilization of RAM as a first-class resource.

Why can't we apply the same techniques for sharing CPUs and I/O devices to memory? In short, it is because memory is a non-renewable resource. Every second there is a fresh new second of CPU time to divide between virtual machines. But memory is assumed to be persistent; memory containing important data for one virtual machine during one second cannot be randomly

given to another virtual machine at the next second. This is complicated in all modern operating systems by RAM utilization techniques such as page caching. Linux has a reason to hoard RAM, because the more RAM it has, the more likely its page cache will contain pages it needs in the future, thus saving costly I/Os.

To be sure, mechanisms exist to take memory away from one virtual machine and give it to another. *Ballooning*, for example, cleverly does this by creating a dynamically-loadable pseudo-driver which resides in each virtual machine and requests pages of memory from the kernel, secretly passing them to the virtual machine manager (VMM) where they can be reassigned to another virtual machine, and later returned if needed. And hot-plug memory techniques can similarly be used to surrender and reclaim memory, albeit at a much coarser granularity. Both of these mechanisms have known weaknesses, not the least of which is they don't solve the thorniest problem: How can it be determined how much memory each virtual machine really needs? That is, which ones are truly "needy" and which ones are selfishly hoarding memory?

IBM's Collaborative Memory Management deeply intrudes into the Linux memory management code and maintains a sophisticated state machine to track pages of memory and communicate status to the VMM. But if changes are being made to the kernel anyway, why not create a true collaboration between the kernel and the VMM?

This is the goal of Transcendent Memory, or *tmem*. Underutilized and unassigned (*fallow*) RAM is collected by the VMM into a central pool. Indirect access to that RAM is then provided by the VMM through a carefully- crafted, page-copy-based interface. Linux kernel changes are required but are relatively small and not only provide valuable information to the VMM, but also furnish additional "magic" memory to the kernel, provide performance benefits in the form of reduced I/O, and mitigate some of the issues that arise from ballooning/hotplug.

In the remaining sections, we will first provide an overview of how tmem works. We will then describe some Linux changes necessary to utilize some of the capabilities of tmem, implementing useful features that we call *precache* and *preswap*. Finally, we will suggest some future directions and conclude.

## 2 Transcendent Memory Overview

We refer to a tmem-modified kernel as a *tmem client* and to the underlying tmem code as a *tmem implementation* or just as *tmem*. The well-specified interface between the two is the *tmem API*. Xen provides a tmem implementation, and the code is structured to be easily portable. A Linux client patch is available for 2.6.30 and we will discuss that shortly. But first, we will describe the operational basics of tmem.

### 2.1 Tmem pool creation

In order to access tmem memory, the kernel must first create a *tmem pool* using the `tmem_new_pool` call. The `tmem_new_pool` call has a number of parameters which will be described in more detail later, but an important one is whether the memory in the pool is needed to be persistent or non-persistent (*ephemeral*). While it might seem a no-brainer to always request persistent memory, we shall see that, due to certain restrictions imposed by tmem, this is not the case.

If tmem successfully creates the pool, it returns a small non-negative integer, called a *pool_id*. Tmem may limit the number of pools that can be created by a tmem client—the Xen implementation uses a limit of 16—so pool creation may fail, in which case `tmem_new_pool` returns a negative errno.

Once a tmem pool is successfully created, the kernel can use the pool_id to perform operations on the pool. These operations are page-based and the individual pages are identified using a three-element tuple called a *handle*. The handle consists of a pool_id, a 64-bit object identifier (*obj_id*), and a 32-bit page identifier (*index*). The tmem client is responsible for choosing the handle and ensuring a one-to-one mapping between handles and pages of data.

Though they need not be used as such, the three handle components can be considered analogous to a filesystem, a file (or inode number) within the filesystem, and a page offset within the file. More generically, the handle can be thought of as a non-linear address referring to a page of data.

A created pool may be *shared* between clients and shared pools may be either ephemeral or persistent. Clients need only share a 128-bit secret and provide it

at pool creation. This is useful, for example, when multiple virtual nodes of a cluster reside on the same physical machine, or as an inter-VM shared memory mechanism. For the purposes of this paper, we will assume that created pools are *private*, not shared, unless otherwise noted.

### 2.2 Tmem basic operations

The two primary operations performed on a tmem pool are `tmem_put_page` (or *put*) and `tmem_get_page` (or *get*). The parameters to `tmem_put_page` consist of a handle and a physical page frame, and the call indicates a request from the kernel for tmem to copy that page into a tmem pool. Similarly, the parameters to `tmem_get_page` consist of an empty physical page frame and a handle, and the call indicates a request to find a page matching that handle and copy it into the kernel's empty page frame.

If tmem elects to perform the copy, it returns the integer value 1. If it elects to NOT perform the copy, it returns the integer value 0. If it is unable to perform the copy for a reason that might be useful information to the client, it returns a negative errno.

In general, a put to an ephemeral pool will rarely fail but a get to an ephemeral pool will often fail. For a persistent pool, a put may frequently fail but, once successfully put, a get will always succeed. Success vs failure may appear random to the kernel because it is governed by factors that are not visible to the kernel.

Note that both get and put perform a true copy of the data. Some memory utilization techniques manipulate virtual mappings to achieve a similar result with presumably less cost. Such techniques often create aliasing issues and suffer significant overhead in TLB flushes. Also, true copy avoids certain corner cases as we shall see.

### 2.3 Tmem coherency

The kernel is responsible for ensuring coherency between its own internal data structures, the disk, and any data put to a tmem pool. Two tmem operations are provided to assist in ensuring this consistency: `tmem_flush_page` takes a handle and the call ensures that a subsequent get with that handle will fail; `tmem_flush_object` takes a pool_id and an obj_id and ensures that a get to ANY page matching that pool_id and obj_id will fail.

In addition, tmem provides certain coherency guarantees that apply to sequences of operations using the same handle: First, put-put-get coherency promises that a *duplicate put* may never silently fail; that is in a put-put-get sequence, the get will never return the stale data from the first put. Second, get-get coherency promises that if the first get fails, the second one will fail also.

Note also that a get to a private ephemeral pool is defined to be destructive, that is, if a get is successful, a subsequent get will fail, as if the successful get were immediately followed by a flush. This implements *exclusive cache* semantics.

### 2.4 Tmem concurrency

In an SMP environment, tmem provides concurrent access to tmem pools but provides no ordering guarantees, so the kernel must provide its own synchronization to avoid races. However, a tmem implementation may optionally serialize operations within the same object. So to maximize concurrency, it is unwise to restrict usage of tmem handles to a single object or a very small set of objects.

### 2.5 Tmem miscellaneous

Tmem has additional capabilities that are beyond the scope of this paper, but we mention several briefly here:

- A tmem implementation may transparently compress pages, trading off cpu time spent compressing and decompressing data to provide more apparent memory space to a client.
- Extensive instrumentation records frequency and performance (cycle count) data for the various tmem operations for each pool and each client; and a tool is available to obtain, parse, and display the data.
- A pagesize other than 4KB can be specified to ensure portability to non-x86 architectures.
- Pool creation provides versioning to allow forwards and backwards compatibility as the tmem API evolves over time.

- Subpage reads, writes and exchange operations are provided.
- Pools can be explicitly destroyed, if necessary, to allow reuse of the limited number of pool_ids.

More information on tmem can be found at `http://oss.oracle.com/projects/tmem`

## 3 Linux and tmem

From the perspective of the Linux kernel, tmem can be thought of as somewhere between a somewhat slow memory device and a very fast disk device. In either case, some quirks must be accommodated. The tmem "device":

- has an unknown and constantly varying size
- may be synchronously and concurrently accessed
- uses object-oriented addressing, where each object is a page of data
- can be configured as persistent or non-persistent

Although these quirks may seem strange to kernel developers, they provide a great deal of flexibility, essentially turning large portions of RAM into a renewable resource. And although a kernel design for using tmem that properly accommodates these quirks might seem mind-boggling, tmem actually maps very nicely to assist Linux memory management code with two thorny problems: page cache *refaults* [vanRiel, OLS'2006] and RAM-based swapping. We call these new tmem-based features *precache* and *preswap*. We will describe both, but first, to illustrate that they are not very intrusive, Figure 1 shows the diffstat for a well-commented example patch against Linux 2.6.30.

This patch not only supports both precache and preswap but:

- can be configured on or off at compile-time
- if configured off, all code added to existing Linux routines compiles into no-ops
- if configured on but Linux is running native, has very low overhead
- if configured on and running on Xen, has very low overhead if tmem is not present (e.g. an older version of Xen) or not enabled
- is nicely-layered for retargeting to other possible future (non-Xen) tmem implementations

### 3.1 Precache

Precache essentially provides a layer in the memory hierarchy between RAM and disk. In tmem terminology, it is a private-ephemeral pool. Private means that data placed into precache can only be accessed by the kernel that puts it there; ephemeral means that data placed there is non-persistent and may disappear at any time. This non-persistence means that only data that can be regenerated should be placed into it which makes it well-suited to be a "second-chance" cache for clean page cache pages.

When Linux is under memory pressure, pages in the page cache must be replaced by more urgently needed data. If the page is dirty, it must first be written to disk. Once written to disk—or if the page was clean to start with—the pageframe is taken from the page cache to be used for another purpose. We call this an *eviction*. After a page is evicted, if the kernel decides that it needs the page after all (and in certain workloads, it frequently does), it must fetch the page from disk, an unfortunate occurrence which is sometimes referred to as a *refault*. With precache, when a page is evicted, the contents of the page are copied, or *put*, to tmem. If the page must be refaulted, a *get* is issued to tmem, and if successful, the contents of the page has been recovered. If unsuccessful, it must be fetched from disk and we are no worse off than before.

Let's now go over the precache mechanism in more detail.

When a tmem-capable filesystem[1] is mounted, a `precache_init` is issued with a pointer to the filesystem's superblock as a parameter. The `precache_init` performs a `tmem_new_pool` call. If pool creation is successful, the returned pool_id is saved in a (new) field of the filesystem superblock.

When the filesystem is accessed to fetch a page from disk, it first issues a `precache_get`, providing

[1] Currently, ext3 is supported; ocfs2 and btrfs are in progress.

```
Changed files:

 fs/buffer.c                              |    5
 fs/ext3/super.c                          |    2
 fs/mpage.c                               |    8 +
 fs/ocfs2/super.c                         |    2
 fs/super.c                               |    5
 include/linux/fs.h                       |    7
 include/linux/swap.h                     |   57 +++++++
 include/linux/sysctl.h                   |    1
 kernel/sysctl.c                          |   12 +
 mm/Kconfig                               |   26 +++
 mm/Makefile                              |    3
 mm/filemap.c                             |   11 +
 mm/page_io.c                             |   12 +
 mm/swapfile.c                            |   46 +++++
 mm/truncate.c                            |   10 +
 drivers/xen/Makefile                     |    1
 include/xen/interface/xen.h              |   22 ++
 arch/x86/include/asm/xen/hypercall.h |    8 +

Newly added files:

 mm/tmem.c                                |   62 +++++++
 include/linux/tmem.h                     |   88 +++++++++++
 include/linux/precache.h                 |   55 +++++++
 mm/precache.c                            |  145 ++++++++++++++++++
 mm/preswap.c                             |  273 ++++++++++++++++++++++++++++++++++
 drivers/xen/tmem.c                       |   97 ++++++++++++
 include/xen/interface/tmem.h             |   43 +++++
```

Figure 1: Diffstat for linux-2.6.30 tmem patch, supporting both precache and preswap (From `http://oss.oracle.com/projects/tmem/files/linux-2.6.30`)

an empty `struct page`, a `struct address_space` mapping pointer, and a page index. The `precache_get` extracts from the mapping pointer the pool_id from the superblock and the inode number, combines these with the page index to build a handle, performs a `tmem_get_page` call, passing the handle and the physical frame number of the empty pageframe, and returns the result of the `tmem_get_page` call. Clearly, the first time each page is needed, the get will fail and the filesystem continues as normal, reading the page from the disk (using the same empty pageframe, inode number, and page index). On subsequent calls, however, the get may succeed, thus eliminating a disk read.

When a page is about to be evicted from page cache, a call to `precache_put` is first performed, passing the `struct page` containing the data, a `struct address_space` mapping pointer, and a page index. The `precache_put` extracts from the mapping pointer the pool_id from the superblock and the inode number, combines these with the page index to build a handle, performs a `tmem_put_page` call, passing the handle and the physical frame number of the data page, and returns the result of the `tmem_put_page` call. In all but the most unusual cases, the put will be successful. However, since the data is ephemeral, there's no guarantee that the data will be available to even an immediately subsequent get, so success or failure, the return value can be ignored.

Now, regardless of guarantee (but depending on the delay and the volume of pages put to the precache), there's a high probability that if the filesystem needs to refault the page, a `precache_get` will be successful. Every successful `precache_get` eliminates a disk read!

One of the challenges for precache is providing coherency. Like any cache, cached data may become stale and must be eliminated. Files that are overwritten, removed, or truncated must be ensured consistent between the page cache, the precache, and the disk. This must be accomplished via careful placement of calls to `precache_flush` and `precache_flush_inode`, the placement of which may differ from filesystem to filesystem. Inadequate placement can lead to data corruption; overzealous placement results in not only a potentially large quantity of unnecessary calls to `tmem_flush_page` and `tmem_flush_object`, but also the removal of pages from precache that might have been the target of a successful get in the near future. Since the flushes are relatively inexpensive and corruption is very costly, better safe than sorry! Optimal placement is not an objective of the initial patch and will require more careful analysis.

Note that one of the unique advantages of precache is that the memory utilized is not directly addressible. This has several useful consequences for the kernel: First, memory that doesn't belong to this kernel can still be used by this kernel. If, for example, a tmem-modified VM has been assigned a maximum of 4GB of RAM, but it is running on a physical machine with 64GB of RAM, precache can use part of the remaining, otherwise invisible, 60GB to cache pages (assuming of course that the memory is fallow, meaning other VMs on the physical machine are not presently using it). This "magic" memory greatly extends the kernel's page cache. Second, memory space that belongs to the kernel but has been temporarily surrendered through ballooning or hotplug activity may be re-acquired synchronously, without waiting for the balloon driver to asynchronously recover it from the VMM (which may require the memory to be obtained, in turn, from a balloon driver of another VM). Third, no `struct page` is required to map precache pages. In the previous 4GB-64GB example, a VM that might periodically need to balloon upwards to 64GB might simply be initially configured with that much memory (e.g. using the "maxmem" parameter to xen). But if that is the case, all pages in the 64GB must have a `struct page`, absorbing a significant fraction of 1GB just for kernel data structures that will almost never be used.

The benefits for the virtualized environment may not be as obvious but are significant: Every page placed in precache is now a renewable resource! If a balloon driver in VM *A* requests a page, it can be synchronously delivered simply by removing the page from the precache of VM *B* without waiting for the kernel and/or balloon driver in VM *B* to decide what page can be surrendered. And if a new VM *C* is to be created, the memory needed to provision it can be obtained by draining the precache of VM *A* and VM *B*. Further, pages placed in precache may be transparently compressed, thus magically expanding the memory available for precached pages by approximately a factor of two vs if the same memory were explicitly assigned to individual VMs.

Of course precache has costs too. Pages will be put to precache that are never used again; and every disk read is now preceded by a get that will often fail; and the flush calls necessary for coherency are also a requirement. Precache is just another form of cache and caches are not free; for any cache, a benchmark can be synthesized that shows cache usage to be disadvantageous. But caches generally prove to be a good thing. For precache, the proof will be, er, in the put'ing.

### 3.2 Preswap

Preswap essentially provides a layer in the swap subsystem between the swap cache and disk. In tmem terminology, it is a private-persistent pool. Again, private means that data placed into preswap can only be accessed by the kernel that put it there; persistent means that data placed there is permanent and can be fetched at will... but only for the life of the kernel that put it there. This semi-permanence precludes the use of preswap as a truly persistent device like a disk, but maps very nicely to the requirements of a swap disk.

In a physical system, sometimes the memory requirements of the application load running on a Linux system exceed available physical memory. To accommodate the possibility that this may occur, most Linux systems are configured with one or more swap devices; usually these are disks or partitions on disks. These swap devices act as overflow for RAM. Since disk access is orders of magnitude slower than RAM, a swap device is used as a last resort; if heavy use is unavoidable, a *swapstorm* may result, resulting in abysmal system performance. The consequences are sufficiently dire that system administrators will buy additional servers and/or purchase enough RAM in an attempt to guarantee that a swapstorm will never happen.

In a virtualized environment, however, a mechanism such as ballooning is often employed to reduce the amount of RAM available to a lightly-loaded VM in an attempt to *overcommit* memory. But if the light load is transient and the memory requirements of the workload on the VM suddenly exceed the reduced RAM available, ballooning is insufficiently responsive to instantaneously increase RAM to the needed level. The unfortunate result is that swapping may become more frequent in a virtualized environment. Worse, in a fully-virtualized data center, the swap device may be on the other end of a shared SAN/NAS rather than on a local disk.

Preswap reduces swapping by using a tmem pool to store swap data in memory that would otherwise be written to disk. Since tmem prioritizes a persistent pool higher than an ephemeral pool, precache pages—from this kernel or from another—can be instantly and transparently reprovisioned as preswap pages. However, in order to ensure that a malicious or defective kernel can't absorb all tmem memory for its own nefarious purposes, tmem enforces a policy that the sum of RAM directly available to a VM and the memory in the VM's persistent tmem pools may not exceed the maximum allocation specified for the VM. In other words, a well-behaved kernel that shares RAM when it is not needed can use preswap; a selfish kernel that never surrenders RAM will be unable to benefit from preswap. Even better, preswap pages may be optionally and transparently compressed, potentially doubling the data that can be put into tmem.

Now that we understand some of preswap's benefits, let's take a closer look at the mechanism.

When a swap device is first configured (via `sys_swapon`, often at system initialization resulting from an entry in `/etc/fstab`), `preswap_init` is called, which in turn calls `tmem_new_pool`, specifying that a persistent pool is to be created. The resulting pool_id is saved in a global variable in the swap subsystem. (Only one pool is created even if more than one swap device is configured.) Part of the responsibility of `sys_swapon` is to allocate a set of data structures to track swapped pages, including a 16-bit-per-page array called `swap_map`. Preswap pages also must be tracked, but a single "present" bit is sufficient and so the tmem-modified `sys_swapon` allocates a 1-bit-per-page `preswap_map` array.

When a page must be swapped out, a block I/O write request must be passed to the block I/O subsystem. The routine that submits this request first makes a call to `preswap_put`, passing only the `struct page` as a parameter. The `preswap_put` call extracts the swap device number and page index (called *type* and *offset* in the language of Linux swap code), combines it with the saved preswap_poolid to create a handle, and passes the handle along with the physical frame number of the page to `tmem_put_page`. If the put was successful, `preswap_put` then records the fact by setting the corresponding bit in the `preswap_map` and returns success (the integer 1). Otherwise, the integer 0 is returned. If `preswap_put` returns success, the page has been placed in preswap, the block I/O write is circumvented, and the `struct page` is marked to indicate that the page has been successfully written.

A similar process occurs when a page is to be swapped in, but two important points are worth noting. First, if the bit in the `preswap_map` corresponding to the page to be swapped in is set, the `tmem_get_page` will always succeed—it is a bug in tmem if it does not! Second, unlike an ephemeral pool, a get from a persistent pool is non-destructive; thus, the bit in the `preswap_map` is not cleared on a successful get. This behavioral difference is required as a swapped page is reference counted by the swap subsystem because multiple processes might have access to the page and, further, might concurrently issue requests to read the same page from disk!

However, this behavior leads to some complications in the implementation of preswap. First, an explicit flush is required to remove a page from preswap. Fortunately, there is precisely one place in the swap code which determines when the reference count to a swap page goes to zero, and so a `preswap_flush` can be placed here. Second, data from rarely used `init`-launched processes may swap out pages and then never swap them back in. This uses precious tmem pool space for rarely used data.

This latter point drives the need for a mechanism to *shrink* pages out of preswap and back into main memory. This `preswap_shrink` mechanism needs to be invoked asynchronously when memory pressure has subsided. To accomplish this, `sys_swapoff`-related routines have been modified to alternately `try_to_unuse` preswap pages instead of swap pages. In the current patch, the mechanism is invoked by writing a sysfs file,

`/sys/proc/vm/preswap`[2]. In the future, this should be automated, perhaps by kswapd.

One interesting preswap corner case worth mentioning is related to the tmem-enforced put-put-get coherency. Since a preswap get is non-destructive, duplicate puts are not uncommon. However, it is remotely possible that the second put might fail. (This can only occur if preswap pages are being compressed AND the data in the second put does not compress as well as the first put AND tmem memory is completely exhausted. But it *does* happen!) If this occurs, an implicit `preswap_flush` is executed, eliminating the data from the first put from the tmem pool. Both `preswap_put` and the routine that calls it must be able to recover from this, e.g. by clearing the corresponding `preswap_map` bit and by ensuring the page is successfully written to the swap disk.

## 4 Future directions

Tmem is a brand new approach to physical memory management in a virtualized environment. As such, we believe we are only beginning to see its potential.

We have done some investigation into *shared* tmem pools. A shared-ephemeral pool can serve nicely as a server-side cache for a cluster filesystem, or perhaps for a network-based filesystem. Like precache, this *shared precache* would reduce the cost of refaults but, in the case of virtual cluster nodes co-residing on the same physical node, a page evicted by one node might be found by a get performed by another node. A prototype of this has been implemented targeting the ocfs2 filesystem, using the 128-bit ocfs2 UUID as the shared secret that must be specified by both nodes when the shared pool is created.

With three quadrants of the private vs shared / persistent vs ephemeral matrix implemented, the fourth, a shared-persistent pool falls out easily. A shared-persistent pool looks like a fine foundation for inter-VM shared memory, and shared memory can be used as a basis for inter-VM communication or other capabilities. Several research projects implementing inter-VM messaging have been published. To our knowledge, none is yet available commercially.

[2]Reading this same sysfs file provides the number of pages written to preswap instead of to disk

Benchmarking is needed. But since nearly all virtualization deployments are implemented around assumptions and constraints that tmem intends to shatter, using yesterday's static benchmarks to approximate tomorrow's highly-dynamic utility data center workloads does a disservice to everyone.

With a well-defined API in place, additional implementations both above and below the API boundary are feasible. A native Linux implementation has been proposed, using standard kernel memory allocation to provision tmem pools. This might seem silly, but could serve as an effective API for compressing otherwise evicted or swapped pages to improve memory utilization when memory pressure increases—something like the `compcache` in the linux-mm project list, but with the capability of compressing page cache pages as well as swap pages. The API might also prove useful for limiting persistence requirements on or restricting access to new memory technologies, such as solid-state disks or blocks of memory marked for hot-delete.

Should tmem prove sufficiently advantageous in optimizing memory utilization across a data center, new tmem clients might be implemented to ensure that, for example, BSD VM's can play nice with Linux VMs. Or proprietary Unix versions in virtual appliance stacks. Or even Windows might be "enlightened" (or binary-patched).

Some argue that tmem-like features are redundant on KVM. Some believe otherwise. It will take a full KVM tmem implementation to decide.

Elevating memory to a full first-class resource opens new avenues for new research and new tools. VMM schedulers are smart enough to take into account CPU-bound VM's vs I/O-bound VMs. But how much of that I/O is refaulting/swapping due to insufficient memory? And can metrics obtained from tracking tmem put/get successes and failures be fed back to improve native Linux page replacement algorithms? Or to help Linux directly self-manage its own memory size without the obfuscations of a balloon driver?

Even further out, might ephemeral memory influence future system design? Does a memory node or memory blade make more sense for memory that is a renewable resource?

## 5 Acknowledgements

The authors thank Jeremy Fitzhardinge, Keir Fraser, Ian Pratt, Jan Beulich, Sunil Mushran, and Joel Becker for valuable feedback and Zhigang Wang for assistance in implementing the Xen control plane tools for the Xen tmem implementation.

## 6 References

R. van Riel, *Measuring Resource Demand on Linux* Proceedings of the Ottawa Linux Symposium 2006.

M. Schwidefsky *et al.*, *Collaborative Memory Management in Hosted Linux Environments* Proceedings of the Ottawa Linux Symposium 2006.

J. Schopp, K Fraser, and M. Silbermann, *Resizing Memory with Balloons and Hotplug*

Transcendent Memory home page, `http://oss.oracle.com/projects/tmem`

# Incremental Checkpointing for Grids

John Mehnert-Spahn, Eugen Feller, Michael Schoettner
*Heinrich Heine University of Duesseldorf, Duesseldorf, NRW, Germany*
{John.Mehnert-Spahn, Eugen.Feller, Michael.Schoettner}@uni-duesseldorf.de

## Abstract

The EU-funded project XtreemOS implements an open-source Linux-based grid operating system. Here, checkpointing (CP) is used to implement fault tolerance and process migration. We have developed an incremental CP solution for saving only memory pages that have been changed since the last CP. We present how we keep track of memory page modifications between CPs using Linux-native radix trees and how we handle virtual memory area changes. We also discuss experiment results with selected examples. Finally, we present a custom memory event connector transparently reporting page write-protection fault of processes to a user mode grid service to adaptively control incremental CP at grid level.

## 1 Introduction

Secure and efficient resource sharing between institutes and companies is increasingly required by research, engineering and industry. Both is provided by grid technologies implementing distributed computing platforms e.g. middleware-approaches like Globus [4] or grid-functionality integrated into native operating systems such as XtreemOS [3].

Grid-inherent dynamicity as well as the unpredictable application behaviour require to transparently move applications from heavily-loaded or close-to-fail grid nodes to idle and healthy ones. Coping with grid fault tolerance is an ongoing research topic. Our approach is based on using existing kernel checkpointers, including the latest Linux checkpointer. In this paper we focus on developing incremental checkpointing in Linux to speed-up checkpoint time. Once, a grid service is able to switch between full and incremental checkpointing, based on monitoring information provided by the kernel, the best-suited strategy can be applied.

Recent Linux innovations allow applications, running on one node, to be isolated from each other regarding resources such as cpus, pids, network, ipcs, file system, etc. Container concept [12] implementations such as cgroups [1] allow the same resource identifier to be used by multiple applications residing in a separate cgroup container. The container concept pushes server consolidation. Furthermore, it is significant for fault tolerance to avoid potential resource conflicts at restart. A checkpointing mechanism is about to be developed and can be used for process migration and fault tolerance. Linux container and checkpointer implementations are required by grid computing in order to support load balancing and fault tolerance.

This paper is structured as follows: Section 2 provides a short overview of the XtreemOS project, Section 3 gives a detailed insight into our concepts and implementation of incremental checkpointing in Linux including a short overview of related work, Section 4 presents evaluation results followed by conclusions and outlook.

## 2 XtreemOS

The EU funded project XtreemOS aims at providing a Linux-based open source Grid operating system coming in three flavours for: single PCs, Single System Images (SSI) cluster and mobile devices. Fundamental XtreemOS functionalities include native Virtual Organisation (VO) support and fault tolerance that have been implemented by extending the native operating system.

Basically, applications can transparently exploit resources distributed in the grid spread across administrative domains. Besides state-of-the-art grid applications, legacy applications can be run in the grid unmodified by relying on the POSIX interface provided by XtreemOS. Developers can easily create new grid applications, using the functionality provided by distributed grid services through the XOSAGA API providing access to security, resource and process manage-

ment. Users are organised in VOs, VO policies can be created to provide fine grained resource access control. Furthermore, XtreemOS comes with a grid file system called XtreemFS [10], allowing for location transparent file access, file replication and file striping.

XtreemOS ships with an integrated grid checkpointing mechanism. The components of the XtreemOS grid checkpointing architecture [13] are shown in Figure 1. The main concept is to rely on existing kernel checkpointer packages, e.g. Berkeley Labs Checkpoint Restart (BLCR) [8], OpenVZ [11], the Linux-native checkpointer for a single PC [1] and the LinuxSSI checkpointer [14] for a SSI cluster.

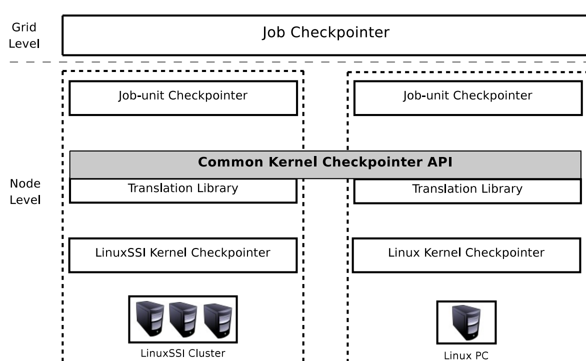



Figure 1: XtreemOS grid checkpointing architecture

The job checkpointer service at the top is responsible for checkpointing/restarting a job consisting of one or multiple job-units. A job-unit checkpointer focuses on saving and restoring a single job-unit. Therefore, it uniformly addresses an underlying kernel checkpointer using the so-called *common kernel checkpointer API*. This API is implemented per checkpointer in a separate translation library. It bridges semantic differences, e.g. each kernel checkpointer distinguishes from another one by an individual calling semantic. Furthermore, there are different process group types supported by the checkpointers. The translation library must check a process group precisely matches a job-unit's processes in order to enforce checkpoint/restart consistency. The library is responsible for providing a uniform interface to developers to transparently register checkpoint/restart callbacks. The API supports the retrieval of a matching kernel checkpointer for an application at job submission, because most kernel checkpointers are incapable of saving and restoring all possible kernel resources. Besides grid-to-kernel level semantic translations, the API also provides inter-kernel checkpointer translation regarding saving/restoring reliable channels by a generic channel flushing protocol.

## 3 Incremental Checkpointing in Linux

### 3.1 Memory page modification detection

Checkpointing overhead can be dramatically reduced by saving only content that has changed since the previous checkpoint. The major challenge here is to *detect* these content modifications. Generally, there are page-based and variable based approaches.

Detection of content changes at variable-granularity-level is described in [7] where a compiler is manually modified to detect variable changes. Thus, the compiler is enabled to insert incremental state saving calls before a variable is changed. In [15] detection of changes variables is enabled without manual intervention, but via an executable editing library. Furthermore, special memory hardware exists that incrementally state saves contained variables [5].

Detection of modified pages can be realised by taking existing page table entry bits, namely the *dirty bit* or the *write bit*, into account.

The *dirty bit* is of high importance for the Linux internal memory management components, especially for the Page Cache. A set *dirty bit* indicates to synchronize cache contained pages with those versions on disk and the swap partition. Just reading the *dirty bit* does not always indicate changed content. After modified cache contained data are written back to disk, the bit is reset, thus, a past modification is not visible anymore. Bookkeeping of modified pages includes resetting the dirty bit to detect new modifications after a taken snapshot, which is dangerous since it affects Page Cache consistency. In [6] page modification detection is done based on the *dirty bit* being mirrored into one of the reserved entry bits.

Our approach to detect modified pages is *write bit* focused. Each time a write-protection page-fault occurs, the *write bit* is set. Such exceptions are detected by the memory management unit. An exception handler is called and resolves the exception by removing the write-protection (*write bit* set to 1). Based on the detection we record modified pages in a bookkeeping control structure. At the initial checkpoint all pages of a program address space are saved. After each checkpointing

operation all writable pages are explicitly made write-protected (*write bit* is reset to 0). In case an application attempts to write on such a page within a checkpoint interval, the triggered page-fault handler removes the write-protection. Thus, detection of modified pages is enabled during the next checkpointing operation. Depending on the application behaviour, generally just a subset of all application pages needs to be saved. In order to detect future write attempts in subsequent checkpoint intervals, the write-protection will be reactivated by explicitly resetting the page *write bit*. Special handling of newly added read-only pages and partially written pages after a checkpointing operation is detailed under Section 3.3.

During restart the last version of a physical page needs to be localized out of multiple checkpoint images. Therefore, a dedicated bookkeeping control structure is required that keeps track of the last page's file location and offset within the checkpoint image file, described in the following section.

We are conscious about TLB entries, which must be flushed explicitly after each incremental checkpoint, since they are not updated with our *write bit* modification. The latter could result in inconsistency. However, each process context switch anyway results in flushing the TLB. Taking multiple checkpoints within one scheduler time slice is hard to achieve due to its shortness.

### 3.2 Bookkeeping of modified pages

Physical page content of an application address space may be spread across multiple incremental checkpoint images each potentially storing all or just a subset of all pages. At restart the complete content and its consistent versions must be loaded.

We use a bookkeeping control structure that keeps track of page locations and is based on the Linux-native *radix tree*. The *radix tree* provides fast lookup and insertion operations (O(1)) which are needed to keep the structure in sync with the process memory structure. In Figure 2 the localisation of a virtual address-related physical page is shown.

Each tree node is identified by a virtual address. A node entry stores the version of a dedicated incremental checkpoint image file (e.g., pages_5.bin for the fifth incremental checkpoint) containing the latest version of

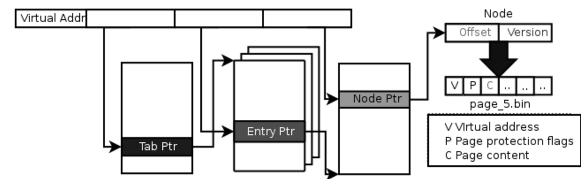



Figure 2: Bookkeeping control structure for modified pages

the page, and the offset in this file, since more than one page may be modified between two incremental checkpoints. The incremental checkpoint image file stores data blocks each containing a virtual address, page protection flags and the page content itself. Of course, all node entries are also saved to disk at each incremental checkpoint.

During a checkpointing operation the bookkeeping control structure is updated. That means, bookkeeping entries targeting not yet referenced pages are added, file locations and offsets of pages that are *present and modified and already referenced* are being updated and saved to disk.

At restart the bookkeeping control structure is read from disk. Its data is used to localize memory pages out of multiple incremental checkpoint files to restore a process' address space.

The mere write-bit based page modification approach alone is not able to keep the bookkeeping control structure in sync with the process memory structure, especially if memory pages have been removed. A requirement is to delete such structure entries to avoid wasting memory. Reading from and writing structure content to disk decreases checkpointing performance.Another issue is the unawareness of newly mapped read-only data that cannot be detected and will lead to inconsistency at restart because the appropriate page content is missing. The solution for both cases is detailed in the next section.

### 3.3 Challenge: memory region changes

Virtual pages belong to a bigger logical unit called memory region or virtual memory area (VMA). Memory regions can be seen as an overlay structure of continuous virtual pages. At one time one virtual address belongs to exactly on memory region. Over time one virtual address may be reassigned to a new memory region. They

are implicitly created from user space (e.g. *mmap* system call) and are created/managed by internal memory management mechanisms.

*Memory region changes in connection with read-only and writable pages must be taken into account when managing the book keeping control structure.* Two criteria must be fulfilled, proper assignment of pages to memory regions, which can be influenced by dynamic region creation/removal, and clean management of bookkeeping control structure entries, outdated entries must be deleted. If the later contains inappropriate content, a restart may fail or result in inconsistency. According to Linux memory region management [2] four cases of memory region changes can occur:

**Rule 1 (region extension):** if a new range of addresses is to be added to a process, the kernel first tries to enlarge an existing memory region. This requires virtual address holes or free address blocks in the process' address space and access rights of the existing region and the additional addresses being equal.

**Rule 2 (region creation):** if a new memory region is created and attached to the process' address space, the kernel tries to merge neighbouring regions, as far as they share the same access rights.

**Rule 3 (region shortening):** a certain address block can be removed from a region. If this address block resides at the beginning or end of a region, the region is shortened.

**Rule 4 (region splitting):** if the address block to be removed resides within an existing region, the region is split into two smaller regions.

The following examples demonstrate the need for an additional criteria than only checking the *write bit* in order to detect memory region changes, and thus page content changes.

**Case 1:** An application maps file A in a separate memory region 2. The application gets checkpointed, afterwards file A gets unmapped, memory region 2 vanishes. The application maps file B, accidentally having same size and using the virtual address block of former region 2. A new memory region 2 will be created.

In case file B has been mapped as read-only a new incremental checkpoint does not include the new memory region 2 content, since no *write bit* has been set (see

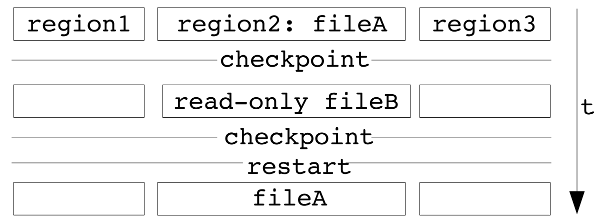



Figure 3: Region2 content with fileB has never been saved, restore old content of fileA

Figure 3). At restart memory region 2 will be recreated containing file A (old memory region 2) content which is wrong.

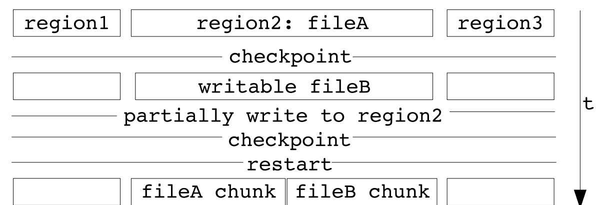



Figure 4: fileB was partially written to, restore mix of old and new content

In case file B has been mapped as writable and if it has been partially written to, a new incremental checkpoint results in saving just the pages of region 2 with the *write bit* being set (see Figure 4). After restart memory region 2 represents a mixture of file A and file B content which is wrong.

**Case 2:** this scenario is similar to the first one of case 1. However, a smaller file B is mapped, and thus a smaller memory region 2 will be created, resulting in a hole of the virtual addresses between new region 2 and region 3.

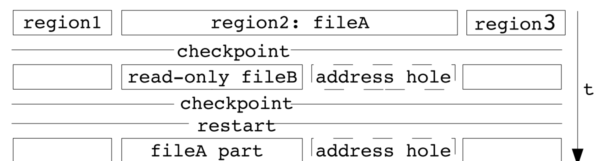



Figure 5: Region2 content with fileB has never been saved, restore part of fileA

In case file B is mapped read-only, no content of region 2 will be saved because no *write bit* is set (see Figure 5). The reduction of virtual addresses of new memory region 2 is not reflected in the bookkeeping control structure. At restart parts of old memory region 2 will be recreated in the new address range of region

2. These bookkeeping control structure contains more entries than supposed to be which may result in wasted memory space.

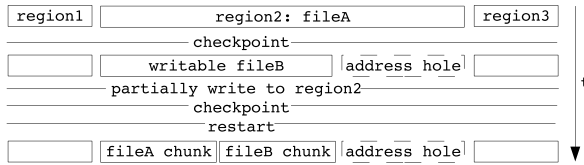



Figure 6: FileB was partially written to, restore mix of old and new content

In case file B has been mapped as writable, and if it has been partially written to, a mixture of old and new region 2 content will be reestablished at restart (see Figure 6). Furthermore, the bookkeeping contains out-of-date data, since it does not reflect a memory region shrinkage.

**Case 3:** three regions exist, a memory hole exists between region 2 and three. At checkpoint time the complete content of all regions is saved. Afterwards, region 2, which maps file A, gets unmapped, a new file B, which is bigger than the previous file A is mapped. New region 2 is placed between region 1 and 3, no memory hole between 1 and 2, as well as 2 and 3 exists.

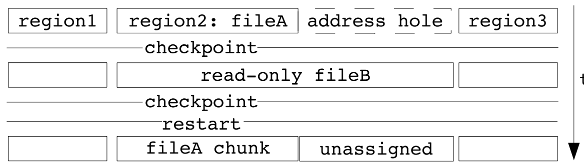



Figure 7: Region2 content with fileB has never been saved, restore fileA and unassigned space

In case file B is mapped read-only, no new region 2 related content is saved at an incremental checkpoint. Especially the additional virtual addresses of new region 2 opposite to old region 2, are not taken into account, since no *write bit* is set. At restart, region 2 contains file A (old region 2) content. Since the bookkeeping control structure is not aware of additional virtual addresses of new region 2 restart is likely to fail or causes inconsistency.

In case file B is mapped writable, and if it has been partially written to, a mixture of old and new content will be reestablished at restart for the address block covered by old region 2. Regarding the additional addresses of

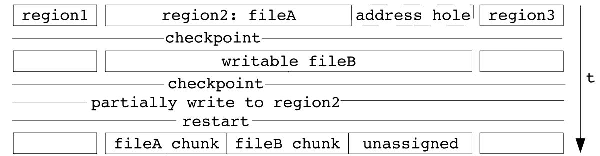



Figure 8: fileB was partially written to, restore mix of old and new content and unassigned space

new region 2, the same effects are expected as explained shortly before.

**Case 4:** in the center of memory region 1 an address block is being write-protected having different access rights than the surrounding region 1 parts. Consequently, the Linux memory management enforces region 1 to be *split* into three parts. Region 1, 2 and 3, with region 2 containing the pages the *mprotect* call has been applied to.

In case the write-protected region 2 is not written to, or is partially written to, the same effects as described under Case 1 occur.

**Case 5:** region 1 contains an address block at the end or at the beginning which gets removed. The region got *shortened*. In case region 2 is read-only, the appropriate bookkeeping control structure entries of the removed address block are not removed. Then, a new region gets created partially or fully covering the previously removed address block.

In case the new region is read-only, or has been partially written to, effects as detailed under Case 1 occur.

### 3.4 Solution: Memory region modification monitor

The special cases mentioned under Section 3.3 occur in combination of memory region modifications with read-only and writable content, e.g. such as shared segments, or anonymous memory region content or memory mapped files.

To tackle these issues we introduce an additional logical layer of modification detection—a memory region modification monitor. This monitor keeps track of memory region changes and thus complements the mere write-bit focused approach of memory page modification detection. Based on monitor data the bookkeeping control structure can be kept in sync with the actual memory structure of a process at checkpoint time. It is sufficient

to update the corresponding book keeping structure entries once, at checkpoint time, instead at each region modification event.

The monitor records region removals and additions. After each checkpoint, monitor data will be flushed. At checkpoint time the monitor entries are used to manage the bookkeeping control structure. Its entries are compared to control structures entries. In case a region has been removed, the start and end address of each monitor entry is used to delete appropriate bookkeeping control structure entries. This ensures control structure efficiency and consistency. In case a region has been added, relevant bookkeeping control structure entries are added and/or updated to reference appropriate checkpoint image contained pages. This allows whole new regions to be saved initially at the first incremental checkpoint after their creation.

Our monitor supports detection of *mmap* and *munmap* calls. Therefore, we insert a monitor notification function into the kernel functions *do_mmap* and *do_munmap*. Per *mmap* call the start and end address of an affected memory region are inserted into the memory region modification monitor. A detected *munmap* call results in deleting the appropriate entry of the monitor.

For example, region creation detection, via the *mmap* call, results in initially saving the whole physical page content at the next checkpoint. Issues as listed under Section 3.3 can be avoided.

The memory region modification monitor has a similar structure as the bookkeeping structure.

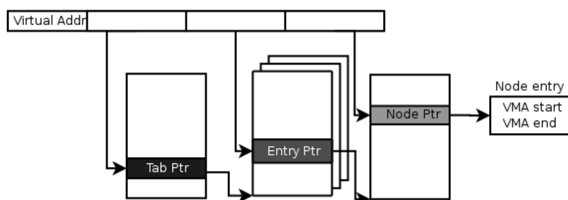



Figure 9: Memory region monitor structure

Its entries are organised in a radix tree providing fast access for entry removal, addition and retrieval. Each entry contains the memory regions start and end address of the covered virtual address range. The structure is shown in Figure 9.

## 4 Incremental grid checkpointing

In order to realise one flavour of adaptive checkpointing, our incremental checkpointing enhanced kernel checkpointer has been integrated into the XtreemOS grid checkpointing architecture.

For the job checkpointer service to know when it is best to use a full or incremental checkpointing, the number of modified pages of an application must be computed. Therefore, the job checkpointer service has been enabled to detect page modifications in a transparent manner for a given set of processes. Without modifying applications, the service can self-decide which checkpoint approach to be used by keeping efficiency.

Reporting page modifications from the kernel space to the user space domain has been achieved by setting up a new Linux Connector [9]. The service registers at a so-called *memory event connector (MEC)* at kernel-level to be informed about *do_page_fault* calls triggered by selected processes. The service receives MEC messages at user space and performs accounting on page faults on a per process base. In case the collected data exceed a certain threshold, the job checkpointing service enforces full checkpointing. Otherwise, incremental checkpointing is used.

## 5 Measurements

The testbed consists of 2 nodes with Intel Core 2 Duo E6850 processors (3 GHz) with 2048 MB DDR2-RAM and being interconnected with gigabit network. A master node runs a tftpboot and a NFS server providing a LinuxSSI image and a Linux environment including the directory for checkpoint image storage to a client node.

Our test application allocates 1 MB of RAM and writes integer values to random locations in 1 millisecond intervals.

Figure 10 shows the checkpoint duration of full and incremental checkpointing if checkpoints are issued in one second intervals.

Both data sets indicate incremental checkpointing taking shorter time especially after the initial checkpoint. Figure 11 shows the resulting image size of full and incremental checkpoints. It appears that one incremental checkpoint image file is smaller than a full checkpoint image, especially after the initial checkpoint. Efficiency of incremental checkpointing relies on the write

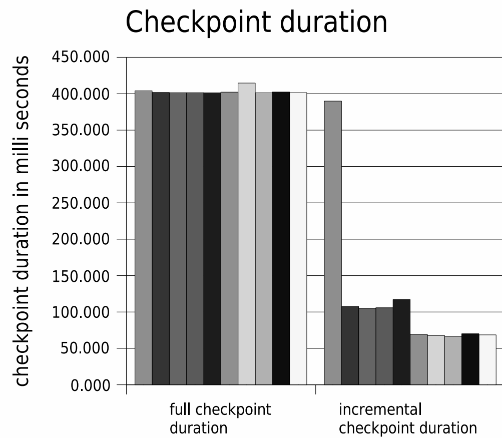



Figure 10: Full and incremental checkpointing duration

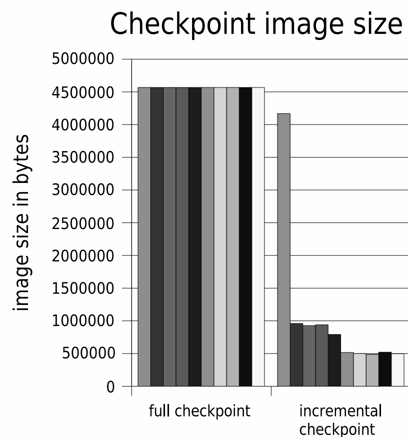



Figure 11: Image size of full and incremental checkpointing

behaviour of an application. Since an additional control structure and a region monitor need to be maintained, incremental checkpointing may become inappropriate, especially the more pages have been changed per checkpoint interval. It is the task of the grid service to figure out the best-suited strategy.

Furthermore, restarting from an incremental checkpoint may result in accessing more than one image file opposite to just one file with full checkpointing. Increased I/O overhead, caused by reading from multiple files, is likely decrease restart performance.

## 6 Conclusion and Outlook

We described the integration of incremental checkpointing into a Linux-based kernel checkpointer. Our basic approach to detect modified pages, which is the most significant prerequisite to be met for incremental checkpointing, is *write bit* based. We use the *write bit* of pages to detect whether pages have been modified and thus need to be saved. Modification of the *write bit* does not interfere with other Linux memory management functionality, e.g. the Page Cache.

Furthermore, we classified five generic and common scenarios of Linux memory region behaviour, where page content modifications cannot be detected with a mere *write bit* focused approach. Basically, combinations of memory region creation and removal in connection with read-only and writable content can lead to inconsistency and failure at restart.

We implemented a memory region modification monitor that takes region changes into account in order to save appropriate content at each incremental checkpoint and avoid the scenarios described under Section 3.3.

Efficient Linux native structures, such as a radix tree, have been used to implement a page-modification bookkeeping control structure and the memory region modification monitor enabling fast management of incremental checkpointing-specific data at checkpoint and restart.

We profit from recent innovations, namely the generic connector concept. A custom memory event connector (MEC) informs a user-space component of page modifications, which improves the symbiosis of kernel- and grid-level checkpointing components.

We are conscious of further events to be monitored regarding region resizings, e.g. caused by *do_mremap*. Besides, we will focus on saving pages contained in the swap area as well. Additionally, we plan to realise concurrent checkpointing to provide more kernel-based functionalities that support adaptive checkpointing at grid level.

## References

[1] Sukadev Bhattiprolu, Eric W. Biederman, Serge Hallyn, and Daniel Lezcano. Virtual servers and checkpoint/restart in mainstream linux. *SIGOPS Oper. Syst. Rev.*, 42(5):104–113, 2008.

[2] D. Bovet and M. Cesati. *Understanding the Linux Kernel, Third Edition*. O'Reilly, 2006.

[3] XtreemOS Consortium. Annex 1 - description of work. Contract funded by the European Commission, April 2006. XtreemOS Integrated Project, IST-033576.

[4] I. Foster and C. Kesselman. The globus project: a status report. *Future Gener. Comput. Syst.*, 15(5-6):607–621, 1999.

[5] R. M. Fujimoto, J.-J. Tsai, and G. Gopalakrishnan. Design and performance of special purpose hardware for time warp. In *ISCA '88: Proceedings of the 15th Annual International Symposium on Computer architecture*, pages 401–409, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[6] Roberto Gioiosa, Jose Carlos Sancho, Song Jiang, and Fabrizio Petrini. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 9, Washington, DC, USA, 2005. IEEE Computer Society.

[7] F. Gomes and A. F. Bosco. *Optimizing incremental state-saving and restoration*. PhD thesis, University of Calgary, Calgary, Alta., Canada, Canada, 1996. A.-U. Brian.

[8] P. H. Hargrove and J. C. Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *In Proceedings of SciDAC 2006*, June 2006.

[9] M. Helsey. Process event connector. 2005.

[10] F. Hupfeld, T. Cortes, B. Kolbeck, E. Focht, M. Hess, J. Malo, J. Marti, J. Stender, and E. Cesario. Xtreemfs - a case for object-based file systems in grids. *Concurrency and Computation: Practice and Experience*, 20(8), 2008.

[11] K. Kolyshkin. Virtualisation in linux. 2006.

[12] D. Lezcano. lxc. 2008.

[13] J. Mehnert-Spahn, T. Ropars, M. schoettner, and C. Morin. Xtreemos grid checkpointing service architecture. 2008.

[14] J. Mehnert-Spahn and M. Schoettner. Design and implementation of basic checkpoint/restart mechanisms in linuxssi d2.2.3. 2007.

[15] Darrin West and Kiran Panesar. Automatic incremental state saving. In *Proc. 10th Workshop on Parallel and Distributed Simulation (PADS 96*, pages 78–85, 1996.

# Putting LTP to test—Validating both the Linux kernel and Test-cases

Subrata Modak
*Linux Technology Center, IBM INDIA,*
subrata@linux.vnet.ibm.com

Balbir Singh
*Linux Technology Center, IBM INDIA,*
balbir@linux.vnet.ibm.com

Masatake YAMATO
*Red Hat Inc.,*
yamato@redhat.com

## Abstract

The **Linux Test Project** (LTP)[5] is receiving renewed interest, and attention due to increased focus on testing, and integration of Linux components from several projects in the Linux Ecosystem. LTP has not only discovered bugs in the Linux kernel, but also inconsistencies between other components such as libraries, the man pages, and the kernel.

In this paper, we will cover our experience in this area and delve into the details of benefits being brought to LTP because of closer interaction with the Linux ecosystem. We will also discuss the adoption of newer technologies for static and dynamic analysis of existing test cases, and show that we can use this approach to reduce any errors in test cases (leading to better end automation of Linux testing). We will also analyze the new LTP code using various test metrics, and look at the requirements for allowing the test cases to handle errors introduced by Fault Injection. We finally propose integration of all such technologies to LTP infrastructure.

## 1 Introduction

The surge in growth of LTP[7] in the last couple of years has also brought along with it a host of issues that needs to be addressed immediately. While numerous patches flow in to create new test cases for the ever-expanding new features of the kernel, as well as fixing the existing ones to adapt to the enhancements of existing kernel features; one immediate and dire problem that has cropped up is to validate the quality of the test cases themselves—those of which will in turn validate the quality of the kernel code.

While the kernel code can be validated through the discovery of new bugs, but, the very question of the test case quality needs to be found and answered. Contrary to the belief that the effectiveness of a test case is limited to only finding bugs, *the importance of the test case is repeatedly established when it continuously proves the stability of the same code*. However, doubts on the quality of test cases are often raised when they fail to expose bugs for too long. There can be different reasons for this. One—the code that it is supposed to test has stabilized enough, and until somebody changes something dramatically; regression will not be discovered soon. In this scenario, although nothing can be done to the test case itself, it still retains the ability to find any regressions in future.

Another reason could be that the test cases themselves are not written as good programs. They have loopholes in the way of becoming good quality code. This is something which can be addressed effectively by analyzing the test code through various static and dynamic analysis tools available to the Open Source Community. In this paper we discuss many of them. Major issues found would be highlighted, along with the remedy to make LTP a much better project.

## 2 Benefits of The Linux Ecosystem

The biggest common misunderstanding about LTP may be that the relationship between LTP and other projects is one-way; LTP just tests the Linux kernel. Another misunderstanding is that the test cases in LTP can be written easily based on man pages as specification. One of the authors (YAMATO) also believed the same, before joining LTP. In reality the relationship between LTP and other projects is complimentary. This fact came to his cognizance gradually through various stages, and surprises while contributing to LTP. He has already ex-

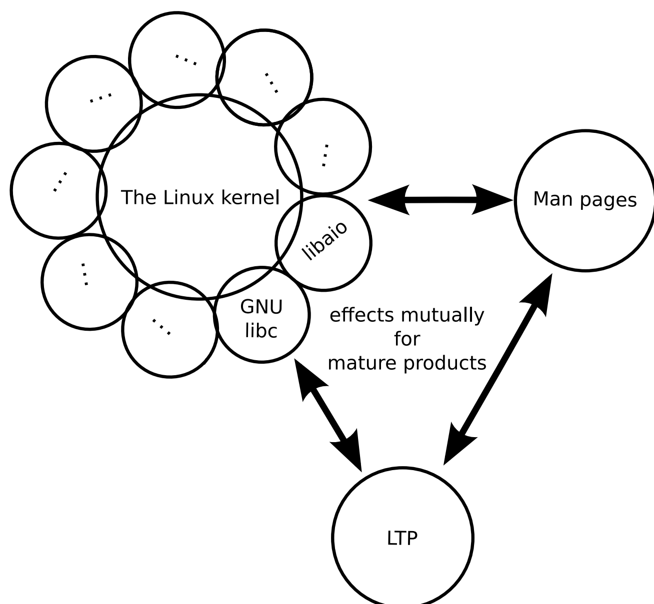



Figure 1: Linux ecosystem around test

pressed the extent of such relationship at ***Linux Ecosystem Around Test***[14].

Figure 1 shows the concept of ***The Linux Ecosystem.*** The projects in the ecosystem contribute mutually. Development efforts and/or by-products in a project are reused in other projects of the ecosystem.

### 2.1 Lessons learned while working on LTP

In the majority of circumstances, test cases were written based on man pages. In most instances they report **SUCCESS**. This fact leads to misunderstanding among test developers. The real worry starts only when a test case reports **FAILURE**. When FAILURE is reported for the first time, a programing error in the test case is suspected. The test case developer wonders misreading man page, and/or misusing the helper libraries of LTP. The test case **FAILURE** generates a series of investigation to find out the root cause of such a failure. And the hunt for the flaw in any component of the ecosystem starts only then.

#### 2.1.1 Suspecting run time bug

If the error could not be found in the test case, the developer moves to the next stage. Armed with the only understanding that LTP tests just the Linux kernel, the developer suspects bug in the Linux kernel. At this point one may choose reporting the error to **Linux Kernel Mailing List**(LKML)[4] or examining the kernel source code. But this understanding alone is not enough. C language level wrapper exists between the test case and the kernel. For many system calls theses wrappers are implemented as really thin layer, and are part of GNU libc[1]. But, wrappers for some system calls do some other work, and belong to libraries other than GNU libc. These library wrapping system calls are called system libraries. Therefore before suspecting and examining Linux kernel, one must also suspect and examine the system library related to the test cases. Here, both the Linux kernel, and the system libraries as a whole are called run time entities.

The example in this section is based on author's working on a test case for `posix_fadvise()` system call[8].

When the author was working on it, the test case calls the system call with a wrong argument, and expects `EINVAL` error. The test case ran successfully on the author's PC but failed on the system used by the *bug reporter*[12].

The difference of test result came from the build configuration of kernel; The function `sys_fadvise64_64` was implemented for the system call and treated `CONFIG_EXT2_FS_XIP` configuration in wrong way[15]. `CONFIG_EXT2_FS_XIP` was *turned off* in the kernel running on the author's PC but might have been *turned on* in the kernel running on the system used by the bug reporter.[1]

Only few lines were needed to fix the bug by changing the `KERNEL_CONFIG` option before building the desired kernel, however the inspection had taken rather long time. The author had inspected from upper layer: GNU libc first, and then the Linux kernel. Historically `posix_fadvise` has two variants; `posix_fadvise64` and `posix_fadvise64_64`. Their implementation look complex because many `#ifdefs` were used to share the code for the variants. Therefore the author had suspected a programming error in those `#ifdefs` causing the bug, and hence the delay.

[1] The author found "Machine Architecture: armv6l" in the bug report.

#### 2.1.2 Suspecting man pages

With expectation that a test case for a system call can be easily based on the man pages, one may easily suspect only the run time environment, kernel and system libraries. But one would rarely suspect the man page. The activity defining the specification of new system call, and the code implementing it are not separate. In other words the system call is not implemented after its man page is written. The maintainer of man pages joins the testing[13] of the implementation of newer system call, and writes the man page only after that. Like the run time, the man pages must also be suspected.

