Proceedings of the Linux Symposium

July 11th–13th, 2012
Ottawa, Ontario
Canada

Contents

Sockets and Beyond: Assessing the Source Code of Network Applications **7**
M. Komu, S. Varjonen, A. Gurtov, S. Tarkoma

Load-Balancing for Improving User Responsiveness on Multicore Embedded Systems **25**
Geunsik Lim, Changwoo Min, YoungIk Eom

Experiences with Power Management Enabling on the Intel Medfield Phone **35**
R. Muralidhar, H. Seshadri, V. Bhimarao, V. Rudramuni, I. Mansoor, S. Thomas, B. K. Veera, Y. Singh, S. Ramachandra

File Systems: More Cooperations - Less Integration. **47**
A. Depoutovitch & A. Warkentin

"Now if we could get a solution to the home directory dotfile hell!" **55**
A. Warkentin

Improving RAID1 Synchronization Performance Using File System Metadata **63**
H. Subramanian & A. Warkentin & A. Depoutovitch

Out of band Systems Management in enterprise Computing Environment **71**
D. Verma, S. Gowda, A. Vellimalai, S. Prabhakar

ClusterShell, a scalable execution framework for parallel tasks **77**
S. Thiell & A. Degrémont & H. Doreau & A. Cedeyn

DEXT3: Block Level Inline Deduplication for EXT3 File System **87**
A. More, Z. Shaikh, V. Salve

ARMvisor: System Virtualization for ARM **93**
J-H. Ding, C-J. Lin, P-H. Chang, C-H. Tsang, W-C. Hsu, Y-C. Chung

Clustering the Kernel **109**
A. Lissy & J. Parpaillon & P. Martineau

Non-scalable locks are dangerous **119**
Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich

Fine Grained Linux I/O Subsystem Enhancements to Harness Solid State Storage **131**
S. Brahmaroutu, R. Patel, H. Rajagopalan, S. Vidyadhara, A. Vellimalai

Optimizing eCryptfs for better performance and security **137**
Li Wang, Y. Wen, J. Kong, X. Yi

Android SDK under Linux **145**
Jean-Francois Messier

Sockets and Beyond: Assessing the Source Code of Network Applications

Miika Komu
Aalto University, Department of Computer Science and Engineering
`miika@iki.fi`

Samu Varjonen, Andrei Gurtov, Sasu Tarkoma
University of Helsinki and Helsinki Institute for Information Technology
`firstname.lastname@hiit.fi`

Abstract

Network applications are typically developed with frameworks that hide the details of low-level networking. The motivation is to allow developers to focus on application-specific logic rather than low-level mechanics of networking, such as name resolution, reliability, asynchronous processing and quality of service. In this article, we characterize statistically how open-source applications use the Sockets API and identify a number of requirements for network applications based on our analysis. The analysis considers five fundamental questions: naming with end-host identifiers, name resolution, multiple end-host identifiers, multiple transport protocols and security. We discuss the significance of these findings for network application frameworks and their development. As two of our key contributions, we present generic solutions for a problem with OpenSSL initialization in C-based applications and a multihoming issue with UDP in all of the analyzed four frameworks.

1 Introduction

The Sockets API is the basis for all internet applications. While the number of applications using it directly is large, some applications use it indirectly through intermediate libraries or frameworks to hide the intricacies of the low-level Sockets API. Nevertheless, the intermediaries still have to interface with the Sockets API. Thus, the Sockets API is important for all network applications either directly or indirectly but has been studied little. To fill in this gap, we have statistically analyzed the usage of Sockets API to characterize how contemporary network applications behave in Ubuntu Linux. In addition to merely characterizing the trends, we have also investigated certain programming pitfalls pertaining the Sockets API.

As a result, we report ten main findings and how they impact a number of relatively new sockets API extensions. To mention few examples, the poor adoption of a new DNS look up function slows down the migration path for the extensions dependent on it, such as the APIs for IPv6 source address selection and HIP. OpenSSL library is initialized incorrectly in many applications, causing potential security vulnerabilities. The management of the dual use of TCP/UDP transports and the dual use of the two IP address families creates redundant complexity in applications.

To escape the unnecessary complexity of the Sockets API, some applications utilize network application frameworks. However, the frameworks are themselves based on the Sockets API and, therefore, subject to the same scrutiny as applications using the Sockets API. For this reason, it is natural to extend the analysis for frameworks.

We chose four example frameworks based on the Sockets API and analyzed them manually in the light of the Sockets API findings. Since frameworks can offer high-level abstractions that do not have to mimic the Sockets API layout, we organized the analysis of the frameworks in a top-down fashion and along generalized dimensions of end-host naming, multiplicity of names and transports, name look up and security. As a highlight of the framework analysis, we discovered a persistent problem with multiplicity of names in all of the four frameworks. To be more precise, the problem was related to multihoming with UDP.

In this article, we describe how to solve some of the discovered issues in applications and frameworks using the Sockets API. We also characterize some of the inherent limitations of the Sockets API, for instance, related to complexity.

2 Background

In this section, we first introduce the parts of the Berkeley Sockets and the POSIX APIs that are required to understand the results described in this article. Then, we briefly introduce four network application frameworks built on top of the two APIs.

2.1 The Sockets API

The Sockets API is the de-facto API for network programming due to its availability for various operating systems and languages. As the API is rather low level and does not support object-oriented languages well, many networking libraries and frameworks offer additional higher-level abstractions to hide the details of the Sockets API.

Unix-based systems typically provide an abstraction of all network, storage and other devices to the applications. The abstraction is realized with *descriptors* which are also sometimes called *handles*. The descriptors are either file or socket descriptors. Both of them have different, specialized accessor functions even though socket descriptors can be operated with some of the file-oriented functions.

When a socket descriptor is created with the `socket()` function, the transport protocol has to be fixed for the socket. In practice, `SOCK_STREAM` constant fixes the transport protocol to TCP and `SOCK_DGRAM` constant to UDP. For IPv4-based communications, an application uses a constant called `AF_INET`, or its alias `PF_INET`, to create an IPv4-based socket. For IPv6, the application uses correspondingly `AF_INET6` or `PF_INET6`.

2.1.1 Name Resolution

An application can look up names from DNS by calling `gethostbyname()` or `gethostbyaddr()` functions. The former looks up the host information from the DNS by its symbolic name (forward look up) and the latter by its numeric name, i.e., IP address (reverse look up). While both of these functions support IPv6, they are obsolete and their modern replacements are the `getnameinfo()` and `getaddrinfo()` functions.

2.1.2 Delivery of Application Data

A client-side application can start sending data immediately after creation of the socket; however, the application typically calls the `connect()` function to associate the socket with a certain destination address and port. The `connect()` call also triggers the TCP handshake for sockets of `SOCK_STREAM` type. Then, the networking stack automatically associates a source address and port with the socket if the application did not choose them explicitly with the `bind()` function. Finally, a `close()` call terminates the socket gracefully and, when the type of the socket is `SOCK_STREAM`, the call also initiates the shutdown procedure for TCP.

Before a server-oriented application can receive incoming datagrams, it has to call a few functions. Minimally with UDP, the application has to define the port number and IP address to listen to by using `bind()`. Typically, TCP-based services supporting multiple simultaneous clients prepare the socket with a call to the `listen()` function for the following `accept()` call. By default, the `accept()` call blocks the application until a TCP connection arrives. The function then "peels off" a new socket descriptor from existing one that separates the particular connection with the client from others.

A constant `INADDR_ANY` is used with `bind()` to listen for incoming datagrams on all network interfaces and addresses of the local host. This wildcard address is typically employed in server-side applications.

An application can deliver and retrieve data from the transport layer in multiple alternative ways. For instance, the `write()` and `read()` functions are file-oriented functions but can also be used with socket descriptors to send and receive data. For these two file-oriented functions, the Sockets API defines its own specialized functions.

For datagram-oriented networking with UDP, the `sendto()` and the `recvfrom()` functions can be used. Complementary functions `sendmsg()` and `recvmsg()` offer more advanced interfaces for applications [19]. They operate on scatter arrays (multiple non-consecutive I/O buffers instead of just one) and also support so-called ancillary data that refers to meta-data and information related to network packet headers.

In addition to providing the rudimentary service of sending and receiving application data, the socket calls also implement access control. The `bind()` and `connect()` limit ingress (but not egress) network access to the socket by setting the allowed local and remote destination end point. Similarly, the `accept()` call effectively constrains remote access to the newly created socket by allowing communications only with the particular client. Functions `send()` and `recv()` are typically used for connection-oriented networking, but can also be used with UDP to limit remote access.

2.1.3 Customizing Networking Stack

The Sockets API provides certain default settings for applications to interact with the transport layer. The settings can be altered in multiple different ways.

With "raw" sockets, a process can basically create its own transport-layer protocol or modify the network-level headers. A privileged process creates a raw socket with constant `SOCK_RAW`.

A more constrained way to alter the default behavior of the networking stack is to set socket options with `setsockopt()`. As an example of the options, the `SO_REUSEADDR` socket option can be used to disable the default "grace period" of a locally reserved transport-layer port. By default, consecutive calls to `bind()` with the same port fail until the grace period has passed. Especially during the development of a networking service, this grace period is usually disabled for convenience because the developed service may have to be restarted quite often for testing purposes.

2.2 Sockets API Extensions

Basic Socket Interface Extensions for IPv6 [5] define additional data structures and constants, including `AF_INET` and `sockaddr_in6`. The extensions also define new DNS resolver functions, `getnameinfo()` and `getaddrinfo()`, as the old ones, `gethostbyname()` and `gethostbyaddr()`, are now obsoleted. The older ones are not thread safe and offer too little control over the resolved addresses. The specification also defines IPv6-mapped IPv4 addresses to improve IPv6 interoperability.

An IPv6 application can typically face a choice of multiple source and destination IPv6 pairs to choose from. Picking a pair may not be a simple task because some of the pairs may not even result in a working connectivity. IPv6 Socket API for Source Address Selection [13] defines extensions that restrict the local or remote address to a certain type, for instance, public or temporary IPv6 addresses. The extensions include new socket options to restrict the selection local addresses when, e.g., a client application connects without specifying the source address. For remote address selection, new flags for the `getaddrinfo()` resolver are proposed. The extensions mainly affect client-side connectivity but can affect also at the server side when UDP is being used.

The Datagram Congestion Control Protocol (DCCP) is similar to TCP but does not guarantee in-order delivery. An application can use it - with minor changes - by using `SOCK_DCCP` constant when a socket is created.

Multihoming is becoming interesting because most of the modern handhelds are equipped with, e.g., 3G and WLAN interfaces. In the scope of this work, we associate "multihoming" to hosts with multiple IP addresses typically introduced by multiple network interfaces. Multihoming could be further be further characterized whether it occurs in the initial phases of the connectivity or during established communications. All of the statistics in this article refer to the former case because the latter requires typically some extra logic in the application or additional support from the lower layers.

When written correctly, UDP-based applications can support multihoming for initial connectivity and the success of this capability is investigated in detail in this article. However, supporting multihoming in TCP-based applications is more difficult to achieve and requires additional extensions. A solution at the application layer is to recreate connections when they are rendered broken. At the transport layer, Multipath TCP [4] is a TCP-specific solution to support multihoming in a way that is compatible with legacy applications with optional APIs for native applications [16].

The Stream Control Transmission Protocol (SCTP, [21]) implements an entirely new transport protocol with full multihoming capabilities. In a nutshell, SCTP offers a reliable, congestion-aware, message-oriented, in-sequence transport protocol. The minimum requirement to enable SCTP in an existing application is to change the protocol type in `socket()` call to SCTP. However, the application can only fully harness the benefits of the protocol by utilizing the `sendmsg()` and `recvmsg()` interface. Also, the protocol supports sharing of a single socket descriptor for multiple simultaneous communication partners; this requires some additional logic in the application.

Transport-independent solutions operating at the lower layers include Host Identity Protocol [11] and Site Multihoming by IPv6 Intermediation (SHIM6) [12]. In brief, HIP offers support for end-host mobility, multihoming and NAT traversal. By contrast, SHIM6 is mainly a multihoming solution. From the API perspective, SHIM6 offers backwards compatible identifiers for IPv6—in the sense that they are routable at the network layer—whereas the identifiers in HIP are non-routable. HIP has its own optional APIs for HIP-aware applications [9] but both protocols share the same optional multihoming APIs [8].

Name-based Sockets are a work-in-progress at the IETF standardization forum. While the details of the specification [23] are rather immature and the specification still lacks official consent of the IETF, the main idea is to provide extensions to the Sockets API that replace IP addresses with DNS-based names. In this way, the responsibility for the management of IP addresses is pushed down in the stack, away from the application layer.

2.3 NAT Traversal

Private address realms [18] were essentially introduced by NATs, but Virtual Private Networks (VPNs) and other tunneling solutions can also make use of private addresses. Originally, the concept of virtual address spaces was created to alleviate the depletion of the IPv4 address space, perhaps, because it appeared that most client hosts did not need publicly-reachable addresses. Consequently, NATs also offer some security as a side effect to the client side because they discard new incoming data flows by default.

To work around NATs, Teredo [7] offers NAT traversal solution based on a transparent tunnel to the applications. The protocol tries to penetrate through NAT boxes to establish a direct end-to-end tunnel but can resort to triangular routing through a proxy in the case of an unsuccessful penetration.

2.4 Transport Layer Security

Transport Layer Security (TLS) [22] is a cryptographic protocol that can be used to protect communications above the transport layer. TLS, and its predecessor Secure Socket Layer (SSL), are the most common way to protect TCP-based communications over the Internet.

In order to use SSL or TLS, a C/C++ application is usually linked to a library implementation such as OpenSSL or GNU TLS. The application then calls the APIs of the TLS/SSL-library instead of using the APIs of the Sockets API. The functions of the library are wrappers around the Sockets API, and are responsible for securing the data inside the TCP stream.

2.5 Network Frameworks

The Sockets API could be characterized as somewhat complicated and error-prone to be programmed directly. It is also “flat” by its nature because it was not designed to accommodate object-oriented languages. For these reasons, a number of libraries and frameworks have been built to hide the details of the Sockets API and to introduce object-oriented interfaces. The Adaptive Communication (ACE) [17] is one such framework.

ACE simplifies the development of networking applications because it offers abstracted APIs based on network software patterns observed in well-written software. Among other things, ACE includes network patterns related to connection establishment and service initialization in addition to facilitating concurrent software and distributed communication services. It supports asynchronous communications by inversion of control, i.e., the framework takes over the control of the program flow and it invokes registered functions of the application when needed.

Boost::Asio is another open source C++ library that offers high-level networking APIs to simplify development of networking applications. Boost::Asio aims to be portable, scalable, and efficient but, most of all, it provides a starting point for implementing further abstraction. Several Boost C++ libraries have already been included in the C++ Technical Report 1 and in C++11. In 2006 a networking proposal based on Asio was submitted to request inclusion in the upcoming Technical Report 2.

Java provides an object-oriented framework for the creation and use of sockets. *Java.net* package (called Java.net from here on) supports TCP (`Socket` class) and UDP (`Datagram` class). These classes implement communication over an IP network.

Twisted is a modular, high-level networking framework for python. Similarly to ACE, Twisted is also based on inversion of control and asynchronous messaging. Twisted has built-in support for multiple application-layer protocols, including IRC, SSH and HTTP. What distinguishes Twisted from the other frameworks is the focus on service-level functionality based adaptable functionality that can be run on top of several application-layer protocols.

3 Materials and Methods

We collected information related to the use of Sockets API usage in open-source applications. In this article, we refer to this information as *indicators*. An indicator refers to a constant, structure or function of the C language. We analyzed the source code for indicators in a static way (based on keywords) rather than dynamically.[1] The collected set of indicators was limited to networking-related keywords obtained from the keyword indexes of two books [20, 15].

We gathered the material for our analysis from all of the released Long-Term Support (LTS) releases of Ubuntu: Dapper Drake 6.06, Hardy Heron 8.04, Lucid Lynx 10.04. Table 1 summarizes the number of software packages gathered per release. In the table, "patched" row expresses how many applications were patched by Ubuntu.

We used sections "main", "multiverse", "universe" and "security" from Ubuntu. The material was gathered on Monday 7th of March 2011 and was constrained to software written using the C language. Since our study was confined to networking applications, we selected only software in the categories of "net", "news", "comm", "mail", and "web" (in Lucid, the last category was renamed "httpd").

We did not limit or favor the set of applications, e.g., based on any popularity metrics. We believed that an application was of at least of some interest if the application was being maintained by someone in Ubuntu. To be more useful for the community, we analyzed all network applications and did not discriminate some "unpopular" minorities. This way, we did not have to choose between different definitions of popularity—perhaps Ubuntu popularity contest would have served as a decent metric for popularity. We did perform an outlier analysis in which we compared the whole set of applications to the most popular applications (100 or more installations). We discovered that the statistical "footprint" of the popular applications is different from the whole. However, the details are omitted because this contradicted with our goals.

[1] Authors believe that a more dynamic or structural analysis would not have revealed any important information on the issues investigated

	Dapper	Hardy	Lucid
Total	1,355	1,472	1,147
Patched	1,222	1,360	979
C	**721**	**756**	**710**
C++	57	77	88
Python	126	148	98
Ruby	19	27	13
Java	9	10	8
Other	423	454	232

Table 1: Number of packages per release version.

In our study, we concentrated on the POSIX networking APIs and Berkeley Sockets API because they form the de-facto, low-level API for all networking applications. However, we extended the API analysis to OpenSSL to study the use of security as well. All of these three APIs have bindings for high-level languages, such as Java and Python, and can be indirectly used from network application frameworks and libraries. As the API bindings used in other languages differs from those used in C language, we excluded other languages from this study.

From the data gathered,[2] we calculated sums and means of the occurrences of each indicator. Then we also calculated a separate "reference" number. This latter was formed by introducing a binary value to denote whether a software package used a particular indicator (1) or not (0), independent of the number of occurrences. The reference number for a specific indicator was collected from all software packages, and these reference numbers were then summed and divided by the number of packages to obtain a *reference ratio*. In other words, the reference ratio describes the extent of an API indicator

[2] `http://www.cs.helsinki.fi/u/sklvarjo/LS12/`

with one normalized score.

We admit that the reference number is a very coarse grained metric; it indicates capability rather than 100% guarantee that the application will use a specific indicator for all its runs. However, its binary (or "flattened") nature has one particular benefit that cancels out an unwanted side effect of the static code analysis, but this is perhaps easiest to describe by example. Let us consider an application where memory allocations and deallocations can be implemented in various ways. The application can call `malloc()` a hundred times but then calls `free()` only once. Merely looking at the volumes of calls would give a wrong impression about memory leaks because the application could have a wrapper function for `free()` that is called a hundred times. In contrast, a reference number of 1 for `malloc()` and 0 for `free()` indicates that the application has definitely one or more memory leak. Correspondingly, the reference ratio describes this for the entire population of the applications.

In our results, we show also reference ratios of combined indicators that were calculated by taking an union or intersection of indicators, depending on the use case. With combined indicators, we used tightly coupled indicators that make sense in the context of each other.

4 Results and Analysis

In this section, we show the most relevant statistical results. We focus on the findings where there is room for improvement or that are relevant to the presented Sockets API extensions. Then, we highlight the most significant patterns or key improvements for the networking applications. Finally, we derive a set of more generic requirements from the key improvements and see how they are met in four different network application frameworks.

4.1 Core Sockets API

In this section, we characterize how applications use the "core" Sockets API. Similarly as in the background, the topics are organized into sections on IPv6, DNS, transport protocols and customization of the networking stack. In the last section, we describe a multihoming issue related to UDP.

In the results, the reference ratios of indicators are usually shown inside brackets. All numeric values are from Ubuntu Lucid unless otherwise mentioned. Figure 1 illustrates some of the most frequent function indicators by their reference ratio and the following sections analyze the most interesting cases in more detail.

4.1.1 IPv6

According to the usage of AF and PF constants, 39.3% were IPv4-only applications, 0.3% IPv6-only, 26.9% hybrid and 33.5% did not reference either of the constants. To recap, while the absolute use of IPv6 was not high, the relative proportion of hybrid applications supporting both protocols was quite high.

4.1.2 Name Resolution

The obsolete DNS name-look-up functions were referenced more than their modern replacements. The obsolete forward look-up function `gethostbyname()` was referenced roughly twice as often as its modern replacement `getaddrinfo()`. Two possible explanations for this are that either that the developers have, for some reason, preferred the obsolete functions, or have neglected to modernize their software.

4.1.3 Packet Transport

Connection and datagram-oriented APIs were roughly as popular. Based on the usage of `SOCK_STREAM` and `SOCK_DGRAM` constants, we accounted for 25.1% TCP-only and 11.0% UDP-only applications. Hybrid applications supporting both protocols accounted for 26.3%, leaving 37.6% of the applications that used neither of the constants. By combining the hybrids with TCP-only applications, the proportion of applications supporting TCP is 51.4% and, correspondingly, 37.3% for UDP. It should not be forgotten that typically all network applications implicitly access DNS over UDP by default.

4.1.4 Customizing Networking Stack

While the Sockets API provides transport-layer abstractions with certain system-level defaults, many applications preferred to customize the networking stack or to

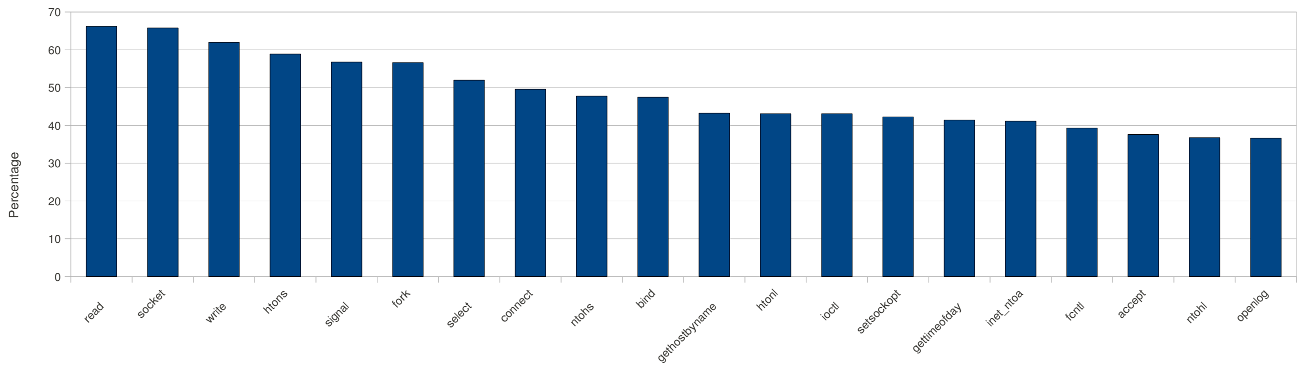


Figure 1: The most frequent functions in Ubuntu Lucid

override some of the parameters. The combined reference ratio of `SOCK_RAW`, `setsockopt()`, `pcap_pkthdr` and `ipq_create_handle()` indicators was 51.4%. In other words, the default abstraction or settings of the Sockets API are not sufficient for the majority of the applications.

It is worth mentioning that we conducted a brute-force search to find frequently occurring socket options sets. As a result, we did not find any recurring sets but merely individual socket options that were popular.

4.1.5 Multihoming and UDP

In this section, we discuss a practical issue related to UDP-based multihoming, but one which could be fixed in most applications by the correct use of `SO_BINDTODEVICE` (2.3%) socket option. The issue affects UDP-based applications accepting incoming connections from multiple interfaces or addresses.

On Linux, we have reason to believe that many UDP-based applications may not handle multihoming properly for initial connections. The multihoming problem for UDP manifests itself only when a client-side application uses a server address that does not match with the default route at the server. The root of the problem lies in egress datagram processing at the server side.

The UDP problem occurs when the client sends a "request" message to the server and the server does not send a "response" using the exact same address pair that was used for the request. Instead, the sloppy server implementation responds to the client without specifying the source address, and the networking stack invariably chooses always the wrong source address - meaning that the client drops the response as it appears to be arriving from a previously unknown IP address.

A straightforward fix is to modify the server-side processing of the software to respect the original IP address, and thus to prevent the network stack from routing the packet incorrectly. In other words, when the server-side application receives a request, it should remember the local address of the received datagram and use it explicitly for sending the response.

Explicit source addressing can be realized by using the modern `sendmsg()` interface. However, a poorly documented alternative to be used especially with the `sendto()` function is the socket option called `SO_BINDTODEVICE`. The socket option is necessary because `bind()` can only be used to specify the local address for the ingress direction (and not the egress).

We discovered the UDP problem by accident with iperf, nc and nc6 software. We have offered fixes to maintainers of these three pieces of software. Nevertheless, the impact of the problem may be larger as a third of the software in our statistics supports UDP explicitly. To be more precise, the lack of `SO_BINDTODEVICE` usage affects 45.7% (as an upper bound) of the UDP-capable software, which accounts for a total of 121 applications. This figure was calculated by finding the intersection of all applications not using `sendmsg()` and `SO_BINDTODEVICE`, albeit still using `sendto()` and `SOCK_DGRAM`. We then divided this by the number of applications using `SOCK_DGRAM`.

4.2 Sockets API Extensions

In this section, we show and analyze statistics on SSL and the adoption of a number of Sockets API extensions.

4.2.1 Security: SSL/TLS Extensions

Roughly 10.9% of the software in the data set used OpenSSL and 2.1% GNU TLS. In this section, we limit the analysis on OpenSSL because it is more popular. Unless separately mentioned, we will, for convenience, use the term SSL to refer both TLS and SSL protocols. We only present reference ratios relative to the applications using OpenSSL because this is more meaningful from the viewpoint of the analysis. In other words, the percentages account only the 77 OpenSSL-capable applications and not the whole set of applications.

The applications using OpenSSL consisted of both client and server software. The majority of the applications using OpenSSL (54%) consisted of email, news and messaging software. The minority included network security and diagnostic, proxy, gateway, http and ftp server, web browsing, printing and database software.

The reference ratios of SSL options remained roughly the same throughout the various Ubuntu releases. The use of SSL options in Ubuntu Lucid is illustrated in Figure 2.

The use of `SSL_get_verify_result()` function (37.7%) indicates that a substantial proportion of SSL-capable software has interest in obtaining the results of the certificate verification. The `SSL_get_peer_certificate()` function (64.9%) is used to obtain the certificate sent by the peer.

The use of the `SSL_CTX_use_privatekey_file()` function (62.3%) implies that a majority of the software is capable of using private keys stored in files. A third (27.3%) of the applications use the `SSL_get_current_cipher()` function to request information about the cipher used for the current session.

The `SSL_accept()` function (41.6%) is the SSL equivalent for `accept()`. The reference ratio of `SSL_connect()` function (76.6%), an SSL equivalent for `connect()`, is higher than for `ssl_accept()` (41.6%). This implies that the data set includes more client-based applications than server-based. Furthermore, we observed that `SSL_shutdown()` (63.6%) is referenced in only about half of the software that also references `SSL_connect()`, indicating that clients leave dangling connections with servers (possibly due to sloppy coding practices).

We noticed that only 71.4% of the SSL-capable software initialized the OpenSSL library correctly. The correct procedure for a typical SSL application is that it should initialize the library with `SSL_library_init()` function (71.4%) and provide readable error strings with `SSL_load_error_strings()` function (89.6%) before any SSL action takes place. However, 10.4% of the SSL-capable software fails to provide adequate error handling.

Only 58.4% of the SSL-capable applications seed the Pseudo Random Number Generator (PRNG) with `RAND_load_file()` (24.7%), `RAND_add()` (6.5%) or `RAND_seed()` (37.7%). This is surprising because incorrect seeding of the PRNG is considered a common security pitfall.

Roughly half of the SSL-capable software set the context options for SSL with `SSL_CTX_set_options` (53.3%); this modifies the default behavior of the SSL implementation. The option `SSL_OP_ALL` (37.7%) enables all bug fixes.

`SSL_OP_NO_SSLV2` option (31.2%) turns off SSLv2 and respectively `SSL_OP_NO_SSLV3` (13.0%) turns off the support for SSLv3. The two options were usually combined so that the application would just use TLSv1.

`SSL_OP_SINGLE_DH_USE` (7.8%) forces the implementation to re-compute the private part of the Diffie-Hellman key exchange for each new connection. With the exception of low-performance CPUs, it is usually recommended that this option to be turned on since it improves security.

The option `SSL_OP_DONT_INSERT_EMPTY_FRAGMENTS` (6.5%) disables protection against an attack on the block-chaining ciphers. The countermeasure is disabled because some of the SSLv3 and TLSv1 implementations are unable to handle it properly.

37.7% of the SSL-capable software prefers to use only TLSv1 (`TLSv1_client_method()`) and 20.1% of the SSL-capable software prefers to fall back from TLSv1 to SSLv3 when the server does not support TLSv1. However, the use of `SSL_OP_NO_TLSV1` option indicates that 7% of the software is able to turn off TLSv1 support completely. `SSL_OP_CIPHER_SERVER_PREFERENCE` is used to indicate that the server's preference in the choosing of the cipher takes precedence. `SSL_OP_NO_SESSION_RESUMPTION_RENEGOTIATION` indicates the

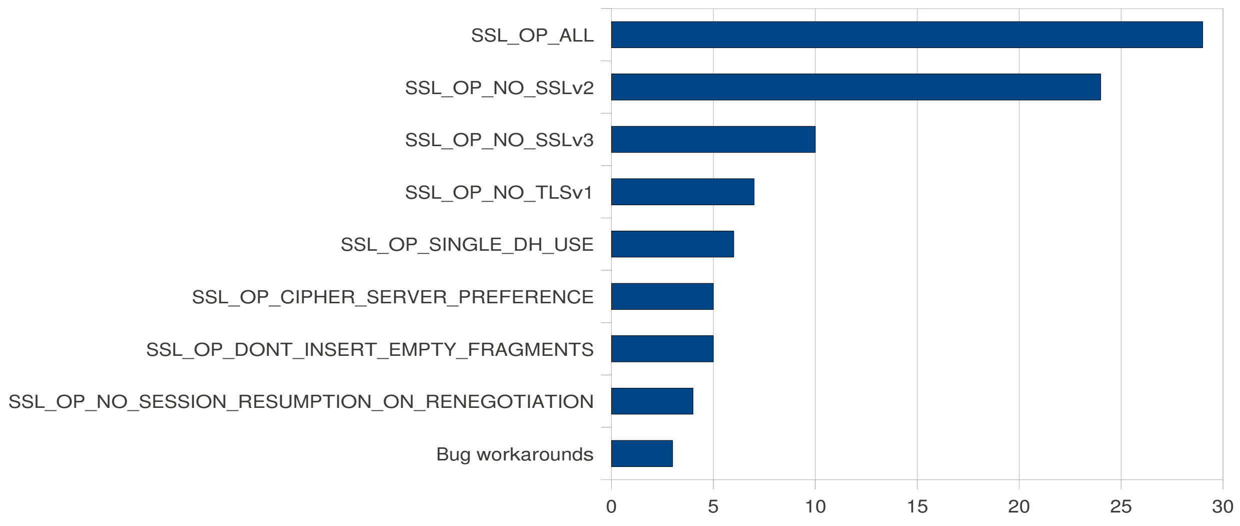


Figure 2: The number of occurrences of the most common SSL options

need for increased security as session resumption is disallowed and a full handshake is always required. The remaining options are workarounds for various bugs.

As a summary of the SSL results, it appears that SSL-capable applications are interested of the details of the security configuration. However, some applications initialize OpenSSL incorrectly and also trade security for backwards compatibility.

4.2.2 IPv6-Related Extensions

During the long transition to IPv6, we believe that the simultaneous co-existence of IPv4 and IPv6 still represents problems for application developers. For example, IPv6 connectivity is still not guaranteed to work everywhere. At the client side, this first appears as a problem with DNS look-ups if they are operating on top of IPv6. Therefore, some applications may try to look up simultaneously over IPv4 and IPv6 [25]. After this, the application may even try to call `connect()` simultaneously over IPv4 and IPv6. While these approaches can decrease the initial latency, they also generate some additional traffic to the Internet and certainly complicate networking logic in the application.

At the server side, the applications also have to maintain two sockets: one for IPv4 and another for IPv6. We believe this unnecessarily complicates the network processing logic of applications and can be abstracted away by utilizing network-application frameworks.

An immediate solution to the concerns regarding address duplication is proposed in RFC4291 [6], which describes IPv6-mapped IPv4 addresses. The idea is to embed IPv4 addresses in IPv6 address structures and thus to provide a unified data structure format for storing addresses in the application.

Mapped addresses can be employed either manually or by the use of `AI_V4MAPPED` flag for the `getaddrinfo()` resolver. However, the application first has to explicitly enable the `IPV6_V6ONLY` socket option (0.1%) before the networking stack will allow the IPv6-based socket to be used for IPv4 networking. By default, IPv4 connectivity with IPv6 sockets is disallowed in Linux because they introduce security risks [10]. As a bad omen, of the total six applications referencing the `AI_V4MAPPED` flag, only one of them set the socket option as safe guard.

The constants introduced by the IPv6 Socket API for Source Address Selection [13] are available in Ubuntu Lucid even though the support is incomplete. The flags to extend the `getaddrinfo()` resolver and the proposed auxiliary functions remain unavailable and only source address selection through socket options is available. Nevertheless, we calculated the proportion of IPv6-capable client-side applications that explicitly choose a source address. As an upper bound, 66.9% percent applications choose source addresses explicitly based the dual use of `connect()` and `bind()`. This means that a majority of IPv6 applications might be potentially interested of the extensions for IPv6 Socket API for Source Address Selection.

4.2.3 Other Protocol Extensions

The use of SCTP was very minimal in our set of applications and only three applications used SCTP. *Netperf* is a software used for benchmarking the network performance of various protocols. *Openser* is a flexible SIP proxy server. Linux Kernel SCTP tools (*lksctp-tools)* can be used for testing SCTP functionality in the userspace.

As with SCTP, DCCP was also very unpopular. It was referenced only from a single software package, despite it being easier to embed in an application by merely using the `SOCK_DCCP` constant in the socket creation.

As described earlier, multipath TCP, HIP and SHIM6 have optional native APIs. The protocols can be used transparently by legacy applications. This might boost their deployment when compared with the mandatory changes in applications for SCTP and DCCP.

The APIs for HIP-aware applications [9] may also face a similar slow adoption path because the APIs require a new domain type for sockets in the Linux kernel. While `getaddrinfo()` function can conveniently look up “wildcard” domain types, the success of this new DNS resolver (23.5%) is still challenged by the deprecated `gethostbyname()` (43.3%). SHIM6 does not face the same problem as it works without any changes to the resolver and connections can be transparently “upgraded” to SHIM6 during the communications.

The shared multihoming API for HIP- and SHIM6-aware applications [8] may have a smoother migration path. The API relies heavily on socket options and little on ancillary options. This strikes a good balance because `setsockopt()` is familiar to application developers (42.8%) and `sendmsg()` / `recvmsg()` with its ancillary option is not embraced by many (7%). The same applies to the API for Multipath TCP [16] that consists solely of socket options.

4.2.4 A Summary of the Sockets API Findings and Their Implications

Table 2 highlights ten of the most important findings in the Sockets APIs. Next, we go through each of them and argue their implications to the development of network applications.

Core Sockets API		
1	IPv4-IPv6 hybrids	26.9%
2	TCP-UDP hybrids	26.3%
3	Obsolete DNS resolver	43.3%
4	UDP-based apps with multihoming issue	45.7%
5	Customize networking stack	51.4%
OpenSSL-based applications		
6	Fails to initialize correctly	28.6%
7	Modifies default behavior	53.3%
8	OpenSSL-capable applications in total	10.9%
Estimations on IPv6-related extensions		
9	Potential misuse with mapped addresses	83.3%
10	Explicit IPv6 Source address selection	66.9%

Table 2: Highlighted indicator sets and their reference ratios

Finding 1. The number of hybrid applications supporting both IPv4 and IPv6 was fairly large. While this is a good sign for the deployment of IPv6, the dual addressing scheme doubles the complexity of address management in applications. At the client side, the application has to choose whether to handle DNS resolution over IPv4 or IPv6, and then create the actual connection with either family. As IPv6 does not even work everywhere yet, the client may initiate communications in parallel with IPv4 and IPv6 to minimize latency. Respectively, server-side applications have to listen for incoming data flows on both families.

Finding 2. Hybrid applications using both TCP and UDP occur as frequently as TCP-only applications. Application developers seem to write many application protocols to be run with both transports. While it is possible to write almost identical code for the two transports, the Sockets API favors different functions for the two. This unnecessarily complicates the application code.

Finding 3. The obsolete DNS resolver was referenced twice as frequently as the new one. This has negative implications on the adoption of new Sockets API extensions that are dependent on the new resolver. As concrete examples, native APIs for HIP and source address selection for IPv6 may experience a slow adoption path.

Finding 4. We discovered a UDP multihoming problem at the server side based on our experiments with three software included in the data set. As an upper bound, we estimated that the same problem affects 45.7% of the UDP-based applications.

Finding 5. Roughly half of the networking software is not satisfied with the default configuration of networking stack and alters it with socket options, raw sockets or other low-level hooking. However, we did not discover any patterns (besides few popular, individually recurring socket options) to propose as new compound socket option profiles for applications.

Findings 6, 7 and 8. Roughly every tenth application was using OpenSSL but surprisingly many failed to initialize it appropriately, thus creating potential security vulnerabilities. Half of the OpenSSL-capable applications were modifying the default configuration in some way. Many of these tweaks improved backwards compatibility at the expense of security. This opens a question why backwards compatibility is not well built into OpenSSL and why so many "knobs" are even offered to the developer.[3]

Finding 9. IPv6-mapped IPv4 addresses should not be leaked to the wire for security reasons. As a solution, the socket option `IPV6_V6ONLY` would prevent this leakage. However, only one out of total six applications using mapped addresses were actually using the socket option. Despite the number of total applications using mapped address in general was statistically small, this is an alarming sign because the number can grow when the number of IPv6 applications increases.

Finding 10. IPv6 source address selection lets an application to choose the type of an IPv6 source address instead of explicitly choosing one particular address. The extensions are not adopted yet, but we estimated the need for them in our set of applications. Our coarse-grained estimate is that two out of three IPv6 applications might utilize the extensions.

We have now characterized current trends with C-based applications using Sockets API directly and highlighted ten important findings. Of these, we believe findings 3, 4, 6 and 9 can be directly used to improved the existing applications in our data set. We believe that most of the remaining ones are difficult to improve without introducing changes to the Sockets API (findings 1, 2, 5) or without breaking interoperability (finding 7). Also, many of the applications appear not to need security at all (finding 8) and the adoption of extensions (finding 10) may just take some time.

[3]Some of the implementations of SSL/TLS are considered "broken"; they do not implement at all or fix incorrectly some of the bugs and/or functionalities in SSL/TLS.

As some of the findings are difficult to adapt to the applications using Sockets API directly, perhaps indirect approaches as offered by network application frameworks may offer easier migration path. For example, the first two findings are related to management of complexity in the Sockets API and frameworks can be used to hide such complexity from the applications.

4.3 Network Application Frameworks

In this section, we investigate four network application frameworks based the Sockets and POSIX API. In a way, these frameworks are just other "applications" using the Sockets API and, thus, similarly susceptible to the same analysis as the applications in the previous sections. However, the benefits of improving a single framework transcend to numerous applications as frameworks are utilized by several applications. The Sockets API may be difficult to change, but can be easier to change the details how a framework implements the complex management of the Sockets API behind its high-level APIs.