The previous paragraph has also a real example. When the author reviewed a test case for `io_cancel` system call written by his colleague, he found strange code in it. The test case expects an error number to return when invalid arguments are passed to `io_cancel`. The strange thing was that the returned value from `io_cancel` was compared with a negative number:

```
io_cancel(ctx,NULL,NULL)==-EFAULT
```

In kernel space, negative integer is used to represent an error. But in user space, generally positive integer is used to represent an error. The test case ran successfully, but man page said the system call is expected to return positive integer when an error occurs.

There is an inconsistency between the run time and the man page for `io_cancel`. While inspecting this issue the author found (1) that the C language wrapper for `io_cancel` is not part of GNU libc, but, is part of `libaio` library. And (2) returning negative integer to represent an error is libaio's own convention. The author sent a bug report for the man pages to the man pages maintainer. As the result, the latest man pages for system calls wrapped by libaio have following NOTES:

> "GNU libc does not provide a wrapper function for this system call. The wrapper provided in libaio for `io_cancel` does not follow the usual C library conventions for indicating error: on error it returns a negated error number."

Simply to say, LTP developer has to suspect everything when test case reports **FAILURE** till it is fixed. The fix may go anywhere. And when it happens, it validates LTP's contribution to ***The Linux Ecosystem***. Good software in the **"Open Source World"** is the outcome of greater collaboration among various OSS projects. It is sometimes true for LTP that the test cases themselves are the outcome. Different from other projects, the development process itself is another outcome: finding inconsistency between run time and the man pages being a prime factor.

### 2.2 Role Reversal (From other projects to LTP)

The test cases for `utimensat` system call in LTP was submitted by the man pages maintainer[13]. The man pages maintainer participates in kernel development by testing and verifying newer kernel code, before the code is shipped as part of official release version of kernel. He reflects such experience when he writes man pages for them. The test code is mainly utilized three times: once for stabilizing the easy development of kernel code, another time for verifying the description of man pages, and last time for LTP. As part of LTP, the test case is run again and again on many different environments.

These days some test cases are written by kernel developers and system library developers; the original authors themselves are the test targets. The number of such test cases have increased. Some of them are imported to LTP by LTP maintainer. Some of them are directly contributed to LTP by the original authors. This is really a good trend. To write test cases, one has to understand the test target. The cost to understand the test target is the most expensive stage to write a test case. Such cost can be reduced if the original author of the test target writes test case for it.

On the other hand, writing test cases by the third person is extremely valuable; such person can have very different view to the test targets from the original authors. And this may be the strongest reason why LTP exists as a project independent from the Linux kernel, system libraries and man pages.

## 3 Dynamic Analysis

In the above 2 sections we explained the relationship between LTP and other projects; and introduced the proposal that we can fix bugs in the run time(the Linux kernel and support libraries) and the specification(man pages) through developing LTP test cases. However,

such kind of bug fixing can be done because LTP takes efforts to make test cases of higher quality. Any action to other projects/proposals starts from reliable test cases. Here after we will describe efforts on how we can bring more reliability to our existing test cases, by analyzing them using run time analysis tools and via static analysis tools.

## 3.1 Memory Leak Analysis

While there has been numerous issues reported during testing/analyzing through ***Valgrind's memcheck*** tool[11], here we look into some of the most significant ones. The following generated error clearly demonstrates that the program has been handling memory allocation/deallocation incorrectly. There has been many such instances found inside LTP, a reflection that all of them needs to be fixed to better handling of memory usage during program execution:

### 3.1.1 Pure Leakage Issues

A memory leak detection on the hackbench[2] tool shows

```
valgrind --leak-check=full
--show-reachable=yes
./hackbench 20 process 1000
LEAK SUMMARY:
definitely lost: 9,840 bytes in 420 blocks.
possibly lost: 0 bytes in 0 blocks.
still reachable: 3,200 bytes in
1 blocks. suppressed:
0 bytes in 0 blocks.
```

The problem *could* be addressed as shown below:

```
if(p) { free(p); (p)=NULL; }
```

at the end of every memory usage, or, before program exit would help solve such problems. And, it indeed helped. Following is the analysis post fix:

```
ERROR SUMMARY: 0 errors from 0 contexts
(suppressed: 3 from 1)
malloc/free: in use at exit:
0 bytes in 0 blocks.
malloc/free:
423 allocs, 423 frees,
14,720 bytes allocated.
All heap blocks were freed
-- no leaks are possible.
```

[2]Hackbench is a part of the LTP test suite.

### 3.1.2 Invalid Read Instances

```
Invalid read of size 8 at 0x401428:
main (clone07.c:141)
Address 0xfffffffffffffff8
is not stack'd, malloc'd or (recently) free'd.
More than 10000000 total errors detected.
I'm not reporting any more.
```

The example above illustrates the example of a test case trying to reference memory, it never owned as the message shows. The run-time has been tolerant of these errors, but they are a sign of code that needs to be revisited and needs our attention.

### 3.1.3 Jump to Invalid Addresses

```
Jump to the invalid address stated on
the next line at 0xFFFFFFFFFF600800: ???
by 0x401650: main (getcpu01.c:125)
Address 0xffffffffff600800 is not stack'd,
malloc'd or (recently) free'd

Process terminating with default action
of signal 11 (SIGSEGV)
Bad permissions for mapped region at
address 0xFFFFFFFFFF600800
at 0xFFFFFFFFFF600800: ???
by 0x401650:
main (getcpu01.c:125)
```

## 3.2 Checking Race Conditions

There are two types of tests in LTP which required to be investigated on this point. On one hand the thread tests needed analysis to see if proper synchronization exists between them, and on the other side we also wanted to see if multiple instances of the same test program(which has a single thread of execution) do not fall into the trap of deadlocks, or, race conditions over commonly accessed system resources.

### 3.2.1 Tests Creating Multiple Threads

This threw up some more surprises. Most of the tests falling under this category revealed that none of them has proper locking code between threads. And following output is common for all of them when analyzed through ***valgrind's helgrind***:

```
Possible data race during
write of size 8 at 0x421E508
Location 0x421E508 has never
been protected by any lock
```

Dynamic analysis shows that there is a potential race condition in the test. Submitting test results based on this test case would require further scrutiny and introspection. There is an urgent need to fix all these issues.

#### 3.2.2 Concurrent Test Execution

Users of LTP reported that many tests inside LTP are not concurrency safe. They pointed out to various such issues and fixes for them were also proposed. We point to few of those.

- **Reserving same Port**
  The following code:

  ```
  sin1.sin_port
   = htons(((getpid() * TST_TOTAL) % 32768)\
   + 11000 + count);
  ```

  is safer than the below one:

  ```
  sin1.sin_port
   = htons((getpid() % 32768) +\
   11000 + count);
  ```

  if more than one process is trying to bind to the same port simultaneously, then the following error can be avoided:

  ```
  sendfile02 2 BROK : call to bind() failed:
   Address already in use
  sendfile02 3 BROK : Remaining cases broken
  ```

- **Generating Keys for Shared Memory Segments**

  The following comment and code snippet addresses concerns in concurrent processes who are trying to create shared memory segments concurrently.

  */* Get a new IPC resource key. Since there is a small chance the getipckey() function returns the same key as the previous one, loop until we have a different key */*

  ```
  do {
   shmkey2 = getipckey();
  }while (shmkey2 == shmkey);
  ```

- **Using sleep() Family to Synchronize**

  Many tests use sleep() family of functions to synchronize between the parent and the child, with the hope that after a specified period of time, one will be able to have a clean access to the resource with the basic assumption that the other has already accessed it. But, this is a common programming error, and many of our old tests are victims to it. It has already been proved that such mechanism is faulty and do not provide foolproof mechanism.

  Many such instances were discovered and fixed using the mechanism of pipes to establish proper communication between parent and child and then going ahead with the desired operation.

- **The ever-famous Reader/Writer problem**

  Since most of these tests were not initially designed keeping concurrent execution in mind, they suffer from this usual design drawback. In one such instance we found test cases seem to fail when multiple instances are run concurrently. The failures occur because the file(they are trying to access) sizes don't match, or, because the number of bytes read don't match the file size. This can be attributed to one parallel instance reading a file before the other instance's write to it has completed. In such situations, either the file size has not been updated in the inode header, or, the file size has been updated, but, the file's write operation has not been updated completely. To fix this concurrency problem, we agreed to check for an existing instance and wait for it to finish before starting another instance. Any other concurrency resolution technique would complicate matters further. A message to the console indicating such a decision in scheduling policy can clarify matters cleanly.

### 3.3 Avoiding Segmentation faults

Certain sections of code try to access memory beyond their scope resulting in segfaults. Proper memory bounds checking before accessing/de-referencing memory will help to avoid such segmentation faults during run time. We encountered some instances of segmentation fault with LTP's provisioning engine *ltp-pan.c*. The following instance of code creates segmentation fault if `coll` is not initialized properly. De-referencing creates the problem further:

```
coll = get_collection(
 filename, optind, argc, argv);
if (coll->cnt == 0) {
```

A properly written code with checks and balances removes such faults:

```
coll = get_collection(
 filename, optind, argc, argv);
if(!coll) exit(1);
if (coll->cnt == 0)
```

### 3.4 Proper Exit Code

Many tests were written without proper exit code. Zeroing on all of them and fixing with appropriate return code is a big challenge given the volume of tests that exist in LTP today. Following is an excerpt of build warning generated during one such compilation:

```
hackbench.c: In function 'main':
hackbench.c:350: warning: control reaches end
 of non-void function
```

A simple `exit(RETURN_CODE)` would solve such issues and promote to better program development.

## 4 Static Analysis

The code that initially gave life to LTP is pretty old, and we were certain that we would hit issues that does not adhere to the latest ANSI C or good coding standards. Even if the code is to follow ANSI C Coding guidelines, still, we were faced with the dilemma of which coding pattern to follow. Being directly responsible to test the Linux kernel, we decided to go ahead with the prevalent standard in the Linux kernel community.

As a means to measure all the violations, we decided to check LTP's health with the most popular Open Source Static Analysis tools like the *SPARSE*[9] and *SPLINT*[10].

### 4.1 SPARSE

A single round of compilation through the code exhibited the anomalies in the program development. We would highlight few of them and probably say/decide how we can fix them.

#### 4.1.1 Non-ANSI definitions

Numerous instances of non-ANSI definitions for various identifiers like the functions/variables were found. For example, the following definition:

```
int dataasciichk(
 listofchars, buffer, bsize,
 offset, errmsg)
char *listofchars;
char *buffer;
int bsize;
int offset;
char **errmsg) {
```

should be replaced with:

```
int dataasciichk(
 char *listofchars,
 char *buffer,
 int bsize,
 int offset,
 char **errmsg) {
```

#### 4.1.2 Non-Static Symbol Declaration

This arose from situations where the functions and other identifiers were not defined as static although they were never used *outside* the contours of the concerned source files. The code:

```
int databinchk(...)
```

should be replaced with:

```
static int databinchk(...)
```

to avoid all such warnings. Given the volume of such messages thrown during compilation, we can definitely say that it is going to be a tough task to fix them all.

#### 4.1.3 Symbol '*XYZ*' re-declared with different type

In older style programming as prevalent code in LTP, the general style is to declare the function prototype at the beginning of the source code, use them in different places, and then finally the definition follows at the end of the source file. Though the compilers can handle *forward references* well, still ***Sparse*** complains about it, and directs you to combine the prototype declaration and definitions together before the symbols are being referred at any point in the program.

#### 4.1.4 Using plain integer as NULL pointer

In many places of our code, integers were directly used instead of referring them through appropriate pointers. The following code snippet:

```
sigprocmask(SIG_UNBLOCK, &newset, 0);
```

should be replaced with:

```
sigprocmask(SIG_UNBLOCK, &newset, NULL);
```

to avoid and fix such warnings.

#### 4.1.5 Uninitialized Identifiers

This is probably the most common type of warning generated by all compilers. The safest bet would be probably to initialize them with proper values, before the undesired bug starts creeping into your program.

#### 4.1.6 Missing type declaration for parameter '*P*'

We found some typical instances of code where a function prototype was just declared:

```
int mkname(char*, int, int);
```

But, when it came to defining that function, the type declarations for certain parameters were missing:

```
int mkname(name, me, idx)
register char   *name;
{
```

The declarations for *me* and *idx* are missing above.

#### 4.1.7 Incompatible types for operation

These types of errors/warnings are thrown when various data types are mixed up, or, they are not properly typecaste in their respective operations. The following piece of code tries to compare whether `void *` is less then an `integer`:

```
if ((shmptr = shmat (shmid, 0, 0)) < 0)
```

### 4.2 SPLINT Analysis

We also found few more static cases through the *SPLINT* tool. They are really interesting enough and showed us how important programming mistakes were made during test case coding. Though many of them are safe to be ignored, still the question remains whether we should just keep ignoring them for their nature being non-fatal. This actually would reflect the concept that we were not clear about when we designed the test, leave aside a proper way to write it.

#### 4.2.1 Return value ignored

These warnings were generated when certain sections of code were found using function calls without collecting the return value of it. The situation is inconsistent, as many other instances of code were seen collecting the function's value. The fundamental flaw is the ambiguity in designing and writing such function prototypes, when the author was not sure what to do with the function ? whether to make it return something, or, just execute a bunch of instructions.

#### 4.2.2 Result returned by function call is not used

If there was no need for the return value of a function, why was it collected in the first place? Moreover, if the purpose is just to execute a function without the need for a return value, then the prototype could have been well defined as `void`.

#### 4.2.3 Path with no return in function declared to return void *

Even there is something interesting. There is a path through a function declared to return a value (interestingly a `void *`) on which there is no return statement. This means the execution may fall through without returning a meaningful result to the caller.

#### 4.2.4 Format string parameter not compile-time constant

The following piece of code should have been written like this

```
fprintf (stdout,"%s %s\n",
 global_progname, VERSION);
```

rather than this

```
char *mesg = "%s %s\n";
 fprintf (stdout, mesg,
  global_progname, VERSION);
```

If format string parameter is not a constant at compile time, then, this can lead to *security vulnerabilities* because the arguments cannot be type checked during compile time.

#### 4.2.5 Possibility of buffer overflow

It is a commonly know fact that use of `sprintf()` has been deprecated, and/or advised to avoid. `snprintf()` is recommended instead as use of function `sprintf()` may lead to buffer overflows. However, our code base contains plenty of them and removing them would really turn out to be challenging.

#### 4.2.6 Suspected infinite loop

Observe the following code:

```
while (child_signal_counter < num_pgrps) {
 alarm(1);
if (debug_flag >= 2)
 printf("%d: Master\
 pausing for done (%d/%d)\n", mypid,\
 child_signal_counter, num_pgrps);
pause();
}
```

No value used in loop test `(child_signal_counter,num_pgrps)` is modified by test or loop body. Hence this appears to be an infinite loop. Nothing in the body of the loop, or, the loop test modifies the value of `child_signal_counter`. Perhaps the specification of a function called in the loop body is missing a modification. Probably the only way of coming out of this loop, and hence this program is to get a signal; as probably specified by `alarm(1)`.

#### 4.2.7 Function parameter values declared as manifest array

Though the following type of declaration is harmless

```
... compute_median (unsigned long
 values[MAX_ITERATIONS],
 unsigned long max_value);
```

as size constant is meaningless here. The size of the array is ignored in this context, since the array formal parameter is treated as a pointer. A more hassle free declaration could be just this

```
... compute_median (unsigned long
 values, unsigned long max_value);
```

## 5 Fault Injection Impact

The ability to alter the course of execution in the kernel through a fault induced path has long been known. The Linux kernel also have the necessary infrastructure to induce random faults in to the various parts of the kernel; thus forcing applications to expect an undesired behavior. The major advantage of using ***Fault Injection*** is to traverse those error paths of the kernel, which in normal circumstances (stable) would not have been touched.

The immediate fallout of such a scenario is an increase in the measurement of the code coverage of the kernel, as, it would guarantee to traverse the faulty path besides the actual execution path. The other advantage would directly go to the developers, who would like to test their kernel code under such varied scenarios.

Though, all these facts are well known, and has been proved by many projects in the Open Source Space, still, such an exercise has never been attempted by the LTP developers. However, even before we started to see the fall out of ***Kernel Fault Injection*** while executing LTP, we were sure that such an exercise will help us in two different ways:

- **Increase Kernel Code Coverage**[2]
- **Help Test Engineers to validate their test code under varied circumstances**

  While writing test cases for certain kernel functionalities, an engineer may test his test cases, by running it over:

  - Stable kernel, and
  - Fault Injected kernel:

    This would give him a bigger insight into his/her test behavior, and would in-fact help him to create a better test case/scenario description by uncovering bugs, if any, in his/her test code.

### 5.1 Experimenting with Fault Injection

We decided to use all the infrastructure provided in **linux-2.6.29** kernel[3], namely:

- `fail_io_timeout`

- `fail_make_request`
- `fail_page_alloc` &
- `failslab`

and use the following parameters of each of these infrastructure[3]:

- `probability`
- `interval`
- `times` &
- `space`

With `space` as `0`, `times` as -1 and `interval` greater than `1`, we varied the `probability` parameter for all the `fail*` subsystems. The following algorithm reflects the way the experiment was carried out:

```
start_code_coverage()
 loop (for each testcase)
  begin
   execute_testcase(inside_stable_kernel)
    begin
     insert_fault_into_kernel()
      loop X Times
       begin
        execute_testcase(inside_fault_kernel)
       end
     restore_kernel_to_normal()
    end
  end
end_code_coverage()
```

The results observed at varied *probability* values were amazing:

- **probability=100%**

  Our test provisioning engine never took off with `probability` value set at this level(100%). We knew that we cannot generate any useful data with such a system. We did not generate any code coverage data for this.

- **probability=30%**

  With `probability` value set to this level(30%), we indeed saw our tests running, but with some major flaws:

  – Failure of many tests

    Many tests failed which otherwise *pass* under normal circumstances. We traced the reasons for such failures owing to the fault in the kernel. A small snippet of `dmesg` output justified our observation. The following failure types

```
<<<test_output>>>
sh: /bin/mktemp: Cannot allocate memory
Usage:
mmapstress07 filename holesize e_pageskip
sparseoff
*holesize should be a multiple of
pagesize
*e_pageskip should be 1 always
*sparseoff should be a multiple of
pagesize
Example: mmapstress07 myfile 4096 1 8192
mmapstress07    1  FAIL  :  Test failed
mmapstress07    0  WARN  :  tst_rmdir():
   TESTDIR was NULL; no removal attempted
```

    were accompanied by `dmesg` entries like

```
FAULT_INJECTION: forcing a failure
Pid: 30589, comm: ltp-pan Not tainted
     2.6.29-gcov #1
Call Trace:
[<c0698374>]should_fail+0x31f/0x3e0
[<c0698266>]?should_fail+0x211/0x3e0
[<c0514e5c>]?should_failslab+0x60/0x73
[<c05123ca>]?slab_should_failslab+0x35/0x48
```

  – Long hours of execution

    Many tests took exceptionally long hours of execution time. But, otherwise, they take seconds to execute. Since, many tests in the bucket started reflecting such abnormal behavior, we had to terminate the experiment owing to the fact that the experiment cannot be continued till infinity.

- **probability=10%**

  We found this particular value more interesting; that it allowed us to run our test bucket for finite time, and simultaneously allowed us to measure the differences in the *code coverage* of test runs between the *stable* and *fault* environments. Though many tests exhibited the earlier scenarios (**probability=30%**), still they did not hinder in completing the tests in finite time frame. However, we chose a very small set of tests, namely the *LTP Syscall tests* [6].

Figure 2 shows the code coverage obtained when the tests were run under stable kernel conditions. Out of accounted `TOTAL_CODE=377538`, **coverage** is **16.4%**, and of `TOTAL_FUNCTIONS=29852`, the tests has touched **21.9%** functions.

**LCOV - code coverage report**

| | | | Found | Hit | Coverage |
|---|---|---|---|---|---|
| **Test:** | Without_fault_injection.info | **Lines:** | 377538 | 61872 | 16.4 % |
| **Date:** | 2009-06-22 | **Functions:** | 29852 | 6547 | 21.9 % |

| Directory | Line Coverage | | | Functions | |
|---|---|---|---|---|---|
| arch/x86/include/asm | | 43.0 % | 533 / 1240 | 46.1 % | 135 / 293 |
| arch/x86/lib | | 58.4 % | 115 / 197 | 59.6 % | 28 / 47 |
| arch/x86/mm | | 13.2 % | 281 / 2129 | 19.4 % | 39 / 201 |
| block | | 32.0 % | 1844 / 5764 | 35.9 % | 215 / 599 |
| drivers/ata | | 16.1 % | 1111 / 6891 | 23.1 % | 104 / 451 |
| drivers/char | | 18.7 % | 1839 / 9836 | 25.5 % | 173 / 678 |
| drivers/net | | 1.1 % | 38 / 3598 | 2.6 % | 5 / 193 |
| drivers/scsi | | 6.4 % | 700 / 10935 | 9.9 % | 80 / 805 |
| fs | | 45.4 % | 9820 / 21645 | 52.8 % | 809 / 1533 |
| fs/debugfs | | 0.0 % | 0 / 274 | 0.0 % | 0 / 53 |
| fs/ext3 | | 46.3 % | 3138 / 6780 | 63.6 % | 199 / 313 |
| fs/proc | | 38.1 % | 1427 / 3742 | 46.5 % | 141 / 303 |
| fs/sysfs | | 33.4 % | 373 / 1118 | 48.9 % | 46 / 94 |
| include/linux | | 52.5 % | 2700 / 5140 | 52.6 % | 798 / 1518 |
| include/net | | 48.2 % | 955 / 1983 | 50.7 % | 237 / 467 |
| ipc | | 63.8 % | 1708 / 2677 | 69.4 % | 127 / 183 |
| kernel | | 33.6 % | 9220 / 27400 | 40.3 % | 1000 / 2484 |
| lib | | 38.7 % | 2046 / 5289 | 41.3 % | 189 / 458 |
| mm | | 37.5 % | 6136 / 16344 | 42.8 % | 527 / 1230 |
| net | | 59.6 % | 598 / 1004 | 53.3 % | 48 / 90 |
| security | | 32.4 % | 492 / 1517 | 43.1 % | 173 / 401 |

*Generated by: LCOV version 1.7*

Figure 2: Code Coverage without Fault Injection

**LCOV - code coverage report**

| | | | Found | Hit | Coverage |
|---|---|---|---|---|---|
| **Test:** | With_10%_fault_injection.info | **Lines:** | 377538 | 64275 | 17.0 % |
| **Date:** | 2009-06-22 | **Functions:** | 29852 | 6742 | 22.6 % |

| Directory | Line Coverage | | | Functions | |
|---|---|---|---|---|---|
| arch/x86/include/asm | | 43.3 % | 537 / 1240 | 46.4 % | 136 / 293 |
| arch/x86/lib | | 58.4 % | 115 / 197 | 59.6 % | 28 / 47 |
| arch/x86/mm | | 13.3 % | 283 / 2129 | 19.4 % | 39 / 201 |
| block | | 35.9 % | 2069 / 5764 | 38.6 % | 231 / 599 |
| drivers/ata | | 17.6 % | 1214 / 6891 | 24.4 % | 110 / 451 |
| drivers/char | | 18.2 % | 1791 / 9836 | 25.5 % | 173 / 678 |
| drivers/net | | 1.1 % | 38 / 3598 | 2.6 % | 5 / 193 |
| drivers/scsi | | 7.3 % | 803 / 10935 | 10.9 % | 88 / 805 |
| fs | | 46.0 % | 9947 / 21645 | 53.5 % | 820 / 1533 |
| fs/debugfs | | 6.2 % | 17 / 274 | 7.5 % | 4 / 53 |
| fs/ext3 | | 47.1 % | 3196 / 6780 | 63.9 % | 200 / 313 |
| fs/proc | | 38.8 % | 1453 / 3742 | 47.2 % | 143 / 303 |
| fs/sysfs | | 37.9 % | 424 / 1118 | 52.1 % | 49 / 94 |
| include/linux | | 53.6 % | 2755 / 5140 | 54.2 % | 823 / 1518 |
| include/net | | 49.8 % | 988 / 1983 | 52.7 % | 246 / 467 |
| ipc | | 63.9 % | 1710 / 2677 | 69.4 % | 127 / 183 |
| kernel | | 34.5 % | 9450 / 27400 | 41.2 % | 1023 / 2484 |
| lib | | 41.3 % | 2185 / 5289 | 44.8 % | 205 / 458 |
| mm | | 38.7 % | 6320 / 16344 | 43.9 % | 540 / 1230 |
| net | | 60.6 % | 608 / 1004 | 54.4 % | 49 / 90 |
| security | | 32.7 % | 496 / 1517 | 43.6 % | 175 / 401 |

*Generated by: LCOV version 1.7*

Figure 3: Code Coverage with Fault Injection

And Figure 3 depicts code coverage when the tests were executed under situation which is a union of stable and fault injection. Out of `TOTAL_CODE=377538`, **coverage** is **17.0%**, and of `TOTAL_FUNCTIONS=29852`, the tests has touched **22.6%** functions.

For sake of visibility and compactness, we highlight only those kernel directories and sub-directories for which significant code coverage increase has happened. Few interesting figures are:

- **3.9%** increase in **block**
- **6.2%** increase in **fs/debugfs**
- **4.5%** increase in **fs/sysfs**
- **1.2%** increase in **mm**

Though the overall increase in `CODE_COVERAGE` of **0.6%**, and `FUNCTION_COVERAGE` of **0.7%** is not significant, but it drives home a point that Code Coverage is bound to increase with `Fault Injection`. The above results are based in minimal set of LTP test cases run, and definitely the figure would be impressive, if the entire test suite is run.

## 6 Conclusion

The usage of Static and Dynamic Analysis tools to test LTP's health has opened up a new plethora of opportunities. These tools would be put to use more in future to validate old/new test cases. We look towards integrating them with LTP infrastructure, and they themselves becoming yardsticks for quality control. Some of the LTP test cases are beginning to show their age, they have helped identify bugs, but with newer technology and tools, it is time to revisit the test cases and shake off the bugs hiding in them, which our regular runtime execution did not expose.

Integration of Fault-Injection creation framework in LTP would be immensely beneficial to developers, who can then design robust testcases to handle these faults better. LTP also looks forward to strengthen its position in the **Linux Ecosystem**, integrate itself with other players in the same ecosystem, so that it can continue to deliver and evolve into better test suite to *Relentlessly Pursue a Better Kernel*.

## Acknowledgement

We would like to thank many of our colleagues and team mates for their inputs to, and, review of drafts of this

paper. And a special thanks to all those LTP developers whose immense contribution keeps this project growing.

## Legal Statement

Copyright © 2009 International Business Machines Corporation and Red Hat, Inc. International Business Machines Corporation ("IBM") and Red Hat, Inc. ("Red Hat") retain the copyright to the submitted paper, but have granted unlimited redistribution rights to all as a condition of submission. This work represents the view of the authors and does not necessarily represent the view of IBM or Red Hat. IBM, IBM logo, ibm.com, and WebSphere, are trademarks of International Business Machines Corporation in the United States, other countries, or both. RED HAT and the Shadowman logo are trademarks of Red Hat, Inc., registered in the United States and other countries. Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries. Other company, product, and service names may be trademarks or service marks of others. References in this publication to IBM products or services do not imply that IBM intends to make them available in all countries in which IBM operates. INTERNATIONAL BUSINESS MACHINES CORPORATION AND RED HAT, INC. PROVIDE THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you. This information could include technical inaccuracies or typographical errors.

## References

[1] Gnu c library. `http://www.gnu.org/software/libc/`.

[2] Lcov - the ltp gcov extension. `http://ltp.sourceforge.net/coverage/lcov.php`.

[3] Linux kernel documentation. `http://www.mjmwired.net/kernel/Documentation/fault-injection/`.

[4] Linux kernel mailing list subscription url. `http://vger.kernel.org/vger-lists.html#linux-kernel`.

[5] Linux test project home page. `http://ltp.sourceforge.net/`.

[6] Ltp source code repository. `http://ltp.cvs.sourceforge.net/viewvc/ltp/ltp/runtest/syscalls`.

[7] Ltp technical papers - what is ltp, how to use ltp, etc. `http://ltp.sourceforge.net/documentation/technical_papers/`.

[8] Man pages entry for posix_fadvise(). `http://www.kernel.org/doc/man-pages/online/pages/man2/posix_fadvise.2.h%tml`.

[9] Sparse - a semantic parser for c. `http://www.kernel.org/pub/software/devel/sparse/`.

[10] Splint - tool for statically checking c programs for security vulnerabilities and coding mistakes. `http://www.splint.org/`.

[11] Valgrind. `http://valgrind.org/`.

[12] Pramod Gurav. [LTP] ltp tests failing. `http://www.mail-archive.com/ltp-list@lists.sourceforge.net/msg00965.htm%l`.

[13] Michael Kerrisk. Linux Foundation fellowship, 6 months in. `http://linux-man-pages.blogspot.com/2008/12/linux-foundation-fellowship%-6-months-in.html`.

[14] Masatake YAMATO. Linux ecosystem around test. `http://people.redhat.com/yamato/talks/around-test.pdf`.

[15] Masatake YAMATO. [PATCH] checking ADVICE of fadvice64_64 even if get_xip_page is given. `http://lkml.org/lkml/2008/1/9/75`.

# Linux-based virtualization for HPC clusters


Lucas Nussbaum
*INRIA - UCB Lyon 1 - ENS Lyon - CNRS*
`lucas.nussbaum@ens-lyon.fr`

Fabienne Anhalt
*INRIA - UCB Lyon 1 - ENS Lyon - CNRS*
`fabienne.anhalt@ens-lyon.fr`

Olivier Mornard
*INRIA - UCB Lyon 1 - ENS Lyon - CNRS*
`olivier.mornard@ens-lyon.fr`

Jean-Patrick Gelas
*INRIA - UCB Lyon 1 - ENS Lyon - CNRS*
`jpgelas@ens-lyon.fr`



## Abstract

There has been an increasing interest into virtualization in the HPC community, as it would allow to easily and efficiently share computing resources between users, and provide a simple solution to checkpointing. However, virtualization raises a number of interesting questions, on performance and overhead, of course, but also on the fairness of the sharing. In this work, we evaluate the suitability of KVM virtual machines in this context, by comparing them with solutions based on Xen. We also outline areas where improvements are needed to provide directions for future works.


## 1 Introduction and motivations

Operating System Virtualization, and all its variants, already largely proved their usefulness in the context of traditional servers. However, in the area of High Performance Computing (HPC), for computing clusters or, on a larger scale, grid or cloud computing, virtualization still has to convince most end users and system administrators of its benefits. The use of virtualization in the context of HPC offers several immediate advantages.

First, any computing center or large scale computing infrastructure under-uses a non-negligible number of physical resources. This is for example due to the fact that all the computing applications are not perfectly embarassingly-parallel. Using virtualization would allow to dynamically allocate resources to jobs, allowing to match their exact performance needs.

Next, many processing jobs do not take full advantage of the multicore architecture available on processing nodes. Deploying several Virtual Machines (VM) per node (e.g., 1 VM per core) would provide an easy way to share physical resources among several jobs.

On most computing grids, a user books a number of resources for a given period of time (also called *lease*). This period of time is generally a rough estimation made by the user of the time required for his application to complete. When the lease expires, results will be lost if the job did not have enough time to finish, and if no checkpointing mechanism is implemented. Good checkpointing mechanisms are difficult to implement, and virtualization provides an easy way to implement it, by freezing and migrating virtual machines.

Finally, the software configuration of computing platforms is generally static, which might be a problem for users with specific needs. Virtualization could allow to deploy customized user environments on the computing nodes, thus allowing users with specific software needs to customize the operating system on which their application will be executed.

Given the arguments listed above, virtualization seems to be an attractive solution for the HPC community. However, using virtualization in this context also has drawbacks.

Indeed, the overhead caused by the additional layers is not well known and controlled, mainly due to a lack of understanding of the underlying virtualization infrastructure.

Another issue is the physical resource sharing. Virtual machines need to access concurrently the physical devices, and it is possible that this sharing mechanism impacts the performance. This raises also the question of the scalability of the number of VMs it is possible to host on a physical machine.

The following section proposes a reminder about the common virtualization solutions currently available. It may help readers to set up the vocabulary in this domain and get familiar with Xen [3] and KVM [9]. Section 3 details our experimental testbed. This section is followed by an evaluation with several micro benchmark (CPU, disk, network) (Section 4) and then with classic HPC benchmarks (Section 5). Finally, before concluding (Section 7) we propose a brief state of the art (Section 6).

# 2 Virtualization

In this section, we describe the different virtualization techniques, and then introduce more specifically Xen [3] and KVM [9].

## 2.1 Virtualization approaches

The goal of virtualization is to partition one physical node (or system) into several independent virtual machines. Common applications of virtualization are server consolidation, and testing and development environments.

One common technique is OS-level virtualization where a single kernel is shared by containers which represent the VMs (*e.g.* VServer). An other approach would be to allow several OS with distinct kernels to run on a single physical machine inside the VMs to give the user a maxium reconfiguration facility.

To manage several of these reconfigurable VM running on a physical node and sharing *de facto* the same hardware, we need a layer acting as a supervisor (a sort of arbiter to access hardware resources). However, as VM include already a supervisor, we call this layer a hypervisor (*i.e.,* a supervisor of supervisors) also called VMM (Virtual Machine Monitor).

The hypervisor manages the requests of VMs and their access to the resources (i.e., IRQ routing, time keeping and message passing between VMs).

Hypervisor virtualization can be divided in two types, Full Virtualization (FV) and Paravirtualization (PV), which can be both combined with hardware-assisted virtualization.

### 2.1.1 Full virtualization

Full virtualization (FV) allows the execution of unmodified guest operating systems by emulating the real system's resources. This is especially useful to run proprietary systems. One pioneer was VMware providing a full virtualization solution. However, providing the guest system with a complete real system interface has an important cost. This cost can be mitigated by using *Hardware-assisted virtualization* discussed in section 2.1.3. KVM takes advantage of this evolution. Currently, in the x86 architecture, the hardware assistance is available in the CPU only, not in the other parts of the computer (like network or video adapters). The gap is then filled by emulation, having an impact on the performance. An alternative solution called hybrid [12] approach consists in using specific paravirtualized drivers which is more efficient than emulation (in terms of CPU consumption) and reaches better performances.

### 2.1.2 Paravirtualization

Paravirtualization (PV) is also based on a hypervisor, but the devices are not emulated. Instead, devices are accessed through lightweight virtual drivers offering better performance.

The drawback is that guest kernels must be upgraded to provide new system calls for the new services. At the lowest level the syscalls are interrupts (0x80) with a function number, which allows to switch from the user mode to the privileged mode in former Linux system call. The newest Linux system uses now a faster method with the syscall/sysenter opcodes (in x86 architecture). In the same way in Xen [3], the OS executes hypercalls with the interrupt 0x82. Like in the Linux system, the use of interrupts is deprecated and replaced by the use of hypercall pages [18], a similar mechanism in Linux called vDSO used to optimize the system call interface.[1] vDSO chooses between int 0x80, sysenter or syscall opcodes (the choice is made by the kernel at boot time).

### 2.1.3 Adding hardware virtualization support

Virtualization software techniques consisting in doing binary translation to trap and virtualize the execu-

[1] `http://www.trilithium.com/johan/2005/08/linux-gate/`

tion of some instructions are very cost inefficient (ex: VMware). Running a VM on a common architecture (ex: IA32 PC) for which it has not been designed is difficult. The original x86 architecture does not comply with the base conditions for being virtualized (equivalence, resource control (safety), efficiency) [15]. In particular, there are some unprivileged instructions changing the state of the processor that can not be trapped.

In 2007, Intel and AMD designed (independently) some virtualization extensions for the x86 architecture [13] (VMX for Intel, Virtual Machine eXtension; and AMD-V/Pacifica for AMD). Each one allows the execution of a hypervisor in order to run an unmodified operating system while minimizing the overhead due to emulation operations.

The kernels can run in privileged mode on the processor, which means on ring 0. Ring 0 is the most privileged level. On a standard system (i.e, not virtualized) this is where the operating system is running. The rings strictly over 0 run instructions in a processor mode called unprotected. Without specific hardware virtualization support, the hypervisor is running in ring 0, but the VM's operating system can not reach this level of privilege (they access ring 1, at best). Thus, in full-virtualization, privileged instructions are emulated, and in paravirtualization the kernel is modified in order to allow those instructions to access ring 0. The hardware assisted virtualization not only proposes new instructions, but also a new privileged access level, called "ring -1", where the hypervisor can run. Thus, guest virtual machines can run in ring 0.

Despite these advantages, using an untouched/ unmodified operating system means a lot of VM traps and then a high CPU consumption used by the emulation of hardware (network manager, video adapter, ...). An alternative solution, called *hybrid* [12], consists in using paravirtualized drivers in combination with the hardware-assisted virtualization.

## 2.2 Introducing Xen and KVM

In this article, we limit our study to free and open source virtualization solutions. Thus, we chose to study and evaluate exclusively the latest releases of Xen and KVM [9] at the time of writing, that are Xen 3.3.1 and KVM 84.

### 2.2.1 Xen

Xen started as a research project by Ian Pratt at Cambridge University. The very first public release of Xen was delivered in October 2003. Then, Ian Pratt created the XenSource company, which develops the project in an open source fashion and distributes customized Xen versions (Xen Enterprise). Major releases 2.0 and 3.0 were delivered respectively in 2004 and 2005. The latest current release available is 3.3.1 (2009). Xen is compatible with x86 processors (Intel or AMD), x86_64 since 3.0, SMP architectures, HyperThreading technology, IA64, PPC. ARM support should also be available soon.

Xen is a hypervisor and has the ability to run guest operating systems, called domains. There are two types of domains. Unprivileged domains (called *DomU*) are the guest systems, while the privileged domain (called *Dom0*) is a special guest with extended capabilities, that contains the applications to control the other guests. *Dom0* is running above the Xen hypervisor when the physical machine starts. It runs a modified Linux kernel, and is generally the only domain able to interact directly with the hardware through the linux kernel drivers. It also allows DomUs to communicate with hardware devices using their virtual drivers.

When a virtual machine hosted in a domU previously described wants to use hardware devices, e.g. the network interface or the block device, the data has to go to dom0 which is than in charge of transmitting it to the physical device. Several mechanisms are invoked to make the transfer between domU and dom0 and to minimize overhead. However, the data path is longer than without virtualization, as shown in Figure 1.

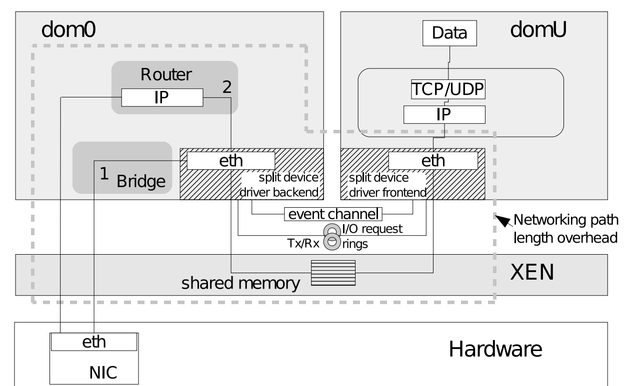



Figure 1: Network path in Xen.

In this example where domU uses the physical network interface to send packets to a remote physical station, packets go through the domU TCP/IP stack, then are transferred to dom0. To make this transfer, dom0 invokes a grant to domU's memory page to fetch the data by page flipping. The other way around, during packet reception on domU, dom0 copies the data into a shared memory segment so that domU can get it [6]. This copy and page flipping mechanisms offer security but are heavy for the performance.

From a technical point of view, on the x86 architecture (with no hardware support), the Xen hypervisor is running in ring 0, kernels in ring 1 and finally applications in ring 3. On the x86_64 architecture, the hypervisor is running in ring 0, and guest domains (*Dom0* and *domUs*) and applications run in ring 3 (i.e., rings 1 and 2 have been removed).

Moreover, the x86 Intel architecture proposes two levels of memory protection. One is based on a segmentation mechanism and the other on page management. Each of these protections may be used to isolate virtualized systems. (NB: Two modes are currently available on x86 architectures: 32 bit mode and 64 bit mode. Only x86 64 bit system architectures (a.k.a IA32e or AMD64) may use both modes). In the 32 bit mode, segment management is fully operational and is used for the memory protection.

In 64 bit mode, segment management almost disappears (for example there is no more segment base and limit management), memory protection through segments is not possible anymore, thus protection through memory page management is used (via the MMU unit)[7]. However, with this mechanism, there are only two levels of protection called normal mode and supervisor mode. Xen must then manage protection through the page level which is the most CPU intensive. Without forgetting that context switching introduced by virtualization is time consuming and so impacts the guest systems performances.

Finally, the inclusion of different parts of Xen in the Linux kernel has been the subject of animated discussions. DomU support is already included, but the inclusion of Dom0 support faced a lot of opposition. In addition to that, Linux distributions use the XenSource-provided patch for 2.6.18, and forward-port this patch to the kernel releases they want to ship in their stable release. The process of forward-porting those patches is difficult, and not supported by the upstream author, leading some distributions to choose to stop supporting Xen recently.

### 2.2.2 KVM

KVM (Kernel based Virtual Machine) is an open source Linux kernel virtualization infrastructure[2] which relies on the hardware virtualization technologies, fully integrated in the Linux kernel. Its first version was introduced in the 2.6.20 Linux kernel tree (released in February 2007). KVM developers are primarily funded by a technology startup called Qumranet, now owned by RedHat. Developers had an original approach. Instead of creating major portions of an operating system kernel themselves, they choose to use the Linux kernel itself as a basis for a hypervisor. Thus, KVM is currently implemented as loadable kernel modules. `kvm.ko`, that provides the core virtualization infrastructure and a processor specific module, `kvm-intel.ko` or `kvm-amd.ko`. The code is relatively small (about 10,000 lines) and simple. This original approach has several benefits. The virtualized environment takes advantage of all the ongoing work made on the Linux kernel itself.

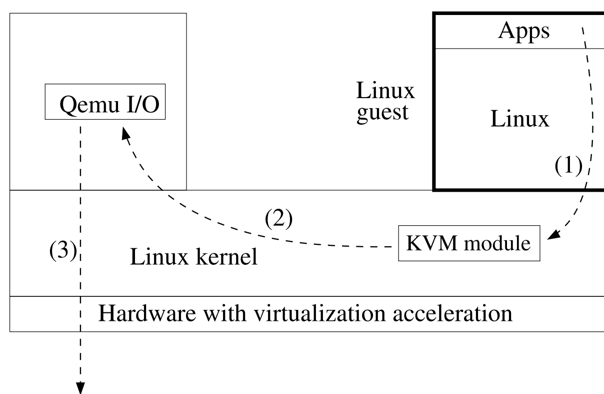



Figure 2: Path of I/O requests in KVM

KVM makes use of hardware virtualization to virtualize processor states (an Intel processor with VT (virtualization technology) extensions, or an AMD processor with SVM extensions (also called AMD-V)). With KVM, each virtual machine is a regular Linux process, scheduled by a standard Linux scheduler. Memory management of the VM is handled from within the kernel but I/O in the current version is handled in user space

[2] `http://www.linux-kvm.org`

through a userspace component also used to instantiate the virtual machines. This component is actually a modified version of QEMU handling I/O hardware emulation: as shown in figure 2, when a process in a guest system issues an I/O request, the request is trapped by KVM, then forwarded to the QEMU instance in the host system's userspace, that issues the real I/O request. KVM emulates virtual devices, such as network interfaces or hard disks. In order to improve performance, recent KVM versions propose a hybrid approach called *virtio* [16]. Virtio is a kernel API that improves the performance of communications between guest systems and the host system by providing a simpler and faster interface than the emulated devices from QEMU. Virtio-based devices exist both for network interfaces and hard disks.

#### 2.2.3 Conclusion

Hardware Assisted Full Virtualization (FV) is often believed to be the best virtualization solution, performance-wise. However, this is not true: paravirtualization approaches may be much better in terms of performance, especially in the context of the IO. In the following sections, we perform several benchmarks outlining the performance differences of the different virtualization techniques and explain why there are such differences.

## 3 Evaluation

In the following experiments, we compare four different virtualization solutions:

**Xen FV** : Xen using full hardware-assisted virtualization (also called Xen HVM for Hardware Virtual Machine)

**Xen PV** : Xen using paravirtualization

**KVM FV** : standard KVM, using the I/O devices emulated by QEMU

**KVM PV** : KVM using the virtio I/O devices

All the experiments were performed on a cluster of Dell PowerEdges 1950 with two dual-core Intel Xeon 5148 LV processors with 8 GB of memory, 300 GB Raid0 / SATA disks and interconnected by 1 Gb/s network links. We used Xen 3.3.1 and KVM 84 for all tests, except when specified otherwise.

We first evaluate all solutions with a set of micro-benchmarks, to evaluate the CPU, the disk accesses and the network separately, then use the HPC Challenge benchmarks, a set of HPC-specific benchmarks.

## 4 Evaluation with micro-benchmarks

In this section, we evaluate the different virtualization solutions with a set of micro-benchmarks.

### 4.1 CPU

In our first experiment, we focused on CPU-intensive applications. We evaluated the overhead caused by using a virtualization solution for such applications, which are obviously crucial for HPC.

Both Xen and KVM support SMP guests, that is, giving the ability to a guest system to use several of the host's processors. We executed a simple application scaling linearly on hosts with four CPUs, then inside guests to which four CPUs had been allocated.

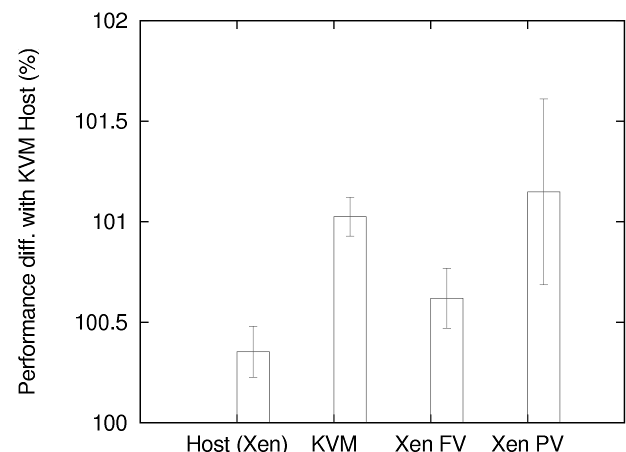



Figure 3: Influence of virtualization on CPU-intensive applications: the same application is executed on a 4-CPU system, then on guests allocated with 4 CPUs.

Figure 3 shows the difference between our host system running Linux 2.6.29, used for KVM, and several other configurations. In all cases, the overhead was minimal (lower than 2%). However, it is worth noting that running the application in the Xen dom0 is slightly slower

than on Linux 2.6.29 (maybe because of improvements in Linux since the 2.6.18 kernel used as Dom0), and that both KVM and Xen guests suffer from a small slowdown.

## 4.2 Disk

While disks can now be considered slow devices (compared to high-speed NICs, for example), it can still be difficult to fully exploit their performance for virtualization solutions.

In this experiment, we compare the different solutions by writing large files using `dd`, using different block sizes. This allows to measure both the influence of per-I/O overhead (for small block sizes) and available I/O bandwidth. We also confirmed our results using `bonnie++`.

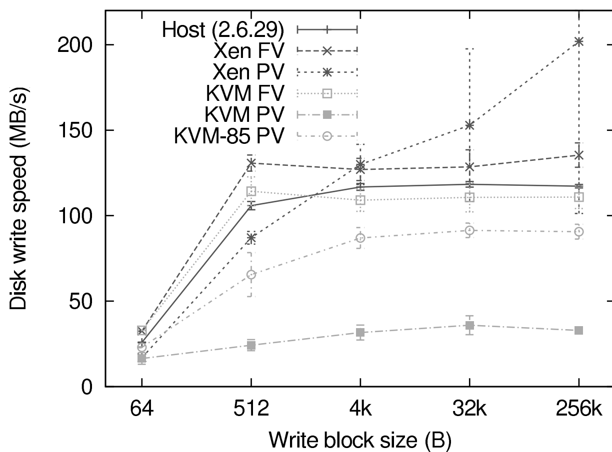



Figure 4: Disk write performance

Results are presented in Figure 4. Each test was run 10 times, and the vertical bars indicate the 95% confidence interval. We used file-backed virtual machines for all tests. Our test systems allowed a maximum write speed of about 120 MB/s on the host system, thanks to the RAID-0 disks setup.

In our tests, Xen PV reported write speeds much higher than the write speed that we obtained from the host system, with a very high variability. While we were not able to confirm that, it seems that Xen reports write completions before they are actually completely committed to disk.

While KVM provided good performance (close to the host system) in full virtualization mode, *virtio* provided more disappointing results. In fact, we identified with `blktrace` that a lot of additional data was written to disk with *virtio*: writing a 1 GB-file resulted in about 1 GB of data written to disk without *virtio*, versus 1.7 GB of data written with *virtio*. This is very likely to be a bug. Since our tests were originally performed with KVM 84, we also re-ran the tests with a version from the KVM git tree, very close to the KVM 85 release date. This more recent version provided better performance, but still far from the one obtained with the other configurations.

It is also worth noting that, while the size of block sizes clearly affects the resulting performance (because of the influence of latency on the performance), it affects all solutions in a similar way.

## 4.3 Network

In this section, the network performance in virtual machines with KVM is compared to Xen network performance using either hardware virtualization or paravirtualization techniques.

To measure throughput, the `iperf` [17] benchmark is used sending TCP flows of 16 kByte messages on the virtual machines. The corresponding CPU cost is measured with the Linux `sar` utility on KVMs and with `xentop` on Xen. Each result is the average of 10 runs of 60 seconds of each test. To each virtual machine, one of the 4 physical CPUs is attributed. The different domains are scheduled in Xen to use the CPUs with default credit-scheduler [19]. In KVM, virtual machines use the emulated e1000 driver for hardware virtualization and virtio for paravirtualization. The virtual machines communicate using virtual tap interfaces and a software bridge interconnecting all virtual machines and the physical network interface. Xen virtual machines under paravirtualization use the virtual split device driver and Xen HVMs use the emulated Realtek 8139 driver. They communicate also using a software bridge in host domain 0.

### 4.3.1 Inter virtual machine communication

In this first experiment, network performance between virtual machines hosted on the same physical machine is evaluated not invoking the use of the physical network

interface and allowing to evaluate the network speed allowed by the CPU under the virtualization mechanisms. This setup is represented on Figure 5.

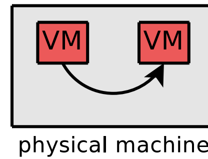



Figure 5: Test setup with two communicating virtual machines hosted on the same physical machine.

The two virtual machines communicate using different mechanisms according to the virtualization technique. In Xen, packets go to the virtual or emulated driver to reach dom0, than to dom0's backend driver to reach the destination virtual machine. On KVM, the packets use also an emulated or virtual interface and are than handled by the kvm module to be sent to the destination virtual machine.

The results in terms of TCP throughput in the four configurations are represented in Table 1.

| | FV | PV |
|---|---|---|
| KVM | 648.3 | 813.2 |
| Xen | 96.05 | 4451 |

Table 1: Average TCP throughput in Mbit/s between two virtual machines hosted on a single physical machine under full virtualization (FV) or paravirtualization (PV).

The best throughput is obtained on Xen paravirtualized guests which can communicate in a very lightweight way achieving nearly native Linux loopback throughput (4530 Mb/s) to the cost of an overall system CPU use of around 180% while native Linux CPU cost is about 120%. However with hardware assisted virtualization, Xen has very poor throughput with even more CPU overhead (about 250% of CPU use) due to the network driver emulation. KVM achieves a throughput between 14 and 18% of native Linux loopback throughput generating a CPU cost between about 150 and 200%.

#### 4.3.2 Communication with a remote host

For communications between virtual machines hosted by distinct physical servers, the packets need to use the physical network interfaces of the hosts. This experiment evaluates the resulting performance. As sending and receiving do not invoke the same mechanisms, sending throughput is evaluated separately from receiving throughput as represent the two configurations on Figure 6.

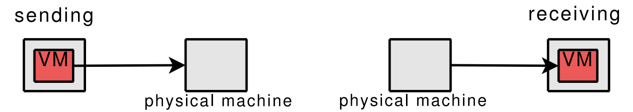



Figure 6: Test setup with a virtual machine communicating with a remote physical host.

Figure 7 shows the throughput obtained on the different types of virtual machines.

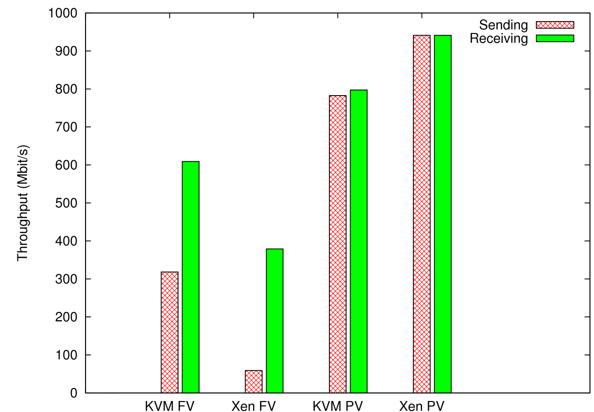



Figure 7: TCP throughput on a virtual machine sending to or receiving from a remote host.

Para-virtualization shows much better results in terms of throughput than hardware virtualization like in the previous experiment, with KVM and Xen. While with Xen para-virtualization, the theoretical TCP throughput of 941 Mb/s is reached, with KVM and the paravirtualized driver, throughput reaches only about 80% of native Linux throughput. In Xen, the network interface is the bottleneck as loopback throughput reaches 4451 Mb/s. In KVM paravirtualization, the virtualization mechanism is the bottleneck, as the same throughput is obtained, whether the network interface card is used or not. With hardware-virtualization, the network performance is very poor, especially with Xen HVM and in the case of sending. This shows that the sending mechanism from the Xen HVM is obviously the bottleneck also in the previous experiment. KVM FV uses about 100% of the CPU, which is assigned to it and can not achieve better throughput, needing more CPU capacity. In the case of paravirtualization with virtio, KVM needs

less CPU, about 76% for sending and 60% for receiving while the physical host system is performing a part of the work to access the physical NIC. The overall system CPU usage is still about 100% of one CPU, but the resulting bandwidth more than doubles in the case of sending.

### 4.3.3 Scalability

This experiment evaluates the impact on throughput while scaling up to either 2, 4 or 8 virtual machines hosted on a single physical host. Figure 8 shows an example with 4 VMs.

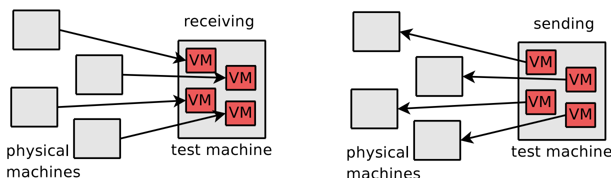



Figure 8: Test setup with 4 virtual machines communicating with 4 remote physical host.

As in the previous experiment, sending an receiving throughput is evaluated separately.

The aggregated TCP throughput obtained on the virtual machines for sending and receiving is represented respectively on Figures 9 and 10 in each configuration (KVM and Xen, with para- or full-virtualization).

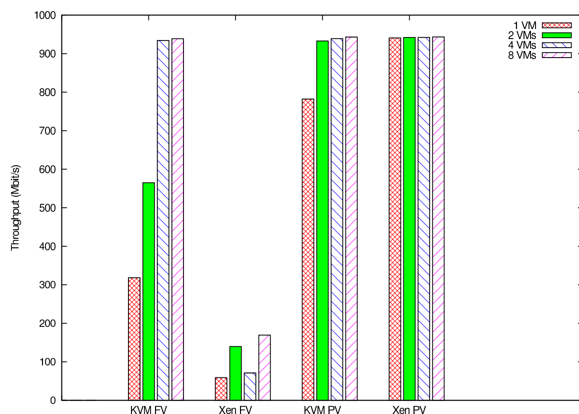



Figure 9: TCP sending throughput on a set of up to 8 VMs.

In both configurations, KVM and Xen, paravirtualization achieves better throughput, like before. Observing the evolution of the aggregate throughput with an

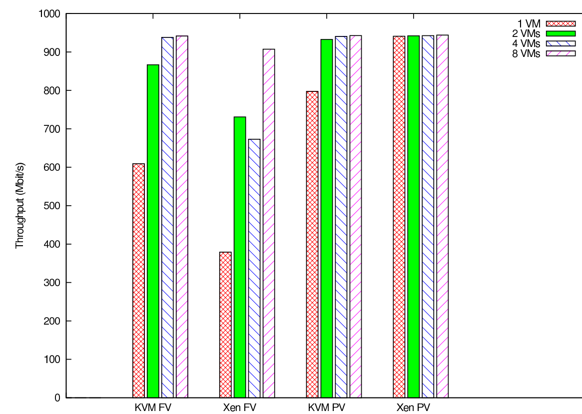



Figure 10: TCP receiving throughput on a set of up to 8 VMs.

increasing number of virtual machines, it can be seen that a bigger number of VMs achieve a better overall throughput than a single virtual machine. In the case of KVM, this might be related to an important CPU overhead necessary for networking. With a single virtual machine sending a single TCP flow, KVM consumes the capacity of an entire CPU whether it uses the emulated e1000 driver or virtio. Using two virtual machines sending two flows, they can each one use one CPU which actually happens for KVM full virtualization where throughput still not reaches the maximum value allowed by the NIC. Paravirtualization needs less instructions making KVM use only between 130 and 165% of the 4 CPU cores and achieving nearly maximum TCP throughput.