4.3.1 Generic Requirements for Modern Frameworks

Instead of applying the highlighted findings described in Section 4.2.4 directly, some modifications were made due to the different nature of network application frameworks.

Firstly, we reorganize the analysis "top down" and split the topics into end-host naming, look up, multiplicity of names and transport protocols and security. We also believe that the reorganization may be useful for extending the analysis in the future.

Secondly, we arrange the highlighted findings according to their topic. A high-level framework does not have to follow the IP address oriented layout of the Sockets API and, thus, we investigate the use of symbolic host names as well. The reconfiguration of the stack (finding 5) was popular but we could not suggest any significant improvements on it, so it is omitted. Finally, we split initiating of parallel connectivity with IPv4 and IPv6 as their own requirements for both transport connections and DNS look ups.

Consequently, the following list reflects the Sockets API findings as modified requirements for network application frameworks:

R1: End-host naming

R1.1 Does the API of the framework support symbolic host names in its APIs, i.e., does the framework hide the details of hostname-to-address resolution from the application? If this is true, the framework conforms to a similar API as proposed by Name Based Sockets as described in section 2.2. A benefit of this approach is that implementing requirements R1.2, R2.2, R3.1 and 3.3 becomes substantially easier.

R1.2 Are the details of IPv6 abstracted away from the application? In general, this requirement facilitates adoption of IPv6. It could also be used for supporting Teredo based NAT traversal transparently in the framework.

R1.3 IPv6-mapped addresses should not be present on the wire for security reasons. Thus, the framework should manually convert mapped addressed to regular IPv4 addresses before passing to any Sockets API calls. Alternatively, the frameworks can use the `AI_V4MAPPED` option as a safe guard to prevent such leakage.

R2: Look up of end-host names

R2.1 Does the framework implement DNS look ups with `getaddrinfo()`? This is important for IPv6 source address selection and native HIP API extensions because they are dependent on this particular function.

R2.2 Does the framework support parallel DNS look ups over IPv4 and IPv6 to optimize latency?

R3: Multiplicity of end-host names

R3.1 IPv6 source address selection is not widely adopted yet but is the framework modular enough to support it especially at the client side? As a concrete example, the framework should support inclusion of new parameters to its counterpart of `connect()` call to support application preferences for source address types.

R3.2 Does the server-side multihoming for UDP work properly? As described earlier, the framework should use `SO_BINDTODEVICE` option or `sendmsg()/recvmsg()` interfaces in a proper way.

R3.3 Does the framework support parallel `connect()` over IPv4 and IPv6 to minimize the latency for connection set-up?

R4: Multiplicity of transport protocols

R4.1 Are TCP and UDP easily interchangeable? “Easy” here means that the developer merely changes one class or parameter but the APIs are the same for TCP and UDP. It should be noted that this has also implications on the adoption of SCTP and DCCP.

R5: Security

R5.1 Does the framework support SSL/TLS?

R5.2 Does the SSL/TLS interface provide reasonable defaults and abstraction so that the developer does not have to configure the details of the security?

R5.3 Does the framework initialize the SSL/TLS implementation automatically?

4.3.2 ACE

ACE version 6.0.0 denotes one end of a transport-layer session with `ACE_INET_Addr` class that can be initiated both based on a symbolic host name and a numeric IP address. Thus, the support for IPv6 is transparent if the developer relies solely on host names and uses `AF_UNSPEC` to instantiate the class. ACE also supports storing of IPv4 addresses in the IPv6-mapped format internally but translates them to the normal IPv4 format before returning them to the requesting application or using on the wire.

In ACE, IP addresses can be specified using strings. This provides a more unified format to name hosts.

ACE supports `getaddrinfo()` function and resorts to `getnameinfo()` only when the OS (e.g. Windows) does not support `getaddrinfo()`.

With UDP, ACE supports both connected (class `ACE_SOCK_CODgram`) and disconnected communications (class `ACE_SOCK_Dgram`). We verified the UDP multihoming problem with test software included in the ACE software bundle. More specifically, we managed to repeat the problem with connected sockets which means that the ACE library shares the same bug as iperf, nc and nc6 software as described earlier. Disconnected UDP communications did not suffer from this problem because ACE does not fix the remote communication end-point for such communications with `connect()`. It should be also noted that a separate class, `ACE_Multihomed_INET_Addr`, supports multiaddressing natively.

A client can connect to a server using TCP with class `ACE_SOCK_Connector` in ACE. The instantiation of the class supports flags which could be used for extending ACE to support IPv6 source address selection in a backwards compatible manner. While the instantiation of connected UDP communications does not have a similar flag, it still includes few integer variables used as binary arguments that could be overloaded with the required functionality. Alternatively, new instantiation functions with different method signature could be defined using C++. As such, ACE seems modular enough to adopt IPv6 source address selection with minor changes.

For basic classes, ACE does not support accepting of communications simultaneously with both IPv4 and IPv6 at the server side. Class `ACE_Multihomed_INET_Addr` has to be used to support such behaviour more seamlessly but it can be used both at the client and server side.

Changing of the transport protocol in ACE is straightforward. Abstract class `ACE_Sock_IO` defines the basic interfaces for sending and transmitting data. The class is implemented by two classes: an application instantiates `ACE_Sock_Stream` class to use TCP or `ACE_SOCK_Dgram` to use UDP. While both TCP and UDP-specific classes supply some additional transport-specific methods, switching from one transport to another occurs merely by renaming the type of the class at the instantiation, assuming the application does not need the transport-specific methods.

ACE supports SSL albeit it is not as interchangeable as TCP with UDP. ACE has wrappers around `accept()` and `connect()` calls in its Acceptor-Connector pattern. This hides the intricacies of SSL but all of the low-level details are still configurable when needed. SSL is initialized automatically and correctly.

4.3.3 Boost::Asio

Boost::Asio version 1.47.0 provides a class for denoting one end of a transport-layer session called `endpoint` that can be initiated through resolving a host name or a numeric IP. By default, the resolver returns a set of endpoints that may contain both IPv4 and IPv6 addresses.[4] These endpoints can be given directly to the `connect()` wrapper in the library that connects sequentially to the addresses found in the endpoint set until it succeeds. Thus, the support for IPv6 is transparent if the developer has chosen to rely on host names. Boost::Asio can store IPv4 addresses in the IPv6-mapped form. By default, the mapped format is used only when the developer explicitly sets the family of the address to be queried to IPv6 and the query results contain no IPv6 addresses. The mapped format is only used internally and converted to IPv4 before use on the wire.

Boost::Asio uses POSIX `getaddrinfo()` when the underlying OS supports it. On systems such as Windows (older than XP) and Cygwin, Boost::Asio emulates `getaddrinfo()` function by calling `gethostbyaddr()` and `gethostbyname()` functions. The resolver in Boost::Asio includes flags that could be used for implementing source address selection (and socket options are supported as well).

Boost::Asio does not support parallel IPv4 and IPv6 queries, nor does it provide support for simultaneous connection set up using both IPv4 and IPv6.

We verified the UDP multihoming problem with example software provided with the Boost::Asio. We managed to repeat the UDP multihoming problem with connected sockets which means that the Boost::Asio library shares the same bug as iperf, nc and nc6 as described earlier.

Boost::Asio defines basic interfaces for sending and receiving data. An application instantiates `ip::tcp::socket` to use TCP or `ip::udp::socket` to use UDP. While both classes provide extra transport-specific methods, switching from one transport to another occurs merely by renaming the type of the class at the in-

[4]IPv6 addresses are queried only when IPv6 loopback is present

stantiation assuming the application does not need the transport-specific methods.

Boost::Asio supports SSL and TLS. The initialization is wrapped into the SSL context creation. In Boost::Asio, the library initialization is actually done twice as `OpenSSL_add_ssl_algorithms()` is a synonym of `SSL_library_init()` and both are called sequentially. PRNG is not automatically initialized with `RAND_load_file()`, `RAND_add()` or `RAND_seed()`, although Boost::Asio implements class `random_device` which can be easily used in combination with `RAND_seed()` to seed the PRNG.

4.3.4 Java.net

Java.net in OpenJDK Build b147 supports both automated connections and manually created ones. Within a single method that inputs a host name, its API hides resolving a host name to an IP address from DNS, creation of the socket and connecting the socket. Alternatively, the application can manage all of the intermediate steps by itself.

The API has a data structure to contain multiple addresses from DNS resolution. The default is to try a connection only with a single address upon request, albeit this is configurable. The internal presentation of a single address, `InetAddress`, can hold an IPv4 or IPv6 address and, therefore, the address family is transparent when the developer resorts solely on the host names. The API supports `v4_mappedaddress` format as an internal presentation format but it is always converted to the normal IPv4 address format before sending data to the network.

Before using IPv6, Java.net checks the existence of the constant `AF_INET6` and that a socket can be associated with a local IPv6 address. If java.net discovers support for IPv6 in the local host, it uses the `getaddrinfo()` but otherwise `gethostbyname()` function for name resolution. DNS queries simultaneously over IPv4 and IPv6 are not supported out-of-the-box. However, the SIP ParallelResolver package in SIP communicator[5] could be used to implement such functionality.

We verified the UDP multihoming problem with example software provided with the java.net. We managed to repeat the UDP multihoming problem with connected sockets. This means that the java.net library shares the same bug as iperf, nc and nc6 as described earlier.

[5]net.java.sip.communicator.util.dns.ParallelResolver

Java.net naming convention favors TCP because a "socket" always refers to a TCP-based socket. If the developer needs a UDP socket, he or she has to instantiate a `DatagaramSocket` class. Swapping between the two protocols is not trivial because TCP-based communication uses streams, where as UDP-based communication uses `DatagramPacket` objects for I/O.

IPv6 source address selection is implementable in java.net. TCP and UDP-based sockets could include a new type of constructor or method, and java has socket options as well. The method for DNS look ups, `InetAddress.getByName()`, is not extensive enough and would need an overloaded method name for the purpose.

Java.net supports both SSL and TLS. Their details are hidden by abstraction, although it is possible to configure them explicitly. All initialization procedures are automatic.

4.3.5 Twisted

With Twisted version 10.2, python-based applications can directly use host names to create TCP-based connections. However, the same does not apply to UDP; the application has to manually resolve the host name into an IP address before use.

With the exception of resolving of AAAA records from the DNS, IPv6 support is essentially missing from Twisted. Thus, mapped addresses and parallel connections over IPv4 and IPv6 remain unsupported due to lack of proper IPv6 support. Some methods and classes include "4" suffix to hard code certain functions only to IPv4 which can hinder IPv6 interoperability.

Introducing IPv6 source address selection to Twisted would be relatively straightforward, assuming IPv6 support is eventually implemented. For example, Twisted methods wrappers for `connect()` function input host names. Therefore, the methods could be adapted to include a new optional argument to specify source address preferences.

The twisted framework uses `gethostbyname()` but has also its own implementation of DNS, both for the client and server side. As IPv6 support is missing, the framework cannot support parallel look ups.

The UDP multihoming issue is also present in Twisted. We observed this by experimenting with a couple of client and server UDP applications in the Twisted source package.

TCP and UDP are quite interchangeable in Twisted when the application uses the `Endpoint` class because it provides abstracted read and write operations. However, two discrepancies exists. First, `Creator` class is tainted by TCP-specific naming conventions in its method `connectTCP()`. Second, applications cannot read or write UDP datagrams directly using host names but first have to resolve them into IP addresses.

Twisted supports TLS and SSL in separate classes. TLS/SSL can be plugged into an application with relative ease due to modularity and high-level abstraction of the framework. The details of SSL/TLS are configurable and Twisted provides defaults for applications that do not need special configurations. With the exception of seeding the PRNG, the rest of the details of TLS/SSL initialization are handled automatically.

4.3.6 A Summary of the Framework Results

We summarize how the requirements were met by each of the four frameworks in Table 3. Some of the requirements were unmet in all of the frameworks. For example, all frameworks failed to support UDP-based multihoming (R3.2) and parallel IPv4/IPv6 connection initialization for clients (R3.3). Also, SSL/TLS initialization (R5.3) was not implemented correctly in all frameworks. In total, 56 % of our requirements were completely met in all of the frameworks.

5 Related and Future Work

At least three other software-based approaches to analyze applications exist in the literature. Camara et al. [3] developed software and models to verify certain errors in applications using the Sockets API. Ammons et al. [1] have investigated machine learning to reverse engineer protocol specifications from source code based on the Sockets API. Palix et al. [14] have automatized finding of faults in the Linux kernel and conducted a longitudinal study.

Req.	ACE	Boost::Asio	Java.net	Twisted
R1.1	✓		✓	(✓)
R1.2	✓	✓	✓	
R1.3	✓	✓	✓	N/A
R2.1	✓	✓	✓	
R2.2				
R3.1	✓	✓	✓	✓
R3.2				
R3.3				
R4.1	✓	✓		(✓)
R5.1	✓	✓	✓	✓
R5.2	✓	✓	✓	✓
R5.3	✓	(✓)	✓	(✓)

Table 3: Summary of how the frameworks meet the requirements

We did not focus on the development of automatized software tools but rather on the discovery of a number of novel improvements to applications and frameworks using the Sockets API. While our findings could be further automatized with the tools utilized by Camara, Ammons and Palix et al., we believe such an investigation would be in the scope of another article.

Similarly to our endeavors with multihoming, Multiple Interfaces working group in the IETF tackles the same problem but in broader sense [2, 24]. Our work supplements their work, as we explained a very specific multihoming problem with UDP, the extent of the problem in Ubuntu Linux and the technical details how the problem can be addressed by developers.

6 Conclusions

In this article, we showed empirical results based on a statistical analysis of open-source network software. Our aim was to understand how the Sockets APIs and its extensions are used by network applications and frameworks. We highlighted ten problems with security, IPv6 and configuration. In addition to describing the generic technical solution, we also reported the extent of the problems. As the most important finding, we discovered that 28.6% of the C-based network applications in Ubuntu are vulnerable to attacks because they fail to initialize OpenSSL properly.

We applied the findings with C-based applications to four example frameworks based on the Sockets API. Contrary to the C-based applications, we analyzed the frameworks in a top-down fashion along generalized dimensions of end-host naming, multiplicity of names and transports, name look up and security. Consequently, we proposed 12 networking requirements that were completely met by a little over half of the frameworks in total. For example, all four frameworks consistently failed to support UDP-based multihoming and parallel IPv4/IPv6 connection initialization for the clients. Also the TLS/SSL initialization issue was present in some of the frameworks. With the suggested technical solutions for Linux, we argue that hand-held devices with multi-access capabilities have improved support for UDP, the end-user experience can be improved by reducing latency in IPv6 environments and security is improved for SSL/TLS in general.

7 Acknowledgments

We would like to thank Tao Wan for his initial work with the topic. We appreciate the discussion with Dmitriy Kuptsov, Antti Louko, Teemu Koponen, Antti Ylä-Jääski, Jukka Nurminen, Andrey Lukyanenko, Boris Nechaev, Zhonghong Ou, Cui Yong, Vern Paxon, Stefan Götz and Suvi Koskinen around the topic. The authors also express their gratitude to anonymous reviewers for their comments. This work was supported by grant numbers 139144 and 135230 from the Academy of Finland.

References

[1] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '02, pages 4–16, New York, NY, USA, 2002. ACM.

[2] M. Blanchet and P. Seite. Multiple Interfaces and Provisioning Domains Problem Statement. RFC 6418 (Informational), November 2011.

[3] P. de la Cámara, M. M. Gallardo, P. Merino, and D. Sanán. Model checking software with well-defined apis: the socket case. In *Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, FMICS '05, pages 17–26, New York, NY, USA, 2005. ACM.

[4] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar. Architectural Guidelines for Multipath TCP Development. RFC 6182 (Informational), March 2011.

[5] R. Gilligan, S. Thomson, J. Bound, J. McCann, and W. Stevens. Basic Socket Interface Extensions for IPv6. RFC 3493 (Informational), February 2003.

[6] R. Hinden and S. Deering. IP Version 6 Addressing Architecture. RFC 4291 (Draft Standard), February 2006. Updated by RFCs 5952, 6052.

[7] C. Huitema. RFC 4380: Teredo: Tunneling IPv6 over UDP through Network Address Translations (NATs), February 2006.

[8] M. Komu, M. Bagnulo, K. Slavov, and S. Sugimoto. Sockets Application Program Interface (API) for Multihoming Shim. RFC 6316 (Informational), July 2011.

[9] M. Komu and T. Henderson. Basic Socket Interface Extensions for the Host Identity Protocol (HIP). RFC 6317 (Experimental), July 2011.

[10] Craig Metz and Jun ichiro itojun Hagino. IPv4-Mapped Addresses on the Wire Considered Harmful, October 2003. Work in progress, expired in Oct, 2003.

[11] Robert Moskowitz, Pekka Nikander, Petri Jokela, and Thomas R. Henderson. RFC 5201: Host Identity Protocol, April 2008.

[12] E. Nordmark and M. Bagnulo. Shim6: Level 3 Multihoming Shim Protocol for IPv6. RFC 5533 (Proposed Standard), June 2009.

[13] E. Nordmark, S. Chakrabarti, and J. Laganier. IPv6 Socket API for Source Address Selection. RFC 5014 (Informational), September 2007.

[14] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia L. Lawall, and Gilles Muller. Faults in linux: ten years later. In Rajiv Gupta and Todd C. Mowry, editors, *ASPLOS*, pages 305–318. ACM, 2011.

[15] Eric Rescorla. *SSL and TLS, Designing and Building Secure Systems*. Addison-Wesley, 2006. Tenth printing.

[16] Michael Scharf and Alan Ford. MPTCP Application Interface Considerations, November 2011. Work in progress, expires in June, 2012.

[17] Douglas C. Schmidt. The adaptive communication environment: An object-oriented network programming toolkit for developing communication software. pages 214–225, 1993.

[18] P. Srisuresh and K. Egevang. Traditional IP Network Address Translator (Traditional NAT). RFC 3022 (Informational), January 2001.

[19] W. Stevens, M. Thomas, E. Nordmark, and T. Jinmei. Advanced Sockets Application Program Interface (API) for IPv6. RFC 3542 (Informational), May 2003.

[20] W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. *Unix Network Programming, Volume 1, The Sockets Networking API*. Addison-Wesley, 2004. Fourth printing.

[21] R. Stewart. RFC 4960: Stream Control Transmission Protocol, September 2007.

[22] T.Dierks and E. Rescorla. RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2, August 2008.

[23] Javier Ubillos, Mingwei Xu, Zhongxing Ming, and Christian Vogt. Name Based Sockets, September 2010. Work in progress, expires in March 2011.

[24] M. Wasserman and P. Seite. Current Practices for Multiple-Interface Hosts. RFC 6419 (Informational), November 2011.

[25] D. Wing and A. Yourtchenko. Happy Eyeballs: Success with Dual-Stack Hosts. RFC 6555 (Proposed Standard), April 2012.

Load-Balancing for Improving User Responsiveness on Multicore Embedded Systems

2012 Linux Symposium

Geunsik Lim
Sungkyungkwan University
Samsung Electronics
leemgs@ece.skku.ac.kr
geunsik.lim@samsung.com

Changwoo Min
Sungkyungkwan University
Samsung Electronics
multics69@ece.skku.ac.kr
changwoo.min@samsung.com

YoungIk Eom
Sungkyungkwan University
yieom@ece.skku.ac.kr

Abstract

Most commercial embedded devices have been deployed with a single processor architecture. The code size and complexity of applications running on embedded devices are rapidly increasing due to the emergence of application business models such as Google Play Store and Apple App Store. As a result, a high-performance multicore CPUs have become a major trend in the embedded market as well as in the personal computer market.