Xen HVM has the most important CPU overhead, especially with 4 or 8 virtual machines, and achieves the poorest throughput. Only dom0 uses almost 250% of the CPUs to forward the traffic of 4 virtual machines. This means that it uses at least 3 CPUs simultaneously and need to share them with the domUs. This sharing needs more context switches. With Xen paravirtualization the overall system CPU utilization does not exceed 160% for dom0 and 8 domUs, allowing to achieve maximum TCP throughput.

In each experiment, the different virtual machines achieve almost the same individual throughput. For this reason, only the aggregated throughput is represented. Per virtual machine throughput corresponds to the aggregated throughput to the number of VMs. This means that the resource sharing is fair between the different virtual machines.

For inter-VM communications on a same physical host

and also for communications using the physical interface, despite its virtualization overhead, Xen paravirtualization achieved the best network throughput having the lowest CPU overhead compared to hardware virtualization and KVM. However, KVM with the virtio API and the paravirtualized drivers can achieve similar throughput if it has enough CPU capacity. This solution could be a good tradeoff between performance and isolation of virtual machines.

## 5 Evaluation with classical HPC benchmarks

In this section, we report on the evaluation of the various virtualization solutions using the HPC Challenge benchmarks [10]. Those benchmarks consist in 7 different tools evaluating the computation speed, the communication performance, or both, like the famous LINPACK/HPL benchmark used to rank the computers for the Top500.[3]

The following results were obtained with HPCC 1.3.1 on a cluster of 32 identical Dell PowerEdge 1950 nodes, with two dual-core Intel Xeon 5148 LV CPUs and 8 GB of RAM. The nodes are connected together using a Gigabit ethernet network. The benchmarks were run on several configurations:

- the host system used to run KVM virtual machines (using Linux 2.6.29);
- the host system used to run Xen virtual machines (Xen dom0, using Linux 2.6.18);
- 32 KVM virtual machines allocated to 4 CPUs each, using *virtio* for network and the classic emulated driver for disk;
- 128 KVM virtual machines allocated to 1 CPU each, using *virtio* for network and the classic emulated driver for disk;
- 32 Xen paravirtualized virtual machines allocated to 4 CPUs each;
- 128 Xen paravirtualized virtual machines allocated to 1 CPU each;
- 32 Xen virtual machines using full virtualization, allocated to 4 CPUs each;
- 128 Xen virtual machines using full virtualization, allocated to 1 CPU each.

[3] http://www.top500.org

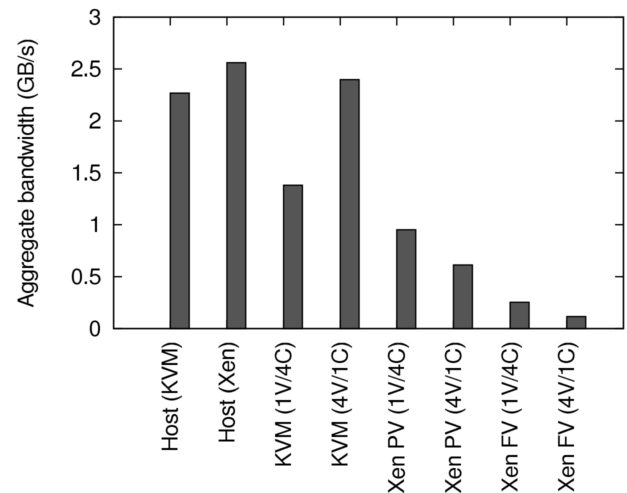



Figure 11: PTRANS benchmark: aggregate network bandwidth

### 5.1 PTRANS benchmark

PTRANS (parallel matrix transpose) excercises the communication network by exchanging large messages between pairs of processors. It is a useful test of the total communications capacity of the interconnect. Results shown in Figure 11 indicate that:

- The setup using four KVM VMs per node performs better than the one using 1 VM per node, and provides performance that is close to native. This might be explained by the fact that having several virtual machines allows the load to be better spread across CPUs;
- Xen setups perform very poorly in that benchmark.

### 5.2 STREAM benchmark

STREAM is a simple benchmark that measures the per-node sustainable memory bandwidth. As shown in Figure 12, all configurations perform in the same way (differences are likely to be caused by measurement artifacts).

### 5.3 Latency and Bandwidth Benchmark

The latency and bandwidth benchmark organizes processes in a randomly-ordered ring. Then, each process receives a message from its predecessor node, then send

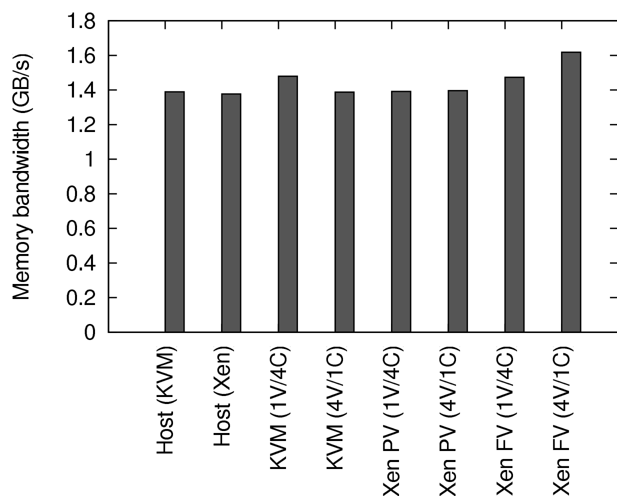



Figure 12: STREAM benchmark: per-node memory bandwidth

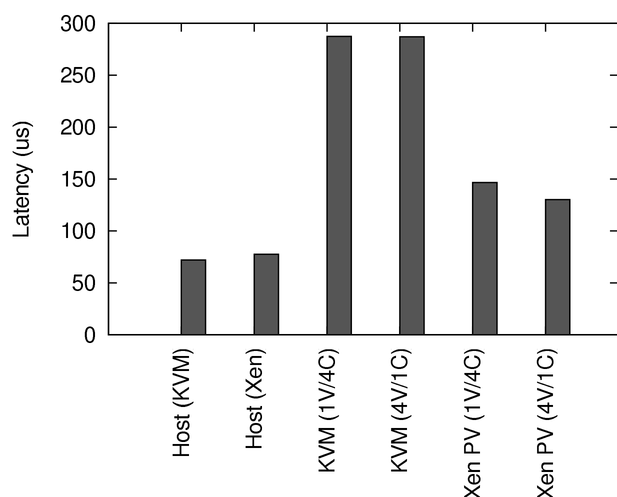



Figure 13: Average node-to-node latency

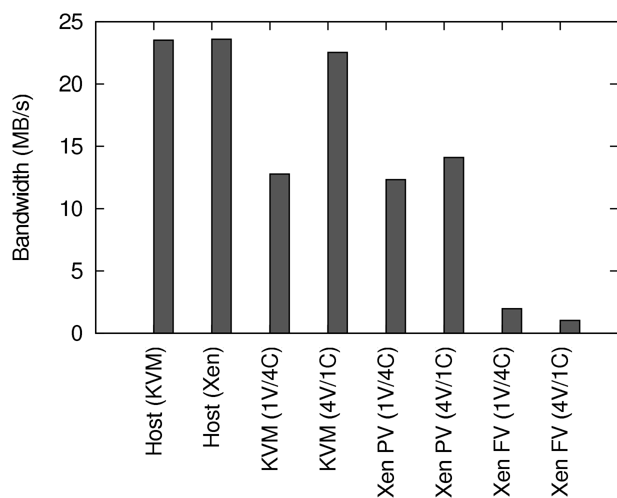



Figure 14: Average node-to-node bandwidth

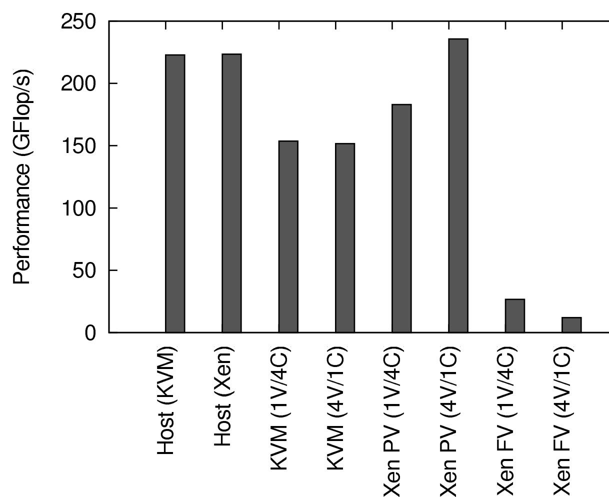



Figure 15: LINPACK benchmark: overall performance

a message to its successor, in a ping-pong manner. 8-byte and 2-MB long messages are used.

Figure 13 presents the latency results. Results for Xen with full virtualization are not included, as the average is 11 ms (1 VM with 4 CPU case) or 8 ms (4 VM with 1 CPU), probably because of the much lower available bandwidth. Xen with paravirtualization performs much better than KVM (146 or 130 $\mu$svs 286 $\mu$s).

Figure 14 presents the bandwidth results, which are similar to those of the PTRANS benchmark.

### 5.4 LINPACK/HPL Benchmark

The LINPACK benchmark combines computation and communications. It is used to determine the annual Top500 ranking of supercomputers. Figure 15 shows the LINPACK results (in GFlop/s) for our configurations.

The best configuration is the Xen setup with 4 virtual machines per node, which reaches 235 GFlops, compared to 222 for the host system. This might be caused by the fact that splitting a physical host into four virtual machines allows for better scalability, compared to when the same kernel is used for the four processes. Also, as we showed in Section 4.3.1, the inter-VM bandwidth is 4.4 Gbps, leading to no performance degradation compared to the host system case.

KVM results show more overhead, likely to be caused by the important communication latency between virtual machines (Section 5.3), which is not compensated by the inter-VM bandwidth. Contrary to what happens

with Xen, both KVM configurations (4 VMs with 1 CPU, and 1 VM with 4 CPU per node) give similar performance.

## 6 Related work

I/O performance is a critical point in virtual machines, and it depends on the kind of virtualization. Previous evaluations compared para-virtualization and full-virtualization like Xen and VMware to OS-level virtualization like VServer[4] and UML.[5] This showed best network and disk access performance for Xen and VServer [5].

Between these two solutions, oftenly VServer performing only control plane virtualization is preferred as in VINI [4] in order to maximise performance. However this offers less isolation, security and reconfigurability, virtualizing at the OS level and so sharing a single OS, while our goal is to have completely isolated systems for more flexibility and security. This lead to concentrate on full- or para-virtualization solutions rather than container based ones.

Xen is probably the most evaluated virtualization solution. Since its appearance in 2003, Xen I/O and especially network virtualization has been constantly improved achieving growing network performance with its successive versions [2] to reach today native Linux throughput on para-virtual machines. Offloading features have been added to virtual NICs in Xen 2.0 and page flipping has been changed to copying to lightweight the operations [11]. Unfairness problems have been corrected in the Credit-Scheduler and the event channel management [14].

Detailed Xen I/O performance has been examined [8] rejecting the Xen data-plane paravirtualization for its performance overhead but proposing Xen virtualization as a viable solution on commodity servers when using direct hardware mapped virtual machines. However, this would not offer the same flexibility requiring dedicated hardware for each virtual machine. Having this isolation and flexibility goal in mind, this paper shows that Xen data-plane virtualization achieves better performance compared to other techniques. In fact, it seems that KVM did not yet reach the same maturity than Xen in I/O management with paravirtualization.

[4] `http://linux-vserver.org`

[5] `http://user-mode-linux.sourceforge.net`

Studies on Xen para-virtualization in the HPC context showed that Xen performs well for HPC in terms of memory acces, and disk I/O [21] and communication and computation [20]. To know if it was due to the paravirtualized driver or to specific Xen implementations, we also compared Xen performance ton KVM performance which is a very recent KVM solutions offering the same isolation features (data-plane virtualization and full isolation) offering also full and paravirtualization.

## 7 Conclusion and future work

In this work, we evaluated different aspects of KVM and Xen, focusing on their adequacy for High Performance Computing. KVM and Xen provide different performance characteristics, and each of them outperforms the other solution in some areas. The only virtualization solution that consistently provided bad performance is Xen with full virtualization. But both Xen with paravirtualization, and the KVM approach (with paravirtualization for the most important devices) clearly have their merits.

We encountered problems when setting up both solutions. Our choice to use Xen 3.3 implied that we had to use the XenSource-provided 2.6.18 kernel, and couldn't rely on an easier-to-use and up-to-date distribution kernel. This brought the usual issues that one encounters when compiling one's own kernel and building software from source. KVM proved to still be a relatively young project (especially its *virtio* support) and also brought some issues, like the unsolved problem with *virtio_disk*.

Finally, while we aimed at providing an overview of Xen and KVM performance, we voluntarily ignored some aspects. The first one is Xen's and KVM's support for exporting PCI devices to virtual machines (*PCI passthrough*). This is important in the context of HPC to give virtual machines access to high-performance networks (Infiniband, Myrinet), but also raises questions on how those devices will be shared by several virtual machines. Another aspect that can be useful in the field of HPC is VM migration, to be able to change the mapping between tasks and compute nodes. In our performance results, we ignored the problem of fairness between several virtual machines: the execution of a task in one VM could have consequences on the other VM of the same physical machine.

Finally, it would be interesting to explore the new virtualization possibilities known as Linux Containers [1]. By providing a more lightweight approach, they could provide a valuable alternative to Xen and KVM.

## Acknowledgements

We would like to thank Pascale Vicat-Blanc Primet for her support and leadership.

## References

[1] Linux containers. http://lxc.sourceforge.net/.

[2] Fabienne Anhalt and Pascale Vicat-Blanc Primet. Analysis and experimental evaluation of data plane virtualization with Xen. In *ICNS 09 : International Conference on Networking and Services*, Valencia, Spain, April 2009.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[4] Andy Bavier, Nick Feamster, Mark Huang, Larry Peterson, and Jennifer Rexford. In vini veritas: realistic and controlled network experimentation. In *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 3–14. ACM, 2006.

[5] Franck Cappello Benjamin Quetier, Vincent Neri. Scalability comparison of four host virtualization tools. *Journal of Grid Computing*, 2006.

[6] David Chisnall. *The Definitive Guide to the Xen Hypervisor*. Prentice Hall, 2007.

[7] Intel Corporation, editor. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, 2008.

[8] Norbert Egi, Adam Greenhalgh, Mark Handley, Mickael Hoerdt, Felipe Huici, and Laurent Mathy. Fairness issues in software virtual routers. In *PRESTO '08*, pages 33–38. ACM, 2008.

[9] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux Virtual Machine Monitor. In *Linux Symposium*, 2007.

[10] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The hpc challenge (hpcc) benchmark suite. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 213, New York, NY, USA, 2006. ACM.

[11] Aravind Menon, Alan L. Cox, and Willy Zwaenepoel. Optimizing network virtualization in xen. In *ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pages 2–2. USENIX Association, 2006.

[12] Jun Nakajima and Asit K. Mallick. Hybrid -virtualization–enhanced virtualization for linux. *Ottawa Linux Symposium*, June 2007.

[13] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. Technical report, Intel Technology Journal, 2006.

[14] Diego Ongaro, Alan L. Cox, and Scott Rixner. Scheduling i/o in virtual machine monitors. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 1–10. ACM, 2008.

[15] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. In *Communications of the ACM 17 (7)*, pages 412–421, July 1974.

[16] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, 2008.

[17] A. Tirumala, F. Qin, J. Dugan, J. Ferguson, and K. Gibbs. iperf : testing the limits of your network. http://dast.nlanr.net/Projects/Iperf.

[18] Willam von Hagen. *Professional Xen Virtualization*. Wiley Publishing, Inc., 2008.

[19] Yaron. Creditscheduler - xen wiki. *http://wiki.xensource.com/xenwiki/CreditScheduler*, 2007.

[20] Lamia Youseff, Rich Wolski, Brent Gorda, and Chandra Krintz. Evaluating the performance impact of xen on mpi and process execution for hpc systems. In *Virtualization Technology in Distributed Computing, 2006. VTDC 2006. First International Workshop on*, 2006.

[21] Lamia Youseff, Rich Wolski, Brent Gorda, and Chandra Krintz. Paravirtualization for hpc systems. In *In Proc. Workshop on Xen in High-Performance Cluster and Grid Computing*, pages 474–486. Springer, 2006.

# I/O Topology

Martin K. Petersen
*Oracle*
martin.petersen@oracle.com

## Abstract

The smallest atomic unit a storage device can access is called a sector. With very few exceptions, a sector size of 512 bytes has been akin to a mathematical constant in the storage industry for decades. That picture is now rapidly changing with hard drives moving to 4KB sectors. Flash-based solid state drives and enterprise RAID arrays also have alignment and block size requirements above and beyond what we have traditionally been honoring.

This paper will present a set of changes that expose the characteristics of the underlying storage to the Linux kernel. This information can be used by partitioning tools and filesystem formatters to lay out data in an optimal fashion.

## 1 Disk Drives and Block Sizes

Until recently what has been commonly referred to as *sector size* has been both the unit used by the programming interface to address a location on disk as well as the size used internally by the drive firmware to organize user data.

In the never-ending quest for increased capacity, disk drive manufacturers are now switching to a 4KB sector size, or *physical block size*. This allows them to increase yield due to less overhead per sector (see Figure 1). I.e. more of the physical capacity can be used for user data as opposed to sync marks, error correction and other fields used internally by the drive firmware.

The industry migration to bigger sectors is just beginning and is scheduled to complete in 2011. Enterprise drives are expected to switch directly to 4KB sectors but can be formatted down to 512-byte blocks at a slight loss in user-visible capacity.

For compatibility with legacy operating systems such as Windows XP, desktop and laptop class disks will continue to present 512-byte sectors to the operating system despite using 4KB blocks internally. This means that we will continue to use increments of 512 bytes to address data on disk. In protocol parlance this is referred to as the *logical block size*. We refer to a sector offset on disk as the *logical block address* or *LBA*.

The backwards compatibility comes at a cost. If the operating system submits a request smaller than 4KB, or if the request submitted is misaligned and straddles two physical blocks, the drive firmware will have to perform a read-modify-write cycle. This incurs a significant performance penalty as the drive will have to perform an extra platter rotation. First the partial sector needs to be read into a buffer, then new data added, and then the same location will need to come back under the head to get written.

For large sequential writes the impact of the read-modify-write cycle is fairly small as only the first and last physical block will be affected. However, small random write workloads are significantly slowed down so it is imperative to prevent misalignment.

## 2 Partitions

Linux normally uses 4KB filesystem blocks and consequently writes smaller than the physical block size are rare. However, we need to make sure that the filesystem is layed out so that the filesystem blocks are aligned with the physical blocks of the underlying disk. Aligning on a 4KB boundary may seem like a trivial task at first but once again backwards compatibility considerations mean that special care must be taken.

Traditionally Linux, like Windows, has been using the DOS partition table format. This format defaults to putting the first partition at sector 63 which is not on

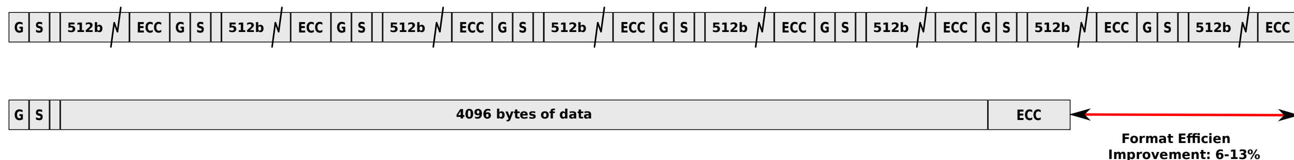



Figure 1: Top line illustrates how 512-byte sectors are stored on disk: Gap, Sync & Address Mark fields, followed by 512 bytes of data and finally an ECC. 8 sectors are required to store 4KB of user data. On the bottom line: A drive with 4KB sectors causes much less overhead.

a 4KB boundary. And consequently all I/O to that partition would be misaligned causing a performance degradation.

Because it is impossible to retroactively change Windows XP to align on something other than sector 63, the drive manufacturers instead opted to ensure that LBA 63 is aligned on the physical block size boundary. This in turn means that the lowest naturally aligned LBA is 7.

There is no guarantee that any subsequent partitions will be aligned on a 4KB boundary, neither from the beginning of the drive, nor from the beginning of the first partition. This means that drive firmware artificially aligning LBA 63 only helps Windows users as XP rarely uses more than one partition. With Linux, however, the first partition is usually used for the `/boot` filesystem where performance is less important.

To make things even more complicated, enterprise class drives are as mentioned above switching to 4KB sectors as well. SAS and Fibre Channel drives will use 4KB logical and physical block sizes. SATA Nearline drives will use 4KB physical and 512-byte logical blocks. In both cases LBA 0 will be naturally aligned, so we can not simply assume that LBA 63 is always aligned on a physical block boundary.

## 3 Alignment and Block Size Reporting

To remedy this both the SCSI and ATA protocols have been expanded with fields that inform us of the drive's alignment. For SCSI drives physical block size and alignment are reported in the `READ CAPACITY(16)` command. And for ATA the same values can be found in `IDENTIFY` words 106 and 209 respectively.

Until now the Linux block layer has used the queue parameter `hardsect_size` to indicate the sector size for an underlying device. In 2.6.31 this value has been deprecated in favor of `logical_block_size` and `physical_block_size` respectively. Both values are exported to user applications via sysfs.

Alignment is also reported, both for the whole block device as well as for each partition. The byte offset from the underlying drive's physical block alignment can be found in sysfs' `alignment_offset` parameter.

These three values enable tools such as `fdisk` and `parted` to align properties on a natural boundary, preventing the read-modify-write cycle and the resulting performance degradation.

## 4 Virtual Block Devices

A significant amount of Linux deployments use either software RAID via the MD driver or logical volume management via the DM driver. In both cases it is crucial to ensure that the virtual block devices exported by the drivers will have their first LBA naturally aligned to the underlying storage.

Virtual block device drivers have traditionally used a stacking function provided by the block layer to ensure that various limits such as the sector size were compatible with the underlying storage. This stacking function has been extended so that alignment and physical block size are taken into account when devices are layered. The drivers can pass in an offset to the stacking function to compensate for space used by their internal superblocks.

The stacking function will make sure that all component devices use compatible alignment and physical block sizes. It even handles corner cases such as combining mismatched devices for example a 512-byte sector drive and a 4KB ditto in a RAID1 setup. In this case the alignment of the 512-byte drive will be scaled up to match that of the 4KB drive.

As it is the case with low-level block devices, both MD and DM will expose the block size and `alignment_offset` parameters in sysfs, meaning that filesystem utilities no longer have to have special cases for extracting this type of information for MD and DM devices. All block devices now export exactly the same set of characteristics.

## 5 Performance Hints

Filesystems such as XFS are designed to be aware of the topology of the underlying storage. When an XFS filesystem is created on top of an MD or LVM device it will query the device to figure out the stripe chunk size as well as the stripe width. XFS will then lay out its important data structures on stripe boundaries.

So far topology reporting has been done using either `ioctl` calls or by wrapping LVM command line tools and parsing their output. However, these approaches are quite inflexible and restricted to virtual block devices.

Hardware RAID arrays have provided means to extract this type of information as well but in a vendor-specific, proprietary fashion. As such it has been up to the system administrator to query the storage device and pass the appropriate layout parameters to the `mkfs` utility.

To remedy this a recent addition to the SCSI Block Commands specification permits devices to export performance characteristics using a common mechanism known as the Block Limits VPD page. Support for this VPD has been added to the SCSI disk driver in 2.6.31 and Linux will now export the values in sysfs if the storage device reports them.

The MD and DM drivers have also been updated and export their respective stripe chunk and stripe width sizes using the same sysfs files. This means that filesystem utilities can gain access to the device performance hints without employing MD and DM specific code.

The first hint is `minimum_io_size` which is the optimal request size granularity for the device, typically the RAID chunk size. A properly aligned multiple of this value is the preferred request size for workloads where a high number of I/O operations per second are desired.

The other hint is `optimal_io_size` which corresponds to the optimal unit of sustained I/O. Typically this is the stripe width for RAID arrays. A properly aligned multiple of this value is the preferred request size for workloads where a high throughput is desired.

## 6 Solid State Drives

Hard drive performance is heavily constricted by the speed at which the read/write heads can be moved across the platter as well as rotational latency. Low-end disk drives spin at 5400 or 7200rpm, whereas enterprise drives spin at 10000 or 15000 rotations per minute, but despite this, a contemporary drive can usually only service between 100 and 250 *IOPS* (I/O operations per second). Once the head is correctly positioned and the platter lined up, however, a modern drive can stream data at a rate in excess of 100 MBps. Consequently, a lot of work has gone into optimizing filesystems and the entire I/O stack to minimize head movement and to place data sequentially on disk.

Flash-based solid state drives are now commonplace in the market. While they look and act like hard drives from a programming perspective they have very different performance characteristics. Because there are no heads to move and no platters to rotate it is possible to achieve a very high degree of parallelism inside the drive. Therefore, an SSD drive has the potential to deliver several orders of magnitude more IOPS than a disk drive.

Internally, flash drives often use page sizes bigger than 512 bytes to organize data, typically 4KB. That makes them similar to harddrives with 512-byte logical, 4KB physical block size. SSD drives can use the same ATA protocol parameters as regular drives to communicate their alignment and block sizes to the kernel.

## 7 Conclusion

Until now the Linux kernel has only been aware of one characteristic of the underlying storage, namely the sector size. Other parameters, such as stripe size and width, have been relegated to special case code in filesystem utilities.

Starting with 2.6.31, the Linux kernel is now aware of more of the hardware capabilities such as physical block size and alignment as well as the extra performance hints exported by some devices. All these characteristics are exported in a common fashion regardless of whether the device is a piece of hardware or a virtual disk exported by MD or DM. The common interface makes it easy for partitioning and filesystem tools, as well as `mdadm` and `dmsetup` to ensure that filesystems and data

are layed out in a way that ensures optimal performance and correctness.

# Step two in DCCP adoption: The Libraries

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 lack of congestion control. DCCP [4] is a new protocol to deliver multimedia in congestion controlled unreliable datagrams.

This paper presents discussions and results in enabling DCCP in open source libraries, as part of our efforts in disseminating the DCCP protocol to developers.

At OLS'08 we presented experimental results [9] using the DCCP implementation in the Linux kernel, where it was shown that DCCP behaves better than UDP in congested environments, while being fair with respect to TCP. This is a work in progress and nowadays DCCP is supported in libraries such as GNU CommonCPP, CCRTP, GNU uCommon, in the GStreamer framework and on Farsight 2.

## 1 Introduction

The Datagram Congestion Control Protocol (DCCP) is a message-oriented Transport Layer protocol that implements reliable connection setup, teardown, ECN, congestion control, and feature negotiation [11]. It was published as RFC 4340 [4] in March of 2006 by Internet Engineering Task Force (IETF) with the main propose of be an Internet protocol for transport multimedia content. In the Linux kernel, the first DCCP implementation was released in version 2.6.14.

The firstly versions of DCCP in the Linux kernel, considering the application developers point of view, was implemented to be used by a very small set of applications, simplest ones based on DCCP socket and not for that advanced multimedia applications. For instance, it was possible to use the socket API to implement a DCCP application to send characters between two hosts. The developers was able to use the common socket functions such as *connect*, *bind* and *accept* in a very similar TCP fashion. By the end of 2007, the DCCP implementation in the Linux kernel became stable and developers began to required DCCP in real development libraries and frameworks.

In the OLS'08 we published a paper and gave a talk discussing about experimental results on the performance of DCCP against UDP and TCP over a wireless network [2]. In that year, we presented that DCCP data flows are fair with respect to others TCP flows, while UDP was very aggressive in terms of network congestion, where in some situations both TCP and DCCP could not transmit any data. This occurs because TCP and DCCP implements congestion control, while UDP does not.

Considering the multimedia application developers requests for providing DCCP in the user space, we have concentrated our efforts on enabling it in a set of selected well-known open source multimedia frameworks. In this paper we present the experiences on enabling DCCP in these frameworks with two goals:

- enable DCCP in the user space to provide the developers an alternative for UDP;
- provide feedback to DCCP developers to improve the DCCP implementation in the Linux kernel.

It is a work in progress and for the first phase we have selected the following libraries: GNU CommonCPP, CCRTP, GNU uCommon, GStreamer framework and Farsight 2. By providing DCCP on these libraries, we aim at disseminating DCCP and making it useful in any Internet applications, while effectively make use of DCCP implementation provided in the kernel – it does not make sense provide DCCP in the Linux kernel and nobody use it.

This paper is organized as follow: in Section 2 are presented overview and background as a base for the rest of the paper. In Section 3 are provided an overview about DCCP and its main features. In Section 4, it is discussed our efforts on enabling DCCP in a set of open source libraries. The current and future works about enabling DCCP in these libraries are described in the Section 7. In Section 8, the conclusions are presented.

## 2 Overview and Background

The motivation for DCCP is based on the growth of Multimedia applications over the Internet in the last few years. The multimedia applications have 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.

Industry and the open source community have developed 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. 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 miles in distance.

These applications have different requirements when compared with application such as HTTP and E-Mail (connection oriented applications). The multimedia applications are delay sensitive, while they make a intensive use of the network bandwidth and tolerate occasional packet loss. Based on the behaviour of the multimedia data flows, this may lead to changes on the design principles of the multimedia application development.

Non-functional requirements such as end-to-end delay (latency) and the variation of the delay (jitter) must be taken into account, regardless the network topology considered [7]. 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 transmitted.

The developers of multimedia applications usually choose to use the UDP protocol for transport the multimedia data. The massive choice for UDP by the multimedia application developers are explained by the fact that UDP introduces less delay in the data transmission in comparison with TCP, for example.

TCP is a connection oriented protocol that provides flow control, congestion control and retransmission of lost packets, which make the protocol appropriate for applications that require reliability during that transmission. Together, these TCP features increase the end-to-end delay during data transmission. Depends on the level of the delay, use TCP is not a best choice.

On the other hand, UDP is a very simple protocol, it does not provide any kind of congestion control, connection hand-shake and packet retransmission in case a packet is lost during the transmission. Together, these features – or the absence of them – in UDP can lead to high levels of network congestion. As a consequence, the network can collapse. In this circumstance, even those TCP (reliable) transmission can become impracticable.

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. DCCP provides a way to gain access to congestion control mechanisms without having to implement them at the Application Layer. It allows for flow-based semantics like in TCP, but does not provide reliable in-order delivery.

## 3 A Bit About DCCP

DCCP was first introduced by Kohler [4] in July, 2001, at the IETF transport group. It provides specific features designed to fulfil 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. DCCP provides a framework that enables addition of new congestion control mechanism, which may be used and specified during the connection handshake, or even negotiated in already established connections. In addition, DCCP 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 [4].

From TCP, DCCP provides 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. This is a UDP limitation that motivated network researchers to specify the STUN [8] (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. Each congestion control algorithm has an identifier called Congestion Control Identifier (CCID). Nowadays there are two standardized congestion control algorithms: CCID 2 [5] and CCID 3 [6].

DCCP is useful for applications with timing constraints on the delivery of data that may become useless to the receiver if reliable in-order delivery combined with congestion avoidance is used. Such applications include streaming media, multiparty online games and Internet telephony. Primary feature of these applications is that old messages quickly become stale, so that getting new messages is preferred than resending lost messages. Currently, such applications have often either settled for TCP or used UDP and implemented their own congestion control mechanisms.

While being useful for these applications, DCCP can also be positioned as a general congestion control mechanism for UDP-based applications, by adding, as needed, a mechanism for reliable and/or in-order delivery on the top of UDP/DCCP. In this context, DCCP allows the use of different – but generally TCP-friendly – congestion control mechanisms [10].

## 4 Libraries

Libraries is a collection of programming functions that can be used to develop a software. A software invokes these functions and, as a result, they provide to the software a return value or take an action. One of the main characteristic of a library is that it provides generic functions that can be shared by a set of software, and each of them combines the library functions with its functions to take actions. By allowing sharing of source code, the use of libraries avoid source code duplication.

In the context of what it is discussed in this paper, the Twinkle [20] soft-phone and Telepathy are two examples of software that adopted the concept explained before. Both projects are free software, one for Voice over IP and the other a library for developing videoconference applications. In the case of Twinkle, it provides many features for communicating: peer-to-peer, conference calls, call redirection, voice mail and instant messaging, all provided by the SIP protocol. The last Twinkle version available provides support for both TCP and UDP, while using Real-Time Protocol (RTP) [3] for signaling audio and video contents. The Figure 1 illustrates two examples of the library sharing. On the top of the stack the Twinkle uses CCRTP, that uses CommonCPP2 and Telepathy, that uses Farsight2 and GStreamer. Both uses the common file socket.h, the standard socket library provided by the operating system. It is the main socket header, where it is found the prototype for the well-known socket functions such as *connect*, *bind*, *send*, *recv* and *accept*.

Telepathy is a framework that can be used to develop communication software, such as VoIP, instant messaging, chat or videoconferencing. It is an open source software, applications use it as a library to simplify the process of developing multimedia applications. Empathy [14], Ekiga [13] and Tapioca [19] are examples of applications that use Telepathy on some of its multimedia service.

The Figure 1 shows a hierarchy of libraries that are used by applications Twinkle and Telepathy. The stack is divided in groups comprising the libraries according to the functionality. Despite the Twinkle and Telepathy are different applications in purpose and use of different libraries, they are at the top of the stack, indicating that they are classified at the highest level for transmitting media streams. The second level of the stack presents the libraries responsible for effectively process the application data, passed though Twinkle or Telepathy, and wrapper them into specific packets based on the protocol used to transport the data.

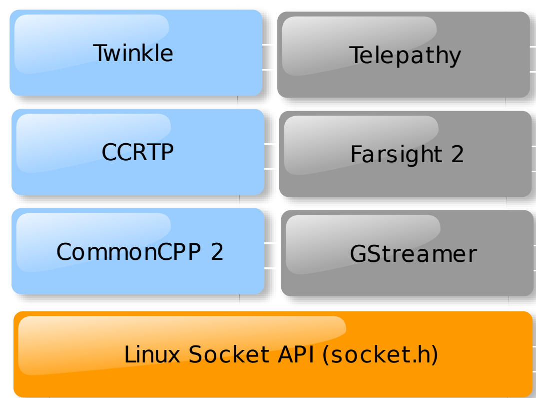



Figure 1: Library usage/sharing between Twinkle and Telepathy

# 5 Enabling DCCP on Libraries

After evaluating the performance of DCCP and presented the results in the OLS' 08, we have started the use of DCCP in the multimedia applications, where Twinkle and Telepathy being considered our starting point. Both of them have a set of libraries dependency and also we also had to change somehow these dependency libraries to accommodate DCCP.

We decided to start the changes in order to enabling DCCP in the selected applications considering the bottom-up approach. In this way, considering the pyramid illustrated in Figure 1, we started from the closest library from the pyramid base to the top of the pyramid. After finishing the work in a specific library, we started to implement DCCP support in the library immediately above and the process was repeated until there is no Twinkle or Telepathy dependencies without DCCP available.

For each libraries modified to make it support DCCP, we have implemented a corresponding example application to test and exploit the features of data transmission using DCCP between pairs. In addition to guide our implementation of DCCP for a given library, this application can be used as a documentation for enabling developer to understand the concepts of the library being used. The example developed in each step was a implementation of a ''hello world" application, where the sender application sends the ''hello world" message and the receiver application receives it.

During the process of changing a certain library, the example application to test the progress of the implementation was continuously executed. This characterized a kind of test-driven development. Once the necessary changes to the libraries were applied, we run the example application and use wireshark [21] to investigate the DCCP traffic transmitted in the network. The Figure 2 shows an example of the DCCP traffic while using DCCP with CommonCPP2. By verifying this, it was possible to certify the the application, through the library that we have provided DCCP support, was indeed transmitting DCCP flows.

## 5.1 Sockets Libraries—First Layer of the Stack

The libraries GStreamer and Commoncpp2, that are in the first layer considering the base stack shown in Figure 1, invokes operating system socket functions. They offer basic functionalities for transmitting data through the connected sockets between a client and a server. In this case, both libraries had to be changed in order to support DCCP, once these libraries only supported TCP and UDP sockets.

The strategy adopted was to add a structure for the DCCP client and server, so that define a connection-oriented sockets. Since GStreamer and CommonCPP uses TCP sockets, we started by coping the TCP implementation, since DCCP and TCP shares the same concept of connection-oriented sockets. Based on TCP implementation, we adapt the code to DCCP parameters passed to the socket functions, such as the *socket* function provided by the operating system. In the next section, we show by using some parts of the code added how we implemented DCCP support in GStreamer and in CommonCPP. Between lines 1-7 of Listing 1, it is shown a set of definitions used to provide DCCP support in the CommonCPP and GStreamer. Between lines 4-7 of Listing 1, it is the constants to read or write DCCP parameters defined by the DCCP implementation in the Linux Kernel. This values are read or written through the functions *getsockopt* and *setsockopt*. For example, the constant DCCP_SOCKOPT_AVAILABLE_CCIDS is passed to *getsockopt* to get the list of CCIDs available in the Linux Kernel. The constant DCCP_SOCKOPT_TX_CCID can be passed either to *getsockopt* or to *setsockopt* to get or set the current CCID, respectively.

In the line 9, it is illustrated how to create a new DCCP socket. Note, IPPROTO_DCCP assumes value 33 because it is the id defined by IANA [18] to DCCP. This

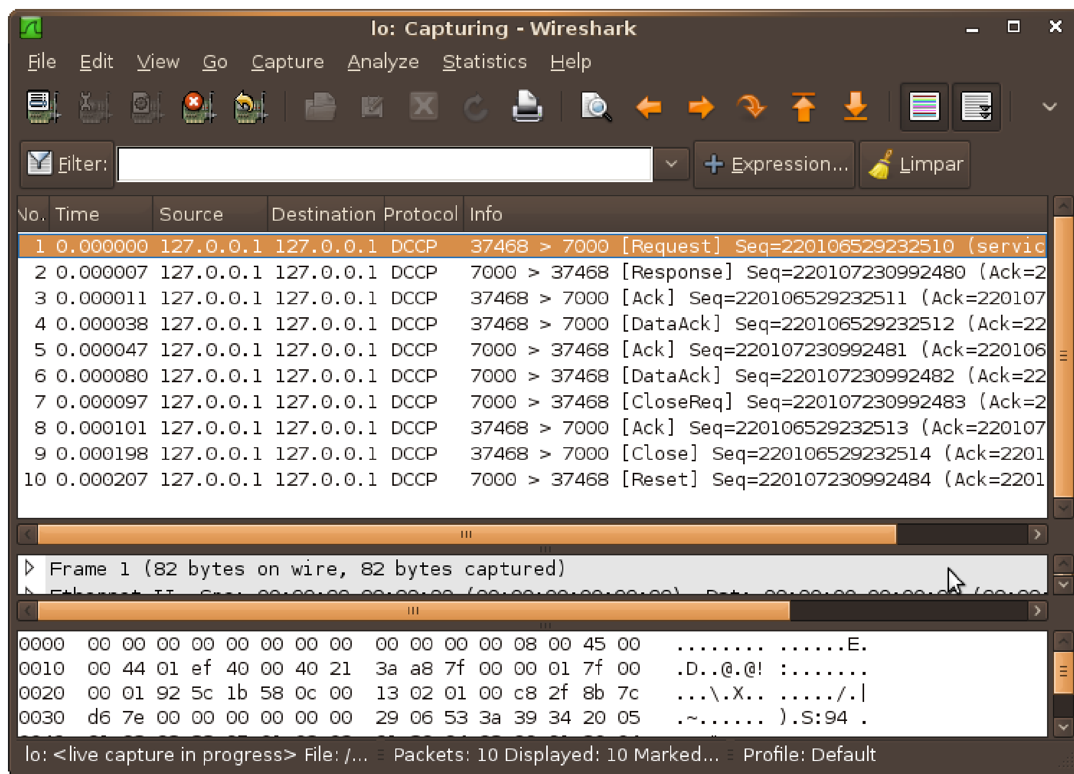



Figure 2: Wireshark filtering DCCP traffic and outputting DCCP packets details.

value is used in the IP packet header to specify which protocol is being used in the transport layer. The common values for this field are 1, 6 and 17 for ICMP, TCP and UDP, respectively. For a complete list of protocol identifier consult reference [17].

```
1 #define SOCK_DCCP                        6
2 #define IPPROTO_DCCP                     33
3 #define SOL_DCCP                         269
4 #define DCCP_SOCKOPT_AVAILABLE_CCIDS     12
5 #define DCCP_SOCKOPT_CCID                13
6 #define DCCP_SOCKOPT_TX_CCID             14
7 #define DCCP_SOCKOPT_RX_CCID             15
8
9 socket(AF_INET, SOCK_DCCP, IPPROTO_DCCP)
```

Listing 1: Definition for DCCP

Before discuss each library that was modified to support DCCP, consider a basic example of DCCP socket shown in the Listing 2. The example was implemented in Python programming language.

```
1 import socket
2
3 socket.SOCK_DCCP = 6
4 socket.IPPROTO_DCCP = 33
5 address = (socket.gethostname(),12345)
6 server = socket.socket(socket.AF_INET,
7           socket.SOCK_DCCP,
8           socket.IPPROTO_DCCP)
9 server.bind(address)
10 server.listen(1)
11 s,a = server.accept()
12 print s.recv(1024)
```

Listing 2: DCCP Server Socket in Python

```
1 import socket
2
3 socket.SOCK_DCCP = 6
4 socket.IPPROTO_DCCP = 33
5 address = (socket.gethostname(),12345)
6 server = socket.socket(socket.AF_INET,
7           socket.SOCK_DCCP,
8           socket.IPPROTO_DCCP)
9 server.bind(address)
10 server.listen(1)
11 s,a = server.accept()
12 print s.recv(1024)
```

Listing 3: DCCP Client Socket in Python

Listing 3 shows the corresponding DCCP client in Python. As it is possible to verify in both client and server examples written in Python, the DCCP socket programming is very simple as TCP socket programming. Basically the unique difference is *socket* func-

tion parameters, where it is necessary to specify IPPROTO=33 (DCCP), as explained before.

The example illustrated in Listing 2 implements a DCCP server that accept a DCCP client connection on port 12345. After connecting, the DCCP server reads 1024 bytes from the DCCP client and exit.

#### 5.1.1 GNU CommonCPP 2

In order to provide DCCP support in CommonCPP, we started by implementing a TCP application to understand CommonCPP API. After making the test application and understand how the CommonCPP works, we investigated the code of the library and located the source codes responsible of handling the sockets by invoking the kernel socket functions. Once located, the TCP implementation code was copied, basically a class named TCPSocket, and modified to create the DCCPSocket class. Listing 4 shows fragments of the DCCPSocket class implemented in the file *src/socket.cpp* of CommonCPP 2 library. The complete code can be found in the CommonCPP repository referred in [15].

After implementing DCCP support for CommonCPP, we have modified the TCP application to make it a DCCP application. Next, we ran the test application and by using wireshark we have validated the implementation by filtering DCCP data packets sent by the test application using the DCCPSocket class.

```
1  \\ Socket class implementation
2     (...)
3  \\ TCPSocket class implementation
4     (...)
5  DCCPSocket::DCCPSocket(const IPV4Address
6                &ia,
7                tpport_t port,
8                unsigned backlog) :
9  Socket(AF_INET, SOCK_DCCP, IPPROTO_DCCP) {
10   struct sockaddr_in addr;
11
12   memset(&addr, 0, sizeof(addr));
13   addr.sin_family = AF_INET;
14   addr.sin_addr = getaddress(ia);
15   addr.sin_port = htons(port);
16   family = IPV4;
17   (...)
18 bool DCCPSocket::setCCID(int ccid) {
19   (...)
20   return (setsockopt(so, SOL_DCCP,
21            DCCP_SOCKOPT_CCID,
22            (char *)&ccid,
23            sizeof (ccid)) >= 0);
24 }
25
26 int DCCPSocket::getTxCCID() {
27   int ccid, ret;
28   socklen_t ccidlen;
29
30   ccidlen = sizeof(ccid);
31   ret = getsockopt(so, SOL_DCCP,
32          DCCP_SOCKOPT_TX_CCID,
33          (char *)&ccid,
34          &ccidlen);
35   if (ret < 0) return -1;
36   return ccid;
37 }
38
39 int DCCPSocket::getRxCCID() {
40   int ccid, ret;
41   socklen_t ccidlen;
42
43   ccidlen = sizeof(ccid);
44   ret = getsockopt(so, SOL_DCCP,
45          DCCP_SOCKOPT_RX_CCID,
46          (char *)&ccid,
47          &ccidlen);
48   if (ret < 0) return -1;
49   return ccid;
50 }
51 (...)
```

Listing 4: Fragments of DCCPSocket class implemented in CommonCPP 2 (src/socket.cpp)

#### 5.1.2 Gstreamer

GStreamer [16] is an open source multimedia framework that allows the programmer to write many types of streaming multimedia applications. Many well-know applications use GStreamer, such as Kaffeine, Amarok, Phonon, Rhythmbox, and Totem. The GStreamer framework facilitates the process of writing multimedia applications, ranging from audio and video playback to streaming multimedia content.

The work initiated by studying the mechanism of data transmission implemented in GStreamer and its concept of plugin-based framework. GStreamer is a plugin-based framework, where each plugin contains elements. Each of these elements provides a specific function – such as encoding, displaying, or rendering data – as well as the ability to read from or write to files. By combining and linking those elements, the programmer can build a pipeline for performing more complex functions. For example, it is possible to create a pipeline for reading

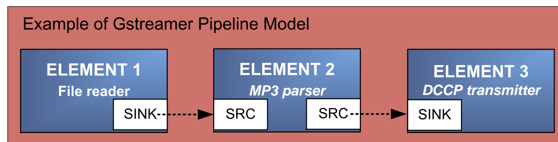



Figure 3: GStreamer Pipeline with three elements: a file reader, an MP3 encoder, and a DCCP transmitter.

from an MP3 file, decoding its contents, and playing the MP3.

Figure 3 represents a GStreamer pipeline composed by three elements. Data flows from Element 1 to Element 2 and finally to Element 3. Element 1 is the source element, which is responsible for providing data to the pipeline, whereas Element 3 is responsible for consuming data from the pipeline. Between the source element and the sink element, the pipeline is permitted to use other elements, such as Element 2 (shown in Figure 3). These intermediary elements are responsible for processing and modifying the content as the data passes along the pipeline.

Based on similar methodology adopted while implementing DCCP support for CommonCPP, we developed the DCCP plugin [9] for GStreamer to deal with data transmission using the DCCP protocol. This plugin has four elements: *dccpserversrc*, *dccpserversink*, *dccpclientsrc*, and *dccpclientsink*. The source elements (*dccpserversrc* and *dccpclientsrc*) are responsible for reading data from a DCCP socket and pushing it into the pipeline, and the sink elements (*dccpserversink* and *dccpclientsink*) are responsible for receiving data from the pipeline and writing it to a DCCP socket.

The *dccpserversrc* and the *dccpserversink* elements behave as the server, but only *dccpserversink* can transmit and only *dccpserversrc* can receive data. When the server element is initialized, it stays in a wait mode, which means the plugin is able to accept a new connection from a client element. The *dccpclientsink* element can connect to *dccpserversrc*, and *dccpclientsrc* can connect to *dccpserversink*.

If a developer wants to send data from the server to the client, you need to use *dccpclientsrc* and *dccpserversink* elements. To send data from the client to server, you need to use the *dccpclientsink* and *dccpserversrc* elements. GStreamer's *gst-launch* command supports the creation of pipelines, and it is also used to debug plugins.

```
gst-launch [!<element> <element params>]+
```

Listing 5: GStreamer gst-launch syntax

Listing 5 illustrates the basic syntax for *gst-launch*. The *gst-launch* command get a list of GStreamer elements with its parameters separated by a exclamation character. Note the ! character, it links the plugin elements, which is similar to the pipe character ("|") very used in the Linux shell prompt. This means that the output of an element is the input to the next specified plugin element.

As an example of the gst-launch command, consider two pipelines to transmit an MP3 stream over the network with DCCP: One works as a DCCP server that streams an MP3 audio file, and the second pipeline is associated with a DCCP client that connects to the remote DCCP server and reproduces the audio content transmitted by the server. To make the example work, you must install GStreamer. In this case, you need the GStreamer-Core, Gst-Base-Plugins, and Gst-Ugly-Plugins packages. Do not worry about the GStreamer installation; GStreamer is a widely used framework available in many Linux package systems for a variety of distributions, such as Debian, Gentoo, Mandriva, Red Hat, and Ubuntu. Once you perform the GStreamer installation, the last step is to compile and install the DCCP Plugin for GStreamer. The Listing 6 shows the command that you can run to install DCCP Plugin for GStreamer, after download it from [12].

```
./autogen --prefix=/usr
make
make install
```

Listing 6: Installing DCCP Plugin for GStreamer

Listing 7 shows a *gst-launch* example that runs a server accepting DCCP connections. Once a client connects, the server starts to stream the audio file named *yourmusic.mp3*. Note that you can specify the CCID with the ccid parameter. This pipeline initializes the server in DCCP port 9011. The server will be waiting for a client to connect to it. When the connection occurs, the server starts to transmit the MP3 stream using CCID-2. The *mp3parse* element is responsible for transmitting a stream. To see more information about mp3parse and the other parameters that are available, run *gst-inspect dccpserversink*.

```
gst-launch filesrc \
  location=yourmusic.mp3 ! \
```

```
mp3parse ! dccpserversink port=9011 \
ccid=2
```

Listing 7: Gst-Launch example starting a DCCP server to stream an mp3 file

Next, start the corresponding client as shown in Listing 8. This GStreamer pipeline initializes the client and connects to the host localhost in port 9011. Once connected, the client starts to receive the MP3 stream, decodes the stream using the *decodebin* element, and pipes the stream to the *alsasink* element, which reproduces the multimedia content in the default audio output device.

```
gst-launch -v dccpclientsrc host=localhost
   port=9011 ccid=2 ! decodebin ! alsasink
```

Listing 8: Gst-Launch example starting a DCCP client to receive an mp3 stream

After implementing the DCCP GStreamer plugin by using socket programming in a similar way done for CommonCPP and validate it using *gst-launch* and *wireshark*, we developed a set of example applications, where client and server applications can stream multimedia content reading from several data sources. For instance, we implemented an application that capture audio from the microphone or from a mp3 file and stream the content to a remote host using DCCP sockets.

The next example shows how to use the GStreamer API to embed DCCP plugin into applications. The application will do the same example explained using *gst-launch*, but this time through the C programming language and GObject, a programming library available for GStreamer application and plugin development. The application creates the same pipeline of the previous examples.

Start by initializing the GStreamer settings, as shown in Listing 9. Note that Listing 9 also defines *GstElements filesrc, mp3parse, and dccpserversink.*

```
#include <string.h>
#include <math.h>
#include <gst/gst.h>

int main(int argc, char **argv) {
  GMainLoop *loop;
  GstElement *pipeline, *filesrc;
  GstElement *mp3parse, *dccpserversink;
  GstBus *bus;

  gst_init(&argc, &argv);
  loop = g_main_loop_new (NULL, FALSE);

  if (argc != 3) {
    g_print("Usage: %s port mp3_location",
            argv[0]);
    return -1;
  }
  return 0;
}
```

Listing 9: Initializing GStreamer Pipeline

The next step is to instantiate a bus callback function to listen to GStreamer pipeline events. A bus is a system that takes care of forwarding messages from the pipeline to the application. The idea is to set up a message handler on the bus that leads the application to control the pipeline when necessary. Put the function shown in Listing 10 above the main function defined in Listing 9.

```
static gboolean bus_event_callback (
  GstBus *bus, GstMessage *msg,
  gpointer data) {

  GMainLoop *loop = (GMainLoop *) data;
  switch (GST_MESSAGE_TYPE (msg)) {
    case GST_MESSAGE_EOS:
      g_print("End-of-stream\n");
      g_main_loop_quit(loop);
      break;
    case GST_MESSAGE_ERROR:
      gchar *debug;
      GError *err;
      gst_message_parse_error(msg, &err,
                              &debug);
      g_free(debug);
      g_print("Error: %s\n",
              err->message);
      g_error_free(err);
      g_main_loop_quit(loop);
      break;
    default:
      break;
  }
  return TRUE;
}
```

Listing 10: Defining GStreamer Bus Event Callback

Every time an event occurs in the pipeline, GStreamer calls the gboolean bus_call function. For example, if you implement a GUI interface for your application, you can show a message announcing the end of the stream or deactivate the GUI stop button when the type of the GStreamer bus message is GST_MESSAGE_EOS. Now comes the most important part of this example—defining the elements and building the GStreamer

pipeline. Insert the code shown in Listing 11 into the main function, after checking the parameter count.

```
pipeline = gst_pipeline_new
           ("dccp-audio-sender");
filesrc = gst_element_factory_make
           ("filesrc", "file-source");
mp3parse = gst_element_factory_make
           ("mp3parse", "mp3parse");
dccpserversink = gst_element_factory_make
           ("dccpserversink",
            "server-sink");
```

Listing 11: Defining GStreamer Elements

Listing 11 first instantiates a new pipeline, dccp-audio-sender, which can be used for future references in the code. Then the code instantiates the *filesrc* element with the name file-source. This element will be used to read the specified MP3 file as an argument of the application. Use the same process to instantiate the elements *mp3parse* and *dccpserversink*. Once all the necessary elements are instantiated, certify that all are properly loaded. For this case, proceed as shown in Listing 12.

```
if (!pipeline || !filesrc ||
    !mp3parse || !dccpserversink) {
  g_print("Element(s) not instantiated");
  return -1;
}
```

Listing 12: Checking GStreamer Elements

The next step is to set the respective element parameters, as shown in Listing 13. For this application, we need to set two parameters: the *port*, where the server will listen and accept client connection from, and the audio file path represented by the parameter *location*.

```
 g_object_set (G_OBJECT (dccpserversink),
           "port", atoi(argv[1]), NULL);
 g_object_set (G_OBJECT (filesrc),
           "location", argv[2], NULL);
```

Listing 13: Setting Elements Parameters

Once all the elements are instantiated and the parameters are defined, it is time to attach the bus callback defined in Listing 10 to the bus of the pipeline. Also, it is need to add the elements to the pipeline and link them, as shown in Listing 14.

```
bus = gst_pipeline_get_bus
       (GST_PIPELINE(pipeline));
gst_bus_add_watch (bus,
       bus_event_callback, loop);
gst_object_unref(bus);
gst_bin_add_many(GST_BIN (pipeline),
       filesrc, mp3parse, dccpserversink,
       NULL);
gst_element_link_many(filesrc, mp3parse,
       dccpserversink, NULL);
```

Listing 14: Linking GStreamer Elements (Server)

Listing 15 shows how to execute the pipeline. Note that GStreamer runs in a main loop (line 5). This means that when this main loop finishes—for example, when the user types Ctrl+C—it is necessary to do some clean up (lines 6 and 10).

```
g_print("Setting to PLAYING\n");
gst_element_set_state
   (pipeline, GST_STATE_PLAYING);
g_print("Running\n");
g_main_loop_run(loop);
g_print("Returned, stopping playback\n");
gst_element_set_state
   (pipeline, GST_STATE_NULL);
g_print("Deleting pipeline\n");
gst_object_unref(GST_OBJECT (pipeline));
```

Listing 15: Executing the GStreamer Pipeline (Server)

The easiest part is to compile the server application—just run the command, which will link the GStreamer libs with the example application, that is in Listing 16. To run the DCCP GStreamer Server execute the command in the line 4 of the Listing 16.

```
$ gcc -Wall $(pkg-config --cflags \
    --libs gstreamer-0.10) \
    -o gst_dccp_server gst_dccp_server.c
$ ./gst_dccp_server 9011 yourmusic.mp3

$ gcc -Wall $(pkg-config --cflags
   --libs gstreamer-0.10)
    gst_dccp_client.c -o gst_dccp_client
$ ./gst_dccp_client localhost 9011
```

Listing 16: Compile and run server and client examples

Note that the example uses port 9011, which the server will use to open the DCCP socket and transmit the stream through the network to the remote DCCP client. Now it is time to build a corresponding client application that acts just like the gst-launch client command discussed previously. The DCCP client application is similar to the server application (Listing 17). Basically, you must initialize GStreamer, check command-line parameters, instantiate the necessary elements, and link them to build the GStreamer pipeline. Finally, to compile and

run the client application, execute the commands of the line 6 and 9 of the Listing 16.

```
 1 #include <string.h>
 2 #include <math.h>
 3 #include <gst/gst.h>
 4
 5 static gboolean bus_event_callback
 6   (GstBus *bus, GstMessage *msg,
 7   gpointer data) {
 8   GMainLoop *loop = (GMainLoop *) data;
 9   switch (GST_MESSAGE_TYPE(msg)) {
10     case GST_MESSAGE_EOS:
11       g_print("End-of-stream\n");
12       g_main_loop_quit(loop);
13       break;
14     case GST_MESSAGE_ERROR:
15       gchar *debug;
16       GError *err;
17       gst_message_parse_error(msg, &err,
18          &debug);
19       g_free(debug);
20       g_print("Error: %s\n",
21          err->message);
22       g_error_free(err);
23       g_main_loop_quit(loop);
24       break;
25     default:
26       break;
27   }
28   return TRUE;
29 }
30
31 int main(int argc, char *argv) {
32   GMainLoop *loop;
33   GstElement *pipeline, *dccpclientsrc;
34   GstElement *decodebin, *alsasink;
35   GstBus *bus;
36
37   gst_init(&argc, &argv);
38   loop = g_main_loop_new(NULL, FALSE);
39   if (argc != 3) {
40      g_print("Usage: %s host Port\n",
41              argv[0]);
42      return -1;
43   }
44
45   pipeline = gst_pipeline_new(
46              "audio-sender");
47   dccpclientsrc = gst_element_factory_make
48             ("dccpclientsrc",
49              "client-source");
50   decodebin = gst_element_factory_make
51                ("decodebin", "decodebin");
52   alsasink = gst_element_factory_make
53                ("alsasink", "alsa-sink");
54   if (!pipeline || !alsasink ||
55       !decodebin || !dccpclientsrc) {
56      g_print(
57       "Element(s) not instantiated\n");
58      return -1;
59   }
60
61   g_object_set(G_OBJECT(dccpclientsrc),
62               "host", argv[1], NULL);
63   g_object_set(G_OBJECT(dccpclientsrc),
64               "port", atoi(argv[2]), NULL);
65   gst_bin_add_many(GST_BIN(pipeline),
66               dccpclientsrc, decodebin,
67               alsasink, NULL);
68   gst_element_link_many(dccpclientsrc,
69               decodebin, alsasink, NULL);
70   bus = gst_pipeline_get_bus
71               (GST_PIPELINE(pipeline));
72   gst_bus_add_watch(bus,
73               bus_event_callback, loop);
74   gst_object_unref(bus);
75
76   g_print("Setting to PLAYING\n");
77   gst_element_set_state(pipeline,
78                   GST_STATE_PLAYING);
79   g_print("Running\n");
80   g_main_loop_run(loop);
81   g_print(
82      "Returned, stopping playback\n");
83   gst_element_set_state(pipeline,
84                   GST_STATE_NULL);
85   g_print("Deleting pipeline\n");
86   gst_object_unref(GST_OBJECT(pipeline));
87   return 0;
88 }
```

Listing 17: DCCP Client source code

## 6 Contributions with the Open Source

In addition to the implementation of DCCP support in the libraries mentioned before, we have also provided some additional contributions while developing the support for DCCP in that libraries. For example, during the development of the DCCP GStreamer plugin, we have noticed that DCCP implementation in the Linux kernel did not provide a mechanism for reading how much bytes is available in the receiving buffer in a given moment of the DCCP connection. We have reported this missing to the DCCP developers, while we contribute with them by implementing and testing this feature in the Linux Kernel. The summary of the patch and the patch itself is available from [1].