Due to this trend, many device manufacturers have been able to adopt more attractive user interfaces and high-performance applications for better user experiences on the multicore systems.

In this paper, we describe how to improve the real-time performance by reducing the user waiting time on multicore systems that use a partitioned per-CPU run queue scheduling technique. Rather than focusing on naive load-balancing scheme for equally balanced CPU usage, our approach tries to minimize the cost of task migration by considering the importance level of running tasks and to optimize per-CPU utilization on multicore embedded systems.

Consequently, our approach improves the real-time characteristics such as cache efficiency, user responsiveness, and latency. Experimental results under heavy background stress show that our approach reduces the average scheduling latency of an urgent task by 2.3 times.

1 Introduction

Performance improvement by increasing the clock speed of a single CPU results in a power consumption problems [19, 8]. Multicore architecture has been widely used to resolve the power consumption problem as well as to improve performance [24]. Even in embedded systems, the multicore architecture has many advantages over the single-core architecture [17].

Modern operating systems provide multicore aware infrastructure including SMP scheduler, synchronization [16], interrupt load-balancer, affinity facilities [22, 3], CPUSETS [25], and CPU isolation [23, 7]. These functions help running tasks adapt to system characteristics very well by considering CPU utilization.

Due to technological changes in the embedded market, OS-level load-balancing techniques have been highlighted more recently in the multicore based embedded environment to achieve high-performance. As an example, the needs of real-time responsiveness characteristics [1] have increased by adopting multicore architecture to execute CPU-intensive embedded applications within the desired time on embedded products such as a 3D DTV and a smart phone.

In embedded multicore systems, efficient load-balancing of CPU-intensive tasks is very important for achieving higher performance and reducing scheduling latency when many tasks running concurrently. Thus, it can be the competitive advantage and differentiation.

In this paper, we propose a new solution, *operation zone based load-balancer*, to improve the real-time performance [30] on multicore systems. It reduces the user

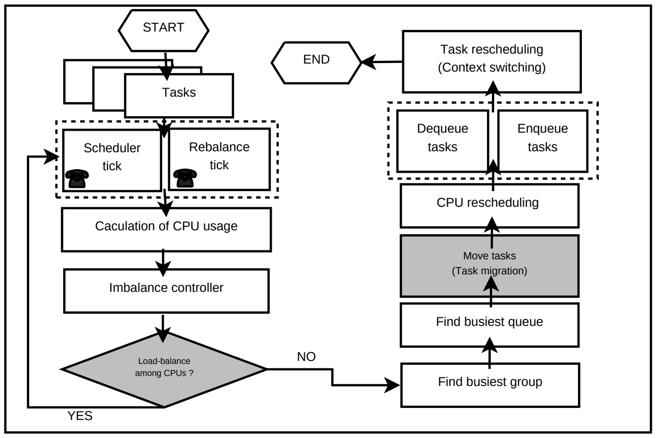


Figure 1: Load-balancing operation on Linux

waiting time by using a partitioned scheduling—or per-CPU run-queue scheduling—technique. Our solution minimizes the cost of task migration [21] by considering the importance level of running tasks and per-CPU utilization rather than focusing on naive CPU load-balancing for balanced CPU usage of tasks.

Finally, we introduce a flexible task migration method according to *load-balancing operation zone*. Our method improves operating system characteristics such as cache efficiency, effective power consumption, user responsiveness, and latency by re-balancing the activities that try to move specific tasks to one of the CPUs on embedded devices. This approach is effective on the multicore-based embedded devices where user responsiveness is especially important from our experience.

2 Load-balancing mechanism on Linux

The current SMP scheduler in Linux kernel periodically executes the load-balancing operation to equally utilize each CPU core whenever load imbalance among CPU cores is detected. Such aggressive load-balancing operations incur unnecessary task migrations even when the CPU cores are not fully utilized, and thus, they incur additional cache invalidation, scheduling latency, and power consumption. If the load sharing of CPUs is not fair, the multicore scheduler [10] makes an effort to solve the system's load imbalance by entering the procedure for load-balancing [11]. Figure 1 shows the overall operational flow when the SMP scheduler [2] performs the load-balancing.

At every timer tick, the SMP scheduler determines whether it needs to start load-balancing [20] or not, based on the number of tasks in the per-CPU run-queue. At first, it calculates the average load of each CPU [12]. If the load imbalance between CPUs is not fair, the load-balancer selects the task with the highest CPU load [13], and then lets the migration thread move the task to the target CPU whose load is relatively low. Before migrating the task, the load-balancer checks whether the task can be instantly moved. If so, it acquires two locks, `busiest->lock` and `this_rq->lock`, for synchronization before moving the task. After the successful task migration, it releases the previously held double-locks [5]. The definitions of the terms in Figure 1 are as follows [10] [18]:

- Rebalance_tick: update the average load of the run-queue.
- Load_balance: inspect the degree of load imbalance of the scheduling domain [27].
- Find_busiest_group: analyze the load of groups within the scheduling domain.
- Find_busiest_queue: search for the busiest CPU within the found group.
- Move_tasks: migrate tasks from the source run-queue to the target run-queue in other CPU.
- Dequeue_tasks: remove tasks from the external run-queue.
- Enqueue_tasks: add tasks into a particular CPU.
- Resched_task: if the priority of moved tasks is higher than that of current running tasks, preempt the current task of a particular CPU.

At every tick, the `scheduler_tick()` function calls `rebalance_tick()` function to adjust the load of the run-queue that is assigned to each CPU. At this time, load-balancer uses `this_cpu` index of local CPU, `this_rq`, `flag`, and `idle (SCHED_IDLE, NOT_IDLE)` to make a decision. The `rebalance_tick()` function determines the number of tasks that exist in the run-queue. It updates the average load of the run-queue by accessing `nr_running` of the run-queue descriptor and `cpu_load` field for all domains from the default domain to the domain of the upper layer. If the load imbalance is found, the SMP scheduler starts the procedure to balance the load of the scheduling domain by calling `load_balance()` function.

It is determined by `idle` value in the `sched_domain` descriptor and other parameters how frequently load-balancing happens. If `idle` value is `SCHED_IDLE`, meaning that the run-queue is empty, `rebalance_tick()` function frequently calls `load_balance()` function. On the contrary, if `idle` value is `NOT_IDLE`, the run-queue is not empty, and `rebalance_tick()` function delays calling `load_balance()` function. For example, if the number of running tasks in the run-queue increases, the SMP scheduler inspects whether the load-balancing time [4] of the scheduling domain belonging to physical CPU needs to be changed from 10 milliseconds to 100 milliseconds.

When `load_balance()` function moves tasks from the busiest group to the run-queue of other CPU, it calculates whether Linux can reduce the load imbalance of the scheduling domain. If `load_balance()` function can reduce the load imbalance of the scheduling domain as a result of the calculation, this function gets parameter information like `this_cpu`, `this_rq`, `sd`, and `idle`, and acquires spin-lock called `this_rq->lock` for synchronization. Then, `load_balance()` function returns `sched_group` descriptor address of the busiest group to the caller after analyzing the load of the group in the scheduling domain by calling `find_busiest_group()` function. At this time, `load_balance()` function returns the information of tasks to the caller to move the tasks into the run-queue of local CPU for the load-balancing of scheduling domain.

The kernel moves the selected tasks from the busiest run-queue to `this_rq` of another CPU. After turning on the flag, it wakes up `migration/*` kernel thread. The migration thread scans the hierarchical scheduling domain from the base domain of the busiest run-queue to the top in order to find the most idle CPU. If it finds relatively idle CPU, it moves one of the tasks in the busiest run-queue to the run-queue of relatively idle CPU (calling `move_tasks()` function). If a task migration is completed, kernel releases two previously held spin-locks, `busiest->lock` and `this_rq->lock`, and finally it finishes the task migration.

`dequeue_task()` function removes a particular task in the run-queue of other CPU. Then, `enqueue_task()` function adds a particular task into the run-queue of local CPU. At this time, if the priority of the moved task is higher than the current task, the moved task will preempt the current task by calling `resched_task()` function to gain the ownership of CPU scheduling.

As we described above, the goal of the load-balancing is to equally utilize each CPU [9], and the load-balancing is performed after periodically checking whether the load of CPUs is fair. The load-balancing overhead is controlled by adjusting frequency of load-balancing operation, `load_balance()` function, according to the number of running tasks in the run-queue of CPU. However, since it always performs load-balancing whenever a load imbalance is found, there is unnecessary load-balancing which does not help to improve overall system performance.

In multicore embedded systems running many user applications at the same time, load imbalance will occur frequently. In general, more CPU load leads to more frequent task migration, and thus, incurs higher cost. The cost can be broken down into direct, indirect, and latency costs as follows:

1. Direct cost: the load-balancing cost by checking the load imbalance of CPUs for utilization and scalability in the multicore system
2. Indirect cost: cache invalidation and power consumption
   (a) cache invalidation cost by task migration among the CPUs
   (b) power consumption by executing more instructions according to aggressive load-balancing
3. Latency cost: scheduling latency and longer non-preemptible period
   (a) scheduling latency of the low priority task because the migration thread moves a number of tasks to another CPU [29]
   (b) longer non-preemptible period by holding the double-locking for task migration

We propose our *operation zone based load-balancer* in the next section to solve those problems.

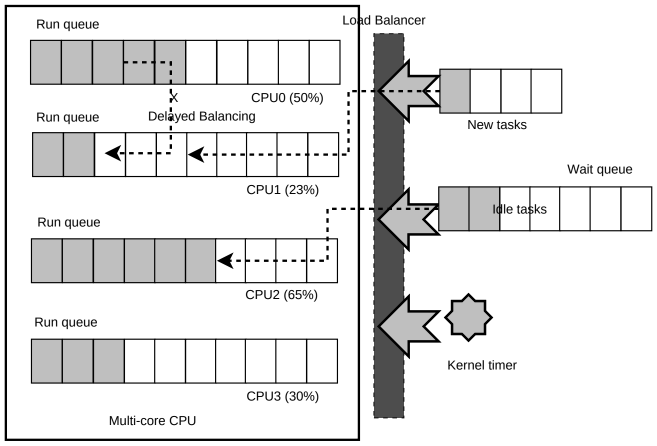


Figure 2: Flexible task migration for low latency

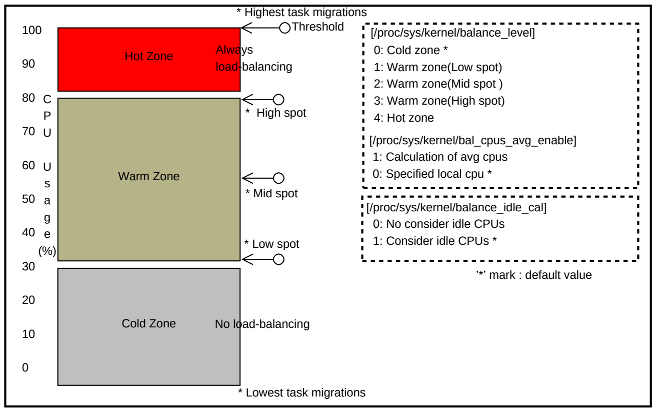
3 Operation zone based load-balancer

In this section, we propose a novel load-balancing scheduler called *operation zone based load-balancer* which flexibly migrates tasks for load-balancing based on *load-balancing operation zone* mechanism which is designed to avoid too frequent unnecessary load-balancing. We can minimize the cost of the load-balancing operation on multicore systems while maintaining overall CPU utilization balanced.

The existing load-balancer described in the previous section regularly checks whether load-balancing is needed or not. On the contrary, our approach checks only when the status of tasks can be changed. As illustrated in Figure 2, *operation zone based load-balancer* checks whether the task load-balancing is needed in the following three cases:

- A task is newly created by the scheduler.
- An idle task wakes up for scheduling.
- A running task belongs to the busiest scheduling group.

The key idea of our approach is that it defers load-balancing when the current utilization of each CPU is not seriously imbalanced. By avoiding frequent unnecessary task migration, we can minimize heavy double-lock overhead and reduce power consumption of a battery backed embedded device. In addition, it controls



```
[/proc/sys/kernel/balance_level]
 0: Cold zone *
 1: Warm zone(Low spot)
 2: Warm zone(Mid spot )
 3: Warm zone(High spot)
 4: Hot zone
[/proc/sys/kernel/bal_cpus_avg_enable]
 1: Calculation of avg cpus
 0: Specified local cpu *

[/proc/sys/kernel/balance_idle_cal]
 0: No consider idle CPUs
 1: Consider idle CPUs *
```

Figure 3: Load-balancing operation zone

the worst-case scenario: one CPU load exceeds 100% even though other CPUs are not fully utilized. For example, when a task in `idle`, `newidle`, or `noactive` state is rescheduled, we can make the case that does not execute `load_balance()` routine.

3.1 Load-balancing operation zone

Our *operation zone based load-balancer* provides *load-balancing operation zone* policy that can be configured to the needs of the system. As illustrated in Figure 3, it provides three multicore load-balancing policies based on the CPU utilization. The *cold zone* policy loosely performs load-balancing operation; it is adequate when the CPU utilization of most tasks is low.

On the contrary, the *hot zone* policy performs load-balancing operation very actively, and it is proper under high CPU utilization. The *warm zone* policy takes the middle between *cold zone* and *hot zone*.

Load-balancing under the *warm zone* policy is not trivial because CPU utilization in *warm zone* tends to fluctuate continuously. To cope with such fluctuations, *warm zone* is again classified into three spots—high, mid, and low—and our approach adjusts scores based on weighted values to further prevent unnecessary task migration caused by the fluctuation. We provide `/proc` interfaces for a system administrator to configure the policy either statically or dynamically. From our experience, we recommend that a system administrator configures the policy statically because of system complexity.

3.1.1 Cold zone

In a multicore system configured with the *cold zone* policy, our operation zone based load-balancing scheduler does not perform any load-balancing if the CPU utilization is in *cold zone*, 0~30%. Since there is no task migration in *cold zone*, a task can keep using the currently assigned CPU. Kernel performs the load-balancing only when the CPU utilization exceeds *cold zone*.

This policy is adequate where the CPU utilization of a device tends to be low except for some special cases. It also helps to extend battery life in battery backed devices.

3.1.2 Hot zone

Task migration in the *hot zone* policy is opposite to that in the *cold zone* policy. If the CPU utilization is in *hot zone*, 80~100%, kernel starts to perform load-balancing. Otherwise, kernel does not execute the procedure of load-balancing at all.

Under the *hot zone* policy, kernel defers load-balancing until the CPU utilization reaches *hot zone*, and thus, we can avoid many task migrations. This approach brings innovative results in the multicore-based system for the real-time critical system although the system throughput is lost.

3.1.3 Warm zone

In case of the *warm zone* policy, a system administrator chooses one of the following three spots to minimize the costs of the load-balancing operation for tasks whose CPU usage is very active.

- High spot (80%): This spot has the highest CPU usage in the *warm zone* policy. The task of *high spot* cannot go up any more in the *warm zone* policy
- Low spot (30%): This spot has the lowest CPU usage in the *warm zone* policy. The task of *low spot* cannot go down any more in the *warm zone* policy.
- Mid spot (50%): This spot is in between high spot and low spot. The weight-based dynamic score adjustment scheme is used to cope with fluctuations of CPU utilization.

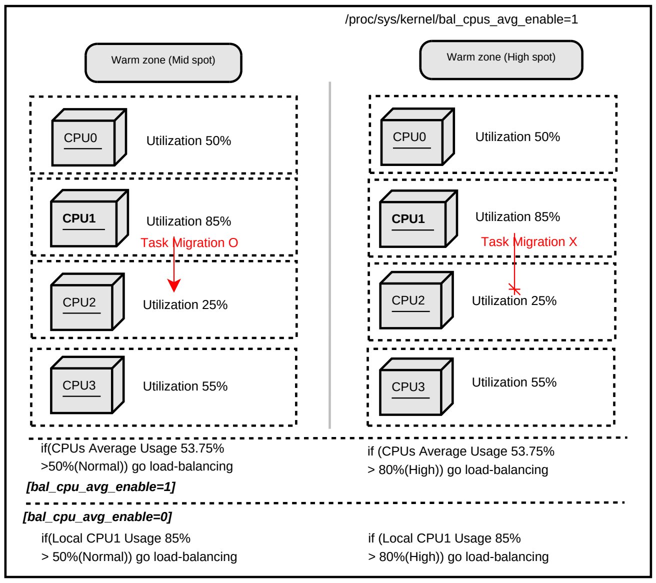

/proc/sys/kernel/bal_cpus_avg_enable=1
Warm zone (Mid spot)
Warm zone (High spot)
CPU0
Utilization 50%
CPU1
Utilization 85%
Task Migration O
Task Migration X
CPU2
Utilization 25%
CPU3
Utilization 55%
if(CPUs Average Usage 53.75% >50%(Normal)) go load-balancing
if (CPUs Average Usage 53.75% > 80%(High)) go load-balancing
[bal_cpu_avg_enable=1]
[bal_cpu_avg_enable=0]
if(Local CPU1 Usage 85% > 50%(Normal)) go load-balancing
if (Local CPU1 Usage 85% > 80%(High)) go load-balancing

Figure 4: Task migration example in warm zone policy

The spot performs the role of controlling the CPU usage of tasks that can be changed according to weight score. In the *warm zone* policy system, weight-based scores are applied to tasks according to the period of the CPU usage ratio based on *low spot*, *mid spot* and *high spot*. The three spots are detailed for controlling active tasks by users. The CPU usages of tasks have penalty points or bonus points according to the weight scores.

Although the score of task can increase or decrease, these tasks cannot exceed the maximum value, *high spot*, and go below the minimum value, *low spot*. If CPU usage of a task is higher than that of the configured spot in the *warm zone* policy, kernel performs load-balancing through task migration. Otherwise, kernel does not execute any load-balancing operation.

For example, we should consider that the CPU utilization of quad-core systems is 50%, 85%, 25% and 55% respectively from CPU0 to CPU3 as Figure 4. If the system is configured in *mid spot* of the *warm zone* policy, the load-balancer starts operations when the average usage of CPU is over 50%. Kernel moves one of the running tasks of run-queue in CPU1 with the highest utilization to the run-queue of CPU2 with the lowest utilization.

In case of *high spot* and the *warm zone* policy, the load-balancer starts the operations when the average usage of CPU is over 80%. Tasks that are lower than the CPU usage of the *warm zone* area is not migrated into another

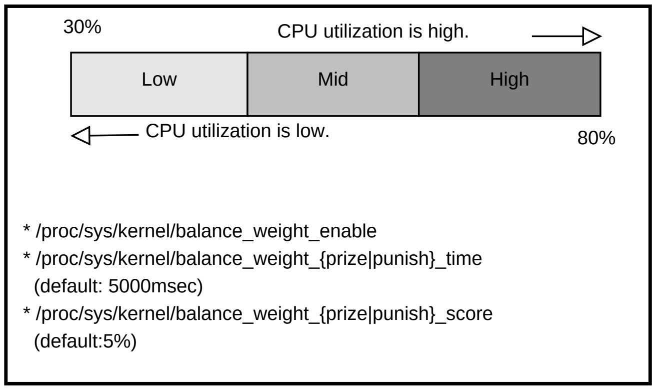

30%
CPU utilization is high.
Low
Mid
High
CPU utilization is low.
80%
* /proc/sys/kernel/balance_weight_enable
* /proc/sys/kernel/balance_weight_{prize|punish}_time (default: 5000msec)
* /proc/sys/kernel/balance_weight_{prize|punish}_score (default:5%)

Figure 5: Weight-based score management

CPU according to migration thread. Figure 4 depicts the example of load-balancing operations on the *warm zone* policy.