We have also contributed to the GStreamer and CommonCPP projects by providing DCCP support patches. Nowadays, both projects officially support DCCP protocol.

## 7 Current and Future works

Nowadays we are in a working in progress to provide DCCP support in Farsigh2 and CCRTP. We are developing a testing application for video conferencing between hosts. For both APIs we have started the process of adding the DCCP support for connection-oriented services. We are in constant contact with the Farsight2 and CCRTP developers. They are helping us to implement DCCP support on them.

For future works, we will provide DCCP support in MPlayer and finalize DCCP support in the Twinkle softphone.

## 8 Conclusion

We have presented the basic concepts of DCCP, the process of enabling DCCP in CommonCPP and in the GStreamer and how to build a DCCP-based application using the GStreamer DCCP plugin. The way how DCCP was implemented in the Linux kernel allowed us to rapidly implement DCCP support in the user-space API, like GStreamer and CommonCPP. The contributions that we have provided in this work will enable new DCCP applications, enabling alternatives for UDP protocol based applications.

Network analysis and testing applications, such as TTCP, tcpdump, and Wireshark already provide support for the DCCP protocol, and multimedia tools such as the open source VLC player accommodate DCCP streaming. As multimedia developers become aware of its benefits, it can expect to hear more about DCCP in the coming years.

## References


[1] Arnaldo Carvalho de Melo, Leandro Melo de Sales, Ian McDonald, and David S. Miller. Implement siocinq/fionread. `http://git.kernel.org/?p=linux/kernel/git/davem/net-2.6.git;a=commitdif%f;h=6273172e1772bf5ce8697bcae145f0f2954fd159`. Last access on July 2009.

[2] Leandro Melo de Sales, Hyggo Oliveira de Almeida, Angelo Perkusich, and Arnaldo Carvalho de Melo. Measuring DCCP for Linux Against TCP and UDP With Wireless Mobile Devices. In *Ottawa Linux Symposium 2008*, volume 1, pages 163–177, 7 2008.

[3] J. Du, D. Putzolu, L. Cline, D. Newell, M. Clark, and D. Ryan. An Extensible Framework for RTP-based Multimedia Applications. In *Proceedings do 7th International Workshop on Network and Operating System Support for Digital Audio and Video*, volume 1, pages 53–60, 1997.

[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 July 2009.

[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 July 2009.

[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 July 2009.

[7] James F. Kurose and Keith W. Ross. *Computer Networks and the Internet: A New Approach*. Addison Wesley, 2 edition, 9 2005.

[8] 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 July 2009.

[9] Leandro Sales, Hyggo Almeida, and Angelo Perkusich. The DCCP Protocol in Three Steps. *Linux Magazine*, (92):56–62, 12 2008.

[10] Leandro M. Sales, Hyggo O. Almeida, Angelo Perkusich, and Marcello Sales Jr. An Experimental Evaluation of DCCP Transport Protocol: A Focus on the Fairness and Hand-off over 802.11g Networks. In *Consumer Communications and Networking Conference Proceedings*, pages 1149–1153, 1 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] E-Phone Team. Dccp plugin for gstreamer. `https://garage.maemo.org/projects/ephone`. Last access on July 2009.

[13] Ekiga Team. Ekiga - open source voip and video conferencing application. `http://ekiga.org/`. Last access on July 2009.

[14] Empathy Team. Empathy - instant-messaging. `http://live.gnome.org/Empathy`. Last access on June, 2009.

[15] GNU CommonCPP Team. Commoncpp source code repository. `http://savannah.gnu.org/projects/commoncpp`. Last access on July 2009.

[16] GStreamer Team. Gstreamer - library for constructing graphs of media-handling components. `http://www.gstreamer.net/`. Last access on July 2009.

[17] IANA Team. Iana - assigned internet protocol numbers. `http://www.iana.org/assignments/protocol-numbers/`. Last access on July 2009.

[18] IANA Team. Iana - internet assigned numbers authority. `http://www.iana.org/`. Last access on July 2009.

[19] Tapioca Team. Tapioca - provides a set of convenience libraries to easily integrate voip and im. `http://tapioca-voip.sourceforge.net/wiki/index.php/Tapioca`. Last access on July 2009.

[20] Twinkle Team. Twinkle - softphone for your voice over ip and instant messaging communcations using the sip protocol. `http://www.twinklephone.com/`. Last access on July 2009.

[21] Wireshark Team. Wireshark - the worldś foremost network protocol analyzer. `http://www.wireshark.org/`. Last access on July 2009.

# Programmatic Kernel Dump Analysis On Linux

Alex Sidorenko
*Hewlett-Packard*
asid@hp.com

## Abstract

Companies providing Linux support rely heavily on kernel dumps created on customers' hosts. Kernel dump analysis is an art and it is impossible to make it fully automatic. The standard tool used for dump-analysis, 'crash', provides a number of useful commands. But when we need to enhance it or to analyze several thousand similar structures, we need programmatic API.

In this paper we describe Python bindings to *crash*[1] and compare it to C-like SIAL extension language. After a general framework discussion we look at some practical tools developed on top of PyKdump, such as `xportshow`. This tool works on kernels 2.4.21-2.6.28 and provides many useful features, such as printing routing tables, emulating `netstat` and summarizing networking system status.

## 1 Why Do We Need Programmatic API?

- adding new features and enhancing functionality quickly
- there are have too many structures to look through all of them ourselves
- running a number of useful tests — each of them can be executed manually, but there are many of them
- running programs on a customer's site if for some reason he cannot send us vmcores
- we can use an already developed tool on live kernels instead of writing new DLKM or Systemtap script

An ability to run scripted tests quickly is extremely important for support organizations. Even though in theory customers should provide a detailed description of the problem, in reality it is not unusual to get vmcore with just the generic description of "the host is unresponsive."

It can mean many different things, for example:

- a critical userspace application (e.g. Oracle) stopped responding
- network connectivity is lost
- the system is just overloaded
- the system is out of memory
- there is a bug in the kernel leading to CPUs executing kernel code forever, with interrupts disabled

In such cases it makes sense to run a number of standard tests to narrow down the problem. For example:

- how much memory is used and whether it is fragmented
- check load averages and runqueues (e.g. are there any RT processes)
- when was the last time NICs transmitted and received data
- is syslogd hanging (this will make all processes doing `syslog()` unresponsive)

It makes sense to run all such tests programmatically to save time and effort. Furthermore, even those lacking the proper skills to do dump analysis themselves can run automated tasks.

[1] `http://sourceforge.net/projects/pykdump`

### 1.1 Extensions Available for Crash

***Crash*** [1] is a standard tool used for dump analysis. There is similar another tool, ***lcrash*** but we will not discuss it here.

***Crash*** can be dynamically extended by writing programs in C and linking them in a special way. After that the extensions can be loaded/unloaded by using builtin `extend` command. But developing in C is rather time-consuming and unpractical, especially if we need to write a custom code quickly. It is much better to use special extensions providing bindings of ***crash*** to higher-level languages. Using such an "extension language," we can develop new programs quickly without a need to compile/link every time we need to modify our script. Here are some known extension languages:

- SIAL–C-like language. Very handy for writing small tools, but problematic for big projects. Is included as part of ***crash*** distribution
- Alicia–Perl wrapper driving ***crash*** via stdin, retrieving results from stdout. Quite slow, as a result. There was no activity for this project on SF site during last 3 years
- *PyKdump*–Python bindings to ***GDB/crash*** internals

From these three frameworks, SIAL is probably the easiest to use for kernel hackers as they already know C. However, *PyKdump* provides a number of features that makes it better than SIAL for big projects:

- better scalability–a program can be split into many files and/or libraries and loading/execution time is reasonable even for huge programs
- Python standard library is extremely powerful
- Python is a high-level language with efficient lists, dictionaries and other useful classes
- error processing is easier because of exception mechanism
- more features for making runtime decisions based on symbolic info from vmlinux
- an ability to run ***crash*** commands and parse their output efficiently

### 1.2 Writing Programs That Work With Different Kernel Revisions

Linux kernel is a moving target. The definitions of kernel structures, global variables and algorithms are different from version to version. If we want to write a program that works for kernel dumps obtained from different kernels, this needs to be taken into account. Some possible approaches are:

- check for kernel version explicitly, use a different code for different versions
- check whether certain global variables exist
- check whether a structure has a specific fieldname

This means that we need to make runtime decisions. C is a strongly typed language:the variable type needs to be explicitly declared and cannot be changed afterwards. Let us consider the following case. There is a global variable. In an older kernel it was declared as

```
struct one var;
```

In a newer kernel the name of the struct has been changed even though its definition is the same:

```
struct two var;
```

We want to access `var.field`

In C-like languages (e.g. SIAL) a possible approach is the following:

```
{"LINUX_2_2_16",
   "(LINUX_RELEASE==0x020210)"},
{"LINUX_2_2_17",
  "(LINUX_RELEASE==0x020211)"},
{"LINUX_2_4_0",
   "(LINUX_RELEASE==0x020400)"},
...
```

Then in some include file crossSupport.h:

```
#if LINUX_2_6_X
    #define TYPEX struct one
#else
    #define TYPEX struct two
#endif
```

Then in the code:

```
#include <crossSupport.h>
void func(...)
{
TYPEX *s=(TYPEX *)var;
    if(var->field ...) {

    }
}
```

This is not very elegant and is rather unreliable. Most commercial distributions base their major release on a specific kernel and then backport bugfixes/features from recent kernels as needed. As a result, variables and structures definition on 2.6.9-based RHEL4 might change even though the kernel is still reported as 2.6.9. A better approach would be to retrieve variable types from vmlinux and use them as they are. In *PyKdump* we can do the following:

```
var = readSymbol("var")
f = var.field
```

Another approach is to base runtime decisions on explicit type information. That is, to check whether a struct has a specific member or what its type is. At this moment SIAL lacks this functionality but it might be added in the future.

## 2 PyKdump Design

Python is a very powerful and extremely popular programming language, at least among userspace application developers. Unfortunately, many kernel hackers only know well C and assembler. There are excellent books available and outstanding documentation provided on Python website [2]. But the syntax of Python operators is close enough to C, so there should be no problem in understanding all examples provided in this paper even for those who know nothing about Python.

### 2.1 Mapping C-structures To Python Objects

The Linux kernel is written in C (plus a bit of assembly). To be able to write useful dump-analysis scripts easily, we need as a minimum:

- to be able to read memory, global variables and struct/union contents
- to be able to write Python code easily looking at related C-sources

For example, if we want to write a program printing routing tables from a dump, we start by looking at its kernel implementation of related /proc routines. It would be convenient to be able to copy and paste pieces of related C-sources to our script, but even with SIAL (using C-like syntax) this does not always work.

While developing Python bindings to ***crash*** internals we used the following approach:

- we map C `struct` and `union` by creating Python objects with attributes matching the respective C field names
- we map other C types to Python types that are close, e.g. C `int` to Python `integer`
- we map C operators to similar Python operators

Python passes everything by reference, there are no pointers. As a result, there are no `*`, `->`, and `&` operators. It is easy to mimic reading and accessing fields of C struct/union in Python as both C and Python have the dot `.` operator:

```
struct blk_major_name {
   struct blk_major_name *next;
   int major;
   char name[16];
} svar;
```

```
s = readSU('struct blk_major_name', addr)
major = s.major
print "%3d       %-11s" % (major, s.name)
```

Here we read `struct blk_major_name` from a given address and print the `major` field. Python has many built-in data types, including integers, floating-point numbers and strings. We return properly typed values automatically, without specifying the type explicitly every time. There is no special pointer type in Python but we can represent pointers by integers. In the example above we expect to get

- s.next as an integer
- s.major as an integer
- s.name as a string

There are some problems with this approach. If we meet `char name[10]` declaration, how do we know whether it is intended to be used as a string or an array of 1-byte integers? We cannot know this from the symbolic information available in vmcore. To work around this, we introduce a special 'SmartString' type which mimics null-terminated strings but lets you access info as if it was a normal array. So if `name` is a SmartString, printing it will result in truncation on NULL byte but we still be able to access any byte using array access:

```
name="abc\0\5\6\7\8\9\10"
print s.name  # will print abc
print s.name[5] # will print 5
```

In most cases you can work with these SmartStrings just like with normal Python strings, but sometimes Python library functions check type explicitly (e.g. you cannot pass *SmartString* to regular expressions functions). You can convert `SmartString s.name` to a normal string using `str()` function, e.g.

```
str(s.name)
```

By default, struct/union members that are defined as char pointers or char arrays, are returned as *SmartString* type. If they have explicit *signed* or *unsigned* specifiers, they are returned as integer arrays.

## 2.2 Dereferencing Pointers in Structs and Unions (Emulating * and -> Operators)

What if we want to follow the 'next' pointer in the example above? The attribute dereference operator `->` in C is really just a syntax sugar that combines pointer dereference with attribute access:

```
/* The same as (*svar).next  */
svar->next;
/* The same as (*(*svar).next).next */
svar->next->next;
```

There is neither `*` nor `->` operators in Python but we still can dereference using alternative approaches. For example, for a pointer dereference we can use `Deref()` function. In C:

```
struct blk_major_name *sptr;
int major = (*sptr).major;
in major1 = sptr->major;
```

In Python (assuming that sptr is an object representing a pointer to structure):

```
major = Deref(sptr).major  # Approach 1
major1 = sptr.major        # Approach 2
```

Please note how we used the dot operator without dereferencing first. In C, it would have failed at the compilation stage. In *PyKdump*, the framework finds that an object is a pointer to a structure, so obviously the dot operator is not a simple field dereference. Consequently it interprets it as `->`. That is, you can use the dot operator in both cases and it will be used in whatever way is needed automatically. For example:

```
/* in C */
s->f1.f2->f3.f4-f5

# In Python
s.f1.f2.f3.f4.f5
```

In C, using dot operator on a pointer would trigger a compilation error. In Python, we make life easier by trying to interpret the dot operator either as `.` or `->`, depending on the object type.

More than that, in *PyKdump* pointers to structures and structures themselves have the same object type. It is similar to Java's approach where we have just references and no pointers.

Please note that the description above is correct only for pointers to structures. Pointers to any other type are represented with a different object class. In particular:

```
/* in C */
struct test *sptr;
struct test **pptr;

# in Python
# sptr is the same as Deref(sptr)
# (the same type), so you can write
sptr.f1

# pptr is completely different,
# Deref(pptr) is not the same as pptr
Deref(pptr).f1
```

To emulate the missing features we can define special attributes (usually called "properties" in OOP). Accessing such an attribute triggers a function call. A potential problem exists in the shape of name collision between internal object attributes and C-attributes as mapped to Python. Luckily, this is a highly improbable event for kernel structures. The "Linux Coding Style" document [3] says: "mixed-case names are frowned upon" so using mixed-case attributes for our own purposes should be safe enough. The "internal" methods of Python classes are all named like `__aname__` and, to reiterate, we have never seen name collision between field names of Linux kernel structures and Python internals.

### 2.3 Emulating & Operator

We can get the address of a global variable using `sym2addr()` function, e.g.

```
addr = sym2addr("init_task")
```

In other cases we start from a struct/union and need to find the address of its member. For example, we have a field which is defined as a struct (not a pointer), e.g.

```
type = struct task_struct {

   volatile long int state;

   ...

   struct list_head tasks;

}
```

When we access `tasks` attribute, we obtain an object representing a structure. For such objects we can use Addr(obj) function to obtain the associated address, e.g.

```
init_task = readSymbol('init_task')
init_task_saddr = Addr(init_task.tasks)
```

This works for objects representing aggregates, strings or pointers as they store the needed address internally. However, for integer/floating type the objects are just native Python integers/floats, there is no address. At this moment the only way to get the address of such a field is to compute it manually using low-level functions, e.g.

```
dev_base = readSymbol("dev_base")
off = member_offset("struct net_device",
                    "next")
addr_next = Addr(dev_base) + off
```

In the future we might wrap integers and floats so that `Addr()` will work for them as well - but this is not implemented yet. (The main reason for this is that we still need to evaluate the impact of these extra wrappers on performance).

### 2.4 Some Special Types

We map all C integer types to Python native `long integer` type and we map C struct/union instances to `class StructResult` instances. Pointers and arrays are normally represented by Python integers and lists. But in some cases we would like to preserve additional information while returning values. As a result, we wrap integers and strings in Python classes. Please note that usually you do not construct objects of these types yourself, as they will be initialized and returned as needed when using readSymbol and similar functions. Two most important cases are tPtr (to represent pointers to different C-types) and SmartString (used for C `char *` pointers and char arrays). When you obtain these objects from readSymbol, they internally store additional useful information.

#### 2.4.1 StructResult

This type represents struct/union and a pointer to struct/union. In kernel sources we usually don't need to "read" structures as they are already in memory. So if we want to access a structure at a given address, we just use the cast operator, e.g.

```
struct sock *s = (struct sock *)addr;
```

In *PyKdump* we read them (ultimately reading bytes from vmcore file...):

```
# read from address addr
s = readSU("struct sock", addr)
# similar to C s.socket
socket = s.socket
# similar to C &s
addr = Addr(s)
```

### 2.4.2 tPtr - A Typed Pointer

When a variable is a pointer, it is an *integer* (address) plus type information.

*tPtr* class inherits from *long* so it can be used as a normal *long* integer. For example, it's OK to use it in arithmetical expressions. In some rare cases the library functions check for type of passed object explicitly. You can always convert *tPtr* to a plain long integer by doing conversion explicitly,

```
i = long(tptr)
```

Please note that at this moment this class is intended mainly for internal use. Objects of this type are returned as needed, but you should not attempt to create them yourself.

The main reason for needing this special type is preserving information while reading global variables (see the description of *readSymbol*).

### 2.4.3 SmartString

This type is used to represent variables and structure fields declared in C as `char *` or `char []`. This is a subclass of the standard Python string, with additional data attached and some methods redefined. We subclass to preserve the pointer value and address of the variable. We might need this to access the pointer itself if `char *` is used as a generic pointer instead of more correct `void *`. Another use for it is mapping char arrays where they are used to store byte values, not to represent an ASCII string. For example, in sources a variable declared like this:

```
char *testvar;
```

If the first 7 bytes of it are `abc\0def` it probably makes sense to interpret it as an ASCII string `abc`. In most cases this would be acceptable, but sometimes we need to access other bytes. We'll be able to do the following in Python:

```
# Read the variable, return
# SmartString object
s = readSymbol("testvar")
# Print this string using C
# NULL-terminated convention (i.e."abc")
print s
# print 2 chars after NULL
print s[4:6]
# Print the address of testvar
print Addr(testvar)
# print the pointer value
print long(testvar)
```

By default, `readSymbol()` reads and stores just the first 256 chars. If you need to read more, you can use the pointer value (retrieved as shown above).

### 2.4.4 Supporting Different Kernels

We have already discussed this briefly in 1.2. Now let us revise it by looking at some examples from real program.

Each object representing a struct/union has a number of attributes, mapped from C. In addition to those attributes, we can add our own. This is usually a sensible approach to isolate the dependencies on a specific kernel. For example, in some kernels `spinlock_t` is declared as

```
typedef struct {
    volatile unsigned int lock;
#ifdef CONFIG_DEBUG_SPINLOCK
    unsigned magic;
#endif
} spinlock_t;
```

but in others the access to a field similar to `lock` is more complicated:

```
typedef struct {
unsigned int slock;
} raw_spinlock_t;

typedef struct {
    raw_spinlock_t raw_lock;
#ifdef CONFIG_GENERIC_LOCKBREAK
    unsigned int break_lock;
#endif
#ifdef CONFIG_DEBUG_SPINLOCK
    unsigned int magic, owner_cpu;
    void *owner;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
    struct lockdep_map dep_map;
#endif
} spinlock_t;
```

We can declare a new attribute that will be equivalent to `lck.lock` for the first kernel but to `lck.raw_lock.slock` for the second kernel. We do this in the following way:

```
sn = "spinlock_t"
structSetAttr(sn, "Slock",
              ["raw_lock.slock", "lock"])
```

This should be called only once. At the moment when this is executed, the framework traverses the list `["raw_lock.slock", "lock"]` and verifies whether the needed structures/fields exist, that is — does a dereference chain specified in a list element make sense? As soon as we find a match, we add this attribute (implemented as a "property" to the class to be used to represent this typedef). If no match was found, `structSetProcAttr` returns `False`. In case of success, later we can do the following:

```
sl = lck.Slock
```

The result will be the value of the function associated with that attribute—there will be no runtime overhead for checking structure definitions. The default function returns the value for the dereference chain that matched. In addition to this, we can specify an alternative function, for example:

```
# Programmatic attrs
def getSrc6(tw):
    # Some code...
    #
    # which returns this
    return val

sn = "struct tcp_timewait_sock"
structSetProcAttr(sn, "Src6", getSrc6)
```

In this case every time we use `tw.Src6` where `tw` is a result of the type `struct tcp_timewait_sock`, the function `getSrc6` will be used

### 2.5 Performance

The performance of Python language itself is more than adequate for our purpose. Python uses two-stage process:

- compile to pseudocode (and write results to files)
- execute pseudocode using a virtual machine

This is similar to Java. There are two JIT compilers to further increase the performance but they are still rather experimental and not ready for production. Still, the performance is excellent — summing 10 million integers in a loop takes less than 3s on a 2-year old laptop. The main performance bottleneck is due to the fact that *PyKdump* sources are compiled without any knowledge about symbolic information from the kernel. This is good as everything is compiled in advance. Even if we write a huge (> 100,000 lines) program, it will be compiled once for all kernels and start time in ***crash*** will be reasonably small.

SIAL uses a different approach: the compilation to pseudocode is done while loading the script. This means that if SIAL script is huge (and it is unclear how scalable is SIAL), loading it will take significant time.

With *PyKdump* programs, loading is very fast. After starting ***crash*** we load a rather small extension. This extension (written in C) consists of an embedded Python interpreter and subroutines to interface ***crash***. Then, when we want to execute a program, the interpreter loads it from files that are already compiled to pseudocode.

Accessing symbolic information from vmlinux is rather slow. We do it only once for each type, after which this information stays in memory until we exit ***crash***. This means that if we need to run several programs using the same structures, they will share this information.

A problem specific to *PyKdump* implementation is that traversing the dereference chain is a rather expensive operation. Here is what happens when we do `s.a`:

- we check the type information for object `s` and verify that it has a member `s`
- we find the address of member `a` and its type
- we create and return a new object of the needed type

If we have a longer dereference chain such as `s.a.b.c` this process is repeated. This is much longer than in C or SIAL where all type analysis is made at the compilation stage and not during runtime. To improve the performance, we use a number of tricks:

- using efficient functions ("readers") to dereference specific member
- metaclasses to build new classes on the fly to represent specific C-type
- using pseudoattributes for long dereference chains—they analyze all needed symbolic info and generate an efficient function to return the result quickly

As a result of all these optimizations, the performance for dereference chains is on par (but still somewhat lower) with SIAL. Arithmetical/logical operations on base types are much faster than those for SIAL.

The performance of real tools is more than adequate. The first run is always slower than subsequent runs. For example, running ***xportshow*** on a live kernel and emulating "netstat -an":

- first run 1.06s (real) 1.00s (CPU)
- second run 0.15s (real) 0.13s (CPU)

## 2.6 Packaging And Usage

Building *PyKdump* from sources is described at `http://pykdump.wiki.sourceforge.net/Building`. It is recommended to build from SVN using the "testing" branch instead of "trunk". "testing" is where we copy recent versions when they are more or less tested; "trunk" is much more experimental. There are some prebuilt packages on SF site (they are rather old).

To use *PyKdump*, you need just a single extension file. It usually has a name ***mpykdump64.so*** on 64-bit hosts but you can rename it as you wish. You start your ***crash*** session as usual and after that load the extension by doing

```
crash32> extend /tmp/mpykdump32.so
/tmp/mpykdump32.so: shared object loaded
```

The extension file contains all needed components:

- embedded Python interpreter
- an interface module to ***crash*** internals
- a subset of Python Standard Library
- some standard tools built on top if *PyKdump*, such as ***xportshow*** and ***crashinfo***

The extension file is constructed as a shared library with ZIP-archive appended. It is acceptable to to add your own programs directly to the extension file by using ***zip*** command. This is mainly useful for distribution; normally you develop and run programs directly from Python files. For example:

```
-----------hello.py--------------
# This is a basic PyKdump program
from pykdump.API import *

print "Hello PyKdump"
---------------------------------
crash32> epython hello.py
Hello PyKdump
crash32> epython hello
Hello PyKdump
```

## 3 XPORTSHOW

***xportshow*** is a tool written using *PyKdump*. It is interesting that in addition to using it for general troubleshooting (e.g. HP Linux support organizations) it has been deployed by computer security experts such as "Volatile Systems."

The tool uses short options similar to those of ***netstat*** plus many long options. You can use multiple '-v' to increase the verbosity, up to '-vvv'. `xportshow -h` lists all options and there is additional documentation at `http://pykdump.wiki.sourceforge.net/xportshow`

You can find some examples of ***xportshow*** outputs at the end of this paper.

## References

[1] `http://people.redhat.com/anderson/`

[2] `http://www.python.org`

[3] `http://www.llnl.gov/linux/slurm/coding_style.pdf`

```
crash32> xportshow -at
tcp   0.0.0.0:42691            0.0.0.0:*                 LISTEN
tcp   127.0.0.1:9161           0.0.0.0:*                 LISTEN
tcp   0.0.0.0:8010             0.0.0.0:*                 LISTEN
tcp   127.0.0.1:9165           0.0.0.0:*                 LISTEN
tcp   0.0.0.0:111              0.0.0.0:*                 LISTEN
tcp6  :::22                    :::*                      LISTEN
tcp   127.0.0.1:631            0.0.0.0:*                 LISTEN
tcp   15.236.177.25:52414      16.236.16.79:5223         ESTABLISHED
tcp   15.236.177.25:53004      69.159.122.174:22         ESTABLISHED
tcp   15.236.177.25:35015      15.37.113.20:143          ESTABLISHED
tcp   127.0.0.1:47939          127.0.0.1:9165            ESTABLISHED
tcp   127.0.0.1:9165           127.0.0.1:47939           ESTABLISHED
tcp   127.0.0.1:9161           127.0.0.1:54388           TIME_WAIT
tcp   127.0.0.1:9161           127.0.0.1:54387           TIME_WAIT

crash32> xportshow -atv
------------------------------------------------------------------------------
<struct tcp_sock 0xf62c8000> TCP
tcp 0.0.0.0:42691 0.0.0.0:* LISTEN
family=PF_INET
backlog=0(16)
max_qlen_log=5 qlen=0 qlen_young=0
------------------------------------------------------------------------------
<struct tcp_sock 0xf7580980>           TCP
tcp   15.236.177.171:51095      16.236.16.79:5223          ESTABLISHED
windows: rcv=63480, snd=32767  advmss=1398 rcv_ws=0 snd_ws=0
nonagle=0 sack_ok=0 tstamp_ok=0
rmem_alloc=0, wmem_alloc=0
rx_queue=0, tx_queue=0
rcvbuf=87380, sndbuf=16384
rcv_tstamp=7.8 s, lsndtime=10.2 s  ago
------------------------------------------------------------------------------
<struct tcp_sock 0xf7954e40> TCP
tcp 127.0.0.1:9161             127.0.0.1:54393            TIME_WAIT
tw_timeout=15000, ttd=1730

crash32> xportshow -ltvv
<struct sock 0xd5c6c600>               TCP
tcp   0.0.0.0:7778             0.0.0.0:*                 LISTEN
         family=PF_INET
         backlog=129(128)
         max_qlen_log=10 qlen=69 qlen_young=1
    --- Accept Queue <struct open_request 0xf001e600>
          laddr=128.8.61.4 raddr=10.148.6.13
          laddr=128.8.61.4 raddr=10.148.2.101
          laddr=128.8.61.4 raddr=10.149.6.7
    --- SYN-Queue
          laddr=128.8.61.4          raddr=128.8.11.24
          laddr=128.8.61.4          raddr=10.148.16.12
          laddr=128.8.61.4          raddr=10.152.0.45
          laddr=128.8.61.4          raddr=10.149.4.8
```

Figure 1: TCP Connections Info

```
crash32> xportshow -uav
-----------------------------------------------------------------------------
<struct udp_sock 0xf791b280>            UDP
udp6  ::1:123                     :::*                          st=7
        rx_queue=0, tx_queue=0
        rcvbuf=110592, sndbuf=110592
        pending=0, corkflag=0, len=0
-----------------------------------------------------------------------------
<struct udp_sock 0xf791b000>            UDP
udp6  :::123                      :::*                          st=7
        rx_queue=0, tx_queue=0
        rcvbuf=110592, sndbuf=110592
        pending=0, corkflag=0, len=0
-----------------------------------------------------------------------------

crash32> xportshow -ax
unix   State          I-node  Path
----------------------------------
unix   LISTEN         17667   /var/run/acpid.socket
unix   LISTEN         17996   @/var/run/hald/dbus-eYQQ7ZQwSxe
unix   LISTEN         17928   /var/run/dbus/system_bus_socket
unix   LISTEN         19733   /dev/gpmctl

crash32> xportshow -awv
-----------------------------------------------------------------------------
<struct raw_sock 0xe7678600>            RAW
raw   0.0.0.0:1                   0.0.0.0:*                     st=7
        rx_queue=0, tx_queue=0
        rcvbuf=131072, sndbuf=2048
```

Figure 2: Other Protocols Info

```
crash32> xportshow --summary
TCP Connection Info
-------------------
        ESTABLISHED      7
          TIME_WAIT      2
             LISTEN      7
                        NAGLE disabled (TCP_NODELAY):     1
UDP Connection Info
-------------------
  13 UDP sockets, 0 in ESTABLISHED
Unix Connection Info
------------------------
        ESTABLISHED    331
              CLOSE     12
             LISTEN     21
Raw sockets info
--------------------
              CLOSE      1
Interfaces Info
---------------
  How long ago (in seconds) interfaces trasmitted/received?
          Name     RX          TX
          ----   ----------  ---------
           lo         1.9       7467.3
         eth0         4.2          7.2
        wmaster0      7467.3         57.3
         eth1      7467.3       7467.3
         tun0         7.2          7.2
```

Figure 3: Summary

# Online Hierarchical Storage Manager

Sandeep K Sinha
*NetApp, India*
sandeepksinha@gmail.com

Rishi B Agrawal
*Symantec, India*
rishi.b.agarwal@gmail.com

Vineet Agarwal
checkout.vineet@gmail.com

Rohit Vashist
rohit.k.vashist@gmail.com

Rohit K Sharma
mailboxrohit19@gmail.com

Sneha Hendre
sneha.hendre@gmail.com

## Abstract

Intel, Sandisk, and Samsung are investing billions of dollars into Solid State Drive technology and manufacturing capacity. Unfortunately due to the extreme cost of building the manufacturing facilities, SSD manufacturing capacity is not likely to exceed HDD manufacturing capability in the near future. Most data centre applications heavily lean toward database applications which use random read/write disk activity. For random read/write activity, the performance of SSDs is 10x to 100x that of a single rotational disk. Unfortunately, the cost is also 10x to 100x that of a single rotational disk. Due to the limited manufacturing capability of SSD, most applications are going to remain on rotational disk for the foreseeable future. Online Hierarchical Storage Manager has been developed to allow SSD and traditional HDD (including RAID) to be seamlessly merged into a single operational environment thus leveraging SSD while using only a modest amount of SSD capacity.

In an OHSM enabled environment, data is migrated to and from the high performing SSD storage to traditional storage based on various user defined policies. Thus, if widely deployed, OHSM has the ability to improve computer performance in a significant way without a commiserate increase in cost. OHSM being developed as open source software also abolishes the licensing issues and the costs involved in using storage solution software. OHSM being "online" signifies the complete abolishment of the downtime and any changes to the existing namespace.

## 1 Introduction

Hierarchical Storage Management is a data management technique that uses devices in an economically efficient manner, thus reducing the storage space and administrative costs associated with managing data.

OHSM is an online hierarchical storage manager for Linux which offers policy based transparent movement of data from one class of storage to another. Being the first attempt towards an open source data manager, it provides a base platform for all further developments in similar areas. It supports policy based migration of files i.e. it defines a set of policies which decide the correct placement tier for a file during its initial creation, as well as block allocation and relocation of the file from one placement tier to another. A placement tier is basically a storage class, which consist of a collection of storage devices with similar properties defined in the policy file based on its speed, cost or any other attribute. These placement tiers can be priority-ordered and can be overlapping as well. These policies are enforced on a OHSM enabled file system through an XML based placement policy file. A placement policy file contains a collection of rules which decides both, the initial file location and the circumstances under which existing files are relocated. Therefore, placement policies have been broadly categorized into placement and relocation policies.

Whenever a file is created it will be allocated in accordance to the placement policy which has been enforced on the file system. If the file fails to match any of the rules specified in the policy file, it falls into the default allocation method that is used by the underlying file system. Similarly for relocation, whenever a file matches any of the relocation policy, it is relocated from the

source tier to destination tier as specified in the relocation policy file. The migration of data is non disruptive and completely transparent to the placement tiers. The placement policy file also contains the mapping information between the storage devices and the respective placement tiers to which they belong. OHSM does not impose constraints on the placement tiers as far as capacity, performance and availability are concerned.

OHSM also provides functionality to remove files on certain events which can be specified through the policy file. Defragmentation of the relocating files could also be achieved by enabling defragmentation at the time of triggering relocation. Though OHSM doesn't guarantee complete defragmentation of data blocks, it does a best-effort attempt. Relocation is an event triggered operation based on single or multiple policies selected from the set of relocation policies aiming to provide greater flexibility and usability to system administrators.

## 2 Design

The idea here is to leverage the underlying device topology of the block-device logical volume and use this information to optimize the block allocation methodologies used by the file system, thus achieving storage efficiency. OHSM provides a framework to implement hierarchy based storage in the existing environment using support from the device mapper. This requires some modifications to the file system's file creation and block allocation routines.

The overall design of the system is composed of a group of inter-operating modules implemented as shared libraries, daemons and kernel modules. OHSM has various components including the user interface, an XML parser, OHSM Admin and the Kernel Driver with each of them offering different functionality to support the various services offered by OHSM.

### 2.1 User interface

Another key component of the OHSM system from an administrator's perspective is the administrative interface. It is comprised of both a graphical as well as a command line interface. Apart from the basic functionality like enabling, disabling and querying the state of the OHSM system, it also provides the administrator an opportunity to generate, modify and validate the XML policy file through a graphical interface. The graphical interface also provides various other statistical data including state of tiers, space utilization, number of files relocated and various other information related to the each placement tier. Apart from these facilities both the interfaces have a lot of services to offer such as getting and setting various OHSM runtime tuneable features, enforcing of placement and relocation policies on file systems and many more. In all, both the GUI and CLI offer a simple and easy to use interface to the administrators.

### 2.2 XML Parser

OHSM is a policy based hierarchical storage manager and it uses an XML based policy file for defining all the placement and relocation policies. The XML parser is responsible not just for parsing the administrator defined XML policy file but also for validating the information provided. The XML parser takes the policy files as input and further parses it to extract various information from it like the tier-device mapping, placement and relocation policies. Then it validates that information against the device topology map and checks for any conflicting policies. In case of any errors or conflicts it either reports back or else it transforms the parsed policy information into relevant data structures and passes it to the OHSM Admin module for further processing.

### 2.3 OHSM Administrator

This is the central communication hub which differentiates and communicates with all other components of the OHSM system. It also helps keep the design modular and simpler. It is also responsible for all the required communications between the user space and kernel driver to facilitate all the requests through the user interface. Most of the error handling is done by the OHSM Admin. All the communication with the user-space device mapper library to get the device topology mapping also goes through the OHSM Admin. The errors received from the parser and the device mapper library are processed and converted into user readable strings and passed to the user interface.

### 2.4 Device Mapper API

OHSM uses the user space device mapper library in order to extract the device topology beneath the logical

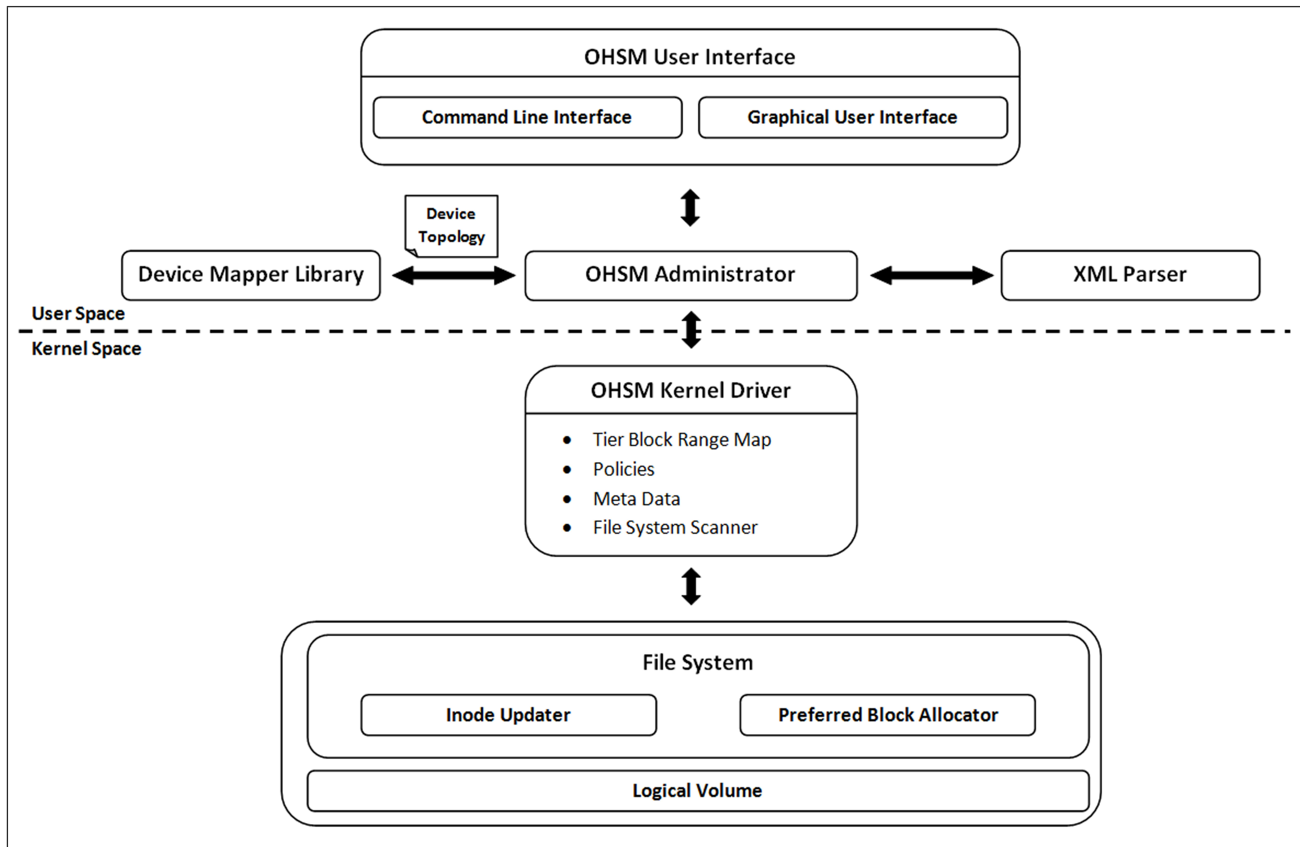



Figure 1: OHSM Architecture

volume on which the file system is already mounted. This helps the file system to optimize its data block allocation to provide better storage efficiency. The device topology along with the tier-device mapping information helps OHSM build its complete internal mapping data structures. The final mapping information resides in OHSM kernel driver. The library also offers certain other callbacks providing data regarding the underlying devices which can be used in later implementations of OHSM. This will help provide a more administrator friendly solution. This information can also be utilized to derive certain heuristics and statistical information regarding the placement tiers.

### 2.5 OHSM Kernel Driver

OHSM kernel driver is the most important component of the OHSM system. This is the core of the system and helps service all the requests from the user space. It stores various metadata and service routines associated with OHSM. It also holds the tier-device mapping table, placement and relocation policies associated with the file systems. The file system scanner and the complete mechanism of relocation have also been implemented as a part of the kernel driver. Apart from these, it also implements the various ioctl service routines. The kernel driver is loaded in the memory during boot time, so that the required information and functionality is available to the file system soon after mounting. Any problem with the kernel driver can lead to a complete freeze of the working OHSM. Hence special care has been taken to handle most of the error conditions gracefully. The kernel driver is the core of OHSM. However, it is the file system implementation that gives it its power. Keeping the two independent allows the file system changes to be minimally invasive and not require major OHSM specific patches to address inode updates and preferred data block allocation needs. The driver comes into picture once the administrator triggers relocation on the file system.

### 2.6 File System

OHSM system can work with most of the GNU/Linux file systems with some basic modifications to the way a file is created and extended. OHSM broadly divides those changes into two sub categories:

#### 2.6.1 Inode Updater

As the name signifies, the changes revolve around the inode allocation mechanism for a file system. In an OHSM enabled environment, it is expected that the initial file creation be governed through some file creation policy. OHSM changes the way the normal file creation works and imposes an additional check of the file's physical characteristics against the placement policies specified. In case of any match, the home tier id is set for that file, eventually directing the file system to serve all the data blocks requests for that file from the pool of blocks belonging to its home tier.

#### 2.6.2 Preferred Block Allocator

In an OHSM enabled environment, all the data block requests for a file are restricted to its home tier. This might lead to a situation wherein there is no free space left on the specified tier. In order to overcome such circumstances, OHSM offers an option for administrators to provide the tiers in a prioritized order, with the most desirable destination listed first. This change overrides the normal block allocation policy of the file system and makes sure that it follows the policies specified by the administrator, if any. Those files that don't qualify to any of the file placement criteria follow the usual semantics of file creation offered by the file system. Also, all further data block requests of such files are serviced from blocks spanning over the complete file system.

## 3 OHSM Value Proposition

The most broadly applicable benefit of the Online Hierarchical Storage Manager is to reduce the average online storage cost by migrating inactive files to a less-expensive placement tier in a hierarchical based storage environment. It should be assumed that the lower placement tiers have a significantly lower per-byte cost than storage in next higher placement tier. In most of the

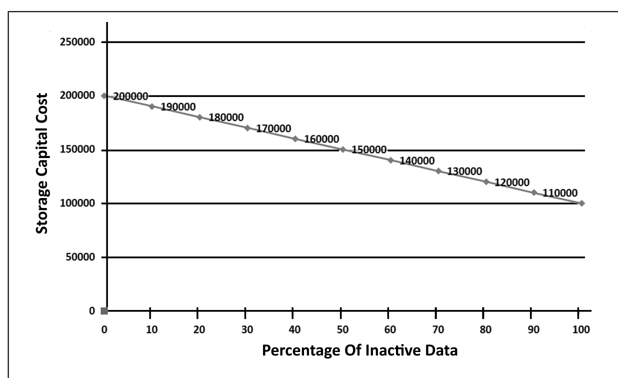



Figure 2: Value Proposition

cases, the cost differential between different types of online storage creates the economic justification for such hierarchical storage environment. If the highest placement tier storage costs around $5 per gigabyte and mid range placement tier storage costs around $2 per gigabyte, an enterprise whose online data is 50% inactive could save around 30% of its storage acquisition cost by moving the inactive files to mid range placement tiers. Larger percentages of inactive files result in higher savings.

For enterprises that keep a significant amount of non-critical data online, a multi-tier storage strategy can offer substantial cost savings without adverse effects on business operations. The challenge in attaining the benefits of multi-tier storage is to get the right files on the right storage tier at the right time. OHSM achieves it very efficiently with the help of policy based allocation and relocation. The purpose of OHSM is to eliminate any administrative cost and complexity in a hierarchical storage environment by automating the relocation of files as levels of I/O activity against them rise and fall, as well as when their sizes, owners, or logical positions in the file system hierarchy change.

## 4 Working with OHSM

Information Lifecycle Management provides effective management of information throughout its useful lifecycle. ILM also provides strategies to allow a computing device to administer the storage systems. These strategies consist of policies which differentiate and administer data based on its usage and priority. In order to work with the Online Hierarchical Storage Manager the administrator needs to be quite aware of the various file

placement and relocation policies supported by OHSM. The administrator needs to put together all this information along with the tier device mapping information into a XML file. This file is passed to OHSM to enable and enforce these policies on a file system. It should not be forgotten that OHSM also offers a graphical user interface to generate the XML policy file. The user can either use the graphical interface or the command line interface in order to enable OHSM. Below we examine various file placement and relocation policies with their respective sample XML policy grammar as supported by OHSM.

### 4.1 Tier Device Map

The tier device mapping information is required in order to define the set of devices that belong to a particular placement tier. Both the file placement and relocation policies are validated against the tier device map information specified in this section. All the information provided in the tier-device map section is also validated against the configuration of the system. Some of the validations involve checks for making sure that the specified devices exist on the system, all the devices are part of the same logical volume over which the file system is mounted, etc. A simple tier device mapping information:

```
<DEVICES>

  <DEV_TIER_INFO>
    <NR_TIERS>3</NR_TIERS>
    <NR_DEVICES>6</NR_DEVICES>
  </DEV_TIER_INFO>

  <DEV_TIER>
    <TIER>1</TIER>
      <DEVICE>/dev/md4</DEVICE>
      <DEVICE>/dev/md5</DEVICE>
    <TIER>2</TIER>
      <DEVICE>/dev/md3</DEVICE>
    <TIER>3</TIER>
      <DEVICE>/dev/md1</DEVICE>
      <DEVICE>/dev/md2</DEVICE>
      <DEVICE>/dev/md6</DEVICE>
  </DEV_TIER>

</DEVICES>
```

### 4.2 File Placement

Transparent and non disruptive relocation of data across various placement tiers is undoubtedly the most obvious use of the OHSM but it is not restricted to that. OHSM also offers various functionality to give targeted files a preferential placement at the time of file creation. We have seen an enterprise using database management system to manage their most critical data and provide that critical data a preference over all other data. OHSM currently supports four file placement policies which can be used individually or can be combined together to form various new rules. If a file qualifies for multiple placement policies then the first match prevails over all others.

For special situations where the target placement tier might run out of free blocks, the administrators have the facility to provide the preferential order of tiers for each such rule. This directs the file system to allocate blocks from a lower tier in case the target placement tier is already full. This feature is optional and can be enabled/disabled as per administrator's discretion. Currently OHSM supports a very primitive set of file placement criterion including file type, user ID, group ID and the logical placement of files in the file system hierarchy, directory name. Allocation policy based on directory name can be both recursive and non-recursive. The file placement policy can be based on the following:

#### 4.2.1 File Type (FTYP)

In today's time, there is a strong likelihood that applications follow a pattern in the file name extension to determine the kind of data that the file holds. This pattern can be utilized to differentiate between different types of files like database files, media files and log files etc. Using the file extension we can also associate the file with different applications most of the time and derive its criticality based upon that. This information can be used to provide preferential placement to various type of files. We can also dedicate placement tiers to specific file types based on its type.

#### 4.2.2 User ID (UID)

In a large server environment the file systems are mostly organized based more on the users rather than the applications. Consider the case of an enterprise, where different users have their home directory on the same shared file system. In such situations there can always be reasons to allot higher placement tiers to various users while restricting others to have a mid range or a lower one.

#### 4.2.3 Group ID (GID)

Similar to file placements based on users, various groups in an organization can share a placement tier. This can be based on the criticality of data they operate on and at times the speed of data retrieval. Consider the case of an engineering team and a marketing group; it may be desirable to have the engineering data on a more reliable placement tier as compared to the marketing team. The accounting group can have opted for a separate placement tier for various other reasons.

#### 4.2.4 Directory Name (DIR)

Often we create directories based on the current time or date. This helps us in keeping the data in a more structured manner. For instance, if someone keeps its reports for the last couple of years, there will be a number of directories present on the file system, for example, report-2007, report-2008 and report-2009 and so on. It is expected that the latest reports will be the most frequently accessed one. Though it is just an assumption, this can be used to provide placement tiers to various structured data classified on the basis of their age. File placement based on directory names can be recursive and non-recursive depending on the specification in the policy file. A simple file placement policy:

```
<ALLOCATION>

  <ALLOC_INFO>
    <NR_USERS>1</NR_USERS>
    <NR_GROUPS>2</NR_GROUPS>
    <NR_TYPES>1</NR_TYPES>
    <NR_DIRS>1</NR_DIRS>
  </ALLOC_INFO>

  <USER_TIER>
    <USER>0</USER> <TIER>1</TIER>
  </USER_TIER>

  <GROUP_TIER>
    <GROUP>0</GROUP> <TIER>1</TIER>
    <GROUP>501</GROUP> <TIER>3</TIER>
  </GROUP_TIER>

  <TYPE_TIER>
    <TYPE>ora</TYPE> <TIER>1</TIER>
  </TYPE_TIER>

  <DIR_TIER>
    <DIR>/foo</DIR> <REC>1</REC> <TIER>1</TIER>
  </DIR_TIER>

</ALLOCATION>
```

### 4.3 File Relocation

One of the most desirable things to have is to store inactive files on placement tiers of lesser quality so that it does not affect the applications adversely. If you look at it from the I/O performance perspective, if the file is accessed rarely and it is mostly inactive, the performance of the storage device underneath that holds it is irrelevant. This makes the ability to relocate files across placement tiers very critical and important. Also, there are situations where you have thousands of small files in a file system. It is seen that under such circumstances most of these files soon become inactive. Some of the scenarios are a document management system, a mail server or any database application using opaque data objects stored as small files. It would be highly desirable to have the ability to relocate data across placement tiers under such circumstances. OHSM currently has support for the following file relocation policy criteria.

#### 4.3.1 File Access Age (FAA)

It signifies the time since the last access to the file which can be one of the most appropriate qualifiers for a downward relocation. Based on the time of last access to the file, it could easily be relocated to a lower placement tier. This can be useful in a search engines or mail server environment, where the files access rates go down as the time increases. This would not be a good candidate as a qualifier for relocation from lower to higher placement tier as this can be misleading at times. There can be files which are just accessed to know that it's not of use and the data is stale. Still, because of the file's access age being quite small, it can get relocated to a higher tier, which would be highly undesirable. The recent introduction to realtime in the kernel really changes how the last access time is managed in a significant way. And use of such a feature might eliminate most of the value of FAA.

#### 4.3.2 File Modification Age (FMA)

This is a true qualifier for a relocation to happen from a lower to a higher placement tier. It can be fairly assumed that a file which has recently been modified or which has a smaller modification age would surely be accessed more frequently in the near future. Hence, this

can be used for deciding upon the conditions for relocation from lower to higher placement tiers. Most of the stub based implementations for HSM also use modification age as one of the primary qualifier to bring back the data from their archival storage to their actual placement tier.

#### 4.3.3 File Size (FSZ)

There may be various situations where it would be desirable to allow a certain size of file to reside on a specific placement tier. The reason is the limited size constraints of the higher placement tiers due to their higher costs. We move the file to a lower placement tier if the file size exceeds a specific threshold. This threshold can be based upon the amount of space that a higher placement tier has. So, the file size qualifier can be easily used to prevent situations where the higher placement tiers don't run out of space.

#### 4.3.4 File I/O Temp (FIOT)

File I/O temperature is defined as the average number of bytes transferred to or from a file over a period of time. This is independent of the file size and is one of the more powerful qualifiers which can be used to automate the process of relocation.

#### 4.3.5 File Access Temp (FAT)

File access temperature is defined as the ratio of the number of times the file has been accessed over a period of time. This helps us to determine the average I/O activity that is taking place on a file against all other files in the file system. Such a measure can be useful to find suitable candidates for relocation from both lower to higher placement tier and vice versa.

#### 4.3.6 FTYP, UID and GID

Relocation policies can also be based on the file type, user ID and group ID qualifiers. Since, the initial allocation can be based on these qualifiers, there is a great chance that when these are combined with other relocation qualifiers, form a finer granularity relocation criterion. A simple file relocation policy:

```
<RELOCATION>
   <NR_RULES>1</NR_RULES>
    <RULE>
         <INFO>1</INFO>
         <RELOCATE>
              <FROM>1</FROM>
              <TO>2</TO>
            <WHEN>
              <FSIZE>50</FSIZE> <REL>LT</REL>
              <FAA>50</FAA> <REL>LT</REL>
            </WHEN>
         </RELOCATE>
    </RULE>
</RELOCATION>
```

## 5 Prototype Implementation for ext2/ext3

The prototype of OHSM involves basic implementation of the idea presented in the previous sections. The general concept of OHSM involves various modules and their relationship with the file system. OHSM consists of roughly four components, namely the User Interface, OHSM Admin, Kernel Driver and File system. Our prototype provides functionality for creating policy files, enabling and disabling of OHSM, and triggering relocation manually. The User interface provides a set of commands to control and monitor the various functionality offered. It also allows the user to create XML based policy files and logical volumes at the same time. In the prototype implementation the user is required to create separate policy files for allocation, relocation and tier device mapping. These policy files were required to be specified at the time of enabling OHSM on a file system. Before OHSM could be enabled, these policy files are required to be parsed and validated for any conflicts. After verifying the policy files, the information is stored in internal data structures. The OHSM Administrator uses the ioctl interface provided by OHSM kernel driver to control and administer the system. On receipt of the data structures the kernel driver replicates these data structures in the kernel, and acknowledges back to the administrator module success or errors if any. On success, EXT3_OHSM_ENABLE flag is set inside the file system's super block. When OHSM is disabled all the data structures are cleared and the flag is reset. In order to achieve this, minor changes were made to struct ext3_inode and a new flag was introduced to be used within struct ext3_super_block.

```
struct ext3_inode {
...
...
__u8 ohsm_home_tid;
__u8 ohsm_dest_tid;
};

/*
 *  Misc. file system flags
 */

#define EXT3_OHSM_ENABLE 0x0008 /* OHSM enabled */
```

When a file is created it is intercepted by the OHSM inode updater and an additional check is made against the allocation policy enforced on the file system. In case a file qualifies, its ohsm_home_tid is set to the corresponding tier id. Otherwise, it remains zero. Later, for any data block requests for files having a non-zero ohsm_home_tid, the call to the block allocation routine is diverted to OHSM's block allocation routine, if OHSM is enabled on the file system. This implementation requires variations in the existing file system structures and its block allocation strategy also known as ranged block allocation. Ranged block allocation improves proficiency of file system in restricting allocation of data blocks to a range of block groups. A table containing the map of tier against the block group ranges is maintained by OHSM kernel driver. Ranged block allocation also uses this information to allocate new data blocks for the file in a specific tier. This block group range table is used by the block allocation routine to identify the block group ranges of device hierarchy. This table is created at the time of enabling OHSM and remains active in memory until the file system remains mounted or OHSM stays enabled. At the time of unmounting or disabling of OHSM this information is dumped on disk to /etc/ohsm. This information is used later to reconstruct this table back when the file system is mounted back or OHSM is re-enabled on a file system.

In user space, the OHSM Admin uses libdevmapper to get the device topology and passes the extents of devices to the kernel driver. The driver later maps these extents to file system specific block group ranges. The ohsm_home_tid field of inode is used as an index in this table to get the specific block group range. The allocation routine bounds the data block allocation within the selected range. Figure 3 illustrates two scenarios. The left half of it illustrates the scenario when a file is created. It shows that initially the ohsm_home_tid is set to zero, which later gets updated by the inode updater where it is qualified against the various file placement policies. If qualified, the files ohsm_home_tid is updated accordingly with the specific tier id. On the right side, it shows the later scenario where there is block allocation request for a file. The block allocator checks for a non-zero ohsm_home_tid and extracts the relevant block group ranges for the same. For a file having ohsm_home_tid equal to zero, the block group range spans the complete file system. The call to the block allocation routine is diverted to Range block allocation in place of file systems normal block allocation routine, which eventually serves the purpose. In case the tier is full, the file's ohsm_home_tid is set to zero and the file system's normal block allocation routine is invoked.

Relocation currently is a triggered event and has to be started manually by the administrator. When relocation is triggered, OHSM kernel driver scans all the inodes in the file system and pushes each qualifying inode to the work queue for relocation. Prior to adding each inode to the work queue, the ohsm_dest_tid is set to the relevant tier for that inode. The workqueue handler routine is implemented in the OHSM kernel driver which picks and does the task of relocation. After the relocation is completed, the ohsm_dest_tid becomes the new ohsm_home_tid for the file. As an optimization, OHSM uses a Tricky copy and swap algorithm to complete relocation as fast as possible.

Tricky copy and swap algorithm starts by allocating a new ghost inode and reading the source inode in memory. It then takes a lock on the source inode to stop any further modifications to the inode in the course of relocation. Later, it reads the data blocks for the source inode and copies them to the destination inode's blocks, block by block. The reading of source block data is done through block buffers and they are then copied to destination buffer. The destination buffer is marked dirty. Finally, when all the data is copied to the ghost inode, the source inode is re-assigned with the contents of destination inode's data blocks by swapping them with that of the ghost inode. The source inode now contains pointers to new data blocks. At this point, the source inode is unlocked, synced and destination inode is released. OHSM Administrator is acknowledged of the completion of these event. See Figure 4 which illustrates the process of relocation.

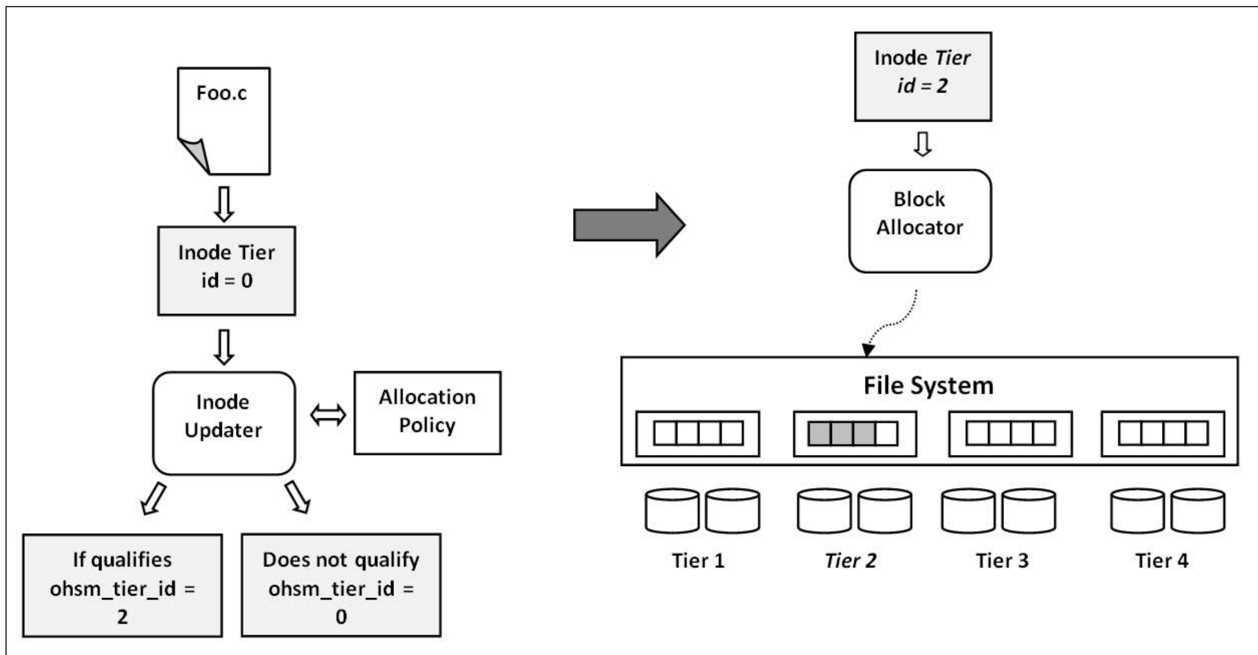



Figure 3: File placement and Block allocation mechanism

## 6 Issues and Concerns

During the course of OHSM's prototype implementation which was done primarily for ext2/ext3 file system, the process revealed several issues. Some of the major ones include the following:

### 6.1 struct ext3_super_block

Currently both the tier-block group range map and the allocation policies reside in OHSM kernel drivers. This enforces a dependency of the file system on OHSM kernel driver. We ensure to handle this situation currently by starting the OHSM services before any local file system is mounted. It required some modification to the system startup script. Since, this information is per file system this information should ideally reside in the super block of the file system. OHSM still struggles to find an easy way out of this. Since, this table can be huge and number of policies can be quite high, keeping such information in the super block is undesirable.

### 6.2 struct ext3_inode

Allocation and relocation are the key components of OHSM and as the object on which OHSM operates is a file, it is very tightly coupled with the inode structure. So, it was required to make on-disk changes in the struct ext3_inode in order to support allocation and relocation. We added "ohsm_home_tid" which stores the tier ID assigned upon allocation of a file and "ohsm_dest_tid" which stores the tier ID assigned during relocation of a file. Furthermore, to support different criteria of relocation like File Access Temp (FAT) and File Input Output Temp (FIOT) respective fields are to be added to the inode structure of the file system. These changes make the compact inode structure slightly bulky. These changes might also disturb any existing file system partitions present on the current system. OHSM plans to use extended attributes in order to avoid this problem and currently lacks a concrete way to handle this.

### 6.3 Exporting internal functions

The complete working of Online Hierarchical Storage Manager requires support from the file system on which it operates. In order to provide support to OHSM a few static functions residing inside the file system are required to be exported. This compromises the integrity of the file system code. For recent file systems like ext4 the required functionality (EXT4_IOC_MOVE_VICTIM) is soon going to be present which will help OHSM to not violate such integrity issues in the future. We are

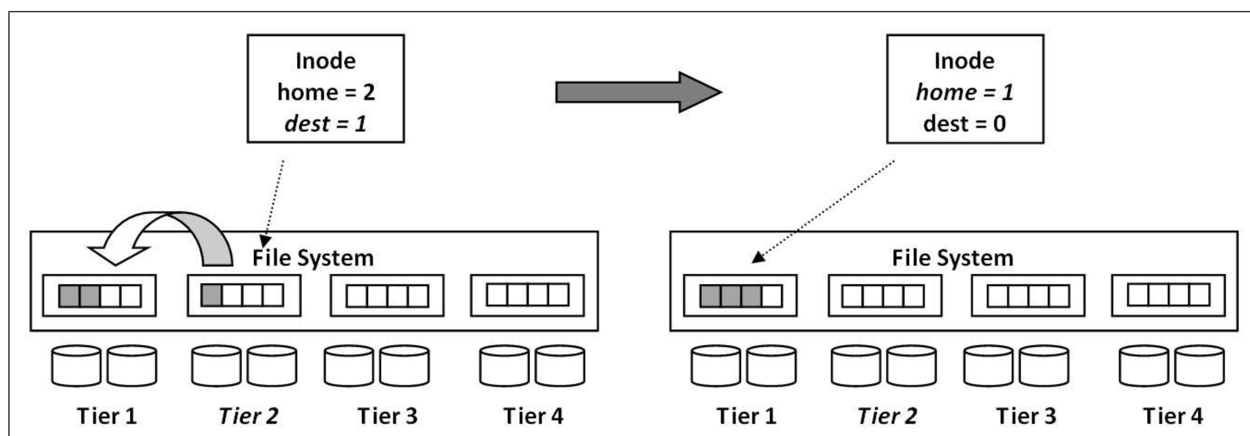



Figure 4: File relocation mechanism

trying to make OHSM completely functional with the ext4 file system during the writing of this paper. We are also monitoring and reviewing the implementation of the patches from Akira Fujita for restricted block allocation.

### 6.4 Crash during relocation

Currently OHSM uses a temporary inode in order to relocate an inode's data blocks from one tier to another. In case, if the system crashes during this relocation, there can be a chance of data loss. Currently we only release the original blocks after the complete relocation is completed. This reserves some space for the time during which relocation is going on. OHSM still needs a better way to handle this. Journaling may be required to overcome this problem.

### 6.5 Inode lock contention

During the process of relocation the inode is locked until all the data blocks attached with the inode are successfully relocated to a new destination tier. The time taken during relocation is file size dependent. This time would be more for large files, so any I/O pending on it will have to wait for that period of time and the requests may even time out. We are in the process of dividing this whole relocation into chunks of 64K in order to reduce this lock contention period.

## 7 One step Further

OHSM looks forward to a list of enhancements in-order to make it complete and stable. Here are a couple of them to start with:

### 7.1 User space implementation

The most desirable aspect with OHSM is to move as much as possible of its implementation into user space. This helps us remove the kernel components and other dependencies of the file system. It will also help maintain the source code integrity for the file system. Such an implementation would surely require a lot of support from the file system. Going ahead with an ioctl based interface would be one of the best options. Eventually the file system would need to support the OHSM based file creation and ranged block allocation. To achieve this without breaking the integrity and consistency of the code is a big challenge.

### 7.2 Automatic Relocation Engine

Currently relocation is a triggered event which in most of the server based environments will not be a pleasant experience for the administrators. Going a step further and designing an automatic relocation engine would be one of the most fascinating features OHSM can offer. The most important challenge in designing such an engine would be to derive the heuristics which would drive such an engine. A very frequent invocation to relocation can damage the file systems performance to a great extent. Also, a long interval between relocations can affect the storage efficiency adversely. So, we need an intelligent mechanism which could be based on the I/O and activities on the file system. Using FAA and FMA as criteria can impose hard restrictions with their values

being constant, which might not always yield optimum results. FIOT and FAT can be the most efficient candidate as they are softer and can be based truly on the file activities in real time. OHSM is still looking forward to good heuristics and measures which would provide an optimum and efficient methods for making relocation a dynamic event.

### 7.3 Optimize mdraid Support

OHSM uses a new block allocation strategy for the file system and also has the underlying device topology. If OHSM is used over mdraid array, OHSM's block allocation strategies can be further optimized to leverage the underlying device layout. This may enhance the I/O speed over the devices in the mdraid array.

## 8 Conclusion

Online Hierarchical Storage Manager for GNU/Linux creates a platform and opens up various opportunities for further work in the area of Hierarchical storage for a Linux based environment. OHSM sets up the basic infrastructure where we can think of systematically merging traditional and SSD based storage devices to reduce the overall cost of the system administration and also attaining a degree of storage efficiency. Moreover due to the support for policy based migration of data, the administrative cost of managing data also reduces. The idea is to effectively reduce the cost of storage administration and at the same time keep the system efficient and consistent. OHSM with some changes to the file systems file creation and block allocation algorithms can achieve its goal of implementing a complete open source storage software solution.

## 9 Acknowledgment

We would like to sincerely thank all the people who helped us in this project, especially Greg Freemyer and Manish Katiyar for providing us their valuable time and support on various technical and design issues. Also we would like to thank Bharti Alatgi and Uma Nagaraj for their keen interest in OHSM from the early beginning and motivating us in the overall course of development.

## 10 References

1 *Retrieving Multimedia Objects From Hierarchical Storage Systems,* Eighteenth IEEE Symposium on Mass Storage Systems and Technologies, 2001.

2 *Planned Extensions to the Linux Ext2Ext3 Filesystem,* Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference.

3 *On Configuring Hierarchical Storage Structures (1998)*, Ali Esmail Dashti, Shahram Gh. In Proceedings of the Joint NASA/IEEE Mass Storage Conference.

4 *Ensuring Performance in Activity-Based File Relocation,* Wu, J.C. Bo Hong Brandt, S.A. Dept. of Comput. Sci., California Univ., Santa Cruz, CA.

5 *DHIS: discriminating hierarchical storage,* Proceedings of SYSTOR 2009: The Israeli Experimental Systems.

# Effect of readahead and file system block reallocation for LBCAS

(LoopBack Content Addressable Storage)

Kuniyasu Suzaki, Toshiki Yagi, Kengo Iijima, Nguyen Anh Quynh,
Yoshihito Watanabe

*National Institute of Advanced Industrial Science and Technology*
*Alpha Systems Inc.*

{k.suzaki,yagi-toshiki,k-iijima,nguyen.anhquynh}@aist.go.jp
watanays@alpha.co.jp

## Abstract

Disk pre-fetching, known as "readahead" of Linux kernel, arranges its coverage size by the rate of cache hit. Fewer readaheads of large window can hide the slow I/O, especially it is effective for virtual block device of virtual machine. High cache hit ratio is achieved by increasing locality of reference, namely, file system block reallocation based on an access profile.

We have developed a data block reallocation tool for ext2/3, called "ext2/3optimizer". The relocation is applied to Linux booting on KVM virtual machine with a virtual disk called LBCAS (LoopBack Content Addressable Storage). We compared the effect with the Linux system call "readahead" which populates the page cache with data of a file in advance. From the experiment, we confirmed that the reallocation of ext2/3optimizer kept larger coverage of readahead and fewer I/O requests than the system call readahead. The reallocation also reduced the overhead of LBCAS and made quick boot.

## 1 Introduction

We have developed a framework of Internet Disk Image Distributor for anonymous operating systems, called "OS Circular[1, 2, 3]". It enables to boot anonymous OS on any real/virtual machines without installation. The disk image is distributed by LBCAS (LoopBack Content Addressable Storage) which manages the virtual disk efficiently. The transferred OS is maintained periodically on the server and fixed vulnerable applications. The partial update is managed by LBCAS efficiently and the user can rollback to old OS.

LBCAS is a kind of loopback block device managed by CAS (Content Addressable Storage) [4, 5, 6]. CAS retrieves a data-block with the hash value of its content. Thus, CAS is a kind of indirect access management of physical address. It is used for permanently information archive because CAS distinguishes every data-block with hash value and reserves old blocks, although direct physical access overwrites the previous data. When block contents are same, they are held together with same hash value and reduce total volume.

Unfortunately, CAS is known as an archive method which behavior is affected by feature of stored data and access patterns [5, 6]. The characteristics differ from a real device and require careful handling. A performance gap is caused by the coordination of disk pre-fetching (i.e., page cache) of existing OS. For example, Linux kernel has the function called "readahead [7, 8]" which pre-fetches some extra blocks from block device. The coverage of readahead is changed by heuristics of page cache. LBCAS should be optimized for the access patterns, namely locality of reference.

In order to increase the locality of reference, we have developed "ext2/3optimizer [9]" to reallocate the data block of ext2/3 file system. The reallocation follows the access profile. The accessed data blocks are arranged to be in line. In this paper we compare the effect of ext2/3optimizer with the Linux system call "readahead" which populates the page cache with data of a file in

advance.

The remainder of this paper is organized in six sections. In section 2 the detail of LBCAS is described. Readahead and its relation to LBCAS are described in section 3. Section 4 reallocation method is described. The performance is evaluated in section 5. Some related topics are discussed in section 6 and conclusion is mentioned in section 7.

## 2 LBCAS: LoopBack Content Addressable Storage

LBCAS is made from an existing block device. The block device is divided by a fixed block size and saved to each small block file. Saved data are also compressed. Each block file has a name of SHA-1 of its contents. The address of block files is managed by mapping table. The mapping table has the relation information of physical address and SHA-1 file name. A virtual block device is reconstructed with the mapping table file on a client. Figure 1 shows the creation of block files and mapping table file "map01.idx".

Block files are treated as network transparent between local and remote. Local storage acts as a cache. The files are measured with SHA-1 hash value of its contents when they are mapped to virtual disk. It keeps the integrity for Internet block device.

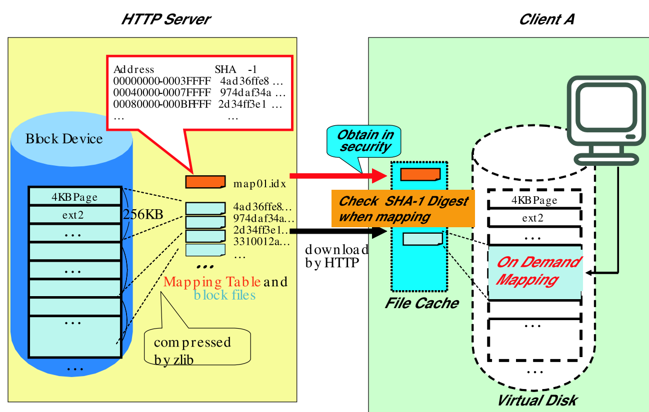



Figure 1: Creation of block files from OS image.

Figure 2 shows the diagram of LBCAS structure. A loopback file is re-constructed with downloaded block files on a client. The main program is implemented as a part of FUSE (File system in USEr space [10]) wrapper program. LBCAS has two level of cache to prevent redundant download and uncompression, which is

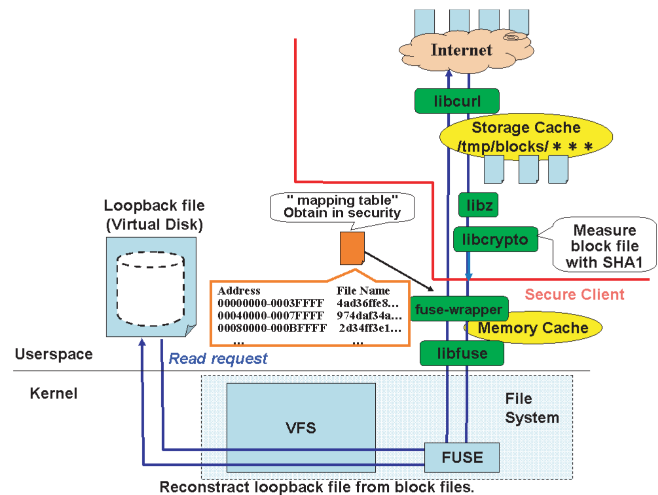



Figure 2: Creation of block files from OS image.

called "storage cache" and "memory cache". The detail of cache is described in next subsection.

A client has to obtain a mapping table file in security. The mapping table file is used to setup LBCAS. When a read request is issued, LBCAS driver searches a relevant block file with the mapping table. If a relevant file exists on a storage cache, the file is used. If not, the file is downloaded from a HTTP server with "libcurl". Each downloaded file is uncompressed by "libz" and measured with the SHA-1 value by "libcrypto". The measurement of SHA-1 value of block file is logged. Even if a block file is broken or falsified, the block is detected. Figure 3 shows the case to detect a falsified block file.

### 2.1 Two level of Cache

Current LBCAS has two level of cache, which is called "storage cache" and "memory cache".

Storage cache saves the downloaded block files at a local storage. It eliminates the download of same block file. If the necessary block files are saved at storage cache, the LBCAS works without network connection. The volume of storage cache is managed by water mark algorithm of LIFO in current implementation. The latest downloaded block files are removed when the volume is over the water mark, because aged block files might be used for boot time.

Memory cache saves the uncompressed block file at the memory of LBCAS driver. It eliminates uncompression

```
1150452051.109: #00000000(845b31ded38e15c1fa8febf97fe0781f23af98c3) :missed.
1150452051.112: #00000000(845b31ded38e15c1fa8febf97fe0781f23af98c3) :hits.
1150452051.112: #00000001(166cbaedbb1cc836e7c95d7d9943efde5a53829e) :missed.
1150452051.113: #00000002(29c4e363dbad648072751ca1f856e5780dd2981d) :missed.
1150452051.114: #00000003(fa8ad05b713a9cf8a701636ca6c353dc58fd6bfd) :missed.
1150452051.114: #00000004(1f82a543fa9310c44eff6a13618beca3cacffc12) :missed.
1150452051.128: #00000004(1f82a543fa9310c44eff6a13618beca3cacffc12) :hits.
1150452051.128: #00000005(916f62a6e2caedc1279a0a74975a406ddb60ec25) :missed.
1150452051.129: #00000006(19111dfc877a4fe241e125d10176d85a99b4bb86) :missed.
1150452051.130: #00000007(950c1d7623b374f8e03309a93041f5adfa3af80f) :missed.
1150452051.130: #00000008(486472b0ee27157d755bd59d623179cfc0034747) :missed.
```

```
1150452375.989: #00000000(845b31ded38e15c1fa8febf97fe0781f23af98c3) :missed.
1150452375.993: #00000000(845b31ded38e15c1fa8febf97fe0781f23af98c3) :hits.
1150452375.993: #00000001(166cbaedbb1cc836e7c95d7d9943efde5a53829e) :missed.
1150452375.994: #00000002(29c4e363dbad648072751ca1f856e5780dd2981d) :missed.
1150452375.995: #00000003(fa8ad05b713a9cf8a701636ca6c353dc58fd6bfd) :missed.
1150452375.996: #00000004(1f82a543fa9310c44eff6a13618beca3cacffc12) :missed.
1150452375.997: #00000004(1f82a543fa9310c44eff6a13618beca3cacffc12) :hits.
1150452375.997: #00000005(916f62a6e2caedc1279a0a74975a406ddb60ec25) :missed.
1150452375.998: #00000006(19111dfc877a4fe241e125d10176d85a99b4bb86) :missed.
E: can't validiate block.
```

Figure 3: Log of LBCAS. The upper shows the correct downloading of block files and the lower shows a falsified block file is detected. "missed" indicates downloading a block file. "hits" indicated finding a block file at local storage.

when same block file is accessed in succession. Memory cache saves 1 block file and the coverage is the size of block file. It should be coordinated with the page cache of existing OS.

### 2.2 Partial Update by Adding Block Files

The update of LBCAS is achieved by adding block files and renewing the mapping table file. The rest block files are reusable. To achieve this function, the file system on LBCAS has to treat block-unit update as ext2/3 file system. ISO9660 file system is not suitable because partial update of ISO9660 changes the location of following blocks.

The updated block is saved to a file with new file name of SHA-1. Collision of file name will be rarely happened. Even if a collision happens, we can check and fix it before uploading the block files on the servers. We can rollback to the previous file system if the old mapping table and block files exist.

### 2.3 Issues of Performance and Behavior on LBCAS

The behavior of CAS is affected by feature of stored data and access patterns [5, 6].

The block size of CAS causes problems of fragmentation and boundary. One problem comes from the size mismatch of the block size of file system. Most file system assumes their block size is 4KB but LBCAS uses larger block size because of efficiency of loopback device and network download. It results in low occupancy rate, redundant download and unnecessary uncompression.

When the occupancy, which is a ratio of effective data in a block file, is low, the overhead of LBCAS is not negligible. The block size of LBCAS should be considered the occupancy. The problem is closely related to locality of reference on transferred OS.

When a read request crosses over the boundary of CAS, CAS requires multiple blocks. If the block size of CAS is too small and requires many blocks for an I/O request, performance gap stands out. The block size of CAS must be balanced to I/O request size. The problem is related to the window size of pre-fetching.

## 3 Readahead(Disk Pre-fetching)

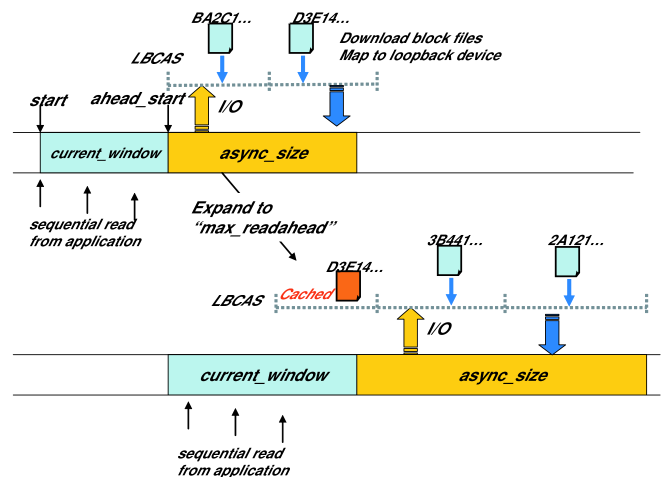



Figure 4: Behavior of readahead on LBCAS

Most operating systems have the function of page cache to reduce I/O operation. When a read request is issued, the kernel reads extra data and saves them to main memory (page cache). It reduces the number of I/O operation and hides the I/O delay. The function in Linux kernel is called "readahead [7, 8]". The coverage of readahead is extended or shrank by the profile of cache hit and miss-hit. Figure 4 shows the action of readahead on LBCAS. When a readahead operation is issued, some block files are downloaded and mapped to the loopback device. When a same block file is required sequentially,

the block file is stored on the memory cache of LBCAS and the uncompression is eliminated. When page cache does not hit, the next readahead shrink the coverage size. The suitable size of coverage achieves efficient usage of cache memory and I/O request.

The readahead causes performance gaps on LBCAS, when the extra coverage crosses over the boundary of LBCAS block (Figure 5). The third read request in the figure crosses over the boundary, and extra block file is downloaded which is never used. It causes big performance penalty compared to real block device. The problem is cased by un-contiguous of necessary blocks.

The un-contiguous blocks are improved by ext2/3optimizer described in next subsection. High hit ratio of page cache and large coverage of readahead means the less I/O requests. The access pattern will be efficient on the LBCAS.

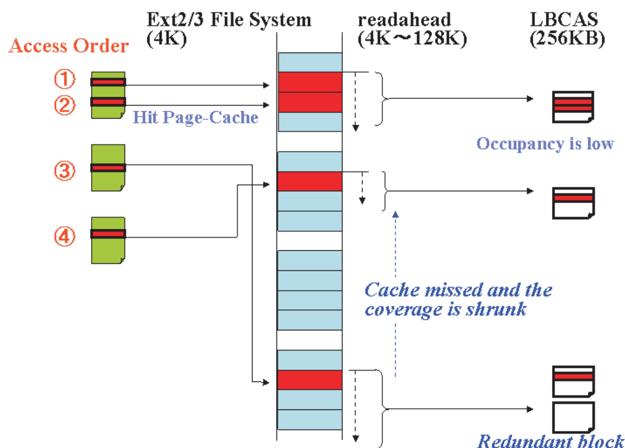



Figure 5: Behavior of readahead on LBCAS

### 3.1 system call "readahead"

Linux kernel has the system call "readahead" from 2.4.13. The system call populates the page cache with data of a file. It is not directly related to the disk pre-fetching but it can achieve the same function from user space, because subsequent reads from that file will not block on disk I/O. Linux distributions have a tool to utilize the readahead system call to make quick boot. The files opened at boot time are listed at "/etc/readahead/boot" and the data of the files are populated on the page cache in advance at boot time.

Unfortunately it requires much memory and has no dynamic flow control. The speed-up depends on individual machine. In this paper we confirm the effect on a virtual machine.

## 4 Block reallocation of File System

Most file systems have defragmentation tools to reallocate blocks of file system. For examples, defrag and ext2resize are tools for ext2. The tools however reallocate blocks from the view of continuation of file and expansion of spare space. Quick access is a side effect of continuation of file.

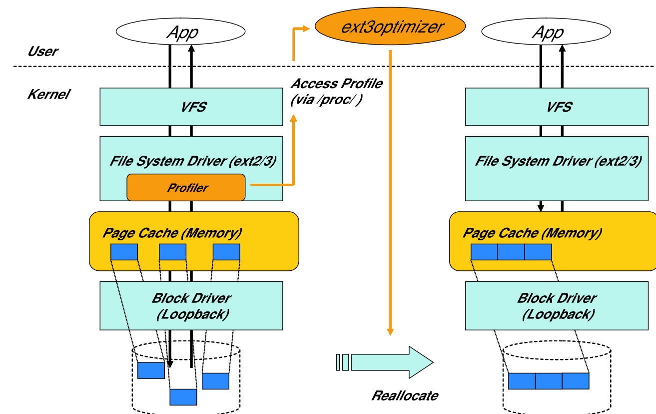



Figure 6: Access profiling and reallocation which increase cache hit ratio and coverage of readahead.

In order to solve the problem, we developed "ext2/3optimizer" [9], which was called ext2optimizer. Ext2/3optimizer takes the profile of accessed blocks of ext2/3 and reallocates the blocks in line. Figure 6 shows the image of profiling and reallocation. The reallocation increases the cache hit ratio and expands the coverage size of readahead. The effect is described in next section.

Ext2/3optimizer change pointers of data blocks of i-node only. It aggregates the data blocks at the head of device and increase locality of reference. The other structure of ext2/3, namely meta-data of ext2/3, is reserved. Figure 7 shows the image of reallocation of ext2/3optimizer.

Block size mismatch problem between file system and LBCAS is reduced by the aggregation of data blocks, because it increases the occupancy of effective data in a block file. Figure 7 shows the higher occupancy reduces the necessary block files. In this case the data on 3 block files is aggregated in 1 block file. The effect is also described in next section.

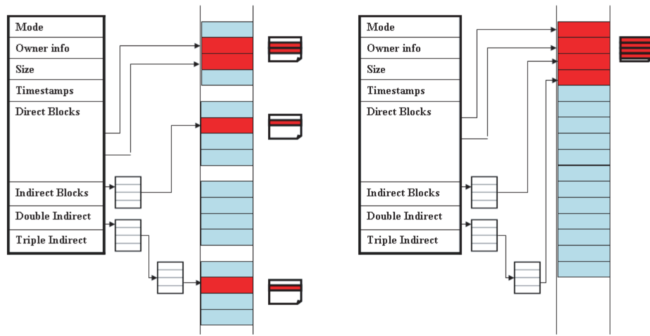



Figure 7: Reallocation of ext2/3optimizer.

## 5 Performance of ext2/3optimizer and user-level readahead on LBCAS

We compared the effect of ext2/3optimizer and user-level readahead (system call readahead) on LBCAS was evaluated. We applied both of them to the gust OS on KVM virtual machine [11] (version 60) with LBCAS, which shows the feasibility of OS migration. Ubuntu 9.04 (Linux kernel 2.6.28) was used for the transferred OS. Ubuntu was installed on 8GB loopback file with ext3 file system on KVM using normal installer. The total volume was 1.98GB. The block files of LBCAS were made from the loopback file.

The access pattern of boot procedure is random and does not read whole contents in a file. Sparse access will be increased, and the coverage of readahead will be narrow. As a consequence, the occupancy of block file will be low and the efficiency of LBCAS becomes worse. In this section we confirmed the characteristics and applied optimizations for it. From after we refer user-level readahead as "u-readahead" in order to distinguish the disk pre-fetch readahead.

### 5.1 Block Reallocation: ext2/3optimizer

Figure 8 shows the data allocation on ext3, which is visualized by DAVL (Disk Allocation Viewer for Linux) [12]. The left figure shows the original data allocation, and right figure shows the data allocation optimized by ext2/3 optimizer.

The green plots in the figure indicate the allocation of meta data of ext3 which was arranged at the right edge. We confirmed that ext2/3optimizer keeps the structure

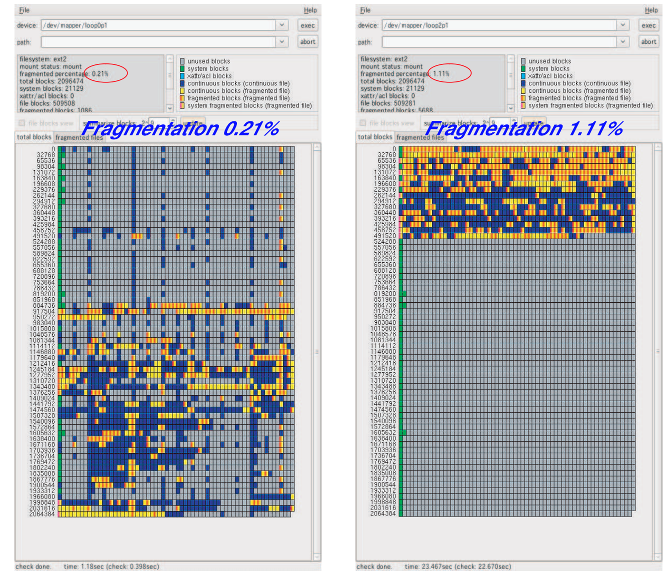



Figure 8: Visualization of data-allocation on ext3 (left is normal and right is ext2/3optimizer) by DAVL.

of ext3. The blue plots indicate the contiguous allocation of data block of file and the yellow plots indicate the non-contiguous allocation. We confirmed that ext2/3optimizer reallocates non-contiguous data at the head of disk. It was the result that ext2/3optimizer exploited the profiled data blocks and aggregated them to the head of the disk. As the result, ext2/3optimizer increased fragmentation from the view of file. DVAL showed that normal ext3 had 0.21% fragmentation but the ext3 optimized by ext2/3optimizer had 1.11%. The relocation however was good for page cache. The coverage of readahead was expected to keep large and occupancy of block file of LBCAS would be high.

Figure 9 shows the access trace of the boot procedure. The x axis indicates the physical address and y axis indicates the elapsed time. The red "+" plots indicate the access on the normal ext3 and the blue "X" plots indicate the access on the ext3 optimized by ext2/3optimizer. The figure showed that the accesses to the normal were scattered. The locality of reference was not good and the effect of page cache and the occupancy of block file of LBCAS would be low. On the other hand, the access to the ext2/3optimizer increased the locality of reference, because the most accesses were the head of disk. The rest spread accesses were the meta data and the volume was little.

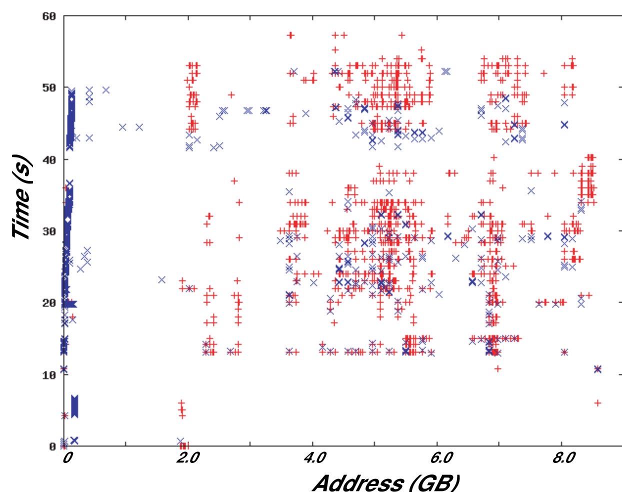



Figure 9: Access trace of boot procedure (RED "+" indicates normal and BLUE "X" indicates ext2/3optimizer.)

### 5.2 User level readahead: system call "readahead"

Ubuntu has the mechanism to populate the page cache with files required at boot time. The files are described at "/etc/readahead/boot" and "/etc/readahead/desktop". The former file listed 937 files and the total volume was 54.1MB. The latter file listed 281 files and the total volume was 25.0MB. the listed files are not all files required boot time. Ubuntu 9.04 requires 2,250 files (203MB) and the half of them are populated on the page cache before they are truly required.

Figure 10 shows the log of bootchart [13], which visualizes the behavior of CPU, I/O, and creation of process at boot time. We confirmed that the same processes were executed at boot time on normal, u-readahead and ext2/3optimizer. The result of u-readahead shows the utilization of I/O increased when u-readahead started. It caused the spike of I/O but the subsequent I/O was little. On the other hand, the I/O was issued on-demand on the normal and ext2/3optimizer. The I/O of ext2/3optimizer was less than the normal. The detail is described in Section 5.3.

Table 1 shows the utilization of CPU and I/O on normal, u-readahead and ext2/3optimizer on 64KB, 128KB, 256KB and 512KB LBCAS. The results shows the u-readahead had higher I/O utilization. It was caused by the redundant read request, because u-readahead read the whole data of files. The I/O utilization of u-readahead was 5-2 times higher than ext2/3optimizer. The results of 512KB LBCAS showed bad I/O utilization on any case. It was caused by the slow response of 512KB LBCAS.

### 5.3 Effect of readahead

Figure 11 shows the Frequency for each readahead coverage size on normal, u-readahead, and ext2/3optimizer.

The figure shows that ext2/3optimizer reduced the small I/O requests. As the result, the frequency of I/O request was reduced to 2,129 from 6,379 and the coverage of readahead was changed to 67KB from 33KB. The total I/O was 140MB and 208MB on ext2/3optimizer and normal respectively. The I/O request is 2 times wider and the frequency of I/O request is 1/3. The effect of frequency is not the inverse of magnification of I/O. The results indicated that the locality of reference is much improved.

On the other hand, u-readahead showed same tendency with normal. The small requests were reduced and the big request were increased a little bit. The total I/O of u-readahead was increased to 231MB from 208MB of the normal. The coverage of readahead was expanded to 41KB but it was small than ext2/3optimizer. The result came from that the u-readahead could not decrease the small I/O, which was cased by the locality of reference. The frequency of I/O was 5,827, which was less than normal 6,379, although the total I/O was increased. The results indicated that ext2/3optimizer was much effective than u-readahead from the view of disk pre-fetch readahead.

### 5.4 Total performance

Figure 12 shows the detail of Ubuntu boot time on KVM from LBCAS. The upper figure shows the ratio of time consumed by LBCAS for each block size. The lower figure shows the consumed time in LBCAS, which was consisted of download time, file read time from storage cache, uncompression time, and time for others.

From the upper figure, we confirmed that ext2/3optimizer was effective for LBCAS at any block size. The LBCAS consuming time was more than 20% on normal, but it was reduced to less than 14% on ext2/3 optimizer. Although the total I/O on u-readahead was more than the normal, the boot time on u-readahead

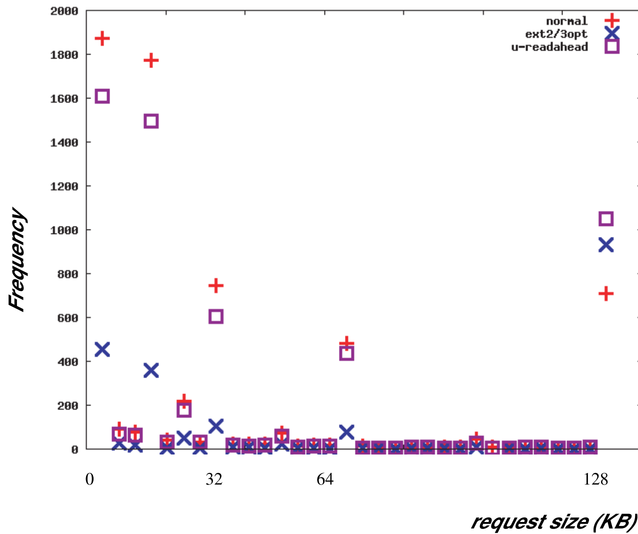



Figure 11: Frequency for each readahead coverage size.

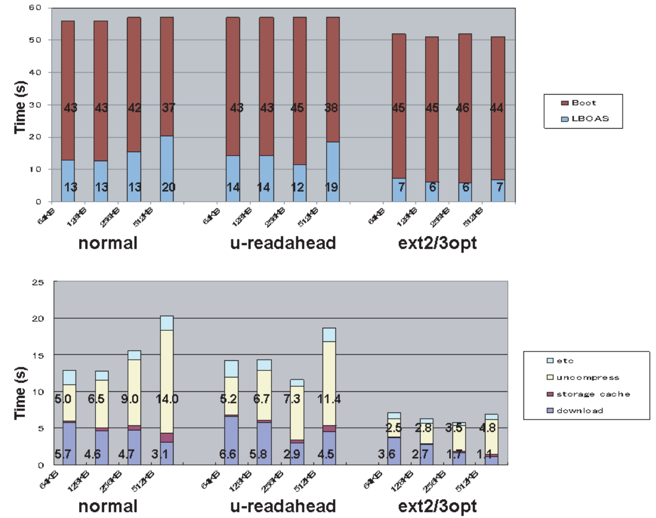



Figure 12: The ratio of consumed time. Upper indicates the ratio of LBCAS in boot procedure. Lower indicates the contents ratio of LBCAS.

was almost same result on normal, The result indicated the page cache populated by u-readahead but it was not effective on KVM.

The lower figure shows the time consuming components of LBCAS. The result shows the most time was consumed by download and uncompression. The download time was longer than the uncompression time at small LBCAS but it was changed at large LBCAS. It was caused by the locality of reference because the small LBCAS was effective from the view of occupancy but it required many block files. On large LBCAS the time of uncompression was increased because of low occupancy in a block file, but the time of download became short because the number of download was fewer and cached on the storage cache. On the 512KB LBCAS the time of uncompression was increased and the total time of LBCAS was the worst.

Table 2 shows the volume transitions at each processing level. The upper table shows the total volume requested from transferred OS: the volume of files which opened by the boot procedure, the block volume which is purely required by the boot procedure, the volume accessed to LBCAS (it includes redundant data covered by readahead). The bottom table shows the status of LBCAS for each block size: the volume of downloaded block files, the volume of uncompressed block files, and the occupancy of effective data in the LBCAS.

From the result, we know that the purely used block was 63% (127MB/203MB) of volume of opened files at boot time. It meant that 37% was not used and it caused inefficient access request of readahead. The readahead for normal ext2 required 208MB access to the LBCAS. The result shows the 81MB (208MB - 127MB) was redundant access. The u-readahead made much worse and 104MB was redundant access. The problem was solved by ext2/3optimizer significantly. The readahead for ext2/3optimize required 140MB. The ext2/3optimizer made 67% better than the normal.

The bottom table shows the status of LBCAS. We confirmed that downloaded files were less than 56MB at any LBCAS size on ext2/3optimizer. However, the normal of 512KB LBCAS requires 144MB, which is 1.67 much larger than 64KB LBCAS (86.1MB). It was caused by bad locality of reference. On ext2/3optimize, the occupancy was almost same on any LBCAS size but it was decreased from 51.5% at 64KB LBCAS to 26.9% at 512KB LBCAS on normal. The result indicated that block reallocation was necessary for LBCAS.

Table 3 shows the frequency of each function of LBCAS for normal, u-readahead, and ext2/3optimizer. I/O requests were issued by guest OS and the frequency was independent of LBCAS. The rest columns indicated the function of LBCAS. The number of uncompress is summation of the number of download and storage cache. The summation of uncompress and memory cache is the total used files on the LBCAS.

The results showed storage cache and memory cache worked well. Especially the two caches were effective on large LBCAS size. The frequency of storage cache and memory cache were more than the frequency of download and uncompression.

The uncompression of ext2/3optimizer was less than half of normal case at each LBCAS size. The result corresponds to time of uncompression at the Figure 12. The decrease affected the performance of LBCAS.

Figure 13 shows the amount of downloaded block file at boot time. The LBCAS size was 256KB. The result shows the ext2/3optimizer reduced the amount of download and made quick boot. The u-readahead downloads many block files around 15 second because it populated the page cache with listed files. It increased the total download but made quick boot. However the boot time of u-readahead was slower than ext2/3optimizer.

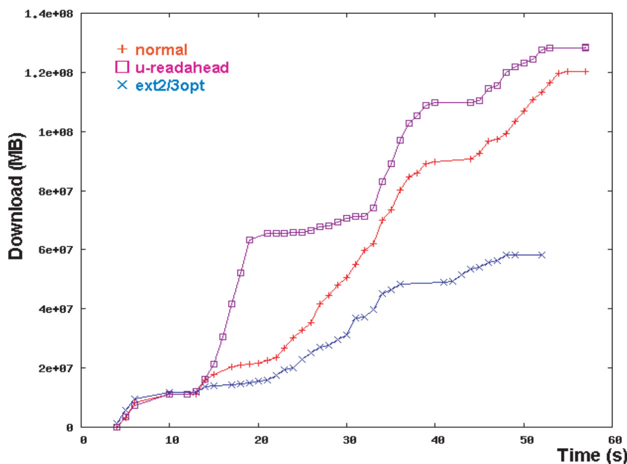



Figure 13: Amount of Downloaded Block File (256KB) at boot time.

## 6 Discussions

The data blocks are reallocated in order to be in line according to the access profile. It results in keeping large coverage of readahead at the boot procedure. It makes quick boot but the data blocks are fragmented from the view of file. The optimization is too tight and it would not fit to another access pattern. If the reallocated data blocks are used in another application, the access pattern can not get large coverage of readahead. However, boot procedure is special and several files are used at boot procedure only. We have to estimate the special files and its ratio, which are not used for other applications.

Most reallocation tools aim to reduce fragmentation and quick access is side effect. Unfortunately the effect of quick access looks to be insufficient, even if the File System has no fragmentation. The original disk image used in section 5 was first install image and there are few fragmentation. The trace of boot procedure, however, showed discrete access. In order to make quick access we should reallocate blocks based on access profile.

A feature of CAS is sharing of block with same hash value. It reduces the total volume of contents. The sharing is not effective on a single OS image but it is effective on some Linux distributions [14] and multi user environment [15]. [14] told the CAS block sharing on Fedora, Ubuntu, and OpenSuse was 10% - 30%. Although our paper does not describe the effect of sharing, the ext2/3optimizer will reduce the effect because it reallocates most of data blocks in ext2/3 file system. If the sharing is important factor, the reallocation tool has to consider the sharing as far as possible.

## 7 Conclusions

We offered an virtual block device called "LBCAS", which manages each block by indirect mapping of SHA-1 value of its contents. The performance was affected by the number of I/O request which is issued by readahead of disk pre-fetching. The number of I/O request is related to the coverage size of readahead. The coverage is expanded by high hit ratio of page cache The hit ratio is increased by locality of reference.

We developed ext2/3optimizer to reallocate the data block of ext2/3 according to the access profile. We applied ext2/3optimizer on Ubuntu 9.04 according to the access profile of boot procedure. We compared the effect with the user-land readahead which uses Linux system call "readahead" and populates the page cache with data of files in advance. As the result, the coverage of readahead expanded to double and the I/O requests reduced in half on ext2/3optimizer. The user-land readahead could also expand the coverage of readahead and reduce the I/O requests but the effect was less than ext2/3optimizer.

The key was the locality of reference which is improved by ext2/3optimizer. The effect of locality of reference also reduced the necessary block files on LBCAS at boot time. The result showed that the optimization was necessary for the OS migration with LBCAS, which is aimed of the OS Circular project.

The source code of tools are available at the project home page.

## References


[1] `http://openlab.jp/oscircular/`

[2] K. Suzaki, *OS Circular: Internet bootable OS Archive*, LinuxConf.Australia, January, 2009.

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

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

[5] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, T. Bressoud, and A. Perrig, *Opportunistic use of content addressable storage for distributed file systems*, Proceedings on USENIX Annual Technical Conference, pages 127–140, San Antonio, TX, June 2003.

[6] Mechiel Lukkein, *Venti analysis and memventi implementation*, Master's thesis of University of Twente, 2008.

[7] WU. Fengguang, XI. Hongsheng, and XU. Chenfeng, *On the design of a new Linux readahead framework*, ACM SIGOPS Operating Systems Review, Volume 42, Issue 5, pp. 75–84, July, 2008.

[8] WU. Fengguang, XI. Hongsheng, J. Li, and N. Zou, *Linux readahead: less tricks for more*, Proceedings of the Linux Symposium, Vol.2, pages 273–284, 2007.

[9] K. Kitagawa, H. Tan, D. Abe, D. Chiba, K. Suzaki, K. Iijima, and T. Yagi, *File System (Ext2) Optimization for Compressed Loopback Device*, 13th International Linux System Technology Conference, pp. 25–33, Nurnberg Germany, September, 2006.

[10] `http://fuse.sourceforge.net/`

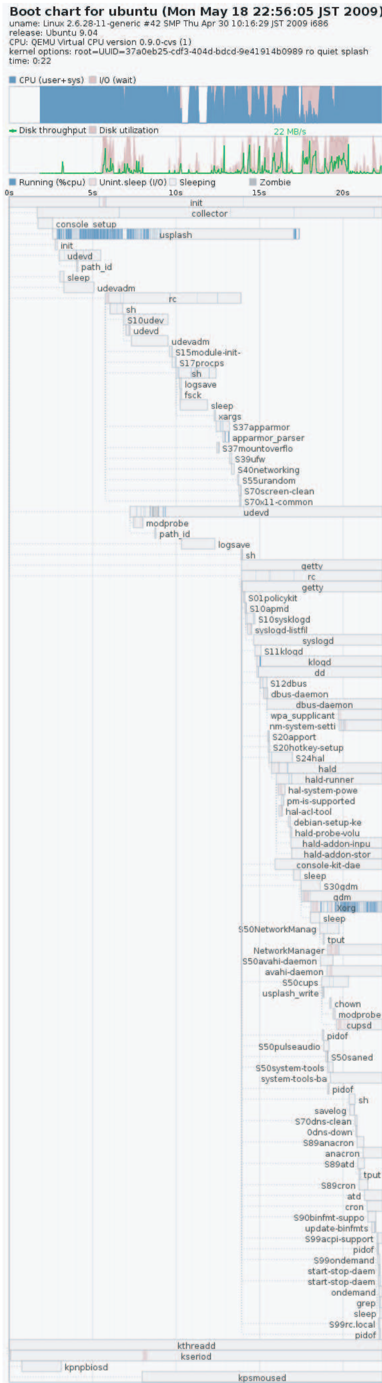
[11] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, *kvm: the Linux Virtual Machine Monitor*, Proceedings of Linux Symposium 2007, Volume 1, pages 225–230, June 2007.

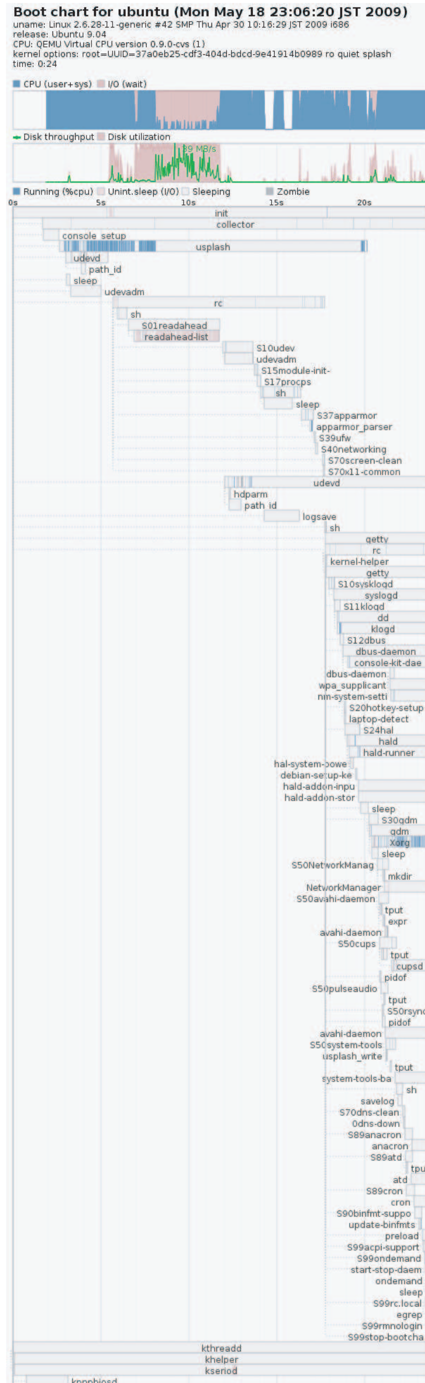
[12] `http://sourceforge.net/projects/davl/`

[13] `http://www.bootchart.org/`

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

[15] P. Nath, M.A. Kozuch, D.R. O'Hallaron, J. Harkes, M. Satyanarayanan, N. Tolia, and M. Toups, *Design Tradeoffs in Applying Content Addressable Storage to Enterprise-scale Systems Based on Virtual Machines*, USENIX Annual Technical Conference, pp. 71–84, Boston MA, 2006.



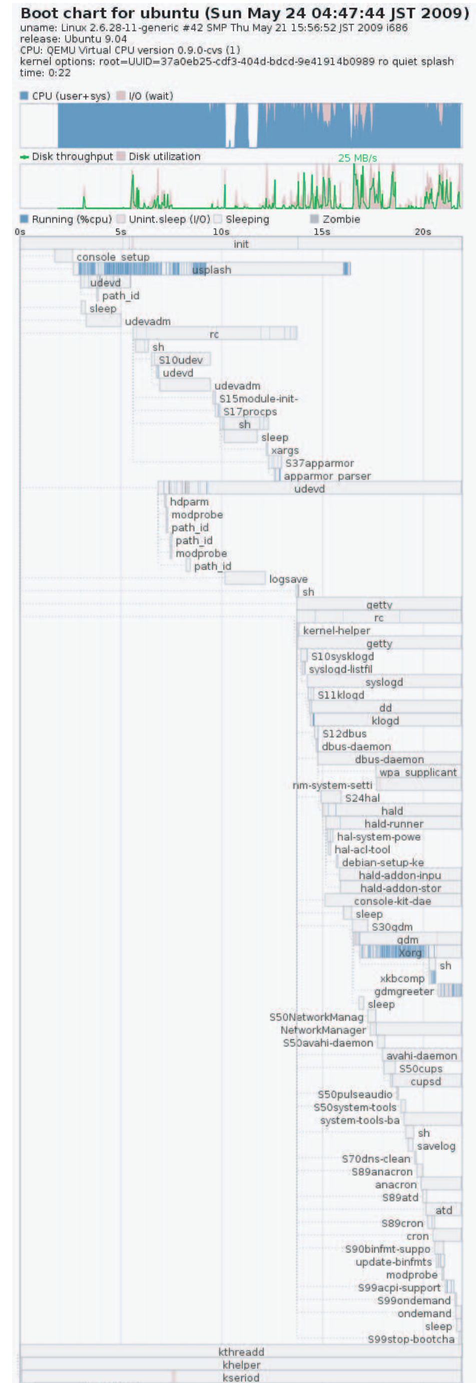




Figure 10: BootChart. The visualization of CPU utilization, I/O utilization and created processes at boot time. The left is normal, the middle is u-readahead, and the right is ext2/3optimizer.

| LBCAS size | normal | | u-readahead | | ext2/3opt | |
|---|---|---|---|---|---|---|
| | CPU (%) | I/O (%) | CPU (%) | I/O (%) | CPU (%) | I/O (%) |
| 64KB | 87.8 | 12.2 | 74.1 | 25.9 | 94.9 | 5.1 |
| 128KB | 84.9 | 15.1 | 81.1 | 18.9 | 94.4 | 5.6 |
| 256KB | 86.0 | 14.0 | 79.4 | 20.6 | 93.7 | 6.3 |
| 512KB | 78.7 | 21.3 | 74.3 | 25.7 | 89.0 | 11.0 |

Table 1: Utilization of CPU and I/O, which was taken by BootChart.

| | Normal | u-readahead | ext2/3optimizer |
|---|---|---|---|
| Volume of files (number, average) | 203MB (2248 Av:92KB) | | |
| Volume of requested blocks | 127MB | | |
| Volume of required access which includes the coverage of Readahead (average number of access and size of readahead) | 208MB (6,379 Av:33KB) | 231MB(5,827 Av:41KB) | 140MB(2,129 Av:67KB) |

| LBCAS size | Downloaded size MB (Uncompressed size MB), Occupancy % | | |
|---|---|---|---|
| 64KB | 86.1(247), 51.5% | 93.4(272), 46.9% | 55.3(144), 88.7% |
| 128KB | 96.8(290), 43.9% | 104(315), 40.3% | 55.3(149), 85.3% |
| 256KB | 114(358), 35.5% | 123(386), 35.0% | 55.6(159), 80.0% |
| 512KB | 144(474), 26.9% | 153(508), 25.1% | 55.6(176), 71.8% |

Table 2: Volume transitions at each processing level. The upper table indicates the volume transition on guest OS. The bottom table indicates the volume transition on LBCAS.

| Normal | Requests (Av. size 33KB) (R) | Download (D) | Storage Cache (S) | Uncompress (U)=(D)+(S) | Memory Cache (M) | Files per request (R)=(1)+(2)+(3) (U)+(M)=(1)+(2)*2+(3)*3 |
|---|---|---|---|---|---|---|
| 64K | 6,338 | 3,958 | 1,663 | 5,621 | 3,647 | (1) 4,148<br>(2) 1,450<br>(3) 740 |
| 128K | 6,381 | 2,321 | 1,729 | 4,050 | 3,793 | (1) 4,919<br>(2) 1,462 |
| 256K | 6,379 | 1,435 | 1,748 | 3,183 | 3,908 | (1) 5,667<br>(2) 712 |
| 512K | 6,395 | 948 | 1,769 | 2,717 | 4,019 | (1) 6,054<br>(2) 341 |
| u-readahead | (Av: size 41KB) | | | | | |
| 64K | 5,825 | 4,344 | 1,172 | 5,516 | 3,626 | (1) 3,537<br>(2) 1,259<br>(3) 1029 |
| 128K | 5,834 | 2,526 | 1,200 | 3,726 | 3,761 | (1)4,181<br>(2) 1,653 |
| 256K | 5,827 | 1,544 | 1,179 | 2,723 | 3,908 | (1) 5,023<br>(2) 804 |
| 512K | 5,822 | 1,015 | 1,172 | 2,187 | 4,023 | (1) 5,434<br>(2) 388 |
| ext2/3opt | (Av: size 67KB) | | | | | |
| 64K | 2,165 | 2,296 | 626 | 2,922 | 1,311 | (1) 941<br>(2) 380<br>(3) 844 |
| 128K | 2,148 | 1,189 | 593 | 1,782 | 1,398 | (1) 1,116<br>(2) 1,032 |
| 256K | 2,129 | 634 | 576 | 1,210 | 1,409 | (1) 1,639<br>(2) 490 |
| 512K | 2,132 | 353 | 517 | 870 | 1,520 | (1) 1,874<br>(2) 258 |

Table 3: Frequency of function of LBCAS. Upper table shows the normal case. Lower table shows the ext2/3optimizer case. “Requests” indicates the number of I/O issued by guest OS. The rest columns show the frequency of each function of LBCAS. “Files per request” indicates the frequency of downloads for files per a request.