Figure 5 shows weight-based load score management for the *warm zone* policy system. When the usage period of CPU is longer than the specified time, five seconds by default, kernel manages bonus points and penalty points to give relative scores to the task that utilizes CPU resources continually and actively. Also, kernel operates the load weight-based *warm zone* policy to support the possibility that the task can use the existing CPU continually.

At this time, tasks that reach the level of the *high spot*, stay in the *warm zone* range although the usage period of CPU is very high. Through these methods, kernel keeps the border of the *warm zone* policy without moving a task to the *hot zone* area.

If a task maintains the high value of the usage of CPU more than five seconds as the default policy based on `/proc/sys/kernel/balance_weight_{prize|punish}_time`, kernel gives the task CPU usage score of -5 which means that CPU utilization is lower. At this point, the CPU usage information of the five seconds period is calculated by the scheduling element of a task via *proc file system*. We assigned the five seconds by default via our experimental experience. This value can be changed by using `/proc/sys/kernel/balance_weight_{prize|punish}_time` by the system administrator to support various embedded devices.

In contrast, if a task consumes the CPU usage of a spot shorter than five seconds, kernel gives the task CPU usage score of +5 which means that CPU utilization is higher. The task CPU usage score of +5 elevates the load-balancing possibility of tasks. Conversely, the task CPU usage score of -5 aims to bring down the load-balancing possibility of tasks.

The value of the *warm zone* policy is static, which means it is determined by a system administrator without dynamic adjustment. Therefore, we need to identify active tasks that consume the usage of CPUs dynamically. The load weight-based score management method calculates a task's usage in order that kernel can consider the characteristics of these tasks. This mechanism helps the multicore-based system manage the efficient load-balancing operation for tasks that have either high CPU usage or low CPU usage.

3.2 Calculating CPU utilization

In our approach, the CPU utilization plays an important role in determining to perform load-balancing. In measuring CPU utilization, our approach provides two ways: calculating CPU utilization for each CPU and averaging CPU utilization of all CPUs. A system administrator also can change behaviors through `proc` interface, `/proc/sys/kernel/balance_cpus_avg_enable`. By default, kernel executes task migration depending on the usage ratio of each CPU.

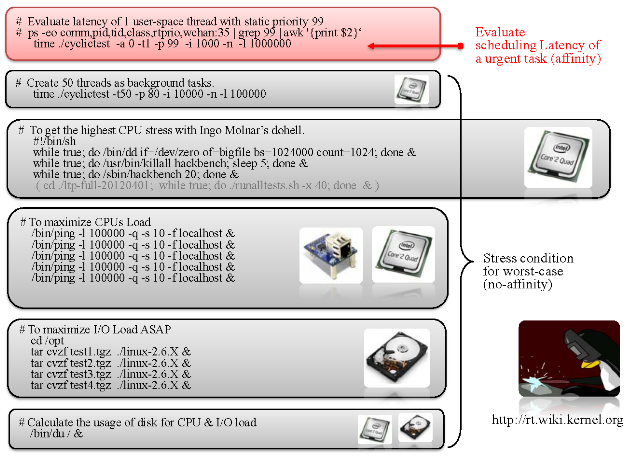
If a system administrator selects `/proc/system/kernel/balance_cpus_avg_enable=1` parameter for their system, kernel executes task migration depending on the average usage of CPUs.

The method to compare load-balancing by using the average usage of CPUs, helps to affinitize the existing CPU as efficiently as possible for some systems. The system needs the characteristics of CPU affinity [14] although the usage of a specific CPU is higher than the value of the *warm zone* policy, e.g. CPU-intensive single-threaded application in the most idle systems.

4 Evaluation

4.1 Evaluation scenario

Figure 6 shows our evaluation scenario to measure the real-time characteristics of running tasks in multicore based embedded systems. In this experiment, we measured how scheduling latency of an urgent task would



```
#  Evaluate latency of 1 user-space thread with static priority 99
#  ps -eo comm,pid,tid,class,rtprio,wchan:35 | grep 99 |awk '{print $2}'
     time ./cyclictest  -a 0 -t1 -p 99  -i 1000 -n  -l 1000000

#  Create 50 threads as background tasks.
     time ./cyclictest -t50 -p 80 -i 10000 -n -l 100000

#  To get the highest CPU stress with Ingo Molnar's dohell.
     #!/bin/sh
     while true; do /bin/dd if=/dev/zero of=bigfile bs=1024000 count=1024; done &
     while true; do /usr/bin/killall hackbench; sleep 5; done &
     while true; do /sbin/hackbench 20; done &
      ( cd ./ltp-full-20120401;  while true; do ./runalltests.sh -x 40; done  & )

# To maximize CPUs Load
    /bin/ping -l 100000 -q -s 10 -f localhost &
    /bin/ping -l 100000 -q -s 10 -f localhost &
    /bin/ping -l 100000 -q -s 10 -f localhost &
    /bin/ping -l 100000 -q -s 10 -f localhost &
    /bin/ping -l 100000 -q -s 10 -f localhost &

# To maximize I/O Load ASAP
    cd /opt
    tar cvzf test1.tgz  ./linux-2.6.X &
    tar cvzf test2.tgz  ./linux-2.6.X &
    tar cvzf test3.tgz  ./linux-2.6.X &
    tar cvzf test4.tgz  ./linux-2.6.X &

# Calculate the usage of disk for CPU & I/O load
    /bin/du / &
```

Evaluate scheduling Latency of a urgent task (affinity)
Stress condition for worst-case (no-affinity)
http://rt.wiki.kernel.org

Figure 6: Evaluation scenario to measure scheduling latency

be reduced under very high CPU load, network stress, and disk I/O.

To measure scheduling latency, we used *cyclictest* utility of *rt-test* package [28] which is mainly used to measure real-time characteristics of *Redhat Enterprise Linux (RHEL)* and real-time Linux. All experiments are performed in `Linux2.6.32` on `IntelQuadcoreQ9400`.

4.2 Experimental result

In Figure 7, we compared the scheduling latency distribution between the existing approach (before) and our proposed approach (after).

Our approach is configured to use *warm zone - high spot* policy. Under heavy background stress reaching to the worst load to the Quad-core system, we measured the scheduling latency of our test thread which repeatedly sleeps and wakes up. Our test thread is pinned to a particular CPU core by setting CPU affinity [15] and is configured as the FIFO policy with priority 99 to gain the best priority.

In the figure, X-axis is the time from the test start, and Y-axis is the scheduling latency in microseconds from when it tries to wake up for rescheduling after a specified sleep time. As Figure 7 shows, the scheduling latency of our test thread is reduced more than two times: from 72 microseconds to 31 microseconds on average.

In order to further understand why our approach reduces scheduling latency more than two times, we traced the

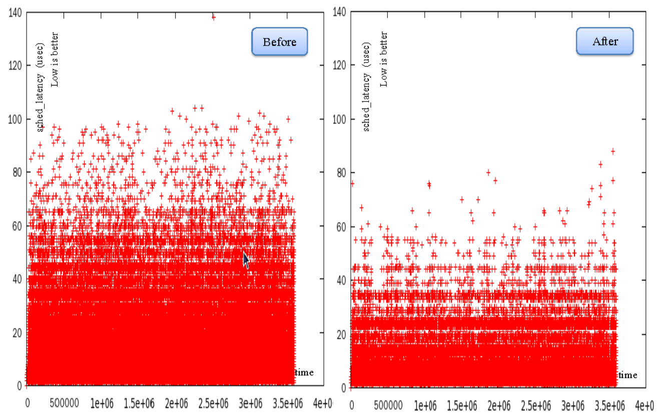


Figure 7: Comparison of scheduling latency distribution

caller/callee relationship of all kernel function during the experiment by using Linux internal function tracer, *ftrace* [26].

The analysis of the collected traces confirms three: first, the scheduling latency of a task can be delayed when migration of other task happens. Second, when task migration happens, non-preemptible periods are increased for acquiring double-locking. Finally, our approach can reduce average scheduling latency of tasks by effectively removing vital costs caused by the load-balancing of the multicore system.

In summary, since the migration thread is a real-time task with the highest priority, acquiring double-locking and performing task migration, the scheduling of the other tasks can be delayed. Since load imbalance frequently happens under a heavily loaded system with many concurrent tasks, the existing very fair load balancer incurs large overhead, and our approach can reduce such overhead effectively.

Our *operation zone based load-balancer* performs load-balancing based on CPU usage with lower overhead while avoiding overloading to a particular CPU that can increase scheduling latency. Moreover, since our approach is implemented only in the operating system, no modifications of user applications are required.

5 Further work

In this paper, we proposed an operation zone based load-balancing mechanism which reduces scheduling latency. Even though it reduces scheduling latency, it does not guarantee deadline for real-time systems where the worst case is most critical. In order to extend our approach to the real-time tasks, we are considering a hybrid approach with the physical CPU shielding technique [6] which dedicates a CPU core for a real-time task. We expect that such approach can improve real-time characteristics of a CPU intensive real-time task.

Another important aspect especially in embedded systems is power consumption. In order to keep longer battery life, embedded devices dynamically turn on and off CPU cores. To further reduce power consumption, we will extend our load-balancing mechanism considering CPU on-line and off-line status.

We experimented with scheduling latency to enhance the user responsiveness on the multicore-based embedded system in this paper. We have to evaluate various scenarios such as direct cost, indirect cost, and latency cost to use our load-balancer as a next generation SMP scheduler.

6 Conclusions

We proposed a novel *operation zone based load-balancing technique* for multicore embedded systems. It minimized task scheduling latency induced by the load-balancing operation. Our experimental results using the *cyclictest* utility [28] showed that it reduced scheduling latency and accordingly, users' waiting time.

Since our approach is purely kernel-level, there is no need to modify user-space libraries and applications. Although the vanilla Linux kernel makes every effort to keep the CPU usage among cores equal, our proposed *operation zone based load-balancer* schedules tasks by considering the CPU usage level to settle the load imbalance.

Our design reduces the non-preemptible intervals that require double-locking for task migration among the CPUs, and the minimized non-preemptible intervals contribute to improving the software real-time characteristics of tasks on the multicore embedded systems.

Our scheduler determines task migration in a flexible way based on the *load-balancing operation zone*. It limits the excess of 100% usage of a particular CPU and suppresses the task migration to reduce high overhead for task migration, cache invalidation, and high synchronization cost. It reduces power consumption and scheduling latency in multicore embedded systems, and thus, we expect that customers can use devices more interactively for longer time.

7 Acknowledgments

We thank Joongjin Kook, Minkoo Seo, and Hyoyoung Kim for their feedback and comments, which were very helpful to improve the content and presentation of this paper. This work was supported by the IT R&D program of MKE/KEIT [10041244, SmartTV 2.0 Software Platform].

References

[1] J.H. Anderson. Real-time scheduling on multicore platforms. In *Real-Time and Embedded Technology and Applications Symposium*, 2006.

[2] ARM Information Center. Implementing DMA on ARM SMP System. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0228a/index.html.

[3] M Astley. Migration policies for multicore fair-share schedulingd choffnes. In *ACM SIGOPS Operating Systems*, 2008.

[4] Ali R. Behrooz A. Shirazi, Krishna M. Kavi. Scheduling and load balancing in parallel and distributed systems. In *IEEE Computer Society Press Los Alamitos*, 1995.

[5] Stefano Bertozzi. Supporting task migration in multi-processor systems-on-chip: a feasibility study. In *In Proceeding DATE '06 Proceedings of the conference on Design, automation and test in Europe*, 2006.

[6] S Brosky. Shielded processors: Guaranteeing sub-millisecond response in standard linux. In *Parallel and Distributed Processing*, 2003.

[7] S Brosky. Shielded cpus: real-time performance in standard linux. In *Linux Journal*, 2004.

[8] Jeonghwan Choi. Thermal-aware task scheduling at the system software level. In *In Proceeding ISLPED '07 Proceedings of the 2007 international symposium on Low power electronics and design*, 2007.

[9] Slo-Li Chu. Adjustable process scheduling mechanism for a multiprocessor embedded system. In *6th WSEAS international conference on applied computer science*, 2006.

[10] Daniel P. Bovet, Marco Cesati. Understanding the Linux Kernel, 3rd Edition. O'Reilly Media.

[11] Mor Harchol-Balter. Exploiting process lifetime distributions for dynamic load balancing. In *ACM Transactions on Computer Systems (TOCS)*, volume 15, 1997.

[12] Toshio Hirosaw. Load balancing control method for a loosely coupled multi-processor system and a device for realizing same. In *Hitachi, Ltd., Tokyo, Japan, Patent No. 4748558*, May 1986.

[13] Steven Hofmeyr. Load balancing on speed. In *PPoPP '10 Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010.

[14] Vahid Kazempour. Performance implications of cache affinity on multicore processors. In *EURO-PAR*, 2008.

[15] KD Abramson. Affinity scheduling of processes on symmetric multiprocessing systems. http://www.google.com/patents/US5506987.

[16] Knauerhase. Using os observations to improve performance in multicore systems. In *Micro IEEE*, May 2008.

[17] Markus Levy. Embedded multicore processors and systems. In *Micro IEEE*, May 2009.

[18] Linus Toavalds. Linux Kernel. http://www.kernel.org.

[19] Andreas Merkel. Memory-aware scheduling for energy efficiency on multicore processors. In *HotPower'08 Proceedings of the 2008 conference on Power aware computing and systems*, 2008.

[20] Nikhil Rao, Google. Improve load balancing when tasks have large weight differential. http://lwn.net/Articles/409860/.

[21] Pittau. Impact of task migration on streaming multimedia for embedded multiprocessors: A quantitative evaluation. In *Embedded Systems for Real-Time Multimedia*, 2007.

[22] Robert A Alfieri. Apparatus and method for improved CPU affinity in a multiprocessor system. http://www.google.com/patents/US5745778.

[23] H PÃűtzl S Soltesz. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS*, 2007.

[24] Suresh. Siddha. Ottawa linux symposium (ols). In *Chip Multi Processing (CMP) aware Linux Kernel Scheduler*, 2005.

[25] Simon Derr, Paul Menage. CPUSETS. http://http://www.kernel.org/doc/Documentation/cgroups/cpusets.txt.

[26] Steven Rostedt. Ftrace (Kernel function tracer). http://people.redhat.com/srostedt.

[27] Suresh Siddha. sched: new sched domain for representing multicore. http://lwn.net/Articles/169277/.

[28] Thomas Gleixner, Clark williams. rt-tests (Real-time test utility). http://git.kernel.org/pub/scm/linux/kernel/git/clrkwllms/rt-tests.git.

[29] V Yodaiken. A real-time linux. In *Proceedings of the Linux Applications*, 1997.

[30] Yuanfang Zhang. Real-time performance and middleware on multicore linux platforms. In *Washington University*, 2008.

Experiences with Power Management Enabling on the Intel Medfield Phone

R. Muralidhar, H. Seshadri, V. Bhimarao, V. Rudramuni, I. Mansoor,
S. Thomas, B. K. Veera, Y. Singh, S. Ramachandra
Intel Corporation

Abstract

Medfield is Intel's first smartphone SOC platform built on a 32 nm process and the platform implements several key innovations in hardware and software to accomplish aggressive power management. It has multiple logical and physical power partitions that enable software/firmware to selectively control power to functional components, and to the entire platform as well, with very low latencies.

This paper describes the architecture, implementation and key experiences from enabling power management on the Intel Medfield phone platform. We describe how the standard Linux and Android power management architectures integrate with the capabilities provided by the platform to provide aggressive power management capabilities. We also present some of the key learning from our power management experiences that we believe will be useful to other Linux/Android-based platforms.

1 Introduction

Medfield is Intel's first smartphone SOC built on a 32 nm process. The platform implements several key innovations in hardware and software to accomplish aggressive power management. It has multiple logical and physical power partitions that enable software/firmware to selectively control power to functional components and to the entire platform as well, with very low latencies.

Android OS (Gingerbread/Ice Cream Sandwich) supports Suspend-to-RAM (a.k.a S3) state by building upon the traditional Linux power management infrastructure and uses concepts of wake locks (application hints about platform resource usage) to achieve S3. The power management infrastructure in Android requires that applications and services request CPU resources with *wake locks* through the Android application framework and native Linux libraries. If there are no active wake locks, Android will suspend the system to S3.

While the S3 implementation in Android helps reduce overall platform power when the device is not actively in use, S3 state does not satisfy applications that require always connected behavior (Instant messengers, VoIP, etc., need to send "keep alive" messages to maintain their active sessions). Entering S3 will result in freezing these applications and connections timing out so the sessions will have to be re-established on resume. The Medfield platform allows such applications to be active and yet achieve good power numbers through S0ix, or Connected Standby states. The main idea behind S0ix is that during an idle window, the platform is in the lowest power state as much as possible. In this state, all platform components are transitioned to an appropriate lower power state (CPU in Cx state, Memory in Self Refresh, components clock or power gated, etc.). As soon a timer or wake event occurs, the platform moves into an "Active state", only the components that are needed are turned on, keeping everything else in low power state. S0ix states are completely transparent to user space applications.

Figure 1 illustrates how S0ix states impact platform power states and how this compares with traditional ACPI-based power management.

This paper is organized as follows. Section 1 is this introduction. Section 2 presents a background of the Linux and Android Power Management architecture. Section 3 describes the key Intel specific power management components on Medfield platform to achieve S3 and S0ix. In Section 3, we will describe our experiences with enabling overall Power management, challenges/issues with handling wake interrupts, handling suspend/resume/runtime PM in different device drivers, and some optimizations that we had to implement on

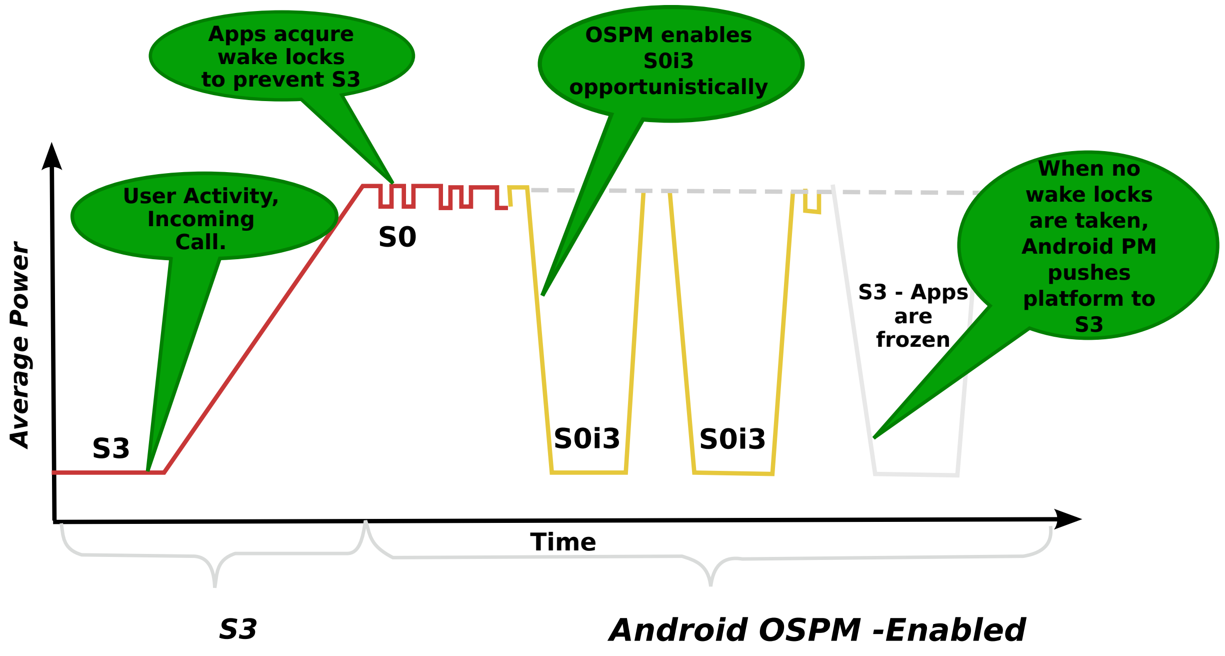

Apps acqure wake locks to prevent S3
OSPM enables S0i3 opportunistically
User Activity, Incoming Call.
When no wake locks are taken, Android PM pushes platform to S3
Average Power
S3
S0
S0i3
S0i3
S3 - Apps are frozen
Time
S3
Android OSPM -Enabled

Figure 1: Platform Power States with S0ix and S3

the platform. We believe that some of these learning will be applicable to other Linux/Android-based SOC platforms as well.