# Scaling software on multi-core through co-scheduling of related tasks

Srivatsa Vaddagiri Bharata B Rao Vaidyanathan Srinivasan Anithra P Janakiraman
Balbir Singh Vijay K Sukthankar

*IBM India Software Labs, Bangalore*
{vatsa, bharata, svaidy, janithra, balbir, vksuktha}@in.ibm.com

## Abstract

Ever increasing demand for more processing power, coupled with problems in designing higher frequency chips are forcing CPU vendors to take the multi-core route. IBM® introduced the first multi-core processor with its POWER4® in 2001, that had two cores in a chip and also 4 chips in a package. Other CPU vendors have followed the trend with dual and quad-core processors becoming increasingly common. It is estimated that by year 2021, there will be chips with 1024 cores on them [6]. Such platforms pose huge challenge on how software effectively utilizes so many cores. One problem of interest is how tasks are scheduled on such platforms. The existing Linux scheduler attempts to distribute tasks equally among all CPU chips. It does not optimize this task placement, taking into consideration that all tasks need not be equal with respect to their use of shared CPU resources (like L2 cache). In this paper, we look at how misplacement of tasks across CPU chips can significantly affect performance and how existing Linux interface to solve that problem is inflexible. We present a new interface which can be used by applications to hint which threads share data closely and thus should be co-scheduled on *neighbouring*[1] CPUs to the extent possible by OS scheduler. We present several results showing the inflexibility of existing interface and how the suggested interface solves those problems.

## 1 Trends in modern system architecture

Modern multi-core processors have innovative and complex cache hierarchy design in order to hide memory access latency and optimize bandwidth on various intra-chip and inter-chip interconnect buses. With faster CPUs, application performance is now becoming bound on the availability of its working data set in *local* CPU cache.

Table 1 aptly illustrates this point using *c2cbench* [1], a benchmark that measures the cost of data transfer between two caches. The benchmark was used to measure throughput for transferring 256KB of data between a producer and consumer thread.[2,3] By controlling the CPUs on which two threads run, the benchmarks measures cost of cache-to-cache transfer. Best throughput is seen when both threads are co-scheduled on sibling cores (which share the same L2/L3 cache). The throughput drops by a factor of 4-6 when the threads are forced to run on cores that don't share the cache hierarchy. An interesting data point from Table 1 that represent typical system cache topology is that co-scheduling producer/consumer tasks on sibling hardware threads gives best performance since they share most of the cache hierarchy. The benefit of cache sharing is outweighing the cost of contention for shared execution resources in the core.

Although memory and inter-chip interconnect bandwidth has been increasing in each generation of processors, the trend seems to indicate that the ratio of access latency between remote and local cache will continue to be significant. Thus we can conclude that task placement can significantly affect performance, especially for scenarios where two or more tasks work closely on shared data.[4] Co-scheduling such related tasks on *neighbouring* CPUs can improve performance by making best use of shared cache hierarchy.

[1] *Neighbouring* CPUs are those that share some or all of a cache hierarchy.

[2] c2cbench -P0 -C1 -prw -crd -d4096 -b256 -s8 -k1 -K0 -I1000

[3] The terms *thread* and *task* are used interchangeably throughout the paper.

[4] The term *thread cluster* is used to refer to a group of tasks that work closely on some shared data.

| *Relative throughput for data sharing (GB/sec)* | *Sibling hardware threads* | *On-chip cores* | *Off-chip cores* |
|---|---|---|---|
| IBM POWER5® | 3.9*a* | 4.3*a* | 1*a* |
| IBM POWER6® | 6.4*b* | 1.4*b* | 1*b* |
| Intel® Xeon Quad Core | N/A | 6.5*c* | 1*c* |
| Intel Core i7 | 4.1*d* | 2.1*d* | 1*d* |

Table 1: Producer-consumer throughput for 256KB transfer

| *Scenario* | *No-co-scheduling case* (million records/sec) | *Co-scheduling case* (million records/sec) | *Impact of co-scheduling* |
|---|---|---|---|
| Two instances | 8.76 | 9.71 | **+10.84%** |
| Single instance | 15.73 | 9.74 | **-38%** |

Table 2: Co-scheduling *ebizzy* instances

| *Scenario* | *No-co-scheduling case* (seconds) | *Co-scheduling case* (seconds) | *Impact of co-scheduling* |
|---|---|---|---|
| Two instances | 209.44 | 207.79 | **+0.78%** |
| Single instance | 107.42 | 203.92 | **-89.8%** |

Table 3: Co-scheduling *kernbench* instances

| *Metric* | *No co-scheduling* (million records/sec) | *Co-scheduling* (million records/sec) | *Impact of co-scheduling* |
|---|---|---|---|
| VM1 Throughput | 5.85 | 5.95 | **+1.7%** |
| VM2 Throughput | 3.64 | 5.65 | **+55.22%** |
| VM3 Throughput | 7.67 | 7.27 | **-5.22%** |

Table 4: Co-scheduling KVM VMs

| *Scenario* | *No co-scheduling* (seconds) | *Co-scheduling* (seconds) | *Impact of co-scheduling* |
|---|---|---|---|
| Two instances | 1x | 0.8846x | **+11.54%** |
| Single instance | 1x | 1.2339x | **-23.39%** |

Table 5: Co-scheduling Trade6 application

## 2 Co-scheduling opportunities

In this section, we look at few opportunities that exist in real world where we can co-schedule related threads on *neighbouring* CPUs for improving performance.

### 2.1 Multiple instances of same applications

In many cases, multiple instances of the same application are launched. For example, multiple users launching same compiler program to compile their program, multiple application servers launched on the same machine as a vertical cluster [2] etc. Probability of data sharing between threads of an instance is higher than between threads across instances. Co-scheduling threads of an instance on *neighbouring* CPUs could potentially yield better performance, *provided* the opportunity exists to utilize remaining CPUs for other work.

Table 2 shows the results of co-scheduling for *ebizzy* [3] benchmark, a workload resembling web application server. The benchmark creates several threads that search for a random key from the same memory region. The memory region thus is shared between all threads of the benchmark.

In first scenario, two instances of *ebizzy* are launched simultaneously on a machine having two dual-core Intel Xeon® CPUs (with 4MB shared L2 cache). In no-co-scheduling case, they were not bound to any CPU and in co-scheduling case, each instance was bound to a separate dual-core CPU. Co-scheduling gives good results in this scenario. In the second scenario, only one instance is launched. Co-scheduling that single instance, which means binding that instance to a single dual-core CPU, does not give good results in this scenario. This is because the single instance, being *hard*-bound to single dual-core CPU, is not effectively making use of all the available (idle) CPUs in the system.

Table 3 shows the results of co-scheduling for *kernbench*, a Linux kernel compilation benchmark. On the same machine described above, two instances of *kernbench* are launched simultaneously in first scenario. Each instance spawns 11 threads for compiling different source files in parallel. Each of those 11 threads will compile its own source file and hence there is very little data sharing between threads of an instance. Co-scheduling in this scenario will not give any benefit and in the second scenario of single instance is actually *hurting* performance.

### 2.2 Virtualization

Power, cooling and real-estate constraints in data centers are forcing customers to consolidate their applications on fewer and powerful machines. Advanced virtualization capabilities of modern processors are being fully utilized to carve several virtual machines (VM) out of a single machine. Each VM gets the illusion as if it has its own set of hardware resources (CPUs, memory etc). The mapping of virtual resources of a VM to underlying physical resources is managed by a hypervisor software. For example, in case of CPUs, the hypervisor will schedule the different virtual CPUs (VCPU) of a VM on different physical CPUs.

Typically each VM hosts a single application, say a database server or webserver. In such a case, data sharing is more likely to occur between threads belonging to the same VM rather than between threads of different VMs. Thus it makes sense to consider co-scheduling different VCPUs of a VM on *neighbouring* CPUs, *provided* the opportunity exists to utilize remaining CPUs for other work.

In an experiment involving KVM based virtualization, 3 VMs, VM1, VM2 and VM3, were launched on a machine having 2 quad-core Intel Xeon CPUs. *ebizzy* benchmark was started simultaneously on all three VMs. In the first case, VMs were not bound to any CPU. In the next case, VM1 and VM2 were bound to two different quad-core CPUs and VM3 was not bound to any CPU. The results shown in Table 4 shows that co-scheduling helps improve the performance of *ebizzy* benchmark running inside VM1 and VM2.

### 2.3 Application Server

Java application servers like WebSphere® Application Server (WAS) are used to host business applications written in J2EE. The same application server can host multiple applications or multiple application instances on the same node. Probability of data sharing is higher between threads of the same application (instance) and hence an application (instance) could form the basis for co-scheduling threads. In case of applications like YouTube or online gaming, it is possible to group threads at a even much finer granularity. For example, all threads serving the same video/photo-album or all threads serving players of the same game instance could be grouped together to form a cluster.

Table 5 shows the result of co-scheduling for Trade6 application on a server with two dual-core Intel Xeon CPUs. Time taken to complete the benchmark is shown on a relative scale, with the *No co-scheduling* case forming the baseline to compare against. In first scenario, two Trade6 instances are launched. Co-scheduling each instance on a separate dual-core CPU results in better performance compared to not co-scheduling any instance. In the second scenario, a single instance is launched. Co-scheduling that single instance (which mean *hard*-binding it to a single dual-core CPU) is actually *hurting* performance in this case, as it does not utilize fully all the available CPU resources.

The key observations from these experiments are:

1. Co-scheduling helps improve performance for certain workloads, where high degree of data sharing exists between threads.

2. Co-scheduling should not be at the cost of idling CPUs. In other words, its better to *break* co-scheduling in favor of utilizing as many required (idle) CPUs.

## 3 Detecting co-scheduling opportunities

In Section 2, we saw that opportunities exists in real-world for improving performance on multi-core systems by co-scheduling related threads. How do we detect such opportunities? In most cases, it is done with manual intervention—after carefully studying the workload and the platform behavior. Co-scheduling is achieved using existing interfaces like *sched_setaffinity* and *cpuset*. Beyond providing the raw support to co-schedule tasks, Linux doesn't have any capability to *automatically* detect co-scheduling opportunities and co-schedule selective tasks based on that.

### 3.1 Automatic detection

[8] describes one mechanism to automatically determine co-scheduling opportunities on IBM Power5-based multi-core platform, based on observing certain HPCs (Hardware Performance Counter) related to cache-miss events. The algorithm described is however quite complex and it remains to be seen how easily it can be adapted to a general purpose operating system like Linux.

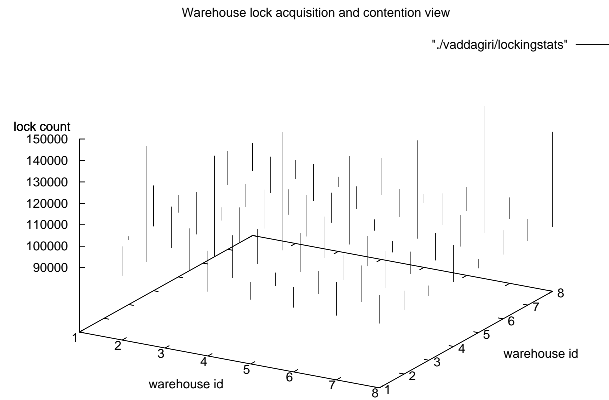



Figure 1: Warehouse lock acquisition and contention view

We present below a more simpler approach which could form the basis of *automatic* co-scheduling. The approach is based on the fact that data sharing between threads generally involves them acquiring the same locks guarding shared data access. Analyzing lock acquisitions can give us a clue on threads that are closely working on shared data. Once such threads groups are detected, we could automatically co-schedule them on *neighbouring* CPUs using the interface described in Section 4.2.

### 3.2 Workload

We ran SPECjbb2000 [4] and modified the default configuration of SPECjbb so that multiple threads (terminals) can simultaneously access the warehouse. In our experiments we used 8 warehouses with 4 threads per warehouse. We instrumented the benchmark to collect information about threads and which warehouse they belonged to.

### 3.3 Results

Figure 1 shows a plot depicting the lock acquisition and contention count for each of the threads by their warehouse ID. The same data is show in numerical tabular form below, Table 3.1.

As can be seen from Table 3.1, the highest locking was seen between the threads belonging to the same warehouse. Figure 1 displays the same graphically. The data was obtained by instrumenting the mutual exclusion paths on a per thread and a per mutex basis. This

| *Warehouse Id* | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | **136774** | 103674 | 91826 | 109088 | 98283 | 99615 | 105770 | 103254 |
| 2 | 103674 | **143964** | 109358 | 109848 | 96172 | 100722 | 106946 | 98890 |
| 3 | 91826 | 109358 | **136828** | 106294 | 108150 | 101856 | 107154 | 94878 |
| 4 | 109088 | 109848 | 106294 | **145206** | 109430 | 104000 | 100534 | 107342 |
| 5 | 98283 | 96172 | 108150 | 109430 | **131296** | 95266 | 102882 | 94316 |
| 6 | 99615 | 100722 | 101856 | 104000 | 95266 | **135796** | 104676 | 101312 |
| 7 | 105770 | 106946 | 107154 | 100534 | 102882 | 104676 | **149144** | 100070 |
| 8 | 103254 | 98890 | 94878 | 107342 | 94316 | 101312 | 100070 | **134370** |

Table 6: Warehouse to warehouse lock acquisition and contention count

data was then summed to extract thread to thread locking statistics by summing lock acquisition counts for each mutex and thread pairs. The warehouse data was obtained by summing the lock statistics for all threads belong to the warehouse.

### 3.4 Observations

The results obtained from the experiments above indicate that

1. Although all threads in a process share the same address space, the working data set could be be different for each thread.
2. A group of threads could share the same working set to form a thread cluster.
3. Co-scheduling such thread clusters on *neighbouring* CPUs should help improve performance (as proven in this case by Figure 6 and Figure 7).

## 4 Co-scheduling interface

Once co-scheduling opportunities are determined, either manually or automatically, co-scheduling related tasks together on *neighbouring* CPUs is accomplished using interfaces such as *sched_setaffinity* or *cpuset*.

### 4.1 *Hard* affinity interface

Both *sched_setaffinity* and *cpuset* provide the ability to control where tasks execute. Using these interfaces it is possible to co-schedule threads of a cluster on *neighbouring* CPUs. The biggest drawback with these interfaces is the *hard*-affinity it creates between tasks and CPUs, because of which it can actually *hurt* performance sometimes (as highlighted by *Single instance* scenario of Table 2). What would be better is a *soft*-affinity interface, which would allow threads to be *soft*-bound to CPUs.

### 4.2 *soft* affinity interface

The *soft*-affinity interface allows applications or administrators to register thread clusters. The CPU scheduler would then *automatically* co-schedule threads of a cluster on *neighbouring* CPUs, *provided* other CPUs can be used for executing other work. In case no other work exists, then scheduler would break co-scheduling of a thread cluster in favor of utilizing all required CPUs for the cluster.

The interface to register thread clusters is built on top of the cgroup process-grouping feature of Linux kernel [7]. A new cgroup subsystem, called *co-scheduler*, was written to mediate between user space and scheduler (Figure 2). The *co-scheduler* subsystem provides a filesystem based API (with help of *cgroup* subsystem) for thread clusters to be registered. The API allows creation/deletion of thread clusters or movement of threads from one cluster to another (Figure 3). The *co-scheduler* subsystem closely tracks the load of each cluster across various CPUs (Figure 4), based on which it will *automatically* co-schedule threads of few clusters on *neighbouring* CPUs. Co-scheduling of threads is accomplished by manipulating their CPU affinity. A high-level flowchart for the working of *co-scheduler* subsystem is shown in Figure 5.

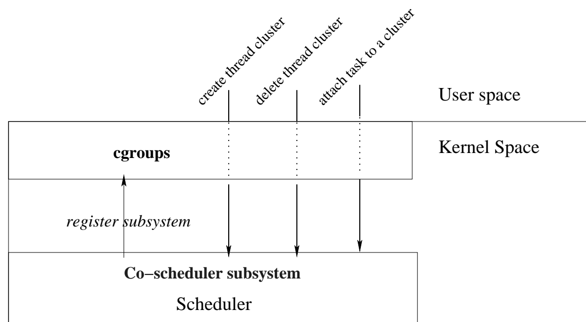



Figure 2: Co-scheduler subsystem

```
# mkdir /cgroup
# mount −t cgroup −o coscheduler none /cgroup
# cd /cgroup
# mkdir cluster1
# mkdir cluster2
# /bin/echo pid1 > cluster1/tasks
# /bin/echo pid2 > cluster2/tasks
```

Figure 3: Registering thread clusters

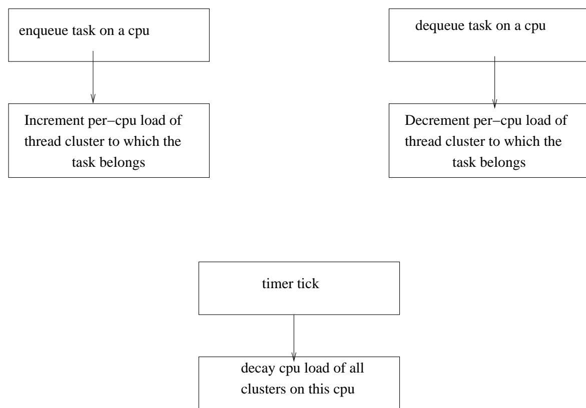



Figure 4: Tracking cluster load

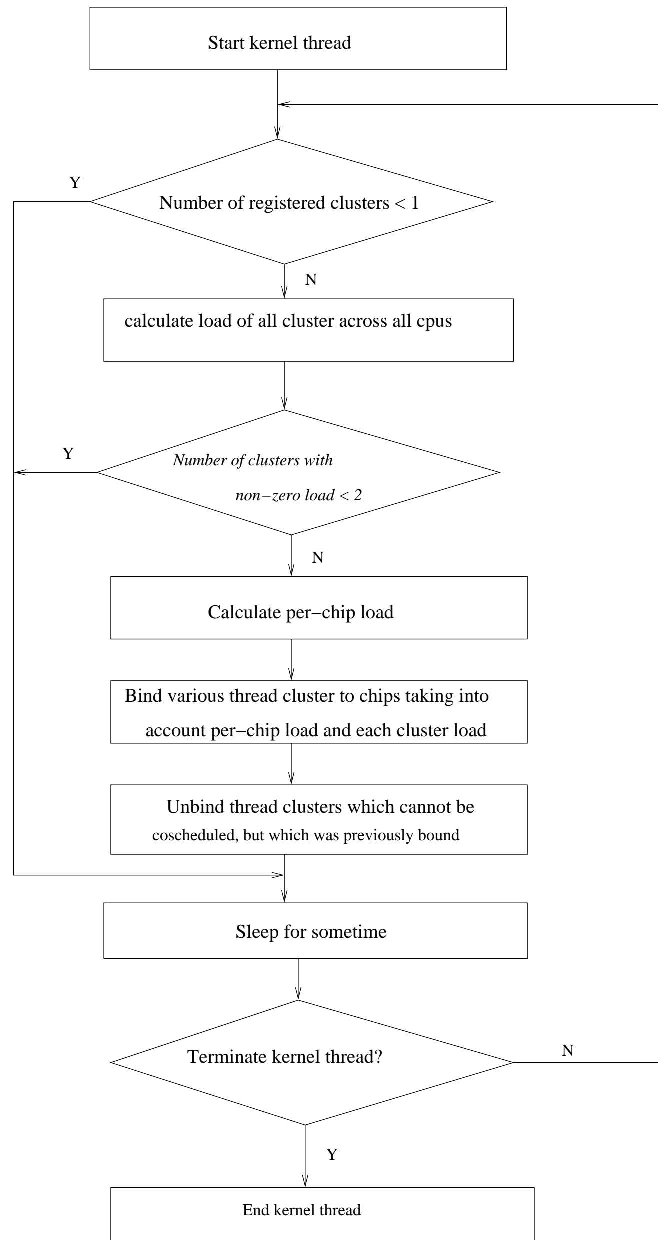



Figure 5: Co-scheduler operation

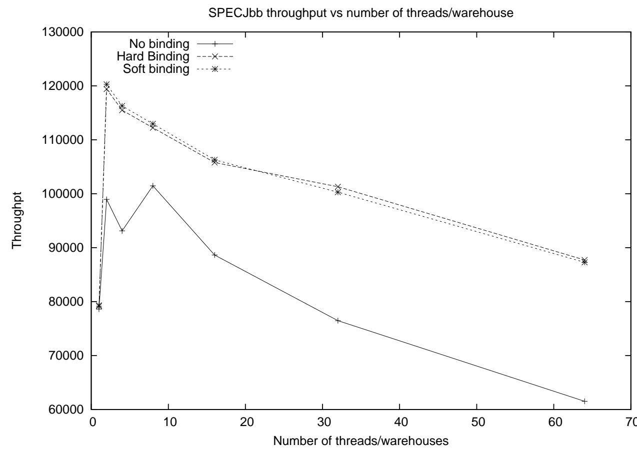



Figure 6: SPECJbb2000—Absolute throughput

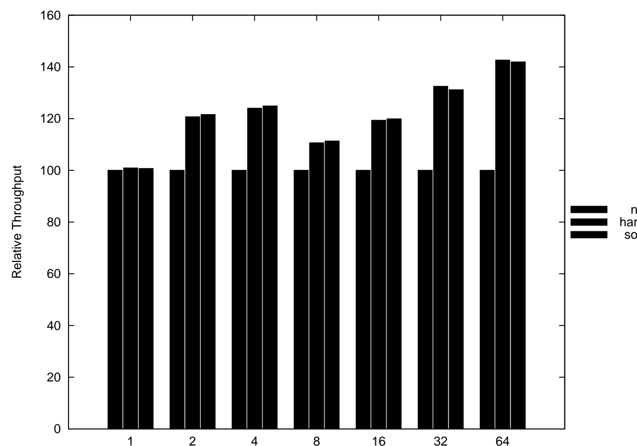



Figure 7: SPECJbb2000—relative throughput

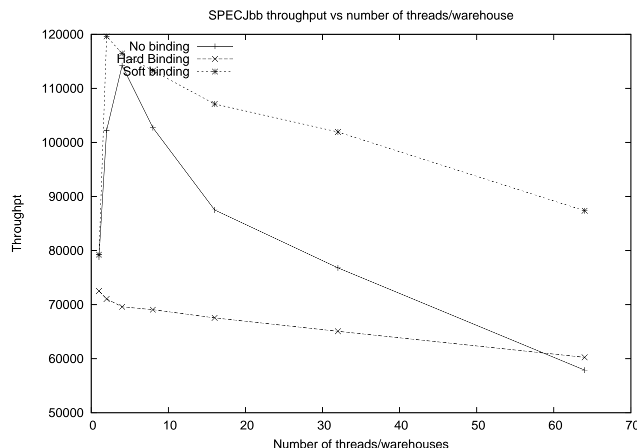



Figure 8: SPECJbb2000—Two warehouses with varying number of threads

#### 4.2.1 Results

Some results comparing *hard-* and *soft*-affinity are provided below:

1. SPECJbb
   SPECJbb [4] is a Java benchmark used to evaluate Java performance. The benchmark creates several warehouses and several threads (or terminals) per warehouse. Threads associated with the same warehouse will very likely access the same data that is associated with the warehouse. The benchmark was modified to bind threads using both the *hard-* and *soft*-affinity interfaces. Figure 6 shows the results of using the interfaces on a system having two dual-core Intel Xeon CPUs. Two warehouses were created and the number of threads per warehouse was varied from 1 to 64. In case of *hard*-affinity, threads belonging to first warehouse were bound (using *sched_setaffinity*) to first dual-core CPU while threads belonging to second warehouse were bound to the second dual-core CPU. In case of *soft*-affinity, threads of the both warehouses were registered as separate clusters. The results show that binding, through either *soft*-affinity or *hard*-affinity, provides better results. Also *soft*-affinity is giving equally good results as *hard*-affinity. Figure 7 shows the same results on a relative scale (with reference to the results obtained without binding any threads).

   Figure 8 shows some results which exposes the weakness with *hard*-affinity. In this case, the number of warehouse was kept constant at 2, while the number of threads/warehouse was varied from 1 to 64. For the *hard*-affinity case, threads of both warehouses were bound to first dual-core CPU, which causes a gross under-utilization of resources. For the *soft*-affinity case, threads of each warehouse were registered as a separate cluster. The results show that *hard*-affinity gives poorer results compared to not binding any threads. Also *soft*-affinity is giving best performance compared to no-binding or *hard*-affinity by deciding to schedule threads of two warehouses on separate dual-core CPUs.

2. Java application server
   IBM Trade Performance Benchmark Sample [5] for WebSphere Application Server or Trade6 is the fourth generation of WebSphere end-to-end benchmark and performance sample application, which

simulates a real-world workload. To study the impact of co-scheduling threads of the same JVM instance, we used up to 5 WebSphere Application Server profiles each running its own installation of Trade6 on a machine having two dual-core Intel Xeon CPUs. Each of the Trade6 instances was configured to use its own DB2 instance as the backend. The Trade6 application was stressed using the WebSphere Studio Workload Simulator engine (iwlengine) which generates a set of requests continuously till a particular runtime is reached.

For the purpose of this experiment the iwlengine script was modified to generate a fixed number of requests.The number of clients was fixed at 50. The results were first collected for 2 instances of Trade6 that were stressed simultaneously. Performance was measured using the iwlengine in terms of throughput and time taken. The threads of each WebSphere instance are likely to access the same data that is associated with the that WebSphere/Trade6 instance. This was exploited using both the *hard*- and *soft*- affinity interfaces. In case of *hard*-affinity, threads belonging to the first instance were bound (using *sched_setaffinity*) to the first dual-core CPU while threads belonging to second instance were bound to the second dual-core CPU. In case of *soft*-affinity, threads of both instances were registered as separate clusters.This experiment was repeated for 3, 4 and 5 application server instances.

Figure 9 shows the results of binding on a relative scale (with reference to the results obtained without binding any threads). The results shows that binding improves the throughput significantly. In some cases *soft*- affinity gives better results which could be attributed to the fact that *soft*- affinity gives priority to CPU utilization over co-scheduling.

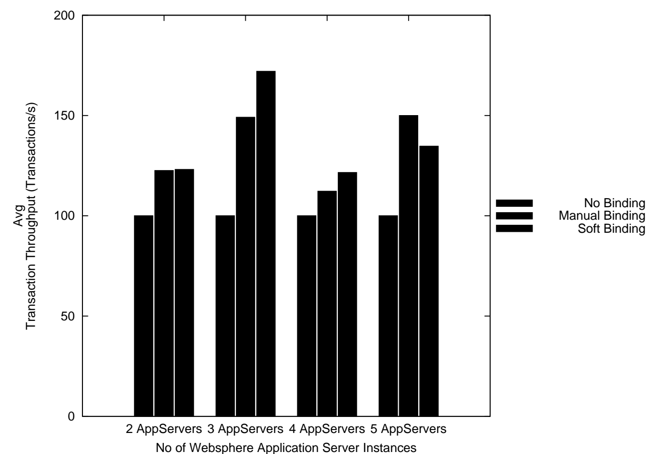



Figure 9: Trade6—Relative throughput

## 5 Acknowledgments

The authors thank IBM management (Premalatha M Nair, Naren A Devaiah, Naveen Kamat, Thomas Domin, Kalpana Margabandhu) for being supportive of this work. A special thanks goes to Manish Gupta (Associate Director, IBM India Research Labs) for prodding the authors to think about multi-core issues and who was instrumental in driving the idea of using lock-contention to form thread-clusters.

## 6 Legal Statement

©International Business Machines Corporation 2009.

Permission to redistribute in accordance with Linux Symposium submission guidelines is granted; all other rights reserved.

This work represents the view of the authors and does not necessarily represent the view of IBM. IBM, IBM logo, ibm.com, and WebSphere, are trademarks of International Business Machines Corporation in the United States, other countries, or both. Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries. Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both. Other company, product, and service names may be trademarks or service marks of others.

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

## References


[1] Cache to cache producer-consumer benchmark. http://sourceforge.net/projects/c2cbench.

[2] Clustering with vertical cluster members. http://publib.boulder.ibm.com/infocenter/wchelp/v6r0m0/index.jsp?topic=%/com.ibm.commerce.admin.doc/tasks/tigvertcluster.htm.

[3] Ebizzy benchmark. http://sourceforge.net/projects/ebizzy/.

[4] Specjbb benchmark. http://www.spec.org/jbb2005/docs/WhitePaper.html.

[5] Trade performance benchmark for websphere application server. http://www.ibm.com/software/webservers/appserv/was/performance.html.

[6] F. Allen. Fran Allen talk on parallel computing. http://www.windley.com/archives/2008/02/fran_allen_compilers_and_parall%el_computing_systems.shtml.

[7] P. B. Menage. Resource control and isolation: Adding generic process containers to the linux kernel. http://ols.108.redhat.com/2007/Reprints/menage-Reprint.pdf.

[8] D. Tam, R. Azimi, and M. Stumm. Thread clustering: Sharing-aware scheduling on smp-cmp-smt multiprocessors. In *in EuroSys*, 2007.

# Converged Networking in the Data Center

Peter P. Waskiewicz Jr.
*LAN Access Division, Intel Corp.*
peter.p.waskiewicz.jr@intel.com

## Abstract

The networking world in Linux has undergone some significant changes in the past two years. With the expansion of multiqueue networking, coupled with the growing abundance of multi-core computers with 10 Gigabit Ethernet, the concept of efficiently converging different network flows becomes a real possibility.

This paper presents the concepts behind network convergence. Using the IEEE 802.1Qaz Priority Grouping and Data Center Bridging concepts to group multiple traffic flows, this paper will demonstrate how different types of traffic, such as storage and LAN traffic, can efficiently coexist on the same physical connection. With the support of multi-core systems and MSI-X, these different traffic flows can achieve latency and throughput comparable to the same traffic types' specialized adapters.

## 1 Introduction

Ethernet continues to march forward in today's computing environment. It has now reached a point where PCI Express devices running at 10GbE are becoming more common and more affordable. The question is, what do we do with all the bandwidth? Is it too much for today's workloads? Fortunately, the adage of "if you build it, they will come" provides answers to these questions.

Data centers have a host of operational costs and upkeep associated with them. Cooling and power costs are the two main areas that data center managers continue to analyze to reduce cost. The reality is as machines become faster and more energy efficient, the cost to power and cool these machines is also reduced. The next question to ask is, how can we push the envelope of efficiency even more?

Converged Networking, also known as Unified Networking, is designed to increase the efficiency of the data center as a whole. In addition to the general power and cooling costs, other areas of focus are the physical amount of servers and their associated cabling that reside in a typical data center. Servers very often have multiple network connections to various network segments, plus they're usually connected to a SAN: either a Fiber Channel fabric or an iSCSI infrastructure. These multiple network and SAN connections mean large amounts of cabling being laid down to attach a server. Converged Networking takes a 10GbE device that is capable of Data Center Bridging in hardware, and consolidates all of those network connections and SAN connections into a single, physical device and cable. The rest of this paper will illustrate the different aspects of Data Center Bridging, which is the networking feature allowing the coexistence of multiple flows on a single physical port. It will first define and describe the different components of DCB. It then will show how DCB consolidates network connections while keeping traffic segregated, and how this can be done in an efficient manner.

## 2 Priority Grouping and Bandwidth Control

### 2.1 Quality of Service

Quality of Service is not a stranger to networking setups today. The QoS layer is composed of three main components: queuing disciplines, or qdiscs (packet schedulers), classifiers (filter engines), and filters [1]. In the Linux kernel, there are many QoS options that can be deployed: One qdisc provides packet-level filtering into different priority-based queues (sch_prio); Another can make bandwidth allocation decisions based on other criteria (sch_htb and sch_cbq). All of these built-in schedulers run in the kernel, as part of the `dev_queue_xmit()` routine in the core networking layer (`qdisc_run()`). While these pieces of the QoS layer can separate traffic flows into different priority queues in the kernel, the priority is isolated to the packet

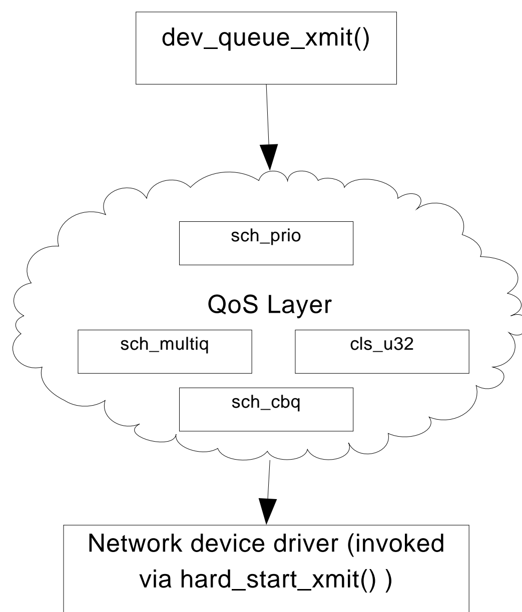



Figure 1: QoS Layer in the Linux kernel

scheduler within the kernel itself. The priorities, along with any bandwidth throttling, are completely isolated to the kernel, and are not propagated to the network. This highlights an issue where these kernel-based priority queues in the qdisc can cause head-of-line-blocking in the network device. For example, if a high priority packet is dequeued from the sch_prio qdisc and sent to the driver, it can still be blocked in the network device by a low priority, bulk data packet that was previously dequeued.

Converged Networking makes use of the QoS layer of the kernel to help identify its network flows. This identification is used by the network driver to decide on which Tx queue to place the outbound packets. Since this model is going to be enforcing a network-wide prioritization of network flows (discussed later), the QoS layer should not enforce any priority when dequeuing packets to the network driver. In other words, Converged Networking will not make use of the sch_prio qdisc. Rather, Converged Networking uses the sch_multiq qdisc, which is a round-robin based queuing discipline. The importance of this is discussed in Section 2.2.

## 2.2 Priority Tagging

Data Center Bridging (DCB) takes the QoS mechanism into hardware. It also defines the network-wide infrastructure for a QoS policy across all switches and end-stations. This allows bandwidth allocations plus prioritization for specific network flows to be honored across all nodes of a network.

The mechanism used to tag packets for prioritization is the 3-bit priority field of the 802.1P/Q tag. This field offers 8 possible priorities into which traffic can be grouped. When a base network driver implements DCB (assuming the device supports DCB in hardware), the driver is expected to insert the VLAN tag, including the priority, before it posts the packet to the transmit DMA engine. One example of a driver that implements this is ixgbe, the Intel® 10GbE PCI Express driver. Both devices supported by this driver, 82598 and 82599, have DCB capabilities in hardware.

The priority tag in the VLAN header is utilized by both the Linux kernel and network infrastructure. In the kernel, vconfig, used to configure VLANs, can modify the priority tag field. Network switches can be configured to modify their switching policies based on the priority tag. In DCB though, it is not required for DCB packets to belong to a traditional VLAN. All that needs to be configured is the priority tag field, and whatever VLAN that was already in the header is preserved. When no VLAN is present, VLAN group 0 is used, meaning the lack of a VLAN. This mechanism allows non-VLAN networks to work with DCB alongside VLAN networks, while maintaining the priority tags for each network flow. The expectation is that the switches being used are DCB-capable, which will guarantee that network scheduling in the switch fabric will be based on the 802.1P tag found in the VLAN header of the packet.

Certain packet types are not tagged though. All of the inter-switch and inter-router frames being passed through the network are not tagged. DCB uses a protocol, LLDP (Link Layer Discovery Protocol), for its DCBx protocol. These frames are not tagged in a DCB network. LLDP and DCBx are discussed in more detail later in this paper.

## 2.3 Bandwidth Groups

Once flows are identified by a priority tag, they are allocated bandwidth on the physical link. DCB uses bandwidth groups to multiplex the prioritized flows. Each bandwidth group is given a percentage of the overall bandwidth on the network device. The bandwidth group can further enforce bandwidth sharing within itself among the priority flows already added to it.

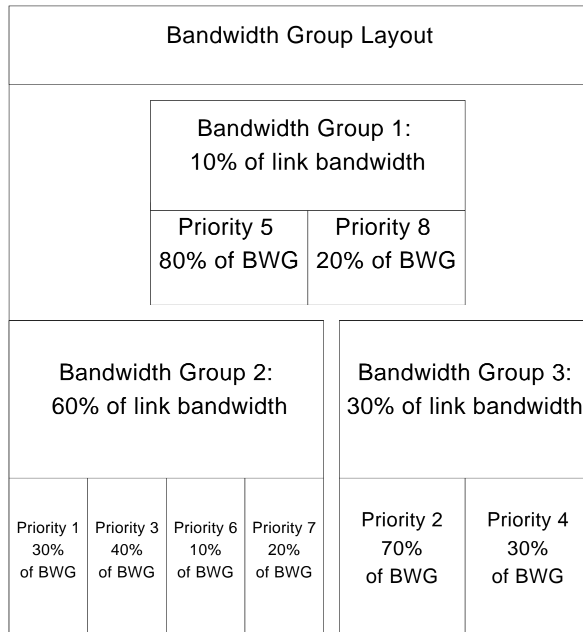



Figure 2: Example Bandwidth Group Layout

Each bandwidth group can be configured to use certain methods of dequeuing packets during a Tx arbitration cycle. The first is group strict priority: It will allow a single priority flow within the bandwidth group to grow its bandwidth consumption to the total of the bandwidth group. This allows a single flow within the group to consume all the bandwidth allocated to the group. This would normally be applied to flows that would run off-hours, and would be in groups that ran on-hours. An example of such a flow is a network backup. The second configuration is link strict priority: This allows any flow from any bandwidth group to grow to the maximum link bandwidth. Obviously this configuration can be dangerous if misconfigured, which could result in the starvation of other flows. However, this mode is necessary to guarantee flows that require maximum bandwidth to get the maximum bandwidth, without needing to reconfigure all bandwidth group layouts [2]. Refer to Figure 2 to see an example Bandwidth Group Layout.

### 2.4 Using TC filters to identify traffic

Now that all the priority flows are distributed into bandwidth groups, traffic flowing down from userspace must be filtered into the underlying queues. There are a few mechanisms that can be used to filter traffic into different queues.

- **`select_queue`** The network stack in recent kernels (2.6.27 and beyond) has a function pointer called `select_queue()`. It is part of the `net_device` struct, and can be overridden by a network driver if desired. A driver would do this if there is a special need to control the Tx queueing specific to an underlying technology. DCB is one of those cases. However, if a network driver hasn't overridden it (which is normal), then a hash is computed by the core network stack. This hash generates a value which is assigned to `skb->queue_mapping`. The skb is then passed to the driver for transmit. The driver then uses this value to select one of its Tx queues to transmit the packet onto the wire.

- **`tc filters`** The userspace tool, tc, can be used to program filters into the qdisc layer. tc is part of the iproute2 package. The filters can match essentially anything in the skb headers from layer 2 and up. The filters use classifiers, such as u32 matching, to match different pieces of the skbs. Once a filter matches, it has an action part of the filter. Most common for qdiscs such as sch_multiq is the skbedit action, which will allow the tc filter to modify the `skb->queue_mapping` in the skb.

DCB needs to make use of both of these mechanisms to properly filter traffic into the priority flows. First, the network driver must override the `select_queue()` function to return queue 0 for all traffic. DCB requires that all unfiltered traffic (i.e. traffic not matching a tc filter) be placed in priority flow 0. The `select_queue()` call is executed prior to the qdisc tc filter section in the core network stack, so if no filter is matched, then the value of `select_queue()` is retained.

tc filters are then added for each network flow that needs to be filtered into a specific priority flow Tx queue.

## 3 Priority Flow Control

In a converged network, various traffic types that normally wouldn't be on an Ethernet-based network are now present. Some of these traffic types are not tolerant of packet loss. Fiber Channel is a good example, and is added to a converged network using Fiber Channel over Ethernet [4]. Fiber Channel is not as tolerant of congestion and packet loss as Internet protocols. Therefore, it must have some form of flow control present to

ensure the frames can be paused, prior to some overrun causing dropped frames.

Using traditional Ethernet flow control is a viable option for these traffic flows. However, the point of Converged Networking is to provide separate, independent network pipes to traffic flows, and not allow one pipe to affect another pipe. Link-based flow control would cause all traffic flows to stop. This is not desired for DCB and Converged Networking.

Priority Flow Control (also known as per-priority pause or PFC) was designed to solve this issue by utilizing utilizes a different packet type from the traditional Ethernet pause. It passes a bitmap of all eight priorities to the link partner, indicating which priorities are currently paused. This way an XOFF/XON pair can be sent for each individual priority flow, while all other flows can continue transmitting and receiving data [3].

# 4 MSI-X and interrupt throttling for latency

## 4.1 Latency requirements

Each priority flow in a DCB network most likely has different latency considerations for the traffic in that flow. For example, high-availability management traffic require very low latency to operate correctly. On the other hand, bulk data transfers, like an FTP transfer, do not require low latency. Other examples of network traffic that have varying latency requirements include Voice Over IP, computer gaming, web surfing, audio streaming, p2p networks, etc. Each of these traffic types treat latency differently, where it either negatively effects the traffic, or doesn't make much difference whatsoever.

## 4.2 Interrupt rates vs. latency

The easiest way to affect the latency of a network device's traffic is to change the interrupt rate of the receive flow interrupt. For example, receive processing running at 8,000 interrupts per second will have a much higher latency than a device running at 100,000 interrupts per second. The trade-off is that the more interrupts a device generates, the higher your CPU utilization will be. Interrupt rates should be tuned to meet the target flow's latency considerations, and will vary based on the contents of that flow.

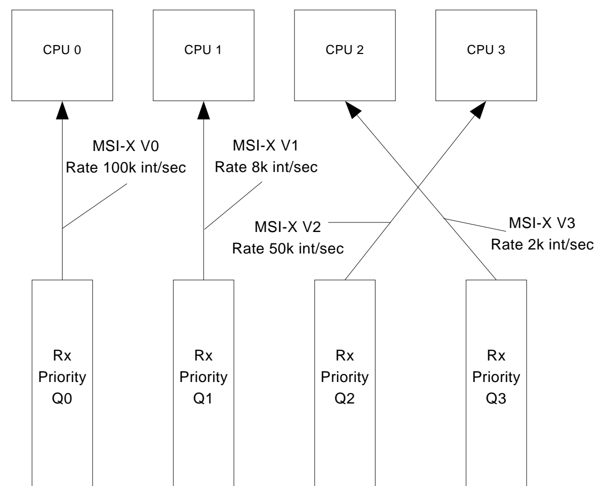



Figure 3: Example MSI-X mapping with variable interrupt rates

## 4.3 MSI-X interrupts

Each traffic flow in DCB may require a unique latency target, therefore requiring a unique interrupt rate. On devices that only support legacy pin interrupts, this cannot be achieved. Rather, the lowest latency must be chosen, and that interrupt rate must be used for the device. This will cause much more CPU overhead than is required for the other flows in your converged network.

MSI-X interrupts (Messaged Signaled Interrupts, Extended) provide the ability to have separate interrupt vectors for each traffic flow. Each vector can be assigned to a receive queue and transmit queue on the network device. Each of those vectors can then be assigned a different interrupt rate, which allows separate traffic latencies for each flow. Refer to Figure 3 to see a sample MSI-X layout with variable interrupt rates.

For another example, the ixgbe's EITR (Extended Interrupt Throttle Rate) registers control the interrupt rates for each interrupt vector. When the device is in MSI-X mode, the device enables an individual EITR register for each MSI-X vector [5]. The driver can then program each EITR separately, accomplishing the need to have fully independent interrupt rates among flows.

# 5 Data Center Bridging Exchange Protocol

DCB has a number of parameters that define how the link operates. The priority group configuration, the

bandwidth allocations, and the priority flow control settings are all part of the overall DCB configuration. Since DCB is a network-wide configuration, there needs to be a mechanism between link partners to negotiate these configuration settings. Data Center Bridging Exchange Protocol, or DCBx, is the protocol that defines how DCB configuration parameters are negotiated on a link. This is a very similar mechanism used to negotiate link parameters, such as auto-negotiated speed, or auto-negotiated flow control settings [6].

### 5.1 LLDP vehicle

DCBx uses the Link Layer Discovery Protocol, or LLDP, to transfer the DCBx configuration frames between link partners. The LLDP frames carry the configuration required to successfully negotiate a DCB link. The protocol also requires that a DCBx negotiation that cannot be resolved (i.e. configuration mismatch) mark the link as failed to negotiate, and disable DCB on the port.

LLDP also carries an application TLV (meaning type, length, and value [6]). This includes information required for applications needing to negotiate parameters with DCBx outside of the stack DCBx parameters. An example is FCoE: FCoE needs to find on which priority it will be resident in DCB. This way, FCoE knows which queues to configure in the base network driver, plus it can make software stack adjustments to properly feed the underlying network driver.

### 5.2 dcbd userspace tools

Linux has a set of userspace tools that implements the DCBx protocol. These tools also push the DCB configuration parameters into the registered network drivers. These tools are part of the dcbd package, which include the dcbd daemon and the dcbtool command line utility. dcbd runs in the background and listens for rtnetlink events on devices that it is managing.

rtnetlink is a Linux kernel interface that allows userspace tools to send messages to kernel components. It is similar to traditional ioctls. Other implementations of rtnetlink interfaces include network interface control (ifconfig commands), VLAN control (vconfig), and Qdisc manipulation (tc).

dcbd learns about network devices when it starts, and when any new device is brought online by listening to link up events from rtnetlink. dcbtool can be used to display the current DCB configuration of a device, manage the configuration of a device, and also toggle DCB mode on and off on a device. The dcbd userspace tools are available on Source Forge at `http://e1000.sf.net`.

## 6 DCB support in the Linux kernel

Data Center Bridging is fully supported in the Linux kernel as of 2.6.29. To date, the only driver making use of DCB features is ixgbe, using the DCB support found in the 82598 and 82599 devices. The main piece of DCB that is resident in the kernel is the configuration API used by dcbd utilities. The layer is called dcbnl, and is an rtnetlink interface. The interface has a full complement of get/set commands to configure each parameter of a DCB device. dcbd uses this interface to pull all DCB-related configuration to feed the DCBx negotiation, and in turn uses the interface to reprogram the device with any new DCB configuration returning from DCBx.

The other portion that DCB makes use of in the kernel is the QoS layer, which was previously discussed. The sch_multiq qdisc is the main workhorse, along with the tc filter act_skbedit action mechanism. These two QoS pieces help filter skbs into their proper priority flows in the underlying base driver.

## 7 Typical DCB deployment model

Now that DCB components have been defined, it's time to take a look at what a DCB deployment model looks like in the data center. The typical deployment today is a composed of two flows, one being storage traffic (FCoE) and the other being all LAN traffic. The LAN traffic can be spread across the other seven priority flows, if the traffic patterns warrant that many prioritized flows in your network. This is a network-dependent setting. Refer to Figure 4 for the order of startup. The steps are numbered.

From here, the FCoE instance will query dcbd through DCBx, using the application TLV, requesting which 802.1P priority it needs to use. Once the priority is provided (default of priority 3), the FCoE tools create tc

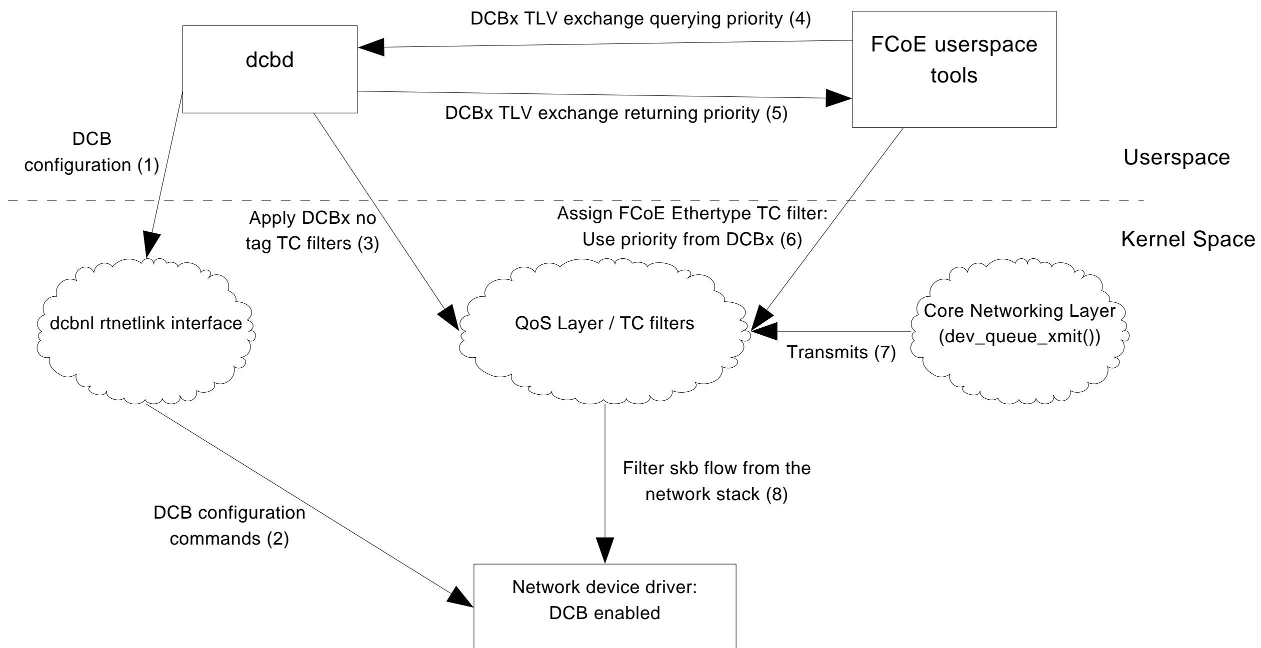



Figure 4: Typical DCB Deployment Model

filters that filter the FCoE ethertype (0x8906) [4]. These filters use the skbedit action to direct the matched flows into flow id 3. The base driver will then use the value of `skb->queue_mapping`, which is set by the skbedit action, to select which transmit queue the base driver allocated for that priority.

On the LAN side, other tc filters can be added by either dcbd or by the system administrator. A typical filter that is added is to match LLDP traffic, and skbedit the `skb->priority` field to be a control frame. That way the base driver can look for that setting, and not set a priority tag on the frame. LLDP frames should not be tagged with priorities within DCB, since they're control traffic.

Once the DCBx negotiation is finished with the switch or link partner, the DCB hardware is ready to use. If all the tc filters are in place, then DCB networking is running.

## 8 Conclusion

Ethernet networks will continue to push the speed envelope. As more 10GbE (and beyond) devices continue to pour into the market, these Ethernet networks will be cheaper to deploy. With data centers pushing the envelope to lower cost of operation with increased computing power and efficiency, Converged Networking will help realize these goals.

## References

[1] Bert Hubert, Thomas Graf, et al. *Linux advanced Routing and Traffic Control* `http://lartc.org`

[2] Manoj Wadekar, Mike Ko, Ravi Shenoy, Mukund Chavan *Priority Groups, Traffic Differentiation over converged link (802.1Qaz)*

[3] Hugh Barrass *Definition for new PAUSE function (802.1Qbb)*

[4] Open-FCoE team *Open-FCoE.org homepage* `http://www.open-fcoe.org`

[5] Intel Corp. *Intel 82598 10GbE Ethernet Controller Datasheet*

[6] Intel Corp., Cisco Systems, Nuova Systems *DCB Capability Exchange Protocol Specification*

# How to (Not) Lose Your Data

Linux as a Reliable Storage Platform


Ric Wheeler
*Red Hat*
rwheeler@redhat.com



## Abstract

Increasingly, Linux is the platform that major vendors use to implement everything from consumer grade NAS devices that you can buy at your local electronics store up to expensive, enterprise grade storage systems. This paper aims to present a high level overview of how some of these systems are put together, how to tune Linux for storage applications and what functionality is either on the horizon or yet to be started in the open source space that will enhance Linux as a storage system. The techniques presented are are also applicable to normal home users who would like to enhance data integrity.


## 1 Why Care about Data Integrity

Taking care of digital data used to be the worry of system administrators. If a computer went down without a backup, few people would ever notice any disruption. Today, the sheer mass of digital data that normal people have makes this a problem for just about anyone with a digital camera or a collection of digital music. Many commercial businesses use Linux-based systems for storing data about their customers like banking records, airline tickets and other critical data. Home users who turn to online sites for storing their photos, music and email on the web also, more than likely, end up using Linux-based systems indirectly.

Linux strives to maintain a unified version of its code, which means that there is just one storage and file system stack that is used by both casual end users and for servers at corporate data centers. The challenge is to provide a system that can leverage high-end storage and its features when possible, without imposing complexity and performance penalties for non-data critical applications. Given the deeply personal nature of the types of data that people store today on Linux, like digital photographs, it is really critical to give users a framework for how to store data reliably.

When designing a reliable storage system, enterprises usually invest in both reliable local storage and a way to replicate their data to storage at a remote site which must also be reliable. This paper aims to provide Linux developers a framework for thinking about how to provide reliable components for people building Linux based storage systems and weigh the trade offs appropriately. The conclusion presents a brief overview of key research in storage systems and a summary of upcoming features in the Linux storage and file system stack that will enhance both enterprise and end user's data integrity.

## 2 Common Causes of Data Loss in Systems

Anyone who deals with storage of digital data, especially long-term storage, understands that even the best storage systems can and will suffer data loss occasionally. This section gives a very high level overview of the most common causes of data loss.

### 2.1 User Errors

The most common errors are typically user errors; for example, accidentally deleting a file, forgetting where you put a specific file, upgrading your system or reformatting a whole disk. Rather than write off this class of data loss as "stupid human errors," the challenge is to design systems that are easy to use, have safe default settings and do not require expensive infrastructure like UPS backup for the servers' power needs.

At the system level, there are a few basic techniques that Linux provides which help mitigate against these types of user errors. A common practice in enterprise data centers is to create periodic snapshots of a file system, say once a day. Snapshots do not protect against storage failure at the block level, but they do give you a point in time picture of the state of your file system that can be

referenced if you accidentally delete a file or do some other regrettable action to your data. In addition, users can create a snapshot of a live file system and use that snapshot as the basis for a local backup or to kick off a consistent remote copy.

## 2.2 Confusing Semantics for Key System Calls

Application writers need to have crisp and clear semantics for basic operations like fsync() and rename() system calls and clear documentation and guidance about how to use them to provide their users reliable data storage. How does an application know with absolute certainty that data that it writes or a new file that it has renamed will survive a power outage or system reboot? To make the challenge more interesting, different applications need different levels of granularity.

At one end of the spectrum, a database typically wants to have this type of clear promise after each commit of a transaction. At the other end of the extreme, it would usually be sufficient for an application like rsync to provide this promise that all of the data is safely stored on the remote system at the end of its execution which would allow the system to flush caches and so on only once for the entire set of files. In the rsync case, the user would be able to simply redo the rsync if the something fails during the initial run without being exposed to any data loss.

Somewhere in the middle of this spectrum are common tools like editors which want to provide atomic updates to files being processed. The rename() system call has long been used to provide this level of atomicity for updating files. For example, an editor that wants to overwrite all or part of file "foo" can do this safely by first creating a temporary copy of the contents of "foo" to a separate file, say "foo.temp." All changes are written to the temporary copy until the editor is ready to persist its changes to disk. At this point, it calls `rename("foo.temp", "foo")` with the expectation that, even in the face of a system crash or power outage, the rename will be atomic. Specifically, after a crash, the user will see either the new contents in file "foo" or the old contents, not some random mix of old and new data or an empty file.

To make this sequence really robust in a generic way, the application should issue one fsync() system call for the new "foo.temp" file and potentially a second fsync() system call on the directory that "foo" lives in to insure that the changes in the name space survive. File systems could automatically insert the appropriate fsync() calls internally, but this could degrade performance for applications that are less concerned about data integrity. How to get the balance right between data integrity and performance is an active debate in the file system developer community.

Clearly, using "fsync()" and "rename()" on every individual file while doing batch updates like the rsync example mentioned above, or when using tar to extract a large number of files, will have a large impact on performance. For some popular file systems like ext3, an effective way to avoid the performance impact of the fsync() per file technique is to have applications break up the extraction into a writing phase in which each file is written to disk without any special promises, and then a second fsync() phase in which the application iterates back over all of the files written and fsync()'s each one in turn in the reverse order that the files were written originally. This technique mitigates the heavy fsync() performance impact since the first fsync() in that second phase will push out the data for all of the preceding files that have just been written.

Different types of storage will show vastly different results since the performance is directly tied to how expensive seek operations are and whether or not the device has a volatile write cache. Testing the various methods on a common 1TB S-ATA disk can give the reader a sense of the impact using common hardware today. Using Fedora10 on a quad core desktop system and ext4, the best rate for writing 40KB files without doing any fsync() calls is around 2,600 files/second. Note that this test basically measures how quickly the file system can write to the page cache and is highly variable.

With the barrier support properly enabled on ext4, the slowest, most cautious method writes only 25 files/second by doing an open, write and and fsync on each file in turn. This rate is roughly half the rate that the drive's seek latency dictates, which corresponds well to the two cache flush operations per fsync that the fsync calls produce when running with barriers.

Finally, using the two phase technique, first writing all of the files in a batch and then iterating back over the batch of files in the reverse order to fsync them one at a time, the rate returns back up to around 143 files/second. If this is not convoluted enough, issuing a sync() system

call before doing the fsync() phase will bring the rate up to around 900 files/second. Looking at the twists and turns required to get reasonable performance and data integrity clearly shows that we need to provide something better if we would like to get application programmers to improve their code.

Clearly, there is a lot of room for reducing the complexity, and giving application writers more intuitive and powerful tools. Over the past few years, file systems developers have been debating several possibilities ranging from some complex mechanisms like exposing transactional semantics to user space applications or providing a robust asynchronous fsync primitive. Like other async calls, the application would use the async fsync interface on each file in a fairly straight forward way and then have a second interface to use when it needs to wait for completion. The advantages of this async approach would be that the file system could optimize the fsync calls internally over a larger set of files.

### 2.3 IO Stack Bugs and Configuration Errors

System software, like the file system or the IO stack, can also be a common cause of data loss when it fails to persist data correctly before a power outage or system crash. Modern file systems and data bases often use journaled transactions as a way to provide robust storage. Transaction based systems need to be able to have a few promises from storage in order to make their transactions robust including the ability to store some information, like a transaction commit block, in a reliable way. Storage devices, including disk drives, have large, volatile write caches which is typically tens of megabytes in size. On power loss, the data stored in that cache will be lost.

To provide robust support for transactions, Linux has supported a fairly brute force mechanism called "write barriers" which effectively give the file system the ability to flush the target device write cache before sending a write with the commit block. A second flush is then initiated in order to make sure that the commit block itself is safe on persistent storage. This technique has a clear performance impact for applications that cause lots of transactions. Most file systems have mount options which enable the barriers correctly, but work is ongoing to make sure that all of the various bits of the block layer like device mapper and the more advanced RAID levels supported by MD will correctly handle barriers operations.

If the system has one of the configurations that do not support barriers properly and it has storage devices with volatile write cache devices, the only safe option is to disable the write cache on the storage devices which can be done with the hdparm command.

Note that external storage arrays typically have large, non-volatile write caches which do not require these barrier operations.Some of these arrays will silently ignore the cache flush commands issued by the Linux barrier operations, but others will honor them by flushing their potentially very large caches which is a gigantic performance hit. To prevent this overhead, file systems mounted on this class of device should be mounted with the barriers disabled.

The preceding set of considerations makes doing the right thing extremely confusing. If a system is using device mapper, the barrier operations will log an error and be disabled which leaves users exposed to potential data loss on power loss. The same story happens with RAID5 or RAID6 and MD devices. Several things need to be fixed in order to reduce the confusion. One very promising set of patches, recently posted by Martin Petersen, exports several characteristics of devices through /sys interfaces. Unfortunately, the nature of the write cache is not currently one of these characteristics, but this is a positive first step. Also, work is ongoing in the device mapper community to properly handle barrier operations. For MD users who use anything but the basic MD1 RAID, the only safe option is to disable the write cache on the individual component devices currently.

### 2.4 Hardware Failures

Hardware failures, specifically disk failures, are what most users would associate with data loss. Single disk drives are relatively reliable components, but can suffer from both hard failures when all data is lost or partial failures where only a portion of data is lost. Other types of hardware failure, like bad memory components, can cause data loss as well. RAID schemes, discussed briefly below, reduce the exposure to data loss by storing the data on multiple components which are assumed to have independent failures. New types of devices, like the increasingly popular SSDs, are largely immune from

some of the causes of failure of traditional drives, but bring their own unique ways of failing that system designers and users will learn more about as the devices increase in number and age.

## 3 Data Loss Timeline

One useful way to think about keeping data safe is as a timeline. Assuming the application has figured out how to properly navigate the confusing maze detailed previously and has correctly stored the data on a storage device, a clock starts counting down for each hardware component in your system. Time runs out when the component actually sustains an error or fails completely. Designing a reliable storage device requires understanding the expected failure rates of the components used to make a system and being able to balance the cost of those components against other considerations like cost, performance and power consumption.

The high level overview of this timeline is:

**Data Creation** The application performs a write of data: for example, the "cp" application is used to create a copy of a file but has not called fsync(). The data is not protected against a power outage or system failure at this point in time.

**Persistently Stored** The data is stored and acknowledged by the storage subsystem: the data is moved from the page cache out to the storage system and the transaction is acknowledged back to the server. At this point, all is right with the data and the storage system has all of the redundant copies it needs to overcome a partial failure.

**Component Failure** A component of the storage system fails partially or completely: failures could be partial failures like a single bad sector on a drive, total failure of a drive or possibly a software or user error that corrupts a file. This error alone might not cause data loss or data unavailability to a user if the data is protected in a RAID group, but it does expose the user to permanent data loss if not repaired before a second failure in the same data stripe. The key consideration in building a robust storage system is to minimize the amount of time spent in this state.

**Failure Detection** The failure is detected by the system: an application tries to read a file back or the RAID software detects a partial or total failure. In RAID arrays, these errors are often detected by the firmware which will continually scan the surface of the individuals drives, searching for partial errors. The critical trade off here is that over aggressive scanning, while reducing the window of time that the system is exposed to a potential data loss, has a negative impact on the performance of the system's normal workload, can prematurely age the components and can consume more power since the devices are kept from entering an idle state.

**Data Repair Initiated** Examples include a new drive is inserted into the RAID group, a file system repair is initiated or a file is restored from tape. Note that there is a potential lag between the detection of the partial error and being able to initiate a repair. In the worst case, if you are repairing a RAID group with one completely failed drive and no spare, this repair phase is blocked until a new drive is physically inserted into the array to replace the failed component.

**Data repair completed** The original file is back and usable by the user. Just like the fourth stage above, there is a trade off here between completing the repair in an aggressive way by consuming the full bandwidth of the device and impacting the foreground workload.

Note that a related class of problem is data unavailability which can be caused by something as common as a power outage or by a long running, offline repair like an invocation of fsck. For time critical data, this can be as critical as permanent data loss.

The next section gives some details about common components used to build storage and gives some measure of how they rate in the time line sketched out here.

## 4 Reliability Building Blocks

A general principle of design for reliable systems is to build systems that tolerate a given number of failures. If you have a system with one drive, your data stands to disappear whenever that single drive fails. If you have two disks in a RAID1 mirror, your system can tolerate the failure of one disk but would still suffer failure if your CPU or DRAM fail. For this reason, enterprise class arrays have redundancy for all critical components: no single failure of a power supply, CPU, DRAM or disk would cause data loss but might cause degraded performance.

In a similar way, a single location like your home office or a data center is a single failure component which could be destroyed by a fire or other disaster. In order to reduce data loss for these catastrophic events, businesses commonly use long distance replication to store data on a remote site.

This section presents a summary of common features in storage and reviews the status of these features in Linux today.

### 4.1 RAID Level Tradeoffs

RAID is the most common form of data protection used today. RAID is normally done at the level of a block device, for example, a file system will send a write down to the block level which will do the appropriate RAID computations transparently. A simple, robust and inefficient RAID level is RAID1—all data is written to each member of the RAID group. For example, a system with four storage devices will write to each of the four devices on every IO. This gives the storage system great fault tolerance since the system could have as many as three of the four drives fail without incurring data loss. The down side of this scheme is that it is horribly inefficient with only 25% of the total capacity of the storage components available for storing user data. This ratio will be referred to as **effective capacity**.

Other RAID levels, with the same number of drives, improve the effective capacity. For example, RAID5 will break each IO into fragments, three data fragments and a fragment which contains parity information. Any single drive can fail and the other disks can be used to regenerate the data from the failed component. In this 4 drive system, a RAID5 scheme provides the user an effective capacity of 75%. In a similar way, RAID6 computes two different parity computations and will be able to survive any dual failure of storage devices, but decreases the effective capacity to 50% in our four drive example above.

Commercial RAID arrays offer a wide range of configurations. Low end systems aimed at consumers start with as few as 2 drives configured into a RAID1 device. Higher end consumer devices move up to a 4 drive RAID5 configuration. Enterprise class RAID arrays provide shelves full of disks. A typical mid-range storage system would have 12 to 15 drives per storage shelf with high-end systems ranging up to a couple thousand drives per array. Clearly, these larger systems present more than one RAID set out to hosts.

One type of failure that can foil any RAID system is an undetected partial failure. The above examples used the common assumption that a storage device would either work correctly or fail completely. While complete failures are not uncommon, it is also relatively common to have storage get corruption that impacts only a few sectors of storage. For example, rotational storage might have localized loss of data due to contamination like dust or lubricant on the platter while SSD devices might have localized data loss due to overuse. Regardless of the cause, the problem is the same—these partial failures can lie undetected for a very long time. In the worst case, they are detected only when a second total failure happens to a different storage device in the same RAID group. As the system tries to rebuild the RAID group, it needs to read data from all of the other components and will invariably detect all latent errors. Each of these latent errors will cause the RAID rebuild to fail for one stripe. In this case, the basic assumption about having independent failures does not protect the user since we notice the latent errors concurrently with the total failure of the other device.

The way to reduce the likelihood of failure during critical times like a RAID rebuild is to do periodic scans of the individual storage devices. For example, once every two weeks, the system will do a full surface scan of each storage device in a RAID group. If you detect an error during the scan, you can attempt to repair the data immediately by recomputing the data from the other devices in the RAID group and attempting to overwrite the failed sector. In many cases, this write will work by either correcting the data in place or by remapping the failed sectors to a pool of extra sectors kept for failures. If the data cannot be recovered, it is time to replace the failed device. In current Linux MD RAID, we have the capability to do this period scan for example.

There are some techniques used by high-end storage systems to make their RAID systems more robust. A very common technique is to have a spare device that is not an active participant in any RAID group. When a drive fails fully, the spare can be used to immediately start rebuild the contents of the missing storage component which decreases the window of time that the RAID group needs to be exposed to a possible second failure. A second trick is to suck as much data as possible from the failed component if it is still partially readable, since it allows the RAID rebuild only the data for stripes that cannot be read.

If the system needs to tolerate more than two component failures, there is a generic set of techniques called erasure encodings that can tolerate $k$ failed components out of the $n$ devices in your system. An example for the mid-level arrays might be an encoding that would survive any 4 failures in a 15-device system.

## 4.2 Remote Replication

Remote replication is another important tool for data protection and provides a remote copy of data that would survive any catastrophic event like a fire or a flood that would destroy any local storage. This section details several varieties of remote replication.

Block level replication can be built using something as simple as a RAID1 device, where one of the components is a remote device like an iSCSI target. Each write will be sent synchronously to the remote site which, depending on distance, can introduce substantial performance hurdles. A more sophisticated scheme could use LVM snapshots to avoid this performance penalty: snapshot a volume and then do the replication to the far site of the snapshot copy while local file system IO is left unhindered. Block level replication is also a feature that is frequently implemented inside of storage arrays that can use either dedicated storage links to the remote sites or direct the replication over normal connections. Block level replication is fairly common in high end data centers but can be a bit challenging to use in an intuitive way.

A more pedestrian way to replicate data is by replication at the file system layer. For anyone who is familiar with rsync, the technique is fairly intuitive: iterate over the entire file system and send the files that have changed to a remote server which will store it on disk. From the point of view of the source file system, the operation should be a fairly straight forward sequence of calls to getdents() in order to build a list of files followed by the application reading and then transmitting the file over the network to the target system. Unfortunately, there are several complications that get in the way of doing this in a straight forward manner.

Some file systems, specifically ext3, can return the file names via getdents() calls in an arbitrary order which, in turn, causes a lot of seeking as the application reads a series of small files in non-sequential order with regards to the disk layout. To improve this performance, applications can sort the list of file names by the inode order. Using a similar test, putting 1 million 40KB files in one directory results give a rate of 55 files/second when read in getdents() order and a rate of 1,381 files/second when read in sorted by inode number. Given that applications can write new files at a rate of 1277 files/second, the sorted remote replication is the only way to keep pace with ingest. Other file systems, like XFS, do a good job of returning the file names in a reasonable order. For these file systems there is no benefit from this technique but the cost of using it is not high. All of this complexity just continues to make the life of application programmers miserable. Note that doing full file system level iteration at full speed for any file system depends on minimizing head movement for traditional disk drives, so any other file system activity can have a severe impact on the performance of the replication.

## 4.3 Data Migration

Data migration is a special form of remote replication in which the intention is to decommission the source storage system once the data is successfully replicated at the target system. Data migration might involve a local migration from a single disk drive which has started to fail to a new drive, or be done from one high end storage array to a second one over a long distance link. It is a fairly common operation both for consumers who routinely replace or upgrade their personal systems and for data centers where high end storage is often rotated out of service after a fixed period, say every three years.

Key points of this class of replication include making absolutely certain that the remote system has a full and persistently stored copy of the data since the source will be taken offline. In the earlier rsync example, it is critical to make sure that the data is not just stored in the remote page cache. One other key consideration is that the source system is typically not new and can be in fairly rough shape, so the iteration can encounter more IO errors than a normal system would encounter. To migrate from an unhealthy source, the IO stack needs to be tuned properly to handle IO errors in a quick and deterministic way and avoid excessive retries. Applications doing the migration need to be able to be equally robust in face of errors: log any failures and keep moving good data to the new system.

Cloud storage can be thought of as another variation on remote replication at the file system layer. The difference is that the target system is normally not a typical

file system or block device that users can access directly. Rather, for each file a user stores in the cloud, that user gets back a object reference that can be used to retrieve the file when needed.

## 5 Research in Reliable Storage Systems

File and storage system research has become one of the more active areas of academic research. One of the highlights of the year for researchers engaged in this area is the USENIX Association's annual FAST conference, where open source, industrial, and academic researchers meet to present key results. The USENIX Association has all of its papers online and freely available, along with recordings and videos of recent presentations. This section presents a very brief summary of key results presented recently.

Some key areas for storage are the analysis of what really fails and how frequent those failures are. For many years, storage companies have collected that data for their own deployed products and guarded that information as an important part of their intellectual property which made it extremely difficult to build either research or real systems based on facts. Two groundbreaking works were presented in FAST in 2007. The first paper was presented by Bianca Schroeder and Garth Gibson from Carnegie Mellon's Parallel Data Laboratory and presented the first large-scale analysis of real-world disk failures [6]. The second work at FAST that year was from Eduardo Pinheiro and his coauthors from Google [2] who shared similar data collected from the huge number of systems in Google's data centers. These works were followed in proceeding years by significant contributions by NetApp [1] and others.

For those interested in coding open source RAID6, or more robust erasure encoded systems, James Plank from the University of Tennessee and his coauthors presented an overview of open source friendly RAID6 [3] and general erasure encoding [4] algorithms. He took careful note of which algorithms he believed to be free of the known patents.

File systems also have received a fair amount of attention from the academic world, with notable contributions to Linux file systems reliability coming from the University of Wisconsin's group, which did an analysis of failures in commodity file systems and produced the prototype code used in ext4's journal checksumming [5]. Several other universities have active Linux based research projects, including Erez Zadok's group and their work on stacking file systems and the scalable file system work being done at the University of California, Santa Cruz.

## 6 Conclusion

The scale of storage systems is increasingly dramatically, both for home consumers and certainly for high end data centers. Current capacity for a single S-ATA drive is 2TB.In the examples used previously in this paper, this single drive will hold over 50 million 40KB files. Migration from a 2TB drive at 25 files/second would take close to 600 hours as compared to just under 12 hours running at the sorted rate of 1,300 files/second. The author has recently been testing Linux file systems on a relatively "small" 80TB LUN exported from EMC's newest Symmetrix, which can hold over two thousand drives. Clearly, scale makes using and improving the techniques discussed above an important challenge.

## 7 References

## References

[1] Weihang Jiang, Chongfeng Hu, and Yuanyuan Zhou. Are Disks the Dominant Contributor for Storage Failures? A Comprehensive Study of Storage Subsystem Failure Characteristics. In *FAST-2008: 6th Usenix Conference on File and Storage Technologies*, February 2008.

[2] E. Pinheiro, W. D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *FAST-2007: 5th Usenix Conference on File and Storage Technologies*, February 2007.

[3] J. S. Plank. The RAID-6 Liberation Codes. In *FAST-2008: 6th Usenix Conference on File and Storage Technologies*, February 2008.

[4] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. Wilcox-O'Hearn. A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries For Storage. In *FAST-2009: 7th Usenix Conference on File and Storage Technologies*, February 2009.

[5] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220, Brighton, United Kingdom, October 2005.

[6] Bianca Schroeder and Garth Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean too you? In *FAST-2007: 5th Usenix Conference on File and Storage Technologies*, February 2007.

# Testing and verification of cluster filesystems

Steven Whitehouse
*Red Hat, UK*
`swhiteho@redhat.com`

## Abstract

Although software testing can never prove a program is correct, it can catch many errors early and plays an important part in the process of declaring a program "stable" and ready for release. Cluster filesystems (e.g., GFS2), by their very nature require hardware-intensive test environments, and are thus also expensive to test. This tends to limit test coverage compared with their simpler, single-node counterparts (ext2/3/4, xfs, etc).

Since reliability is a key feature of any filesystem, this paper considers a number of techniques which may be used to help simulate a cluster on a single node, reducing the cost of testing and increasing the coverage in the process. Although GFS2 is taken as the example filesystem, the techniques described are generic and apply to any similar filesystem.

## 1 Introduction

There is a catch-22 situation in filesystem development where users are unsure of trusting their data to a new filesystem until its been in use by other people for a period of time. Software verification and testing is one way to try and break this cycle and to help build confidence.

Testing cluster filesystems is particularly problematic because they, by definition, require clusters to test on, and since these are expensive to run it is often difficult to get enough testing time, particularly on larger configurations.

In addition, it is often those running larger clusters who are the most sensitive to downtime, and thus reliability is often at the top of the list of requirements.

When supporting filesystems in the field, things will occasionally go wrong, and in that case it is imperative to identify the cause of the failure as quickly as possible, and thus to be able to rectify it. Filesystems therefore should be designed with this consideration in mind.

This paper considers the possible methods by which a cluster filesystem (taking GFS2 as the example) can be tested and debugged. We consider some of the more recent developments based around the Linux tracing infrastructure. We include performance testing as well as testing in order to verify functional correctness.

### 1.1 Proof

In an ideal world, it would be possible to prove the correctness of software. In reality, it is impossible to prove the correctness of anything but the simplest software; the main difficulty is that the combination of all the possible inputs to the system and all the possible states is so large that it is impossible to prove the system correct within a reasonable amount of time. This prevents exhaustive testing, but even with techniques to try and cut down on the total number of states which need to be tested, it is still usually too difficult even for fairly simple systems.

A further complication arises from the possibility that there is an error in the original specification, and in that case a system maybe provably correct, but still fail due to some unforeseen circumstance.

### 1.2 Source analysis

By carefully annotating the source, some errors can be removed at compile time. The `sparse` tool can detect a number of issues relating to endian conversion, arithmetic with invalid types (e.g., bitwise) and similar issues. It isn't able to detect a huge range of errors, but it does catch a number of basic issues, given a disciplined to annotation of the source code.

`gcc` can also assist in this, of course, as more tests are built in as can careful use of `const` etc. in the source files.

### 1.3 Not quite exhaustive testing

In this category are methods like lockdep. The idea is to run a "normal" workload on the filesystem under test and then to monitor the operations performed for invalid sequences. The advantage of this method is that it will catch potential deadlocks even when the code paths didn't happen to run (in the test) at the same time, but where it is possible that they might cause a deadlock in the future.

Occasionally this will cause false positives, but those can be removed by suitable annotation or reordering of operations. One example of that is the configfs `mkdir()` issue where the locking is correct, but done in a different way to that expected by lockdep.

### 1.4 Feature & regression testing

As features are added to GFS2 [1, 2], new tests are added or adapted from other filesystems. In addition, Red Hat's QE department perform regular regression tests over each release of Red Hat Enterprise Linux before release.

Although this prevents previous issues from reappearing and helps to ensure that the basic functionality is working, the total state space is so much larger that it will never be able to catch all the issues.

### 1.5 Early exposure to users

One of the core philosophies of Open Source is that code should be released early and often. This can be a very successful strategy for projects which have a large audience. However, people who have clusters don't usually have them sitting idle just waiting for a new release of software to test on them.

As a result, the benefit of this form of testing is less than with the more common features, such as single node filesystems. There is still a substantial benefit to be gained from people building GFS2 in a variety of different configurations, but most of the bugs reported have related to build issues.

### 1.6 Performance testing

Once all the functionality has been tested, the next aim is to be able to narrow down any performance issues, at least to the section of code where they occur. At that point more targeted methods can then be used to identify bottlenecks.

One example of performance testing is Askant [3] (in the contrib section of the gfs2-utils git tree), which uses a combination of static analysis to annotate blktrace output with the details of the on-disk structures being written.

### 1.7 Testing issues

The main issue raised above is that it is tricky to test a cluster file system without a cluster, which makes testing long-winded and expensive. As a result of that, we plan to try and simulate the effect of being in a cluster as closely as possible, but only using a single node.

## 2 Glocks

Since glocks are the core of GFS2, they are also the initial target of the verification and testing effort. Experience has shown that many different issues in the cluster can result in a deadlock which is first encountered at the glock level.

A glock is a caching mechanism, both for locks and also for the data and metadata associated with them. Each glock can have a number of "holders" associated with it, each of which represents one lock request from the higher layers. System calls relating to GFS2 queue and dequeue holders from the glock to protect the critical section of code.

Each glock corresponds exactly to one DLM lock. The glock state machine provides a local locking mechanism which is designed to reduce the number of remote locking calls made by caching the locks in a particular state until a remote node requires the lock, or until the VM reclaims the glock from the glock LRU list via the shrinker.

The glock state machine is based on a work queue. For performance reasons, we would prefer to use tasklets, however in the current implementation we need to submit I/O from that context which prohibits their use. One of the objectives of the performance tests in this area is to try and work out just how efficient this code is, and whether any improvements can reasonably be made.

The glock debugfs interface allows the visualisation of the internal state of the glocks, the holders and it also includes some summary details of the objects being locked in some cases. Each line of the file either begins `G:` with no indentation (which refers to the glock itself) or it begins with a different letter, indented with a single space, and refers to the structures (`H:` is a holder, `I:` an inode, and `R:` a resource group) associated with the glock immediately above it in the file.

An example is shown in figure 1 which is a series of excerpts (from an approx 18M file) generated by `cat /sys/kernel/debug/gfs2/unity:myfs/glocks >my.lock` during a run of the `postmark` benchmark on a single node GFS2 filesystem. The glocks in the figure have been selected in order to show some of the more interesting features of the glock dumps.

The glock states are either `EX` (exclusive), `DF` (deferred), `SH` (shared) or `UN` (unlocked). These states correspond directly with DLM lock modes except for `UN` which may represent either the DLM null lock state, or that GFS2 doesn't hold a DLM lock. The `s:` field of the glock indicates the current state of the lock and the same field in the holder indicates the requested mode. If the lock is granted, the holder will have the `H` bit set in its flags (`f:` field) otherwise it will have the `W` wait bit set.

The full listing of all the flags for both the holder and the glock are set out in the two tables 3 and 2.

In the current upstream GFS2, once a DLM lock has been taken out it is only ever demoted to the null state (and never unlocked) unless the glock has reached the end of its life and is being freed. This means that holding a reference on a glock that has been promoted to any mode other than `NL` will also result in a reference on the associated DLM lock and thus preserve the content of the lock value block (LVB).

The content of lock value blocks is not currently available via the glock debugfs interface, although we may well add this in the future.

The `n:` field (number) indicates the number associated with each item. For glocks that is the type number followed by the glock number so that, as seen in Figure 1, the first glock is `n:5/75320`—i.e., an iopen glock which relates to inode 75320. In the case of inode and iopen glocks, the glock number is always identical to the inode's disk block number.

| Type Number | Lock type | Use |
|---|---|---|
| 1 | Trans | Transaction lock |
| 2 | Inode | Inode metadata & data |
| 3 | Rgrp | Resource group metadata |
| 4 | Meta | The superblock |
| 5 | Iopen | Inode last closer detection |
| 6 | Flock | `flock(2)` syscall |
| 8 | Quota | Quota operations |
| 9 | Journal | Journal mutex |

Table 1: Glock types

One of the more important glock flags, is the `l` (locked) flag. This is the bit lock which is used to arbitrate access to the glock state when a state change is to be performed. It is set when the state machine is about to send a remote lock request via the DLM, and only cleared when the complete operation has been performed. Sometimes this can mean that more than one lock request will have been sent, with various invalidations occurring between times.

When a remote callback is received from a node which wants to get a lock in a mode which conflicts with that being held on the local node, then one or other of the two flags `D` (demote) or `d` (demote pending) is set. In order to prevent starvation conditions when there is contention on a particular lock, each lock is assigned a minimum hold time. A node which has not yet had the lock for the minimum hold time is allowed to retain that lock until the time interval has expired.

If the time interval has expired, then the `D` (demote) flag will be set and the state required will be recorded. In that case the next time there are no granted locks on the holders queue, the lock will be demoted.

If the time interval has not expired, then the `d` (demote pending) flag is set instead. This also schedules the state machine to clear `d` (demote pending) and set `D` (demote) when the minimum hold time has expired.

The `I` (initial) flag is set when the glock has been assigned a DLM lock. This happens when the glock is first used and the `I` flag will then remain set until the glock is finally freed (which the DLM lock is unlocked).

The most important holder flags are `H` (holder) and `W` (wait), as mentioned earlier, since they are set on granted

```
G:  s:SH n:5/75320 f:I t:SH d:EX/0 a:0 r:3
 H: s:SH f:EH e:0 p:4466 [postmark] gfs2_inode_lookup+0x14e/0x260 [gfs2]
G:  s:EX n:3/258028 f:yI t:EX d:EX/0 a:3 r:4
 H: s:EX f:tH e:0 p:4466 [postmark] gfs2_inplace_reserve_i+0x177/0x780 [gfs2]
 R: n:258028 f:05 b:22256/22256 i:16800
G:  s:EX n:2/219916 f:yfI t:EX d:EX/0 a:0 r:3
 I: n:75661/219916 t:8 f:0x10 d:0x00000000 s:7522/7522
G:  s:SH n:5/127205 f:I t:SH d:EX/0 a:0 r:3
 H: s:SH f:EH e:0 p:4466 [postmark] gfs2_inode_lookup+0x14e/0x260 [gfs2]
G:  s:EX n:2/50382 f:yfI t:EX d:EX/0 a:0 r:2
G:  s:SH n:5/302519 f:I t:SH d:EX/0 a:0 r:3
 H: s:SH f:EH e:0 p:4466 [postmark] gfs2_inode_lookup+0x14e/0x260 [gfs2]
G:  s:SH n:5/313874 f:I t:SH d:EX/0 a:0 r:3
 H: s:SH f:EH e:0 p:4466 [postmark] gfs2_inode_lookup+0x14e/0x260 [gfs2]
G:  s:SH n:5/271916 f:I t:SH d:EX/0 a:0 r:3
 H: s:SH f:EH e:0 p:4466 [postmark] gfs2_inode_lookup+0x14e/0x260 [gfs2]
G:  s:SH n:5/312732 f:I t:SH d:EX/0 a:0 r:3
 H: s:SH f:EH e:0 p:4466 [postmark] gfs2_inode_lookup+0x14e/0x260 [gfs2]
```

Figure 1: Example glock dump from debugsfs (edited for size)

Table 2: Glock flags

| Flag | Name | Meaning |
|---|---|---|
| l | Locked | The glock is in the process of changing state |
| D | Demote | A demote request (local or remote) |
| d | Pending demote | A deferred (remote) demote request |
| p | Demote in progress | The glock is in the process of responding to a demote request |
| y | Dirty | Data needs flushing to disk before releasing this glock |
| f | Log flush | The log needs to be committed before releasing this glock |
| i | Invalidate in progress | In the process of invalidating pages under this glock |
| r | Reply pending | Reply received from remote node is awaiting processing |
| I | Initial | Set when DLM lock is associated with this glock |
| F | Frozen | Replies from remote nodes ignored - recovery is in progress |

Table 3: Glock holder flags

| Flag | Name | Meaning |
|---|---|---|
| t | Try | A “try” lock |
| T | Try 1CB | A “try” lock which sends a callback |
| e | No expire | Ignore subsequent lock cancel requests |
| A | Any | Any compatible lock mode is acceptable |
| p | Priority | Enqueue holder at the head of the queue |
| a | Async | Don’t wait for glock result (will poll for result later) |
| E | Exact | Must have exact lock mode |
| c | No cache | When unlocked, demote DLM lock immediately |
| H | Holder | Indicates that requested lock is granted |
| W | Wait | Set while waiting for request to complete |
| F | First | Set when holder is the first to be granted for this lock |

lock requests and queued lock requests respectively. The ordering of the holders in the list is important; if there are any granted holders, they will always be at the head of the queue, followed by any queued holders.

If there are no granted holders, then the first holder in the list will be the one which triggers the next state change. Since demote requests are always considered higher priority than requests from the filesystem, that might not always directly result in a change to the state requested.

The glock subsystem supports two kinds of "try" locks. These are useful both because they allow the taking of locks out of the normal order (with suitable back-off and retry) and because they can be used to help avoid resources in use by other nodes. The normal `t` (try) lock is basically just that; the so called `T` (try 1CB) is identical, except that the DLM will send a single callback to current incompatible lock holders.

One uses of the `T` (try 1CB) is with the iopen locks which are used to arbitrate among the nodes when an inode's `n_link` count is zero, as to which of the nodes will be responsible for deallocating the inode. The iopen glock is normally held in the shared state, but when the `n_link` count becomes zero and `->delete_inode()` is called, it will request an exclusive lock with `T` (try 1CB) set. It will continue to deallocate the inode if the lock is granted. If the lock is not granted it will result in the node(s) which was/were preventing the grant of the lock marking their glock(s) with the `D` (demote) flag which is checked at `->drop_inode()` time in order to ensure that the deallocation is not forgotten.

This means that inodes which have zero link count, but are still open, will be deallocated by the node on which the final `close()` occurs. Also, at the same time as the inode's link count is decremented to zero, the inode is marked as being in the special state of having zero link count, but still in use in the resource group bitmap. This functions like ext3's orphan list in that it allows any subsequent reader of the bitmap to know that there is potentially space which might be reclaimed, and to have a go at reclaiming it.

### 2.1 Callback injection

In recent kernels, there is a sysfs interface which allows the injection of demote requests from user space. Since GFS2 has no communication with other nodes, except for the callbacks issued in response to DLM lock requests, this provides a way to simulate, on a single node the same callbacks as if it was part of a cluster.

In combination with the GFS2 trace points and blktrace, it becomes possible to check that the correct locks are being held when the block requests I/O are issued and completed. Also, the overhead of doing this is small enough that it can be left running during normal system operation without slowing it significantly (given a not too unreasonable rate of simulated callbacks).

It is generally safe to inject callbacks for inode glocks (type 2) however, for reasons which should be clear from the discussion in the previous section, injecting callbacks for iopen glocks (type 5) is likely to result in a corrupted filesystem.

Callback injection is intended for testing in carefully designed test environments, and is not something that we would encourage general use of.

## 3 Trace points

Linux has an event tracing subsystem which provides a high-performance method of tracking certain operations which are switchable at run time, can be filtered and provided to user space either as a series of human readable ASCII messages or via a binary output format.

The goal in placing trace points in GFS2 was to find a minimum possible number of trace points which can elicit the maximum possible amount of information about the correctness and performance of GFS2 whilst at the same time ensuring that they are generic enough that they will not need to be altered during future developments.

With that in mind, the trace points in GFS2 have been put into three major categories reflecting the main functions of the filesystem.

### 3.1 glock subsystem

For some time there has been a debugfs file available which gives a static view of the glock state as described above. It has been extremely useful in debugging deadlock issues, however although it shows what state the

cluster is currently in, its main failing is that it doesn't always show how it got into that state.

The trace points have also been designed with a view to being able to confirm the correctness of the cache control by combining them with the blktrace output and with knowledge of the on-disk layout. It is then possible to check that any given I/O has been issued and completed under the correct lock, and that no races are present.

The on-disk layout could be parsed in a static manner before the test starts, and then updated can be gained by using the block map trace points.

On the performance side, there are specific questions which we want to be able to answer relating to latencies of performing the various cache control operations. With the current set of trace points it is possible to measure the latencies of granting a new or cached glock, and receiving a callback and flushing the cache.

Information gathered from this will be then used to help target our efforts in improving the scalability of the filesystem.

In the initial implementation of the glock tracing patch, it was an extension to `blktrace` rather than the event tracer that it is currently. There was only one trace point at that time, which was `gfs2_glock_state_change` right in the heart of the glock layer.

Due to the way in which the glock layer is designed, it is only valid to read data or metadata relating to an inode when the glock is in the shared, deferred or exclusive states. It is only valid to write data or metadata when the glock is in the deferred or exclusive states.

Given knowledge of which disk blocks belong to which inode, a task made much easier by the `FIEMAP ioctl()` it should be possible to take the combined information from `blktrace` and `gfs2_glock_state_change` and check that this is indeed the case.

Since the scope of the tracing in GFS2 has been expanded, it is also possible to use the bmap trace points to elicit the same information. Also, due to some further recent patches, GFS2 also tags all of its metadata I/O with the metadata flag so that any violations should be easier to spot.

The `gfs2_glock_put` trace point was introduced so that tracing applications would be able to reduce the amount of state that they needed to remember, and also to allow dynamic tracking of the numbers of allocated glocks.

In order to keep track of demote requests the `gfs2_demote_rq` provides enough information to know when a request has been received and its source (local or remote). The plan is to use this to help measure performance.

Also of interest in the performance area is the time taken from a lock request being submitted to it being granted. This can be derived from the trace points `gfs2_glock_queue` and `gfs2_promote`.

### 3.2 Block map subsystem

Block mapping is a task central to any filesystem. GFS2 uses a traditional bitmap based system with two bits per block. The main aim of the trace points in this subsystem is to allow monitoring of the time taken to allocate and map blocks.

Additionally, with knowledge of the blocks being mapped into inodes, it is possible to keep track of the filesystem layout dynamically and extend tools to take advantage of this.

The `gfs2_bmap` trace point is called twice for each bmap operation. Once at the start to display the bmap request, and once at the end to display the result.

To keep track of allocated blocks, `gfs2_block_alloc` is called not only on allocations, but also on freeing of blocks.

### 3.3 Journal/log subsystem

The trace points in this subsystem track blocks being added to and removed from the journal (`gfs2_pin`), as well as the time taken to commit the transactions to the log (`gfs2_log_flush`). This can be very useful when trying to balance the memory usage of a large log against filesystem performance.

The `gfs2_log_blocks` keeps track of the reserved blocks in the log, which can help show if the log is too small for the workload, for example.

## References


[1] "The GFS2 Filesystem," Steven Whitehouse, *Proceedings of the Linux Symposium*, Ottawa, Canada, June, 2007.

[2] "Global File System," Steven Whitehouse *et al.*, *Wikipedia*, `http://en.wikipedia.org/wiki/Global_File_System`

[3] "Askant: A Linux file system performance analysis tool," Andrew Price, *Final year project*, Swansea University, Summer 2008.

# Fixing PCI Suspend and Resume

Rafael J. Wysocki
*University of Warsaw, Faculty of Physics, Hoża 69, 00-681 Warsaw*
rjw@sisk.pl

## Abstract

Interrupt handlers implemented by some PCI device drivers can misbehave if the device they are supposed to handle is not in the state they expect it to be in. If this happens, interrupt storm may occur, potentially leading to a system lockup. Unfortunately, if the device in question uses shared interrupts, this can easily happen during suspend to RAM, after the device has been put into a low power state. It is even more likely that this will happen at resume time, before the device is brought back to the state it was in before the suspend. On some machines this leads to intermittent resume failures that are very difficult to diagnose.

In Linux kernels prior to 2.6.29-rc3 the power management core did not do anything to prevent such failures from happening, but during the 2.6.29 development cycle we started to address the issue. Still, the solution finally implemented in the 2.6.29 kernel is partial, because it only covers the devices that support the native PCI power management and it only affects the resume part of the code. The complete solution, which has been included into 2.6.30 and which is described in the present paper, required us to make some radical changes, including a rearrangement of the major steps performed by the kernel during suspend and resume. However, not only should it make suspend and resume much more reliable on a number of systems, but it should also allow the writers of PCI device drivers to simplify their code, because some standard PCI device power management operations will now be carried out by the core.

## 1 Introduction

From a computer user's point of view suspend to RAM appears to be a relatively simple operation. There is an event that triggers it, for example a button is pressed or a laptop lid is closed, and in a few seconds the machine is put into a state in which power is only provided to its memory chips in order to preserve their contents. This state will further be referred to as the *memory sleep* state.

Analogously, during resume, after a specific event which may be pressing of a button, opening a laptop lid or receiving a magic packet from a network, the system is brought from the memory sleep state back to the working state in a few seconds. The whole operation does not seem to be very complicated, but at the kernel level it involves the execution of large amount of code that is not run in any other circumstances. Moreover, if one part of this code fails, it usually means that the system will not reach the working state again or, even if it does, its functionality will be adversely affected.

It usually is hard to diagnose such failures, especially if they happen sufficiently late during suspend or sufficiently early during resume, since in that cases there is no practical way to get any useful debugging information out of the failing machine. Quite often the only chance to get some insight into the problem is when it appears as a regression and we are able to identify the exact change that caused it to appear.

Something like this happened during the 2.6.28 development cycle, when resume started to break on one of the author's test machines in a reproducible way and it seemed to be a result of a change of the PCI core code responsible for allocating PCI resources. It was confusing, because the change in question should not have affected suspend and resume in any way, but in the process of debugging it Linus Torvalds suggested that it might be a result of mishandling shared PCI interrupts during resume. Namely, due to the way in which the majority of PCI device drivers handled suspend and resume, there was a time window in which an interrupt could arrive before the devices were put into the states they had been in before the suspend. Then, if one of the involved interrupt handlers could not cope with this situation, the system would crash [LINUS1].

Following this suggestion, we created a patch that made the PCI power management core code put all devices supporting the native PCI power management into the full power state (*D*0) and restore their standard configuration registers to the pre-suspend state before enabling the CPU to receive hardware interrupts [PATCH1]. That indeed made the problem go away on the affected system and it was confirmed to fix resume on Linus' own machine, so it has been included into the 2.6.29 kernel. Still, the devices that do not support the native PCI power management might also be affected by the issue with shared interrupts during resume, and the patch did not cover the suspend part of the code at all. Thus, we had to do more to fix the problem completely, but that turned out to be quite difficult [LINUS2].

The main obstacle was the need to use platform hooks for changing the power state of PCI devices that did not support the native power management, because these hooks could not be executed with interrupts disabled on the CPU. To overcome it we had to rework the core kernel's suspend and resume code so that it prevented hardware interrupts from reaching device drivers when PCI devices might not be in the right states without disabling interrupts on the CPU. Moreover, the ACPI specification[1] wants us to put devices into low power states before calling the platform firmware to prepare itself for the system power transition and we had to take this into account as well [ACPI-SPEC]. Consequently, to meet all of the constraints, we rearranged the major steps carried out by the kernel during suspend and resume, which allowed us to develop the complete solution that is present in 2.6.30.

In what follows we describe the problem in greater detail and show a scenario in which it could manifest itself. For this purpose, among other things, we examine the structure of the kernel's suspend and resume code, and explain how it made the observed failures possible. Next, we present both the partial solution used in 2.6.29 and the final one recently merged into the main kernel tree. Finally, we discuss the consequences of this for the authors of PCI device drivers.

## 2 Description of the problem

To understand the problem and the approach used to fix it, one needs to know how the kernel's suspend and resume code works, but that has been presented elsewhere

[1] ACPI 2.0 or later.

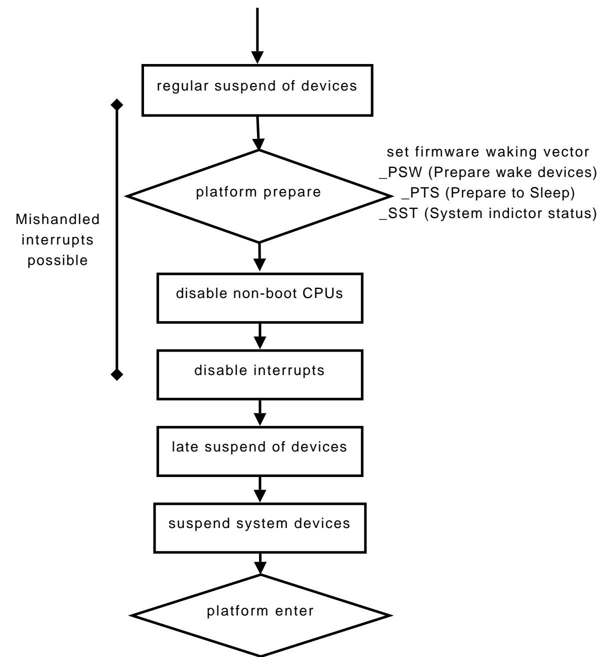



Figure 1: Suspend code structure (2.6.29).

and we are not going to repeat this entire discussion [S2RAM]. Still, we need to recall the parts of it the problem description below is based on.

In the 2.6.29 and earlier kernels there are two phases of suspending devices. The first of them, that further will be referred to as the *regular suspend of devices*, is carried out right before invoking the platform `.prepare()` callback which executes the `_PTS` global control method on ACPI systems. After `.prepare()` has returned, the non-boot CPUs are put off line[2] and interrupts are disabled on the only remaining active CPU. Then, the second phase of suspending devices, that we will refer to as the *late suspend of devices*, is performed. Next, the special system devices called *sysdevs*, such as the local APIC of the boot processor and I/O-APICs, are suspended and the platform is called to put the system into the memory sleep state [S2RAM]. The structure of the suspend code in the 2.6.29 and earlier kernels is schematically shown in Figure 1.

The resume part of the code is organized analogously. First, *sysdevs* are resumed right after the platform firmware has returned control to the kernel. Next, the kernel carries out the first phase of resuming devices

[2] This is accomplished with the help of the CPU hotplug infrastructure.

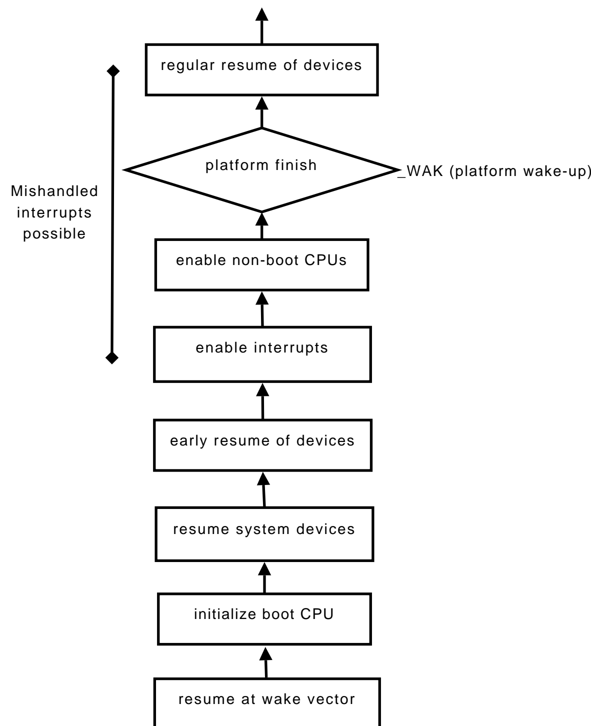



Figure 2: Resume code structure (2.6.29).

that we will refer to as the *early resume of devices*. After that, interrupts are enabled on the only active CPU and the other CPUs are brought back on line. Subsequently, the platform `.finish()` callback is run which causes the `_WAK` global control method to be executed on ACPI systems. Then, the kernel starts the second phase of resuming devices, that further will be referred to as the *regular resume of devices* [S2RAM]. The structure of the resume code in the 2.6.29 and earlier kernels is schematically illustrated in Figure 2.

Thus, in principle, every PCI device driver can implement two suspend callbacks, a *regular* one, to be called before the platform `.prepare()` routine, and a *late* one, to be executed with interrupts disabled. It can also define two analogous resume callbacks, an *early* one, to be executed with interrupts disabled, and a *regular* one, to be executed after the platform `.finish()` callback. However, according to the ACPI specification devices should be put into low power state before the `_PTS` global control method is run [ACPI-SPEC], so the vast majority of drivers implement the regular suspend and resume callbacks only. The late suspend and early resume callbacks are only provided by a few drivers for special purposes. Accordingly, during suspend and analogously during resume there is a time interval in which devices may not be operational or even accessible to their drivers, but the processors can receive interrupts. Then, interrupt handlers may be invoked, even if their devices do not generate any interrupts and if they are not prepared to cope with that situation, the consequences may be dire [LINUS3]. The time intervals in which this is possible during suspend and resume are marked in Figures 1 and 2, respectively, with a vertical line on the left-hand side.

Of course, a PCI device in a low-power state cannot generate interrupts. Yet, if that device uses shared interrupts, then its driver's interrupt handler may be invoked as a result of an interrupt generated by one of the other devices sharing an interrupt vector with it. Moreover, this actually is quite probable, because the suspend and resume callbacks provided by device drivers are executed sequentially [S2RAM], so it is *guaranteed* that one of the devices sharing the interrupt vector will be suspended earlier and resumed later than the other ones[3]. Therefore, if one of these devices is handled by a driver with an interrupt handler that is not designed to work correctly even if the device is not in the right state, things are likely to go wrong. Namely, if the device that has not been suspended yet or that has already been resumed generates an interrupt, the other devices' interrupt handlers will be invoked, and if they fail, the system is going to crash.

This generally is more likely to happen during resume. Specifically, while during suspend the devices are either fully operational or in low power states, so they behave more or less in accordance with the drivers' expectations, during resume they are usually in *D*0 (full power state), but they need not have been initialized yet. Then, before they eventually get initialized, they may respond to the drivers' attempts to access them in a confusing way. Furthermore, during resume interrupts may be generated as a result of a chipset glitch or something similar and that also may confuse interrupt handlers that are not prepared to cope with devices in unexpected states.

Certainly, if every interrupt handler had been able to cope with an already suspended or not yet resumed device, the problem would not have manifested itself. Unfortunately, this evidently is not the case and it would not have been practical to try to fix all of the affected

[3]The order of resuming devices is reverse with respect to the order of suspending them.

drivers [LINUS2]. For this reason the power management core (PM core) code had to be modified to prevent the problem from happening.

## 3 Preliminary fix

To fix the problem described in Section 2, we used the observation that it would not appear during resume if the standard configuration registers of PCI devices were restored to the pre-suspend state before enabling the CPUs to receive hardware interrupts. Analogously, during suspend the problem could not happen if devices were put into low power states after disabling interrupts on the boot CPU, but this turned out to be more difficult to implement, so we first focused on the resume fix.

To implement it, we had to make the PM core restore the standard configuration registers of PCI devices in the early phase of resume, before executing the early resume callbacks provided by PCI device drivers. Fortunately, that was not too difficult to achieve, thanks to the organization of the device resume code. Namely, the lowest-level resume code[4] does not execute the drivers' resume callbacks directly [S2RAM]. Instead, it invokes the resume callbacks defined by the `pci_bus_type` bus type driver[5], which in turn are responsible for executing the drivers' callbacks. More precisely, the `pci_bus_type` bus type implements a `dev_pm_ops` object, called `pci_dev_pm_ops`, that points to a set of power management callbacks executed by the lowest-level power management code in various phases of suspend and resume[6]. In particular, the `pci_pm_resume_noirq()` routine is called during the early resume of devices and it is responsible for executing the early resume callback provided by each PCI device driver. Thus, it was sufficient to make `pci_pm_resume_noirq()` restore the standard configuration registers of each PCI device before executing the early resume callback provided by its driver, if there was one.

For this purpose, however, we had to ensure the accessibility of the device's standard configuration registers before attempting to restore their pre-suspend values. Of course, in theory, the configuration space of a PCI device should be accessible in any power state, except for $D3_{cold}$ [PCI-PM]; but in practice it is better to put all devices into $D0$ before restoring their configuration registers. Still, we could not use `pci_set_power_state()` to do that, because this function might sleep and therefore it would not be valid to call it with interrupts disabled.

There are two reasons why `pci_set_power_state()` may sleep. First, when changing the power state of a device from $D3$ to $D0$ there is the mandatory 10 ms delay, necessary to allow the device to actually enter the full power state [PCI-PM], and it is implemented in `pci_set_power_state()` with the help of `msleep()`. Second, `pci_set_power_state()` executes a platform callback that in principle may be necessary to set up some power resources (e.g. power planes, reference clock) required to power up the device and this callback may sleep. Thus, during the early resume of devices we could only use the native PCI power management mechanism, implemented in `pci_raw_set_power_state()`, for putting devices into $D0$. Moreover, to be able to prevent `pci_raw_set_power_state()` from calling `msleep()` with interrupts disabled we had to add a new parameter to it. Accordingly, the function called `pci_restore_standard_config()`, shown in Figure 3, was introduced and `pci_pm_resume_noirq()` was modified to call it before executing the device driver's resume callback [PATCH1].

Since the configuration spaces of PCI devices were going to be restored during the early resume, it was necessary to make sure that they would be saved during suspend. For this reason `pci_pm_suspend()` was modified to execute `pci_save_state()` before returning to the caller. However, it could not do that unconditionally, because many device drivers saved the PCI configuration spaces of their devices by themselves before putting the devices into low power states (usually $D3_{hot}$). Of course, in that case, the contents of PCI configuration registers saved by the driver before the device was put into the low power state should not be replaced by their contents saved by `pci_pm_suspend()` when the device *was* already in that state. Therefore, the `state_saved` flag was added to the `pci_dev` structure and `pci_save_state()` was changed to set this flag on every execution, so that `pci_pm_suspend()` could use it do decide whether or not to save the PCI configuration registers of given device. This flag was also

[4] Located in `drivers/base/power/main.c`

[5] Defined in `drivers/pci/pci-driver.c`

[6] It also points to several hibernation-specific callbacks, but they are out of the scope of the present discussion.

```
int pci_restore_standard_config(struct pci_dev *dev)
{
  pci_power_t prev_state;
  int error;

  pci_update_current_state(dev, PCI_D0);

  prev_state = dev->current_state;
  if (prev_state == PCI_D0)
    goto Restore;

  error = pci_raw_set_power_state(dev, PCI_D0, false);
  if (error)
    return error;

  /*
   * This assumes that we won't get a bus in B2 or B3
   * from the BIOS, but we've made this assumption
   * forever and it appears to be universally satisfied.
   */
  switch(prev_state) {
  case PCI_D3cold:
  case PCI_D3hot:
    mdelay(pci_pm_d3_delay);
    break;
  case PCI_D2:
    udelay(PCI_PM_D2_DELAY);
    break;
  }

  pci_update_current_state(dev, PCI_D0);

 Restore:
  return dev->state_saved ?
    pci_restore_state(dev) : 0;
}
```

Figure 3: Function called during early resume to restore configuration space of a PCI device (2.6.29).

used by `pci_restore_standard_config()` to decide whether or not the device's configuration space ought to be restored [PATCH1].

The changes described above caused the standard configuration registers of PCI devices supporting the native PCI power management to be saved during suspend and restored during early resume with interrupts disabled on the CPU. That turned out to fix the problem described in Section 2 on a number of test machines, but it obviously did not cover devices requiring additional power resources controlled by the platform to be set up for putting them into *D*0. It also did not cover the suspend-specific part of the problem in which interrupt handlers might be invoked although their devices had already been put into low power states. Still, both of these shortcomings were related to the fact that the platform hooks necessary to change the power state of some devices could not be executed with interrupts disabled and we were not able to use `pci_set_power_state()` during the late suspend and early resume of devices. Hence, to remove the limitations, it was necessary to either modify the platform hooks so that they could be executed with interrupts disabled, or change the suspend and resume code so that interrupts were enabled on the CPU during the late and early phases of device suspend and resume, respectively [LINUS4].

## 4 Complete solution

By using the approach presented in Section 3 we were able to partially fix the problem described in Section 2 for devices that supported the PCI native power management. Still, more in-depth changes were necessary to fix it completely and for all devices. Namely, for this purpose we had to make it possible to execute platform callbacks used for changing the power states of devices during the late phase of suspend and the early phase of resume.

There were two possible ways to achieve this goal. First, the platform callbacks, most importantly the ACPI ones, could be modified so that executing them with interrupts disabled was valid, which would have been the case if they had not taken any mutexes and generally if they had avoided using functions that might sleep. This, however, turned out to be impractical due to the complexity of the ACPI code and to the fact that substantial part of it, known as ACPICA[7], was shared with some other operating systems, like BSD [ACPICA]. Second, the core suspend and resume code could be modified so that the CPUs were able to receive hardware interrupts while executing the late suspend and early resume callbacks provided by device drivers. Specifically, as suggested by Linus Torvalds, instead of disabling interrupts on the CPU before the late suspend of devices, we could prevent device drivers from receiving interrupts by running `irq_disable()` for all interrupt vectors except for the timer ones [LINUS5]. Then, even though the CPUs would be able to receive and acknowledge hardware interrupts, the interrupts would be disabled from the drivers' point of view. Moreover, since timer interrupts would still be handled normally, it would be valid to call the potentially sleeping functions from the device drivers' late suspend routines and if we did the analogous change in the resume part

[7] ACPI Common Architecture.

of the code, we would be able to invoke these functions from the drivers' early resume callbacks and from `pci_restore_standard_config()` as well.

This approach was used in the 2.6.30-rc3 and later kernels. It allowed us to put PCI devices into $D0$ from within `pci_restore_standard_config()` with the help of `pci_set_power_state()` which caused the platform callback changing the device's power state to be called as appropriate. Analogously, we could use `pci_set_power_state()` to put PCI devices into low power states during the late suspend of devices. Still, at the same time we wanted to preserve the suspend and resume code ordering following from the ACPI specification[8] stating that devices ought to be put into low power states before the execution of the `_PTS` global control method which, in turn, ought to happen before disabling the non-boot CPUs [ACPI-SPEC]. Similarly, during resume we were supposed to enable the non-boot CPUs before executing the `_WAK` ACPI global control method which, in turn, ought to happen before putting devices into $D0$. Thus, we proposed to make device drivers' late suspend callbacks be executed before the platform `.prepare()` callback and, analogously, their early resume callbacks be executed after the platform `.finish()` callback. In principle, this change should not matter to device drivers and it would allow us to avoid some otherwise inevitable complications related to reordering ACPI callbacks with respect to one another [RJW1].

Unfortunately, it turned out that some platforms, like for example ARM PXA, used the `.prepare()` and `.finish()` callbacks to communicate with power control devices over an I2C bus the controller of which is effectively turned off and on during the late suspend and early resume of devices, respectively [RMK]. As a result, suspend and resume would not have worked on these platforms if the `.prepare()` and `.finish()` callbacks had been executed, respectively, after the late suspend and before the early resume of devices. For this reason we decided to introduce two additional platform callbacks for suspend and resume, `.prepare_late()` and `.wake()`, to be executed in these two places and we used them for running the code that was previously run in `.prepare()` and `.finish()`, respectively, on ACPI systems [RJW2].

Consequently, in the 2.6.30-rc3 and later kernels the ordering of the suspend and resume code is different from its ordering in the 2.6.29 and earlier kernels. Now, after the first phase of suspending devices (i.e. regular suspend) and the execution of the `.prepare()` platform callback, device drivers are prevented from receiving hardware interrupts which is immediately followed by the second phase of suspending devices (i.e. late suspend). Then, the platform `.prepare_late()` callback is executed, the non-boot CPUs are disabled and interrupts are disabled on the only on-line CPU. Finally, *sysdevs* are suspended and the platform is called to put the system into the memory sleep state. Analogously, during resume, after the platform firmware has transferred control to the kernel, *sysdevs* are resumed, interrupts are enabled on the only on-line CPU and the other CPUs are brought back on line. Next, the platform `.wake()` callback is executed and device drivers' early resume callbacks are run. Finally, device drivers are allowed to receive hardware interrupts, the platform `.finish()` callback is executed and the regular resume of devices is carried out. The structure of the suspend and resume code in the 2.6.30-rc3 and later kernels is illustrated in Figure 4.

To make suspend and resume work as described in the previous paragraph, we needed a reversible mechanism for preventing device drivers from receiving interrupts after the regular suspend of devices. For this purpose we had to make some changes to the generic interrupt management code.[9] Roughly, we wanted `irq_disable()` to be called for every interrupt vector, except for the timer ones, right after the completion of the regular suspend of devices, but we also needed to ensure that `irq_enable()` would be called for these interrupt vectors during resume. Thus the `IRQ_SUSPENDED` interrupt status flags was introduced and used to mark the interrupt vectors disabled during suspend, so that they could be re-enabled during resume. Moreover, we had to take the locking and reference counting done by the interrupt management code into account, so we introduced the helper function `__disable_irq()` complementary to `__enable_irq()` and we made both of these functions take an additional Boolean parameter specifying whether or not they were called by the kernel's core suspend and resume code. [It calls them via `suspend_device_irqs()` and `resume_device_irqs()` shown in Figures 5 and 6, respectively.]

In addition, the fact that some platforms used wake-up

[8] ACPI 2.0 or later.

[9] Located in `kernel/irq/manage.c`.