2 Medfield Platform Power Management Architecture

Medfield (or Atom Z2460) is Intel's first 32 nm smartphone SOC; the Atom Saltwell core runs at up to 1.6 GHz with 512KB of L2 cache, a PowerVR SGX 540 GPU at 400 MHz, a dual channel LPDDR2 memory interface (PoP LPDDR2 (2 x 32 bit support)), and ISP from Silicon Hive, additional I/O, and an external Power Management delivery unit. Figure 2 shows the high level architecture of the Medfield SOC platform.

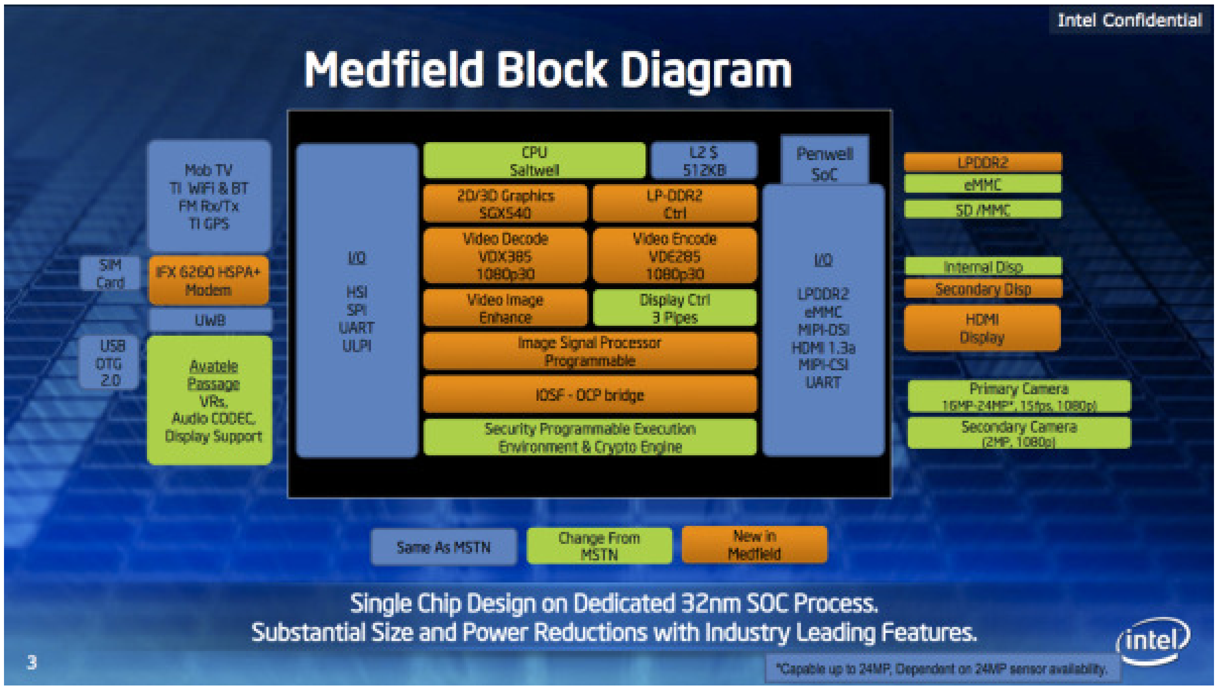
The Saltwell CPU is a dual-issue, in-order architecture with Hyper Threading support. The integer pipeline is sixteen stages long—the longer pipeline was introduced to help reduce power consumption by lengthening some of the decode stages and increasing cache latency to avoid burning through the core's power budget. There are no dedicated integer multiply or divide units, they are all shared with the floating point hardware. The CPU supports several different operating frequencies and power modes. At the lowest power level is its C6 state. Here the core and L2 cache are both power gated with their state saved in a lower power on-die SRAM. Total power consumption in C6 of the processor island is effectively zero. In addition to the 512KB L2 cache there is a separate 256KB SRAM which is lower power and on its own voltage plane. When Saltwell goes into its deepest sleep state, the CPU state and some L2 cache data gets parked here, allowing the CPU voltage to be lowered even more than the SRAM voltage. As expected, with hyperthreading the OS sees two logical cores to execute tasks on.

2.1 Core Linux Changes for x86 Smartphones

The Medfield platform is not PC-compatible in several aspects—no BIOS, no ACPI, no legacy devices, no PCI enumeration in South complex, no real IOAPIC, etc. Most of these changes are available in the upstream kernel now. Refer to [2], which discusses more details of the core kernel changes done for x86-based Intel platforms. This section briefly summarizes the key changes.

The following key changes were made to the underlying kernel in order to minimize changes from existing IA operating systems, and also provide software programming compatibility for key components of the platform (such as, PCI enumeration, IOAPIC interrupt controller, etc.):

1. Simple Firmware Interface (SFI) to replace ACPI



Single Chip Design on Dedicated 32nm SOC Process.
Substantial Size and Power Reductions with Industry Leading Features.

Figure 2: Medfield Platform Architecture

in order to report standard capabilities like CPU P-states, GPIOs, etc.

2. PCI Enumeration for South complex devices

3. IOAPIC Emulation

2.1.1 Simple Firmware Interface

Simple Firmware Interface (SFI) is a method for platform firmware to export static tables to the operating system. Platform firmware prepares SFI tables upon system initialization for the benefit of the OS (CPU P-states, GPIOs, for example). The OS consults the SFI tables to augment platform knowledge that it gets from native hardware interfaces, such as CPUID and PCI. More details on SFI can be found in [3].

2.1.2 PCI Enumeration

Penwell north complex devices are true PCI devices (Graphics, Display, Video encode/decode, ISP), but the South complex devices are fake PCI devices. All these south complex devices are enumerated as PCI devices through a PCI shim (Fake PCI MMCFG space written by firmware into main memory during platform boot) in the kernel. The PCI config space contains both true and fake PCI devices. MMCFG location is stored in SFI. Although this mechanism leverages existing device enumeration mechanism and reuses generic PCI drivers, this approach has its shortcomings in that it cannot detect device presence automatically. Also, PCI shim is read only, therefore, cannot handle writes to PCI config space.

2.1.3 Interrupt Routing and IOAPIC Emulation

Platform specific interrupt routing information is obtained from system firmware via PCI MMCFG space and SFI tables. Also, the south complex System controller Unit (SCU) maintains IOAPIC redirection tables that establish mapping between IRQ line and interrupt vectors.

3 Background: Linux and Android Power Management

Traditional ACPI-defined low power states for the platform are Hibernate to disk (S4) and Suspend to Ram (S3). A detailed treatment of the Linux power management architecture can be found in [5]. The kernel includes platform drivers responsible for carrying out low-level suspend and resume operations required by particular platforms. The platform drivers are used by the PM Core, which is itself generic; it runs on a variety of platforms for which appropriate platform drivers are available, including ACPI-compatible personal computers (PCs) and ARM platforms. Additionally, Linux kernel 2.6.33 and beyond mandate that all device drivers implement Linux Runtime Power Management, which is a framework through which device drivers can implement autonomous power management when idle. This is aggressively used in Medfield platform. On Medfield Android, we only support S3 and subsequent references to standby mean only S3.

3.1 Linux Suspend Resume Flow

When the system goes into the S3 state, the phases are: `prepare`, `suspend`, `suspend_noirq`. This is illustrated in Figure 3 and described in detail in the Linux kernel documentation.

- `prepare` - This may prepare the device or driver in some way for the upcoming system power transition (for example, by allocating additional memory required for this purpose) but it should not put the device into a low-power state.
- The `suspend` methods should quiesce the device, save the device registers and put it into the appropriate low-power state. It may enable wakeup events.
- The `suspend_noirq` phase occurs after IRQ handlers have been disabled, which means that the driver's interrupt handler will not be called while the callback method is running. This method should save the values of the device's registers that weren't saved previously and will finally put the device into the appropriate low-power state. Most device drivers need not implement this callback. However, bus types allowing devices to share interrupt vectors, like PCI, generally need it.

When resuming from standby or memory sleep, the phases are: `resume_noirq`, `resume`, `complete`.

- The `resume_noirq` callback methods should perform actions needed before the driver's interrupt handler is invoked.
- The `resume` method should bring the device back to its operating state so that it can perform normal I/O.
- The `complete` method should undo the actions of the prepare phase.

3.2 Android Power Management Architecture

Android Power Management infrastructure is split across the User space and Kernel layer. *Wake Locks* form a critical part of the framework. A *Wake Lock* can be defined as a request by the applications and services for some of the platform resources (CPU, display, etc.). The Android Framework exposes power management to services and applications through the PowerManager class. All calls from applications to acquire/release wake locks into Power Management should go through the Android runtime PowerManager API.

Kernel drivers can register with the Android Power Manager driver so that they are notified immediately prior to power down or after power up—drivers must register `early_suspend()` and `late_resume()` handlers, which are called when display power state changes. Please refer to [1] and [4] for more details.

4 Power Management Architecture in Medfield

Medfield platform provides fine-tuned knobs for platform level power management and expects the Operating System Power Manager (OSPM) to direct most of these power transitions of the subsystem. OS Power managers like ACPI, APM, etc. traditionally directs the platform to various power states (S3/S4, for example) depending on different power policy set by the user. In Medfield, the OS Power Manager guides the power states that the subsystems and CPU need depending on the Power policy set by the user. The HW then makes the policy decision. This is done by dedicated Power Management Units (PMU) that reside on the Platform. This gives the flexibility of making finer power state transitions which are normally not possible through traditional OS power management methods.

4.0.1 Power Management Capabilities

As is well known, the major sources of power dissipation in CMOS devices, as described in [8] and [9] are:

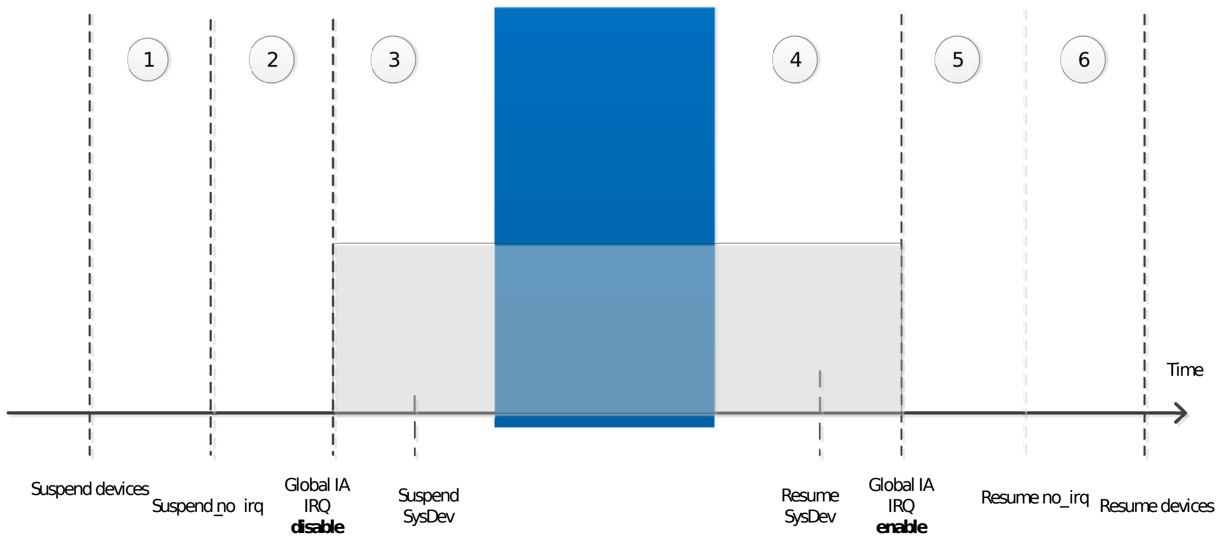


Figure 3: Linux Suspend Resume Flow

Switching power or *dynamic power* and *Leakage power*.

Switching or dynamic power represents the power required to charge and discharge circuit nodes. Broadly speaking, dynamic power depends on supply voltage (actually the square of supply voltage, $V^2 f$), clock frequency (f), node capacitance C (which in turn, depends on wire lengths), and switching activity factor (how frequently wires transition from 0 to 1, or from 1 to 0). Techniques such as clock gating are used to save energy by reducing activity factors during a hardware units idle periods. The clock frequency f, in addition to influencing power dissipation, also influences the supply voltage. Typically, higher clock frequencies will mean maintaining a higher supply voltage. Thus, the combined $V^2 f$ portion of the dynamic power equation has a cubic impact on power dissipation. Strategies such as dynamic voltage and frequency scaling (DVFS) try to exploit this relationship to reduce *(V, f)* accordingly.

Leakage power results due to current dissipation even when devices are not switching. The main reason behind this leakage is that transistors do not have ideal switching characteristics, and thereby leak a non-zero amount of current even for voltages lower than the threshold voltage. Hence power gating the entire logic (if possible) can ideally reduce the leakage power; this comes with additional responsibilities of saving/restoring the state, firewalling, etc.

The power management architecture in Medfield is built around these ideas aggressively that we can turn off subsystems without affecting the end user functionality and usability of the system. This is enabled by several platform hardware and software changes:

1. On die *clock and power gating* of subsystems
2. *Subsystem active idle states* that are OS transparent as well as driver managed
3. *Platform idle states* - extending idleness to the entire platform when all devices are idle

Device Power Management Capabilities - D0ix

All components, including the CPU, can be clock or power gated (individually, or as a combination). The CPU itself has its usual power states; C0 implies full power, full performance, and C6 is a deep sleep state where power is shut off to the entire CPU and state is saved in a small amount of active SRAM. The different power states supported by the Saltwell CPU are shown in Table 1.

Traditionally (according to ACPI, for example), subsystems/devices can be in active power state (D0) or in low power state (D1/D2/D3). Most subsystems/platforms implement D0 and D3, however, not many platforms/systems implement really active idle states, where the platform is active, but subsystems, even though are idle are in lower power state. In Medfield, devices can be in one of the following power states, traditionally called D-states:

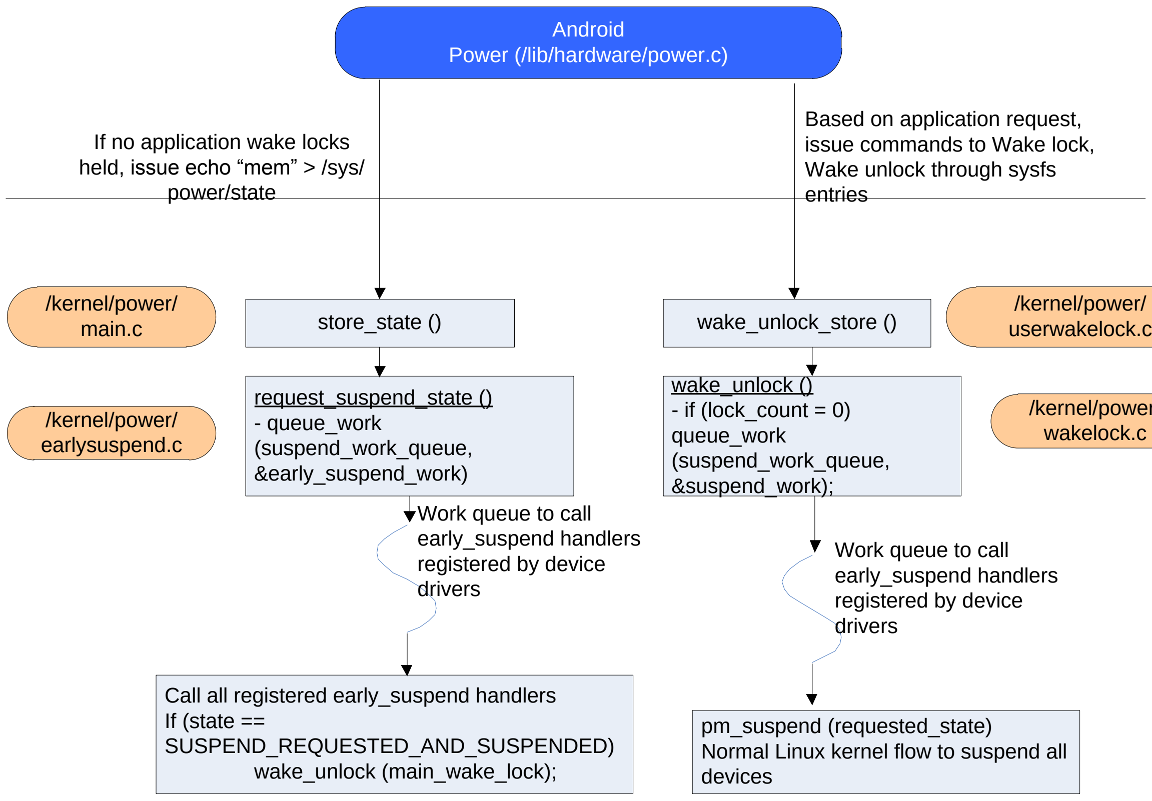

Android
Power (/lib/hardware/power.c)
If no application wake locks held, issue echo "mem" > /sys/power/state
Based on application request, issue commands to Wake lock, Wake unlock through sysfs entries
/kernel/power/main.c
store_state ()
wake_unlock_store ()
/kernel/power/userwakelock.c
/kernel/power/earlysuspend.c
request_suspend_state () - queue_work (suspend_work_queue, &early_suspend_work)
wake_unlock () - if (lock_count = 0) queue_work (suspend_work_queue, &suspend_work);
/kernel/power/wakelock.c
Work queue to call early_suspend handlers registered by device drivers
Work queue to call early_suspend handlers registered by device drivers
Call all registered early_suspend handlers If (state == SUSPEND_REQUESTED_AND_SUSPENDED) wake_unlock (main_wake_lock);
pm_suspend (requested_state) Normal Linux kernel flow to suspend all devices

Figure 4: Android Suspend Resume Flow

Feature	C0 HFM	C0 LFM	C1-C2	C4	C6
Core Voltage	ON	ON	ON	ON	OFF
Core Clock	ON	ON	OFF	OFF	OFF
L1 Cache	ON	ON	Flushed	Flushed	OFF
L2 Cache	ON	ON	ON	Partial Flushed	OFF
Wakeup time	Active	Active	Least	More	Highest

Table 1: Summary of Saltwell CPU Power States

1. D0 - Normal operational state

2. D0i1 - OS-transparent clock gated state

3. D0i3 - Driver directed management of the subsystem with no OS control of the subsystem. The device driver coordinates and manages the subsystem state (and saves/restores state as needed) for power transitions.

4. D3 - OS directed management of the subsystem. The device driver is involved in the management of the subsystem and it must perform state retention and restoration in the driver. OSPM will manage transitioning of power state of the device and the device driver must be involved in the power state transition.

All devices will be managed through the runtime Linux power management infrastructure. Device drivers must implement D0i3 (driver managed autonomous power management) through the Linux Runtime power management framework, and aggressively (and intelligently) manage the power of their corresponding subsystems. Additionally, device drivers must also support standard Linux suspend/resume callbacks for implementing D3.