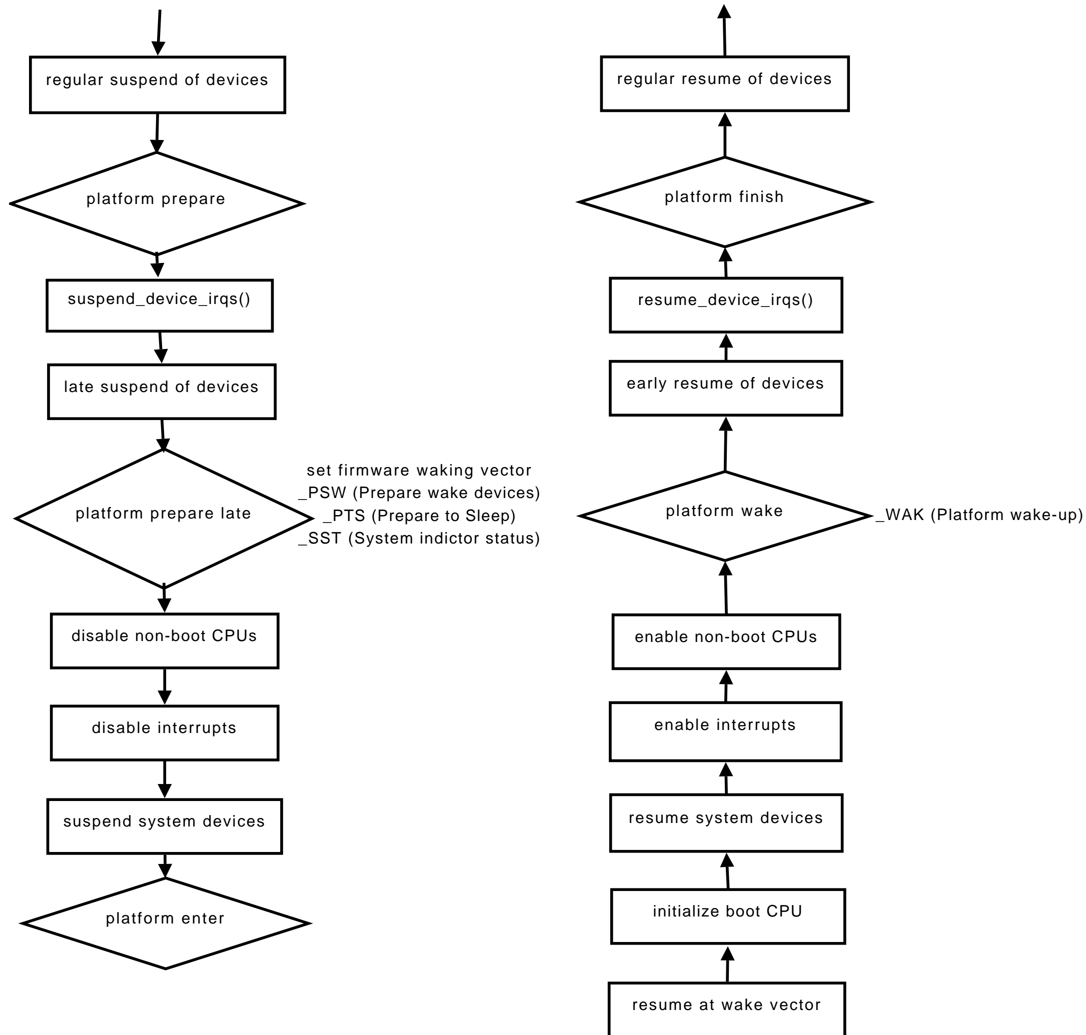



Figure 4: Suspend (left) and resume (right) code structure (2.6.30-rc3 and later).

interrupts to abort suspend, if necessary, had to be taken into consideration. Specifically, in the 2.6.29 and earlier kernels it was possible to mark an interrupt vector with the `IRQ_WAKEUP` status flag and make the platform abort suspend if that interrupt was pending after *sysdevs* had been suspended[10]. However, that might not work any more after introducing `suspend_device_irqs()` and `resume_device_irqs()` and rearranging the core suspend and resume code as described above, because the wake-up interrupts generated after the late suspend of devices would be acknowledged by one of the CPUs. In that case they would not appear to the platform code as pending, since they had been acknowledged by the CPU and the suspend would not be aborted. That would have been a significant change of behavior relative to the 2.6.29 kernel and we had to avoid it. For this purpose we used the observation that the `IRQ_PENDING` flag was set for interrupts acknowledged by the CPUs after `suspend_device_irqs()` had run, so it was sufficient to check this flag along with `IRQ_WAKEUP` after putting the non-boot CPUs off line and disabling interrupts on the remaining active one. Therefore, we introduced the function `check_wakeup_irqs()` shown in Figure 7 and made `sysdev_suspend()` call it and fail if the error code was returned [RJW3].

With all of the above modifications in place we could change `pci_restore_standard_config()` so that it used `pci_set_power_state()` to put devices into *D*0 before attempting to restore their standard PCI configuration registers with the help of `pci_`

[10] The *x86* architecture has never used wake-up interrupts.

```
void suspend_device_irqs(void) {
  struct irq_desc *desc;
  int irq;

  for_each_irq_desc(irq, desc) {
    unsigned long flags;

    spin_lock_irqsave(&desc->lock, flags);
    __disable_irq(desc, irq, true);
    spin_unlock_irqrestore(&desc->lock, flags);
  }

  for_each_irq_desc(irq, desc)
    if (desc->status & IRQ_SUSPENDED)
      synchronize_irq(irq);
}
```

Figure 5: Function called during suspend to prevent device drivers from receiving interrupts (2.6.30-rc3 and later)

```
void resume_device_irqs(void) {
  struct irq_desc *desc;
  int irq;

  for_each_irq_desc(irq, desc) {
    unsigned long flags;

    if (!(desc->status & IRQ_SUSPENDED))
      continue;

    spin_lock_irqsave(&desc->lock, flags);
    __enable_irq(desc, irq, true);
    spin_unlock_irqrestore(&desc->lock, flags);
  }
}
```

Figure 6: Function called during resume to allow device drivers to receive interrupts (2.6.30-rc3 and later)

`restore_state()` [RJW4]. As a result, it has been substantially simplified[11] as shown in Figure 8. We also changed the suspend callbacks of the `pci_bus_type` driver so that the late suspend callback saved the configuration spaces of the devices for which they were not saved by the drivers [RJW5]. This made it possible to develop a working PCI device driver supporting power management that would not touch the standard PCI configuration registers of the device in its suspend and resume callbacks allowing the PCI PM core to handle them as appropriate [RJW6].

[11] It also has been moved to `drivers/pci/pci-driver.c`.

```
int check_wakeup_irqs(void) {
  struct irq_desc *desc;
  int irq;

  for_each_irq_desc(irq, desc)
    if ((desc->status & IRQ_WAKEUP)
        && (desc->status & IRQ_PENDING))
      return -EBUSY;

  return 0;
}
```

Figure 7: Function called to check for wake-up interrupts right before suspending sysdevs (2.6.30-rc3 and later)

```
static int pci_restore_standard_config(struct pci_dev
                                        *pci_dev) {
  pci_update_current_state(pci_dev, PCI_UNKNOWN);

  if (pci_dev->current_state != PCI_D0) {
    int error = pci_set_power_state(pci_dev, PCI_D0);
    if (error)
      return error;
  }

  return pci_dev->state_saved ?
    pci_restore_state(pci_dev) : 0;
}
```

Figure 8: Function called during early resume to restore configuration space of a PCI device (2.6.30-rc3 and later)

## 5 Consequences

Using the approach presented in Section 4 has profound consequences for the writers of PCI device drivers wanting to support suspend and resume.[12] Namely, it allows them to let the PCI power management (PM) core take care of the "ugly" details related to PCI power management, such as the saving and restoration of the standard configuration registers, putting the device into a low power state during suspend and into $D0$ during resume, and preparing it to wake up the system from the memory sleep state, if desired. Since PCI device drivers have traditionally had problems with getting these things right, allowing them to leave it all to the core appears to be a big improvement. Of course, the drivers can still power manage the devices by themselves, which may even be

[12] In our not so humble opinion, every new PCI device driver ought to support suspend and resume.

necessary in some more complicated cases, but generally it is better if their authors avoid doing that, unless they know *very well* what they are doing.

In general, PCI device drivers can implement suspend and resume callbacks in two different ways. First, they can implement the `.suspend()`, `.suspend_late()`, `.resume()`, and `.resume_early()` callbacks available in the `pci_driver` structure, as shown in Figure 9. In that case, as indicated by the names of the callbacks, `.suspend()` will be executed in the first phase of suspending devices (regular suspend), `.suspend_late()` will be executed in the second phase of suspending devices (late suspend) and analogously for the resume callbacks. Drivers do not have to implement all of these callbacks, but if at least one of them is implemented, the PCI PM core will regard the driver as a "legacy" one and will apply special rules to the device handled by it. In particular, such a device will not be power managed by the PCI PM core during suspend and it will not be prepared by the core to wake up the system. Apart from this, the suspend and resume callbacks defined in `pci_driver` are used for hibernation as well as for suspend to RAM. Accordingly, the suspend callbacks take an additional argument of type `pm_message_t` specifying the context in which they are called (suspend to RAM or hibernation), but the resume callbacks do not take any additional arguments, so it is the driver's responsibility to preserve the context information over the suspend-resume (or hibernation-resume) cycle if needed.

```
struct pci_driver {
  ...
  int (*suspend) (struct pci_dev *dev, pm_message_t state);
  int (*suspend_late) (struct pci_dev *dev, pm_message_t state);
  int (*resume_early) (struct pci_dev *dev);
  int (*resume) (struct pci_dev *dev);
  ...
};
```

Figure 9: Members of `struct pci_driver` used during suspend to RAM and resume

The second way to implement suspend and resume callbacks in a PCI device driver is to use a `dev_pm_ops` object pointed to by the `driver.pm` member of the `pci_driver` structure.[13] This object, if present, contains pointers to several power management callbacks that can be implemented by a device driver. The majority of them are hibernation-specific and we are not going to discuss them here, but the ones shown in Figure 10 are used during suspend to RAM and resume. Still, the first two of them, `.prepare()` and `.complete()`, are only of interest to the authors of complicated drivers involving the management of children devices that may be registered and unregistered at any time, so we will not discuss them either. The remaining four callbacks are direct counterparts of the "legacy" ones discussed in the previous paragraph, where the `_noirq` suffix in the name means that the callback is a "late" or "early" one. In other words, `.suspend_noirq()` plays the role of `.suspend_late()` discussed previously and analogously for `.resume_noirq()`.

```
struct dev_pm_ops {
  int (*prepare)(struct device *dev);
  void (*complete)(struct device *dev);
  int (*suspend)(struct device *dev);
  int (*resume)(struct device *dev);
  ...
  int (*suspend_noirq)(struct device *dev);
  int (*resume_noirq)(struct device *dev);
  ...
};
```

Figure 10: Members of `struct dev_pm_ops` used during suspend to RAM and resume

As already stated, a PCI device driver implementing the "legacy" suspend and resume callbacks *has to* take care of putting the device into a low power state and, if necessary, preparing it to wake up the system during suspend, although it need not put the device into *D*0 during resume, since the PCI PM core is going to do that anyway via `pci_restore_standard_config()`. Moreover, during suspend the device should be put into the low power state by `.suspend_late()`, since otherwise the problem described in Section 2 may appear. In turn, a PCI device driver supporting suspend and resume through a `dev_pm_ops` object need not power manage the device during suspend and resume at all. However, if its author decides to put device into a low power state[14] and prepare it for waking up the system during suspend, these operations should be carried out in `.suspend_noirq()` to avoid the problem described in Section 2. Thus, it is possible to consider the "late" and "early" callbacks as the ones in which the actual power management of the device takes place, while

[13]The `struct dev_pm_ops` structure is defined in `include/linux/pm.h`.

[14]The standard PCI configuration registers of the device must be saved before that happens, since otherwise the restoration of their contents during resume may lead to undefined behavior.

the remaining "regular" callbacks can be regarded as the ones causing the driver to stop—`.suspend()`—or start—`.resume()`—using the device without changing the power state and related properties of it. Of course, in the kernels preceding 2.6.30-rc1 it was impossible to classify the suspend and resume callbacks provided by device drivers this way.

As follows from the above discussion, it is recommended and in the majority of cases more convenient to support suspend and resume by using a `dev_pm_ops` object rather then by implementing the "legacy" suspend and resume callbacks. Thus, it seems reasonable to give an example showing how to replace the "legacy" callbacks by a `dev_pm_ops` object in an existing driver and clearly illustrating the benefit of doing so. For this purpose consider the suspend and resume callbacks implemented by the `r8169` network driver shown in Figures 11 and 12, respectively, and observe that the PCI-specific operations carried out by `rtl8169_suspend()` should in fact be moved to a "late" suspend callback. Still, that will not be necessary if the driver provides the support for suspend and resume through a `dev_pm_ops` object.

```
static int rtl8169_suspend(struct pci_dev *pdev, pm_message_t state)
{
  struct net_device *dev = pci_get_drvdata(pdev);
  struct rtl8169_private *tp = netdev_priv(dev);
  void __iomem *ioaddr = tp->mmio_addr;

  if (!netif_running(dev))
    goto out_pci_suspend;

  netif_device_detach(dev);
  netif_stop_queue(dev);

  spin_lock_irq(&tp->lock);
  rtl8169_asic_down(ioaddr);
  rtl8169_rx_missed(dev, ioaddr);
  spin_unlock_irq(&tp->lock);

out_pci_suspend:
  pci_save_state(pdev);
  pci_enable_wake(pdev, pci_choose_state(pdev, state),
    (tp->features & RTL_FEATURE_WOL) ? 1 : 0);
  pci_set_power_state(pdev, pci_choose_state(pdev, state));

  return 0;
}
```

Figure 11: Suspend callback of the `r8169` driver (2.6.29)

To replace the "legacy" callbacks provided by the `r8169` driver with an implementation based on a `dev_pm_ops` object one can move the non-PCI part of `rtl8169_suspend()` to a separate function, like the one shown in Figure 13, and drop all of the PCI-specific

```
static int rtl8169_resume(struct pci_dev *pdev) {
  struct net_device *dev = pci_get_drvdata(pdev);

  pci_set_power_state(pdev, PCI_D0);
  pci_restore_state(pdev);
  pci_enable_wake(pdev, PCI_D0, 0);

  if (!netif_running(dev))
    goto out;

  netif_device_attach(dev);

  rtl8169_schedule_work(dev, rtl8169_reset_task);
out:
  return 0;
}
```

Figure 12: Resume callback of the `r8169` driver (2.6.29)

operations from both the suspend and resume routines. Of course, the `dev_pm_ops` object has to be defined too, but this is really straightforward as illustrated by the code in Figure 14 showing a possible implementation of it for the `r8169` driver[15]. Finally, the `driver.pm` member of the driver's `pci_driver` object has to be made point to the `dev_pm_ops` object defined by the driver, like the `rtl8169_pm_ops` object shown in Figure 14 in this particular case.

```
static void rtl8169_net_suspend(struct net_device *dev) {
  struct rtl8169_private *tp = netdev_priv(dev);
  void __iomem *ioaddr = tp->mmio_addr;

  if (!netif_running(dev))
    return;

  netif_device_detach(dev);
  netif_stop_queue(dev);

  spin_lock_irq(&tp->lock);
  rtl8169_asic_down(ioaddr);
  rtl8169_rx_missed(dev, ioaddr);
  spin_unlock_irq(&tp->lock);
}
```

Figure 13: Non-PCI part of the `r8169` driver's suspend callback.

The benefit of implementing suspend and resume support in the new way, as illustrated in Figures 13 and 14, to the `r8169` driver is that it need not worry about the PCI-specific handling of the device during suspend

[15] This particular definition means that the suspend and resume callbacks are going to be used for hibernation as well as for suspend to RAM and the PCI PM core is supposed to take care of the PCI-specific handling of the device in both cases.

```
static int rtl8169_suspend(struct device *device) {
  struct pci_dev *pdev = to_pci_dev(device);
  struct net_device *dev = pci_get_drvdata(pdev);

  rtl8169_net_suspend(dev);

  return 0;
}

static int rtl8169_resume(struct device *device) {
  struct pci_dev *pdev = to_pci_dev(device);
  struct net_device *dev = pci_get_drvdata(pdev);

  if (!netif_running(dev))
    goto out;

  netif_device_attach(dev);

  rtl8169_schedule_work(dev, rtl8169_reset_task);
out:
  return 0;
}

static struct dev_pm_ops rtl8169_pm_ops = {
  .suspend = rtl8169_suspend,
  .resume = rtl8169_resume,
  .freeze = rtl8169_suspend,
  .thaw = rtl8169_resume,
  .poweroff = rtl8169_suspend,
  .restore = rtl8169_resume,
};
```

Figure 14: Simplified suspend and resume support for the `r8169` driver

and resume and it need not implement any "late" and "early" callbacks. Moreover, it can use the same simplified "regular" callbacks for both suspend to RAM and hibernation allowing the PCI PM core to take care of the PCI-specific operations that ought to be carried out in any of these cases. In fact, if that implementation of suspend and resume support is used, the driver only needs to stop using the device during the regular suspend of devices and start using it during the regular resume of devices without doing anything else. This turns out out be the case for the majority of PCI device drivers currently in the kernel tree.

## 6 Conclusion

By using the approach presented in Sections 3 and 4 to solve the problem described in Section 2 we made it possible to significantly simplify suspend and resume callbacks provided by PCI device drivers. In particular, as shown in Section 5, a PCI device driver's suspend and resume callbacks can be implemented in such a way that all of the PCI-specific operations related to the power management of the device will be carried out by the PCI PM core.

Since the PCI PM core is now going to put every PCI device into $D0$ and restore its standard configuration registers during the early resume of devices, PCI device drivers need not do that any more. Therefore, it would be reasonable to remove these operations from all of the existing resume callbacks provided by PCI device drivers. Moreover, the drivers that put devices into low power states in their regular suspend callbacks should be modified to do it in their late suspend callbacks. Still, it may be even more beneficial to use a `dev_pm_ops` object for implementing suspend and resume support, as illustrated in Section 5, in which case the driver can leave the PCI-specific power management of the device to the PCI PM core.

The changes discussed in Sections 3 and 4 also made it possible to look at the device drivers' suspend and resume callbacks from a new perspective. Namely, the "regular" suspend callback, executed in the first phase of suspend, can be regarded as the one that should make the driver stop using the device, while the "late" suspend callback can be treated as the one preparing the device to wake up the system, if necessary, and putting it into a low power state. Analogously, the "early" resume callback, executed during the early resume of devices, can be regarded as the one that should put the device into the full power state and restore its pre-suspend configuration, while the "regular" resume callback can be treated as the one preparing the driver to use the device again. Of course, for PCI devices some or even all of the tasks of the "late" suspend and "early" resume callbacks can be completed by the PCI PM core, as described above.

## 7 Acknowledgements

The author thanks Linus Torvalds for his help and support of the work presented in this paper, Ingo Molnar and Thomas Gleixner for their help with the modifications of the core interrupt management code and Jesse Barnes for his help with the PCI part. He also thanks everyone who commented and tested patches or contributed to the work presented in this paper in any other way.

## References

[LINUS1] L. Torvalds, *Re: Regression from 2.6.26: Hibernation (possibly suspend) broken* (`http:`

//marc.info/?l=linux-kernel&m=122852860315385&w=4).

[PATCH1] R. J. Wysocki, *PCI PM: Restore standard config registers of all devices early* (http://marc.info/?l=linux-pci&m=123213931514157&w=2).

[LINUS2] L. Torvalds, *Re: PCI PM: Restore standard config registers of all devices early* (http://marc.info/?l=linux-kernel&m=123360797907858&w=2).

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

[S2RAM] L. Brown, R. J. Wysocki, *Suspend to RAM in Linux* (Proceedings of the Linux Symposium, Ottawa 2008 http://ols.fedoraproject.org/OLS/Reprints-2008/brown-reprint.pdf).

[LINUS3] L. Torvalds, *Re: What should PCI core do during suspend-resume?* (http://marc.info/?l=linux-netdev&m=123343922809244&w=2).

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

[LINUS4] L. Torvalds, *Re: PCI PM: Restore standard config registers of all devices early* (http://marc.info/?l=linux-kernel&m=123361303317554&w=2).

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

[LINUS5] L. Torvalds, *Re: PCI PM: Restore standard config registers of all devices early* (http://marc.info/?l=linux-kernel&m=123361270416948&w=2).

[RJW1] R. J. Wysocki, *Re: PCI PM: Restore standard config registers of all devices early* (http://marc.info/?l=linux-kernel&m=123365338201811&w=2).

[RMK] R. King, *900af0d breaks some embedded suspend/resume* (http://marc.info/?l=linux-kernel&m=124026014415587&w=2).

[RJW2] R. J. Wysocki, *PM/Suspend: Introduce two new platform callbacks to avoid breakage* (http://marc.info/?l=linux-kernel&m=124022459914679&w=2).

[RJW3] R. J. Wysocki, *PM: Introduce functions for suspending and resuming device interrupts* (http://marc.info/?l=linux-kernel&m=123703066215140&w=2).

[RJW4] R. J. Wysocki, *PCI PM: Use `pci_set_power_state` during early resume* (http://marc.info/?l=linux-kernel&m=123703114515664&w=2).

[RJW5] R. J. Wysocki, *PCI PM: Put devices into low power states during late suspend (rev. 2)* (http://marc.info/?l=linux-kernel&m=123703114515667&w=2).

[RJW6] R. J. Wysocki, *NET/r8169: Rework suspend and resume* (http://marc.info/?l=linux-kernel&m=123895686321519&w=4).

# Real-Time Performance Analysis in Linux-Based Robotic Systems

Hobin Yoon, Jungmoo Song, and Jamee Lee
*Advanced Software Laboratories,*
*Samsung Advanced Institute of Technology,*
*Samsung Electronics Co. Ltd.*
{hobin.yoon, jmsong, jamee.lee}@samsung.com

## Abstract

Mobile or humanoid robots collect environmental data and reflect back as robotic behaviors via various sensors and actuators. It is crucial this occurs within a specified time. Although real-time flavored Linux has been used to control robot arms and legs for quite a while, it has not been reported much whether the current real-time features in Linux could still meet this requirement for a much more complicated system - a humanoid with about 60 servo motors and sensors with multiple algorithms such as recognition, decision, and navigation running simultaneously. In this paper, in order to meet such requirement, adopting EtherCAT technology is introduced and its Linux implementation is illustrated. In addition, results of real-time experiments and timing analysis on a multi-core processor are presented showing Linux is a viable solution to be successfully deployed in various robotic systems.

## 1 Introduction

One of the key requirements of mobile or humanoid robot is precise control period. It is crucial in robot design in several ways. First it guarantees response time so that robot is able to react properly from external stimulus within a specified time. For example, when a robot hits an obstacle while it walks, if a proper re-balancing of the motion is not executed in a fraction of time, it falls down to the ground. Second, it enables smooth control of each joint which is controlled by a micro controller. Each micro controller tries to compensate movement of each servo motor if it goes too fast or slow and high jitter brings about high current consumption and even noise.

To achieve real-time communication of distributed devices, a field-bus system is used. We have selected EtherCAT over other field-bus systems for its flexible topology, simple configuration, and cost-effectiveness [18, 2]. The technology is supported and promoted by ETC (EtherCAT Technology Group) and standardized by IEC (International Electrotechnical Commission) in 2007.

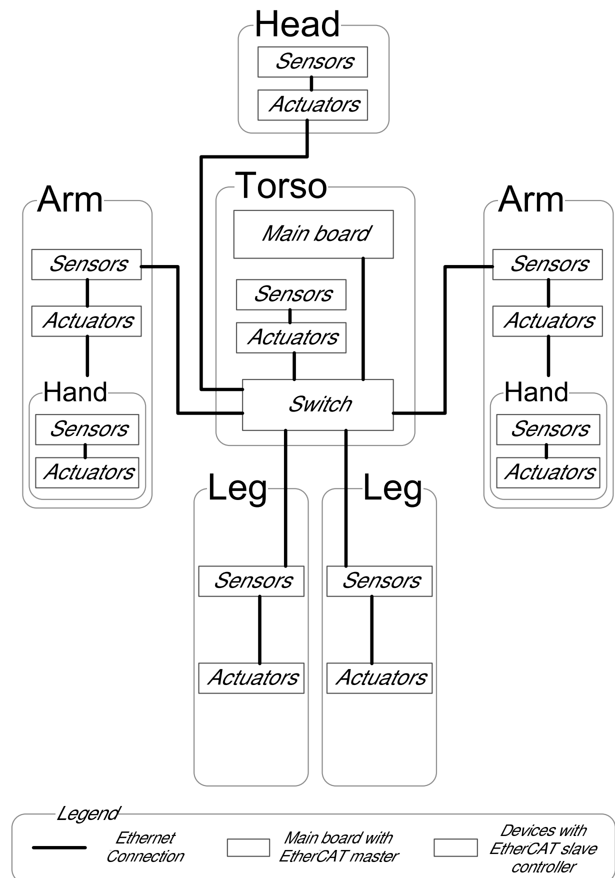



Figure 1: Deployment of EtherCAT master and slave devices

Figure 1 shows schematic diagram of EtherCAT in our robot system. Network interface card on main board in torso plays an EtherCAT master and all other rectangles represent EtherCAT slave devices. They are connected by Ethernet cables which forms various topologies such

as star, tree, and daisy chain. Each EtherCAT slave device is connected by a couple of actuators or sensors.

For EtherCAT master implementation, EtherLab was selected from other master implementations for its proper license and active community [12]. EtherCAT slave hardware is implemented by a few vendors as low-price ASIC. We have selected Beckhoff ET1100 [9].

Real-time scheduling is essential for precise control period. Traditionally, RT OSes such as QNS, RTLinux, VxWorks and Windows CE have been major players in real-time computing. Linux has been evolved a lot for the past few years in terms of real-time. Since in-kernel preemption on kernel 2.4, a lot of real-time enhancements has been added including thread-context interrupt handling, preemptible mutex, priority-inheritance mutex, high-resolution timer, user-space real-time mutex. With the help of these efforts, Linux is becoming comparable with traditional RT OSes [11, 13].

Although many efforts have been made to enhance preemption latency, there are still lots of non-preemptible critical sections and interrupt off regions. Some of the major sources of these latencies are disk IO and network IO [11].

There are several clock sources on x86 architecture such as PIT, ACPI PM, HPET, TSC. The most reliable counter with highest priority is chosen on Linux kernel start-up. TSC is usually chosen for its highest resolution. One shortcoming of the TSC was its inability to adapt dynamic voltage scaling, however, it is solved by constant TSC.

We use a multi-core processor for better efficiency in terms of power usage. However, it has a drawback in deterministic timing. On SMP kernel, preemption latency increases as more processors are added, because they contend for shared interrupt-off region and/or preempt-off region. Affinitizing task and interrupt handling can reduce preemption latency to some extent [11]. To deal more with real-time, some robotic systems use multiple OSes and boards to separate real-time task and non real-time task, although it adds more complexity and power consumption to the system [8, 20].

Tuning real-time application is dependent on application model and often underlying hardware, so it requires a lot of experiments. We followed good real-time programming guides [4, 17, 19] and the experimental result will be presented.

This paper is organized as follows. Section 2 describes design and implementation of RCKS (Robot Control Kernel Subsystem). Section 3 presents real-time performance analysis of our robotic system. Section 4 addresses further tunings. Finally, Section 5 presents concluding remarks.

## 2 Design and Implementation of RCKS

Figure 2 shows software architecture of our robotic system which especially details in kernel components. At the bottom of the layer, modified Ethernet device driver communicates with NIC. It has been modified to fetch received packets without interrupt. As a controlling task is guaranteed to be executed periodically, interrupts from network device driver were considered redundant. EtherCAT master which is layered on top of NIC does EtherCAT protocol handling and monitoring slave device status [12]. ECCI (EtherCAT Control Interface) is implemented on top of EtherCAT master as an interface to user-space applications.

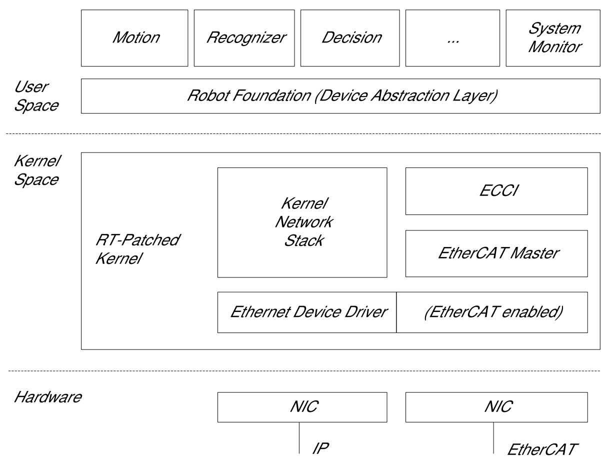



Figure 2: Software architecture of Robot Control Kernel Subsystem

We started from adopting EtherCAT master implementation. EtherLab is implemented in kernel space for two reasons. One is to avoid mode switching between kernel-space and user-space and the other is to communicate directly with network device driver. The driving application is also implemented as a kernel module [12]. Its design is optimized for performance, thereby suits for relatively small applications. However, our robot - as a humanoid robot - needs complex application logic and uses many user-space libraries that makes it inevitable to implement these in user-space. ECCI was

implemented to provide interface to user-space applications while keeping EtherCAT protocol handling intact in kernel-space. The interface includes configuring slaves, controlling actuators, reading sensor data and notifying slave status changes. It also presents a proc file system interface for exporting timing statistics.

In every cycle, ECCI receives one read-write request from the real-time task, Motion Controller. Asynchronously to this request, several non real-time tasks make read requests to ECCI. ECCI internally maintains cache of cyclic data obtained from EtherCAT master for efficiency and controls concurrent accesses from multiple tasks using a mutex which is enabled by FUTEX or PREEMPT_RT. To prevent long waiting of real-time task, ECCI employs RT-mutex [5]. RT-mutex supports priority inheritance and priority queuing which help our real-time task wait at most 1 non real-time task as shown in Figure 3.

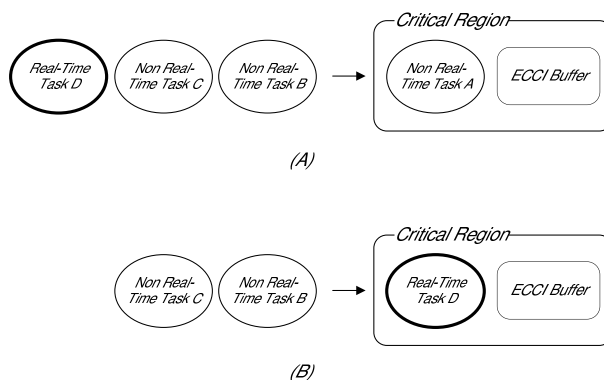



Figure 3: Concurrency control of ECCI buffer by RT-mutex: (A) Real-time task D arrives after non real-time task B and C. (B) Real-time task D acquires lock before non real-time task B and C.

Data flow in each cycle is depicted in Figure 4. The Motion Controller process sends commands which traverse through several layers to reach each actuator. Similarly, each sensor's data go through the layers to get to the Motion Controller. Data flow starts from the Motion Controller process. The process issues a read-write command which, in turn, fetches sensor data from the NIC's (Network Interface Card) buffer and composes and sends actuator commands. The sensor data have been ready at the NIC's buffer in previous cycle. The actuator commands are packetized in Ethernet frame which traverses through all slave devices. EtherCAT master returns immediately without waiting for the Ethernet frame and the thread of execution returns back to the Motion Controller. Now, the Motion Controller computes next cycle's motion plan and goes to sleep to keep steady control period. EtherCAT datagram which has been encapsulated in Ethernet packet is updated as it passes through the ESC (EtherCAT Slave Controller) on each slave device. The ESC generates interrupt to the micro controller which fetches new data from ESC's internal memory, controls actuators, gathers sensor data, and updates ESC's memory.

Sensor data take 1.5 to 2.5 cycles to reach to the Motion Controller depending on the time of occurrence. Command from the Motion Controller takes 1.5 cycles to be delivered to each actuator. Therefore it takes 3 to 4 cycles until our robot reacts to an external event—that is, 3 to 4 ms.

## 3 Real-Time Performance Analysis

The accuracy of the Motion Controller's control period depends on the accuracy of the sleep time in Figure 4. If the motion planning consumes reasonable amount of time, optimizing control period is essentially similar to optimizing the preemption latency of Linux kernel, and general real-time performance tunings can be applied.

### 3.1 General Real-Time Tunings

Linux kernel provides several tuning knobs for real-time applications. We applied some of the typical real-time tunings to achieve deterministic timing of the Motion Controller.

First, the Motion Controller process should have the highest real-time priority. It sleeps at the end of every cycle to keep constant control period which makes the task being moved from run queue to wait queue in Linux kernel. When the time expires it comes back to run queue. After that, when Linux scheduler examines the run queue, our Motion Controller should be on the highest priority run queue. Linux system call sched_setscheduler() provides this facility.

Second, dedicating one CPU for the Motion Controller is desirable. CPU shielding is a strategy in multi-processor system which dedicates one CPU to a real-time task and other CPUs to non real-time tasks and interrupt handlers. This is beneficial to the Motion Controller for two reasons. First, it prevents latencies caused by a non real-time task or interrupt handler. They

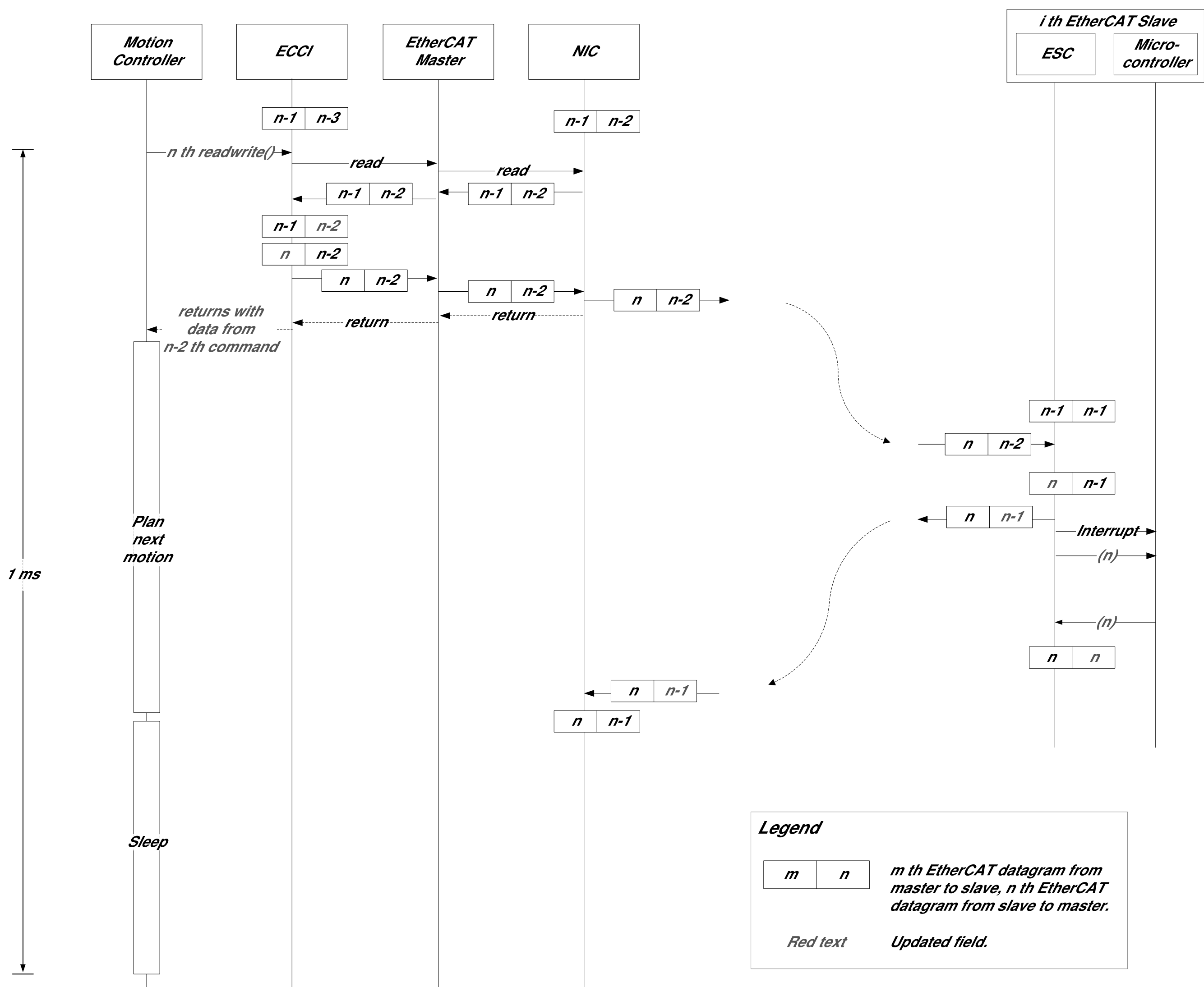



Figure 4: Data flow in a cycle. Data is exchanged through several layers.

may be in an interrupt-off and/or preemption-off region when the Motion Controller is about to be executed, thereby increasing latency. Second, high cache coherence - high coherence of instruction cache, data cache, and TLB (Translation Lookaside Buffer) - helps fast execution of the Motion Controller. Linux provides system call sched_setaffinity() for setting CPU affinity of a process. For interrupt affinity, proc file system interface /proc/irq/<irq_number>/smp_affinity and kernel API set_ioapic_affinity_irq() are provided. taskset is also a useful tool to get and set a process's CPU affinity from a shell. In addition, kernel can be configured to support CPUSETS which constrains the CPU and memory placement of tasks [1]. It is desirable to setup a resource management policy and enforce it using a global resource manager from which all child processes inherits the policy.

Last, the Motion Controller should not be paged-out to prevent high cost of fetching the page from swap area. Linux provides system call mlock() for locking the process's virtual address space into RAM.

### 3.2 Spinning nanosleep

RTLinux provides TIMER_ADVANCE option in its clock_nanosleep() API to enhance accuracy of sleep time. This mechanism wakes a task up before its deadline has arrived and puts it into a busy-wait loop until the deadline has arrived. This busy-wait loop improves latency for real-time task, but the process is in a busy-wait loop while waiting for the deadline [6, 10]. We

implemented this idea in Linux and applied to the Motion Controller. Since our robotic system dedicate one CPU core to the Motion Controller exclusively, spinning in the Motion Controller doesn't affect performance of other tasks. One shortcoming of this busy-waiting is increased power consumption in the core. However, this increase would be small enough compared to the overall power consumption of the robot where most power is consumed by the actuators of the joints. One word of caution is that spinning nanosleep should really sleep for some time or yield CPU to other higher or equal priority tasks before spinning on CPU. Otherwise it causes the starvation of other important kernel threads like watchdog, migration and timer thread which can lead to abnormal system behavior.

### 3.3 Experiment Planning

For the robot to move smoothly, it is important for the Motion Controller to send command to ECCI at the exact time of each cycle. We measured the time and generated statistics. The idea is similar to the latency analysis of cyclictest or realfeel [21, 7, 3]. Ideal period of between each consecutive time should be 1 ms; however, in practice, before the Motion Controller wakes up from sleep, the kernel may be in a critical section, which results in an additional wake-up delay of the Motion Controller.

To verify the effect of real-time tunings to the Motion Controller, we tested with all the combinations of the following options.

- Highest Priority
- CPU Shielding
- Memory Locking

In addition, spinning nanosleep is tested with all the above options turned on. Maximum spinning time is set to 50 us. When more sleep time is requested, it first nanosleep()s until 50 us remains and spins for the remaining time.

To ensure real-time performance of an operational system, it is advised to keep system load under 50% [16]. Nevertheless measuring performance under heavy load is important to observe worst-case performance. Figure 5 shows the test script which generates extremely high disk IO and network IO [14]. Figure 6 describes test environment.

- Linux kernel version: 2.6.26.8-rt16
- CPU: x86 2.4GHz Quad-core
- RAM: 2GBytes
- Robot slave devices: 59 sensors and actuators

Figure 6: Test Environment

### 3.4 Experimental Results

Figure 7 and 8 shows test results on unloaded system and on heavily loaded system respectively. Each combination of test was performed for 10 minutes.

Without any real-time tuning, measured maximum control period was 1,482 us on unloaded system and 346,148 us on heavily loaded system. The latter was intolerable in our robotic system.

With general real-time tunings applied—with maximum priority, memory locking and CPU shielding set—unloaded system showed average 1,017.60 us and maximum 1,044 us control period and heavily loaded system showed average 1,006.12 us and maximum 1,100 us control period.

Memory locking showed little improvement compared with other real-time tunings. It is assumed that the little improvement was due to the enough physical memory —which is 2GBytes—on our test environment which might not cause many page fault and paging-out.

Spinning nanosleep, in addition to general real-time tunings, showed best real-time performance. Control period was average 1,002.77 us and maximum 1,020 us on unloaded system and average 1,002.11 us and maximum 1,071 us on heavily loaded system which are satisfactory for smooth motion control of our robot.

Maximum values are highly unpredictable and may vary from experiment to experiment, which is because predicting longest kernel path—nested interrupt-off and preemption-off regions—is nearly impossible on heavily loaded system. For example, if the test duration is too short, the order of performance can differ from Figure 7 and 8, however when distribution is considered—like from low 99% range to low 99.999% range—we could conclude that the results are easily reproducible.

```
while true; do dd if=/dev/zero of=bigfile bs=1024000 count=1024; done &
while true; do killall hackbench; sleep 5; done &
while true; do $HACK_BENCH 20; done &
ping -l 100000 -s 10 -f localhost &
while true; do du -s / > /dev/null 2>&1 ; done &
```

Figure 5: Stress on Testing

## 4 Further Enhancements

Further fine-tunings are possible depending on system. In real-time systems, the ext2 file system is recommended if journaling is not required. Runlevel should be set to multi-user mode without the graphical interface to avoid additional load. Out-of-memory killer could be customized to select victim process which is least significant in terms of a robotic system. Proper tuning of sched_nr_migrate parameter is desired to limit the number of task that will move at a time [4, 17, 19].

Delayed locking technique can be applied to our system which execute a real-time task at a predefined interval [15]. This technique allows a non real-time task to enter a critical section only if the operation does not disturb the future execution of the real-time application.

## 5 Conclusions

Our robotic system needed a real-time OS for deterministic control of actuators and sensors, and, at the same time needed a general OS for running many processes simultaneously and using rich user-space libraries. Linux with complete in-kernel preemption patch was selected to meet the requirements and our robot system had the benefit of exploiting plentiful open-source software available in Linux.

Robot control kernel subsystem is implemented using EtherLab, an EtherCAT master implementation to control various sensors and actuators in real-time. Robot-specific application logic is implemented in user-space, while EtherCAT-specific protocol handling stays in kernel-space so the kernel can be robust from user-space bugs.

On our system, 1,000 us control period was met most of the time, with average 1,002.11 us and maximum 1,071 us on heavily loaded system which was good enough for smooth motion control. As our robotic system is still in development, we expect to get better real-time performance with further fine tunings.

## References


[1] CPUSETS. Linux kernel documentation: `kernel/Documentation/cpusets.txt`.

[2] EtherCAT Technical Introduction and Overview. `http://www.packagingdigest.com/contents/pdf/EtherCAT_Introduction_en.pd%f`.

[3] Linux Real Time Patch Review - Vanilla vs. RT patch comparison. `http://www.captain.at/howto-linux-real-time-patch.php`.

[4] Real-Time Linux Wiki. Project site: `http://rt.wiki.kernel.org`.

[5] RT-mutex subsystem with PI support. Linux kernel documentation: `kernel/Documentation/rt-mutex.txt`.

[6] RTLinuxPro CPU Reservation Technology. `http://www.linuxdevices.com/articles/AT7665542109.html`.

[7] Andrew Webber. Realfeel Test of the Preemptible Kernel Patch. `http://www.linuxjournal.com/article/6405`.

[8] Berthold Bäuml and Gerd Hirzinger. When hard realtime matters: Software for complex mechatronic systems. *Robotics and Autonomous Systems*, 56(1):5–13, 2008.

[9] Beckhoff. *Hardware Data Sheet ET1100 EtherCAT Slave Controller*, Jan 2008.

[10] Cort Dougan and Zwane Mwaikambo. Lies, Misdirection, and Real-Time Measurements. `http://www.ddj.com/cpp/184401780`.

[11] S. Dietrich and D. Walker. The evolution of real-time linux. In *Proceedings of Seventh Real-Time Linux Workshop*, Nov 2005.

[12] EtherLab. *IgH EtherCAT Master 1.4.0 Preliminary Documentation*, Feb 2009.

[13] Thomas Gleixner and Douglas Niehaus. Hrtimers and beyond: Transforming the linux time subsystems. In *Ottawa Linux Symposium*, 2006.

[14] Ingo Molnar. dohell script. Linux kernel mailing list: `http://lkml.org/lkml/2005/6/22/347`.

[15] Jupyung Lee and Kyu-Ho Park. Delayed locking technique for improving real-time performance of embedded linux by prediction of timer interrupt. In *Real Time and Embedded Technology and Applications Symposium, 2005. RTAS 2005. 11th IEEE*, pages 487–496, March 2005.

[16] Paul E McKenny. 'real time' vs 'real fast': How to choose? In *Ottawa Linux Symposium*, 2008.

[17] Montavista. *Real-time Application Programmer's Guide*, 2008.

[18] S. Potra and G. Sebestyen. Ethercat protocol implementation issues on an embedded linux platform. In *Automation, Quality and Testing, Robotics, 2006 IEEE International Conference on*, volume 1, pages 420–425, May 2006.

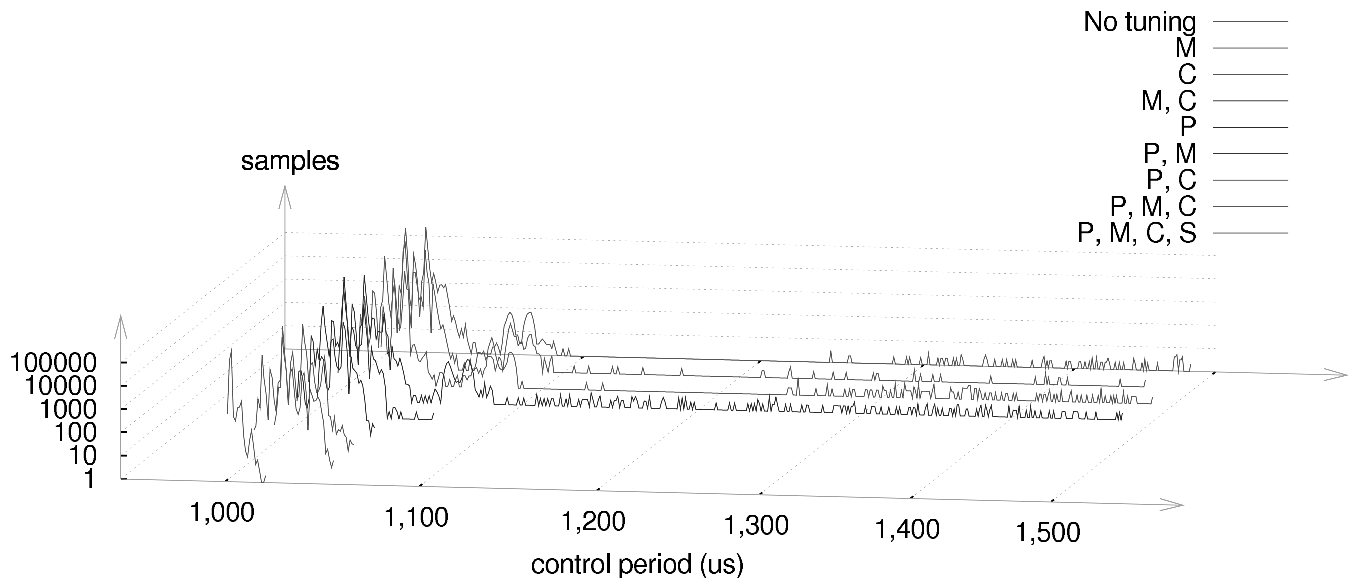
[19] Redhat. *Red Hat Enterprise MRG 1.1 Realtime Tuning Guide*, 2008.

[20] R. Tellez, F. Ferro, S. Garcia, E. Gomez, E. Jorge, D. Mora, D. Pinyol, J. Oliver, O. Torres, J. Velazquez, and D. Faconti. Reem-b: An autonomous lightweight human-size humanoid robot. In *Humanoid Robots, 2008. Humanoids 2008. 8th IEEE-RAS International Conference on*, pages 462–468, Dec. 2008.

[21] Thomas Gleixsner. Cyclictest. `http://rt.wiki.kernel.org/index.php/Cyclictest`.

| | | No Tuning | M | C | MC | P | PM | PC | PMC | PMCS |
|---|---|---|---|---|---|---|---|---|---|---|
| | min | 1,005.00 | 1,006.00 | 1,005.00 | 1,005.00 | 1,005.00 | 1,005.00 | 1,004.00 | 1,004.00 | 1,001.00 |
| Low 99% | max | 1,023.00 | 1,032.00 | 1,030.00 | 1,028.00 | 1,029.00 | 1,030.00 | 1,029.00 | 1,032.00 | 1,006.00 |
| | avg | 1,013.45 | 1,020.54 | 1,020.01 | 1,014.19 | 1,019.46 | 1,019.75 | 1,014.48 | 1,017.43 | 1,002.72 |
| | SD | 5.02 | 4.94 | 4.51 | 5.10 | 4.03 | 4.37 | 5.34 | 4.21 | 0.48 |
| Low 99.9% | max | 1,060.00 | 1,061.00 | 1,072.00 | 1,062.00 | 1,034.00 | 1,035.00 | 1,030.00 | 1,037.00 | 1,008.00 |
| | avg | 1,013.56 | 1,020.65 | 1,020.11 | 1,014.33 | 1,019.56 | 1,019.86 | 1,014.61 | 1,017.58 | 1,002.76 |
| | SD | 5.17 | 5.06 | 4.65 | 5.33 | 4.15 | 4.50 | 5.50 | 4.48 | 0.65 |
| Low 99.99% | max | 1,351.00 | 1,085.00 | 1,355.00 | 1,331.00 | 1,038.00 | 1,038.00 | 1,034.00 | 1,039.00 | 1,013.00 |
| | avg | 1,013.62 | 1,020.70 | 1,020.19 | 1,014.41 | 1,019.57 | 1,019.88 | 1,014.62 | 1,017.60 | 1,002.77 |
| | SD | 5.76 | 5.32 | 5.82 | 6.16 | 4.17 | 4.53 | 5.52 | 4.52 | 0.69 |
| Low 99.999% | max | 1,474.00 | 1,394.00 | 1,463.00 | 1,444.00 | 1,043.00 | 1,040.00 | 1,040.00 | 1,041.00 | 1,016.00 |
| | avg | 1,013.66 | 1,020.71 | 1,020.22 | 1,014.44 | 1,019.57 | 1,019.88 | 1,014.63 | 1,017.60 | 1,002.77 |
| | SD | 6.96 | 5.59 | 6.86 | 7.08 | 4.18 | 4.53 | 5.52 | 4.52 | 0.70 |
| 100% | max | 1,482.00 | 1,464.00 | 1,484.00 | 1,477.00 | 1,064.00 | 1,044.00 | 1,044.00 | 1,044.00 | 1,020.00 |
| | avg | 1,013.67 | 1,020.71 | 1,020.23 | 1,014.45 | 1,019.58 | 1,019.88 | 1,014.63 | 1,017.60 | 1,002.77 |
| | SD | 7.12 | 5.74 | 7.02 | 7.23 | 4.18 | 4.53 | 5.52 | 4.52 | 0.70 |

Maximum value and statistics of control periods on unloaded system. 100% row shows distribution of entire range, while other rows show data distribution without peak values. (Unit:us, P: maximum priority, M: memory locking, C: CPU shielding, S: spinning nanosleep)

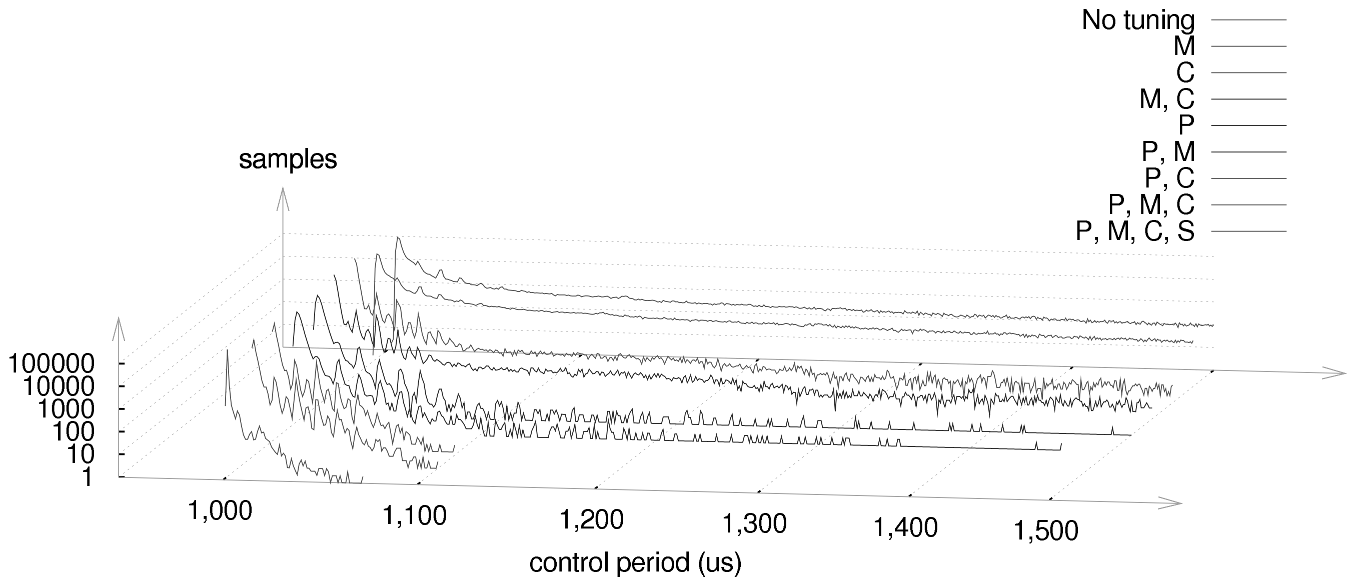



Distribution of control periods

Figure 7: Control periods on unloaded loaded system

| | | No Tuning | M | C | MC | P | PM | PC | PMC | PMCS |
|---|---|---|---|---|---|---|---|---|---|---|
| | min | 1,002.00 | 1,003.00 | 1,004.00 | 1,004.00 | 1,004.00 | 1,004.00 | 1,004.00 | 1,004.00 | 1,001.00 |
| Low 99% | max | 36,967.00 | 37,019.00 | 1,249.00 | 1,215.00 | 1,020.00 | 1,026.00 | 1,027.00 | 1,027.00 | 1,004.00 |
| | avg | 2,072.05 | 2,066.02 | 1,006.85 | 1,006.66 | 1,006.96 | 1,007.08 | 1,005.83 | 1,005.81 | 1,002.03 |
| | SD | 3,935.35 | 3,915.51 | 13.03 | 11.83 | 1.85 | 2.11 | 3.28 | 3.23 | 0.20 |
| Low 99.9% | max | 88,376.00 | 87,678.00 | 2,004.00 | 2,004.00 | 1,050.00 | 1,051.00 | 1,048.00 | 1,048.00 | 1,019.00 |
| | avg | 2,529.41 | 2,521.51 | 1,013.85 | 1,013.33 | 1,007.17 | 1,007.30 | 1,006.09 | 1,006.07 | 1,002.08 |
| | SD | 6,314.70 | 6,283.76 | 78.55 | 76.08 | 2.90 | 3.25 | 4.29 | 4.26 | 0.74 |
| Low 99.99% | max | 156,548.00 | 152,964.00 | 16,513.00 | 19,003.00 | 1,173.00 | 1,160.00 | 1,068.00 | 1,071.00 | 1,038.00 |
| | avg | 2,624.61 | 2,615.18 | 1,016.26 | 1,016.09 | 1,007.23 | 1,007.36 | 1,006.13 | 1,006.11 | 1,002.10 |
| | SD | 7,081.89 | 7,029.08 | 138.80 | 161.27 | 3.57 | 3.84 | 4.52 | 4.51 | 0.99 |
| Low 99.999% | max | 236,548.00 | 228,477.00 | 39,079.00 | 33,098.00 | 1,387.00 | 1,319.00 | 1,081.00 | 1,088.00 | 1,055.00 |
| | avg | 2,640.65 | 2,630.67 | 1,018.49 | 1,018.29 | 1,007.25 | 1,007.38 | 1,006.13 | 1,006.12 | 1,002.11 |
| | SD | 7,284.39 | 7,219.58 | 280.70 | 285.45 | 4.23 | 4.36 | 4.56 | 4.56 | 1.06 |
| 100% | max | 346,148.00 | 284,092.00 | 50,158.00 | 56,006.00 | 1,510.00 | 1,464.00 | 1,098.00 | 1,100.00 | 1,071.00 |
| | avg | 2,643.59 | 2,633.28 | 1,018.92 | 1,018.73 | 1,007.25 | 1,007.39 | 1,006.14 | 1,006.12 | 1,002.11 |
| | SD | 7,341.15 | 7,264.35 | 310.66 | 317.71 | 4.46 | 4.53 | 4.57 | 4.57 | 1.08 |

Maximum value and statistics of control periods on heavily loaded system. 100% row shows distribution of entire range, while other rows show data distribution without peak values. (Unit:us, P: maximum priority, M: memory locking, C: CPU shielding, S: spinning nanosleep)




Distribution of control periods (Values greater than 1,500 us were not depicted)

Figure 8: Control periods on heavily loaded system