4.1 Power Management Architecture

The key components of power management architecture on Medfield are:

1. Standard cpuidle- and cpufreq-based CPU power and performance management components (native drivers and governors).
2. Platform-specific S0ix extensions to the cpuidle driver (intel_idle-based) for Medfield's Saltwell CPU
3. Power Manager Unit (PMU) driver - This driver interfaces with both North and South Complex Power Management Units (PMUs). It also provides platform-specific implementation of deep idle states to the intel_idle-based processor drive and coordinates with the rest of the platform using standard kernel Power Management interfaces like PM_QOS, Linux Runtime PM, etc.
4. PMU Firmware that coordinates power management between the Platform PMUs: P-UNIT for north complex (CPU, Gfx blocks, ISP), and SCU for south complex (everything else: IO devices, storage, comms, etc.)

CPUIDLE driver performs idle state power management. It plugs into existing CPU Idle infrastructure and extends current intel_idle processor driver for the Medfield CPU (code-named Saltwell). It also exposes new platform idle states deeper than traditional C6—these actually correspond to deep idle states for the entire platform, when there is sufficient idleness on the platform. More details about cpuidle can be found in [6].

CPU frequency is managed by the cpufreq driver. The Medfield cpufreq-based P-state driver uses the existing cpufreq infrastructure and exposes the CPU frequency states to the governors. The most common/generic cpufreq governor is the ondemand governor. ondemand is a dynamic in-kernel cpufreq governor that can change CPU frequency depending on CPU utilization. It was first introduced in the linux-2.6.9 kernel. It has a simplistic policy that provides significant benefits to the platform by making use of fast frequency-switching features of the processors to effectively manage their frequencies depending on the CPU load. For a good overview of how DVFS support is provided by these generic Linux modules, please refer to [7].

The PMU driver communicates with the CPU idle driver, platform device drivers, and the PMU firmware to coordinate platform power state transitions. Based on the guidance/hint from idle prediction, the PMU driver opportunistically extends CPU idleness to rest of the platform. In order to do this most efficiently, all device drivers must also be implementing and autonomous power management through Linux Runtime power management. The PMU driver provides a platform-specific callback to the CPU idle framework so that long periods of idleness can be extended to the entire platform. Once CPU and devices are all idle, this driver programs the north and south complex PMUs to implement the required power transitions. The state we enter is called a **S0ix** state.

Android S3 states are directly mapped to S0i3, the only difference being that timers are disabled in S3 state (as compared to S0ix where OS timers can wake up the platform). This is illustrated in Table 2.

PMU driver performs the following actions to emulate S3 over S0i3 :

1. *Program Wake configuration*: PMU driver disables timers as wake source, therefore only events like USB or Comms events will cause a platform wake
2. *Prepare for S3*: Here PMU driver triggers CPU state to be offloaded to a separate SRAM and issues a command to the SCU to enter S0i3.
3. *Enter C6* on both CPU threads with MWAIT instruction. This will guide the CPU to a package-level C6, thereby allowing the PMUs to proceed with S0ix entry sequence in firmware.

With the above actions, the platform enters S3 (S0i3 with timers disabled). It is to be noted that the CPU state that was saved includes everything until the point of MWAIT instruction execution so that on resume, the CPU will start executing from the next instruction after MWAIT. In some cases entering package-level C6 might still fail (break interrupt for example). In that case, SCU will wait for a timeout period before aborting the S0ix/S3 entry.

On exit from S3 the PMU driver gets an interrupt with status register specifying the wake source. This is followed by the actual device wake interrupt. During the resume flow, PMU driver thread will resume devices and trigger thaw_processes() and resume all the devices.

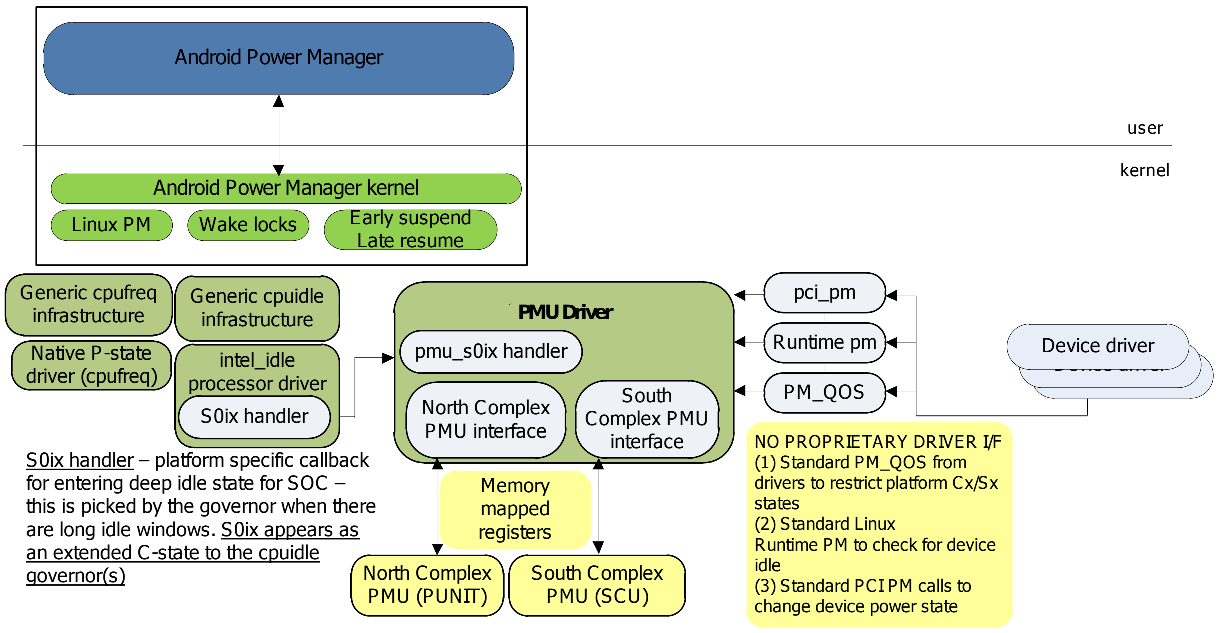

Android Power Manager
user
kernel
Android Power Manager kernel
Linux PM
Wake locks
Early suspend
Late resume
Generic cpufreq infrastructure
Generic cpuidle infrastructure
Native P-state driver (cpufreq)
intel_idle processor driver
S0ix handler
PMU Driver
pmu_s0ix handler
North Complex PMU interface
South Complex PMU interface
pci_pm
Runtime pm
PM_QOS
Device driver
S0ix handler – platform specific callback for entering deep idle state for SOC – this is picked by the governor when there are long idle windows. S0ix appears as an extended C-state to the cpuidle governor(s)
Memory mapped registers
North Complex PMU (PUNIT)
South Complex PMU (SCU)
NO PROPRIETARY DRIVER I/F
(1) Standard PM_QOS from drivers to restrict platform Cx/Sx states
(2) Standard Linux Runtime PM to check for device idle
(3) Standard PCI PM calls to change device power state

Figure 5: Medfield Platform Architecture

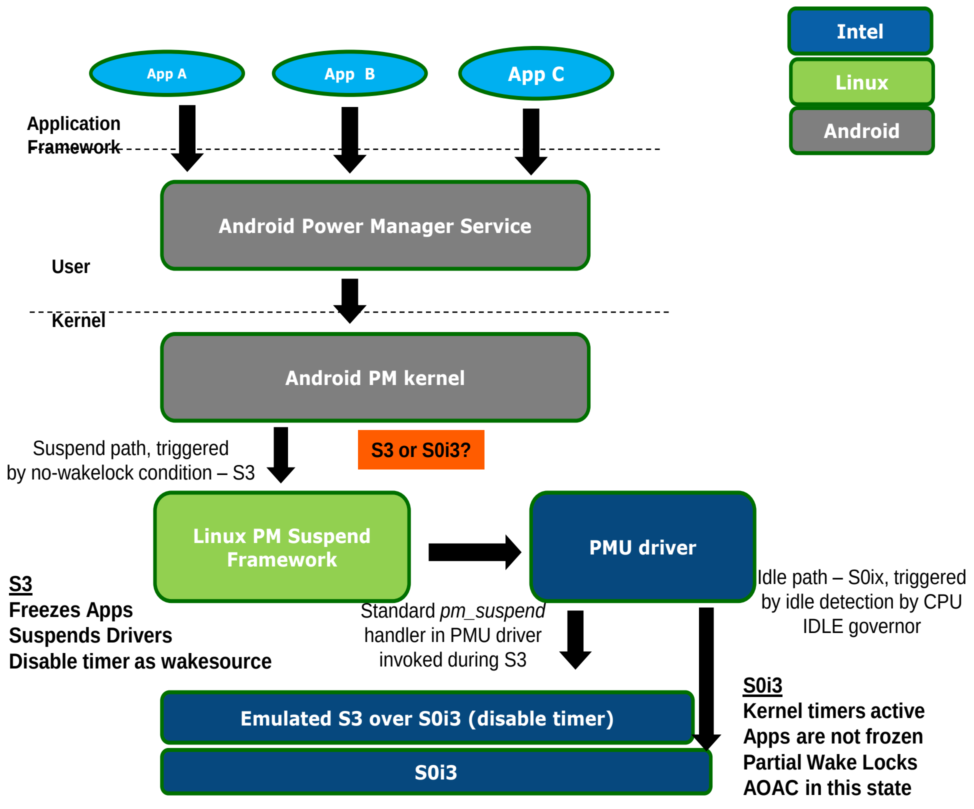

App A
App B
App C
Intel
Linux
Android
Application Framework
Android Power Manager Service
User
Kernel
Android PM kernel
Suspend path, triggered by no-wakelock condition – S3
S3 or S0i3?
Linux PM Suspend Framework
PMU driver
Idle path – S0ix, triggered by idle detection by CPU IDLE governor
S3
Freezes Apps
Suspends Drivers
Disable timer as wakesource
Standard pm_suspend handler in PMU driver invoked during S3
S0i3
Kernel timers active
Apps are not frozen
Partial Wake Locks
AOAC in this state
Emulated S3 over S0i3 (disable timer)
S0i3

Figure 6: Implementing Android Power Management in Medfield

Islands	S0:C0-C6	S0i1	S0i3	S3
CPU	C-state dependent	OFF	OFF	OFF
C6 SRAM, Wake logic	ON	ON	OFF	OFF
DDR	ON/Self-refresh (SR)	SR	SR	SR
Power Manager	ON	ON	OFF	OFF
Graphics	ON/power gated (PG)	PG	OFF	OFF
Video Decode	ON/power gated (PG)	PG	OFF	OFF
Video Encode	ON/power gated (PG)	PG	OFF	OFF
Display Controller	ON/power gated (PG)	PG	OFF	OFF
Display	ON	OFF	OFF	OFF
Device drivers	ON/D0ix	D0i3	D0i3	D3
Applications	Active	Active	Active	Frozen

Table 2: What is on in S0, S0ix, S3?

5 Experiences with Enabling Power Management

This section summarizes some of the most important learning we had enabling power management on the Medfield platform. Some of this learning are relevant to other Linux/Android-based SOCs as well.

5.1 Runtime Power Management Implementation in Device Drivers

All Medfield device drivers implement the Linux Runtime PM framework, whereby drivers autonomously detect their idleness, and guide their corresponding devices to a low power state. Since runtime PM was relatively a new subject in the Linux, we had to spend a significant amount of time in making sure that we have the right implementation in all the drivers. This was one of the key elements for getting the standby functionality stable and robust.

1. **Idle Detection**: Due to the absence of a general rule to detect idleness, we had to establish a process to detect idleness specific to a driver/device. As an example, I2C driver implemented runtime PM based on a rule that if the I2C bus is not being used by devices for a certain idle time, it would put itself into a low power state. As soon as platform sensors were enabled (which were hanging off the I2C bus), none of the sensors allowed to enter a deep sleep state as they were constantly accessing the bus. We had to fine tune the idle detection value to something more optimal that would allow the platform to enter extended idle periods.
   Recommendation: *Idle detection would be more effective if done through a combination of hardware capability (OS-visible, device specific idle/activity counters in HW) and software (guidance from OS/drivers) that will allow tuning/optimizations.*

2. **Runtime PM callbacks**: All the PCI drivers had implemented legacy suspend and resume handlers and had also implemented runtime PM—these two cannot co-exist (if not implemented correctly)—this led to conflict between the device state as maintained by Runtime PM core and Standard Linux kernel PM Core (resulting in kernel panics during suspend/resume phase). This was subsequently fixed manually in all such offending device drivers.
   Recommendation: *All drivers must implement power management correctly as mandated by the Linux kernel recommendations, and must also take into account the new features being added to the kernel as the power management support therein evolves and matures.*

5.2 Interrupt handling

1. **Accessing hardware devices after resuming from D0ix/S0ix**: Device drivers must ensure that corresponding hardware is powered up before accessing device registers. For example, when an interrupt lands on the USB driver it would first try to check if the interrupt was for itself by accessing registers in the USB host controller. The device

driver must ensure that the hardware is powered up before accessing any registers. Specifically, device drivers were modified to move their hardware access code outside the IRQ handler into a kernel bottom half handler and ensuring that the hardware is powered up by doing a `pm_runtime_get_sync()` function.
Recommendation: *Wakes and interrupt delivery logic must be foolproof and the device drivers must also be intelligent to handle such cases.*

2. **Implementing Correct PM Functions**: Some devices can have multiple ways in which interrupts are triggered—in-band and out-of-band through a GPIO. One such device was HSI. We observed that HSI was causing hangs during entry path—when the driver's `suspend` function was invoked, the driver had suspended its device. But when a sideband wake occurred just a little later in the S3 entry phase, it had already suspended and lost its state, thereby causing a kernel panic. This was fixed by having the HSI driver implement `suspend_noirq()` handler which would ensure that no interrupts would land unexpectedly.
Recommendation: *Device drivers must follow the Linux kernel power management recommendations and implement all the relevant callbacks for suspend/resume and runtime PM.*

3. **Enabling Wake Interrupts**: We faced issues with wakes happening from power button, WLAN wakes, etc, from an OS perspective, the interrupt seemed to be lost when the driver had resumed from S3. What was happening was this: when platform resumes from S3, all resume handlers get called (in some sequence). If the default kernel IRQ handler does not see any registered IRQ handler for a specific interrupt (which would have happened during suspend phase where driver deregisters its IRQ handler), it handles it by default and sends an EOI down to the IOAPIC. Thus, such interrupts were handled by default by the kernel since the drivers had not resumed yet. The Linux kernel has support for such conditions—device drivers can indicate using the `IRQF_NO_SUSPEND` flag that its IRQ handler should not be completely removed in S3. If a driver sets this flag, the kernel will invoke the corresponding IRQ handler on high priority, even before the resume handlers are called.
Recommendation: *Drivers with wake capability must use `IRQF_NO_SUSPEND` flag and implement `suspend_noirq()` handlers if there can be multiple (in-band, out-of-band) wake sources.*

5.3 Optimizing for power and performance

A lot of work went into optimizing the platform for Power and Performance (PnP) for all the important use cases on the phone. This section summarizes some of the key experiences and learning.

1. Optimizing platform wakes: A bulk of optimizations around platform power and performance came from optimizing the number of wakes that bring the platform out of standby states. These optimizations spanned firmware, device driver, middleware and applications.

2. S0ix Latency Optimizations: Optimizing standby (S0ix as well as S3) latencies is a critical component of ensuring that the penalty for entering/exiting standby is amortized by the benefits of the low power achieved in those states.

3. Performance optimizations: A lot of optimizations were done across applications and middleware for fine tuning different aspects of performance. For example, we fine tuned the `ondemand` frequency governor, to get the most optimum thresholds for the platform. The threshold ratios (80/20, for example) correspond to how fast the platform can go to higher frequencies and the increments of coming down. We fine tuned the thresholds for the platform based on different characteristics. While we eventually achieved an optimal setting for the platform, clearly this seems to be in the domain of heuristics and is more empirical rather than really knowing what thresholds are best. Currently there are a lot of ongoing discussions to optimize and overhaul the `cpufreq` and `ondemand` governor infrastructure. For future platforms, we believe this is a good area to invest and optimize.

5.4 Tools

All of the above fixes and optimizations would not have been possible without proper hardware and software tools.

1. **Voltage rail level analysis**: A setup with a detailed data acquisition system (DAQ) to acquire rail level power consumption is a MUST when we are dealing with power optimization of handheld devices. Many of the above analysis and optimization was done with such a setup.

2. **Tools like `ftrace` and `powertop`**: Ftrace and Powertop are tools which are already in use within the Linux community. The former helps us with profiling the code which causes CPU activity and the latter helps us in analyzing the wake sources (both software and hardware).

   There were also internal tools that helped for debugging, analyzing device D0ix residencies, memory bandwidth/utilization, etc.

6 Summary

This paper described the architecture, implementation and key learning from enabling aggressive power management on Medfield. The paper explained the standard Linux and Android Power Management architecture and the key architectural enablers for aggressive power management on Medfield—standard Android PM (S3) as well as S0ix, Intel's innovation in HW/SW that enables aggressive low power idle states. Finally we presented some of the key learning from platform-wide power management enabling and optimizations that we believe are important to other SOCs.

7 Acknowledgments

Many teams in Intel have been instrumental in enabling Power Management on Medfield in various phases of architecture, design, pre- and post-silicon validation, software integration and optimization, etc. The authors would like to acknowledge the following individuals/teams (in no particular order): Bruce Fleming, Belli Kuttanna, Ajaya Durg, Ticky Thakkar, Randy Hall, Kalyan Muthukumar, Padma Apparao, Richard Quinzio, the entire Power management and power/performance team, Robert Karas, Jon Brauer, Sreedhara DS, Abhijit Kulkarni, Srividya Karumuri, Pramod HG, Rupal Parikh, Ryan Pinto, Mark Gross, Yanmin Zhang, Nicolas Roux, Pierre Tardy, Christophe Fiat and many others who have helped out in different phases/aspects of Power Management debug/enabling.

References

[1] Android Developer Reference, *http://developer.android.com*

[2] Jacob Pan, *Porting the Linux kernel to x86 MID Platforms*, Embedded Linux Conference 2010.

[3] Simple Firmware Interface, *http://www.simplefirmware.org*

[4] R. Wysocki, *Technical background of the Android Suspend Blockers Controversy*, http://www.lwn.net/images/pdf/suspend_blockers.pdf.

[5] L. Brown, R. Wysocki, *Suspend to RAM in Linux*, In Proceedings of the Ottawa Linux Symposium 2008.

[6] V. Pallipadi, A. Belay, S, *cpuidle: do nothing, efficiently*, In Proceedings of the Ottawa Linux Symposium 2007.

[7] V. Pallipadi, A. Starikovskiy, *The ondemand governor: past, present and future*, In Proceedings of Linux Symposium, 2006.

[8] A. Chandrakasan, S. Sheng, and R. Brodersen, *Low-power CMOS digital design, 1992.*

[9] A. Ghosh, S. Devadas, K. Keutzer, and J. White, *Estimation of average switching activity in combinational and sequential circuits. In Proceedings of the 29th ACM/IEEE conference on Design automation*, Pages 253âĂŞ259. IEEE Computer Society Press, 1992.

[10] Android PowerManagement, `http://developer.android.com/reference/android/os/PowerManager.html`

File Systems: More Cooperations - Less Integration.

Alex Depoutovitch
VMWare, Inc.
aldep@vmware.com

Andrei Warkentin
VMWare, Inc.
andreiw@vmware.com

Abstract

Conventionally, file systems manage storage space available to user programs and provide it through the file interface. Information about the physical location of used and unused space is hidden from users. This makes the file system free space unavailable to other storage stack kernel components due to performance or layering violation reasons. This forces file systems architects to integrate additional functionality, like snapshotting and volume management, inside file systems increasing their complexity.

We propose a simple and easy-to-implement file system interface that allows different software components to efficiently share free storage space with a file system at a block level. We demonstrate the benefits of the new interface by optimizing an existing volume manager to store snapshot data in the file system free space, instead of requiring the space to be reserved in advance, which would make it unavailable for other uses.

1 Introduction

A typical operating system storage stack consists of functional elements that provide many important features: volume management, snapshots, high availability, data redundancy, file management and more. The traditional UNIX approach is to separate each piece of functionality into its own kernel component and to stack these components on top of one another (Fig. 1). Communication typically occurs only between the adjacent components using a linear block device abstraction.

This design creates several problems. First, users have to know at installation time how much space must be reserved for the file system and for each of the kernel components. This reservation cannot be easily changed after the initial configuration. For example, when deploying a file system on top of an LVM volume manager, users have to know in advance if they intend to use snapshots, and they must choose the size of the file system so as to leave enough storage space to hold the snapshots [2]. This reserved space is not available to user programs and snapshots cannot grow larger than the reserved space. The end result is wasted storage capacity and unnecessary over-provisioning. Second, kernel components underneath the file system have to resort to storing their data within blocks at fixed locations. For example, Linux MD software RAID expects its internal metadata in the last blocks of the block device [4]. Because of this, MD cannot be installed without relocating an existing file system [1].

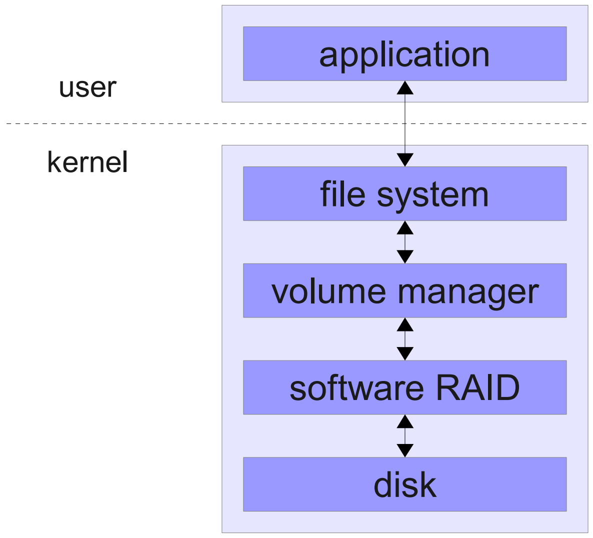


Figure 1: Storage stack

To solve the above problems, many modern file systems, like ZFS, BTRFS, and WAFL, tend to incorporate a wide range of additional functionality, such as software RAID arrays, a volume manager, and/or snapshots [1, 6, 11]. Integrating all these features in a single software component brings a lot of advantages, such as flexibility and ease of storage configuration, shared free space pool, and potentially more robust storage al-

[1]There is an option to store the metadata on a separate file system, but this requires a separate file system to be available.

location strategies. However, additional functionality of such "monolithic" file systems results in a complex and large source code. For example, BTRFS contains twice as many lines of code as EXT4, and four times more than EXT3. ZFS source code is even larger. A large code-base is more error-prone and more difficult to modify. BTRFS development started in 2008, and the stable version is yet to be released. In addition, these file systems put users into an "all or nothing" position: if one needs BTRFS volume management or snapshotting features, one has to embrace its slower performance in many benchmarks [13].

We believe that a file system can and should provide centralized storage space management for stateful components of the storage stack without integrating them inside the file system code. File systems are ideally suited for this role as they already implement disk space management. We propose a new interface, called *Block Register* (BR), to a file system that allows both user-mode applications and kernel components to dynamically reserve storage from a single shared common pool of free blocks. With BR interface, stateful storage stack components do not need to reserve storage space outside the file system, but can ask the file system to reserve the necessary space for them.

With the help of the new interface, file systems (such as EXT4) can benefit from more effective storage resource utilization, reduced wasted space, and flexible system configuration, without integrating additional functionality inside the file system. The BR interface also solves the problem of fixed metadata location, allowing kernel components to dynamically query for the location of necessary blocks. We demonstrate the benefits of the BR interface by integrating it with LVM snapshot mechanism and allowing snapshots to use file system free blocks instead of pre-allocated storage space. It is important that our interface can be exposed by existing file systems with no changes to their disk layout and minimal or no changes to their code.

2 Block Register Interface

The main purpose of the Block Register interface is to allow kernel components to ask the file system to reserve blocks of storage, increase and decrease number of reserved blocks, query for blocks that have been reserved before, and release blocks back for future reuse. While we want this interface to provide both fine-grained control and extensive functionality, we also make sure that it can be implemented by any general-purpose file system without core changes to its design or its on-disk layout, and with no or minimal code changes. The interface needs to be generic and easy to use with simple abstractions. Although we think that kernel components will be the primary users of the Block Register interface, user mode applications might benefit from it as well, for example, to get access to data saved by kernel components. Therefore, the Block Register interface has to be accessible from both user and kernel modes.

brreserve(name, block_num)

brquery(name, offset, block_num) returns set of blocks

brexpand(name, block_num)

brtruncate(name, block_num)

brfree(name)

Table 1: Block Register interface functions.

The main abstraction we provide is a *reservation*. A reservation is a set of blocks on a block device, identified by their block numbers. These blocks belong to the caller and can be accessed directly by calling a block device driver for the device on which the file system is located. The file system will neither try to use these blocks itself nor include them in other reservation requests. Each reservation can be uniquely identified by its name. In the Table 1, we list functions of the Block Register API. Currently, we define 5 functions. When a caller needs to reserve a set of blocks, it calls the `brreserve()` function, specifying a unique name for this reservation and the number of blocks requested. In response, the file system reserves the required the number of blocks from its pool of free blocks.

When a caller needs to get the list of blocks of an existing reservation, it calls the `brquery()` function, passing it the name of the existing reservation and the range of blocks required. The range is specified by the offset of the first block in the query and the number of blocks to return. An existing reservation can grow and shrink dynamically by calls to `brexpand()` and `brtruncate()`. `brexpand()` takes the reservation name and the number of blocks to expand by, while `brtruncate()` takes the physical block to return back to the file system.

Finally, an existing reservation can be removed by calling the `brfree()` function. After a reservation is removed, all blocks belonging to the reservation are returned to the file system's pool of free blocks.

3 Linux Implementation

In this section, we will explore how the Block Register interface could be implemented in the EXT4 file system. Although we use some EXT4- and Linux-specific features, they are used only as a shortcut in the prototype implementation of the BR interface.

Although this is not necessary, we decided to represent Block Register reservations as regular files, using a common name space for files and reservations. Thus, `brreserve()` and `brfree()` can be mapped to file creation and deletion. Leveraging the existing file name space has other advantages with respect to access and management. Since reservations are basically normal files, they can be easily accessed with existing file system utilities and interfaces. Because the BR interface is oriented towards block I/O, application data access through the file system cache results in a data coherence problem and, therefore, should be prohibited. While open, reservation files must be also protected from modifications of file block layout, such as truncation. On Linux, this can be solved by marking the file as immutable with the `S_IMMUTABLE` flag.

Having the block reservations being treated as files has an additional implication. Invoking `brquery()` for every access would have a significant performance impact. Because of this, the caller of the BR interface may want to cache the results of previous `brquery()` calls. Thus, the file system cannot perform any optimization on the file, such as defragmentation, copy-on-write, and deduplication. To enforce this requirement, we relied on the Linux `S_SWAPFILE` inode flag, used to mark a file as unmovable.

We implemented `brexpand()` and `brtruncate()` using the `fallocate()` function implemented in Linux and EXT4 which allows changing the file size without performing actual writes to the file. There are, however, a few specifics of `fallocate()` behaviour that have to be taken into consideration. First, `fallocate()` makes no guarantees on the physical contiguity of allocated space. This may affect I/O performance. Second, `fallocate()` call results in write operations to the underlying block device, thus special care has to be taken by the caller in order to avoid deadlocks.

`brquery()` was implemented using the `bmap()` function, which returns the physical device block for the given logical file block. `bmap()` may also trigger some read requests to the underlying storage.

4 Snapshots with Block Register Interface

In order to evaluate the Block Register interface, we modified Linux Logical Volume Manager (*LVM*) to use our new interface to allocate storage necessary to create and maintain snapshots of block devices [2]. In this section, we briefly describe the changes we made to the snapshotting code so that it can make use of the Block Register interface.

Snapshots preserve the state of a block device at a particular point in time. They are widely used for many different purposes, such as backups, data-recovery, or sandboxing. LVM snapshots are implemented using device mapper. Snapshots require additional storage to hold the original version of the modified data. The size of the additional storage depends on the amount of data modified during the lifetime of the snapshot. Currently, users planning to use snapshots create a file system only on a part of available storage, reserving the rest of it for snapshot data. The space reserved for snapshots cannot be used to store files, and its size is fixed after the file system creation.

We modified dm-snap, the device mapper target responsible for creating snapshots, to use the Block Register interface to allocate space to store the snapshot data as needed instead of using space reserved in advance. dm-snap uses this space to maintain the set of records, called an exception store, that describe chunks of the original device that have been been modified and copied to the snapshot device as well as the old contents of these chunks. We created a new exception store persistence mechanism which uses block reservations instead of a block device for storing data. We based our code on the existing exception store code, dm-snap-persistent, and reused its metadata layout.

In order to create a snapshot of a file system that supports the Block Register interface, the user specifies the device with the file system and the name to be used for a block reservation on that file system. This reservation will contain the snapshot data. The new dm-snap code calls `brreserve()` for the initial block reservation. Responding to this request, the file system creates a new reservation. dm-snap queries blocks allocated to this reservation and creates a new snapshot using the reserved blocks.

After a certain amount of changes to the file system, dm-snap will need additional blocks to maintain the original version of the file system's data. In order to get these blocks, dm-snap calls `brexpand()` to request the new set of blocks. Because expanding the reservation might cause additional changes to the file system metadata, this call cannot be made in the context of a file system operation, otherwise, deadlock might occur. In order to avoid the deadlock, the new dm-snap maintains a pool of available free blocks. If the pool falls below a "low watermark", a separate thread wakes up and requests additional blocks through the Block Register interface. The value of the low watermark depends on the write throughput of the file system.

To improve performance and prevent callbacks to the file system in the context of an I/O operation, dm-snap caches the data returned by `brquery()` in memory and queries this data with a binary search combined with a most-recently-used lookup. The implementation of the cache mechanism has been largely based on the obsolete dm-loop prototype [3].

During the operating system boot process, before the file system can be mounted in read-write mode, it has to be mounted in read-only mode, so LVM can query the file system for blocks reserved for snapshot data. After that, LVM enables the copy-on-write mechanism for the block device containing the file system and exposes the snapshots to the user. Once the COW mechanism is enabled, the file system can be remounted in read-write mode.

5 Evaluation

In order to evaluate our changes, we incorporated them into the Linux 3.2.1 kernel and measured their impact on performance of the file system in the presence of a snapshot. We compared our snapshot implementation using the Block Register interface and the standard LVM snapshot implementation. The results are presented in Figure 2. Each measurement shows the performance of the file system in the corresponding benchmark with an active snapshot implemented using the Block Register interface relative to performance of the same benchmark with a snapshot taken using the standard Linux LVM implementation. Each result is the average of 3 runs. Values greater than 100% mean that the Block Register implementation outperforms the standard Linux implementation.

In the first set of experiments, we created a 50GB test file on the file system, took a snapshot of the file system, and started issuing I/O requests to the file on the original file system. During each experiment we maintained the number of outstanding I/O operations (*OIO*) to be constant (either 1 or 8) and measured the number of completed I/O operations per second. Each I/O operation had a size of 4096 bytes. We used direct synchronous I/O to avoid caching effects. Since read operations do not involve snapshotting, we did not notice any difference between our and the standard Linux snapshot implementations. However, our snapshot implementation behaved better while serving write operations. In random write operations, performance improvement varies from 1% for OIO=1 to 8% for OIO=8. Performance gain for sequential write operations is more significant: 9% for OIO=1 and 26% for OIO=8.

In the second set of experiments, we created a snapshot of an empty file system and measured the time necessary to unpack 5 copies of the Linux 3.2.1 kernel source tree from an archive to the file system and then delete all created files. The last column on Figure 2 shows the result of this experiment. Our implementation performs on a par with the standard Linux implementation in this benchmark.

We believe that the performance improvement comes from a shorter disk head seek span during write operations with our snapshot implementation. As shown on Figure 3, copy-on-write operation for a block requires two additional seek operations. The first is to seek to the location of the block B2 in the snapshot to write the original data of the block B1, and the second is to seek back to the original block B1 to write the new data. In the standard Linux snapshot implementation, storage blocks containing snapshot data are located in the reserved area outside the file system. This forces the disk to perform seek operations on average over the span of the whole file system. In our implementation, a file system has a chance of placing these two blocks closer to each other, thus reducing the seek time. Initially, when

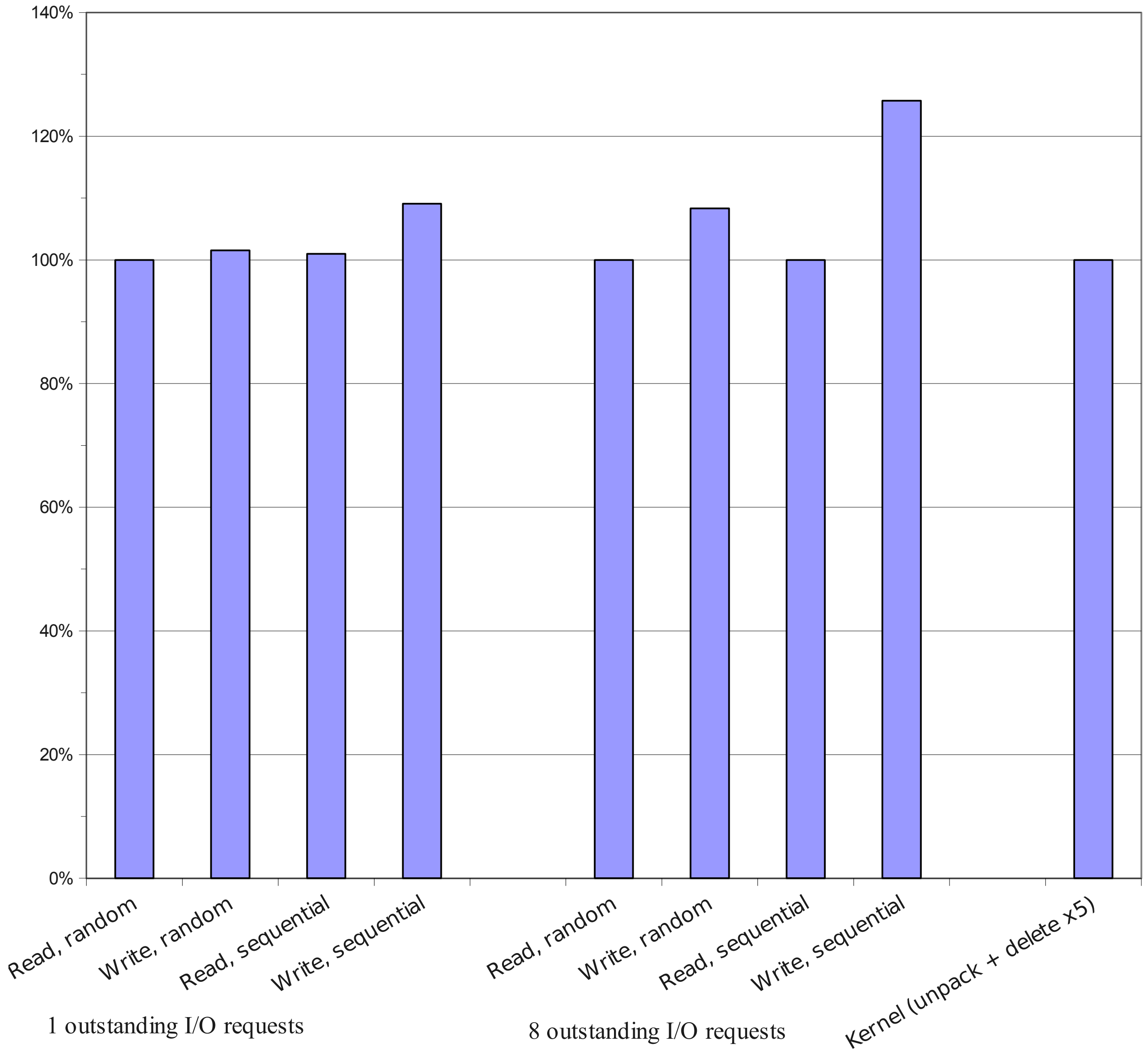


Figure 2: Performance of snapshots using Block Register relative to LVM

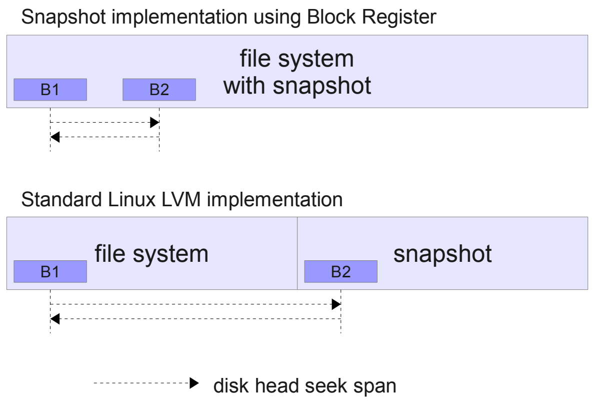

Snapshot implementation using Block Register
file system
with snapshot
B1
B2
Standard Linux LVM implementation
file system
snapshot
B1
B2
disk head seek span

Figure 3: Disk head seek span during copy-on-write operation

we designed the Block Register interface, we did not target performance improvements. This result comes as a pleasant side effect and leaves room for future work on snapshot placement optimizations.

6 Open Issues

The implementation of the Block Register interface raises several important problems. The best way to solve them is yet to be identified. In this section, we describe some of these problems.

So far, we considered only file systems located on a single partition, disk, or a hardware RAID array. Currently, BR interface will not work in case of an additional component, which performs non-linear mapping of disk blocks, placed between the file system and the user of the BR interface. An example of such a component is a software implementation of RAID 0.

Block reservation and query may result in additional I/O operations. Therefore, special care has to be taken, e.g. calling them asynchronously, to avoid deadlock when calling these functions while serving other I/O requests. Another solution is to mark I/O caused by reservation requests with special hints, so that kernel components can give them priority and guarantee that they will not be blocked.

Blocks that belong to reservations must be protected from some file system operations, such as defragmentation or accidental modifications by user-mode applications.

The block allocation algorithm depends on the file system and, therefore, may be sub-optimal in some cases because file systems are tuned for the most common case.

Finally, there are some implementation limitations, such as caching reservation block numbers in memory, which may become a problem for very large and fragmented reservations.

7 Future Work

We are going to investigate additions to the BR interface that will allow other components of the storage stack to interact with a file system without being incorporated into it. The goal of this interaction is to achieve flexibility and functionality similar to those provided by "monolithic" file systems, like BTRFS or ZFS.

Other kernel block layer components may benefit from the Block Register interface. We are planning to modify MD software RAID and DRBD, the Distributed Reliable Block Device [8], to use the Block Register interface for storing their metadata within the file system.

Another interesting opportunity comes from our implementation of reservations as regular files in a file system. Because of that, they could be accessed (with necessary precautions, such as taking measures to avoid the cache coherence problem) by user mode-programs. This allows, for example, to copy or archive contents of the file system along with snapshots using familiar utilities, like *cp* or *tar*.

We are also going to investigate changes to block allocation algorithms that allow kernel components to request reservations in the proximity of a specific location. This can improve performance of some operations, like copy-on-write or update of dirty block bitmap, by enabling proximal I/O [14].

8 Related Work

Some file systems, such as BTRFS, ZFS or WAFL, implement snapshots internally and also store original version of the modified data in their own free space [1, 6, 11]. Our approach uses a similar technique, however, it does not depend on a particular file system implementation.

Thin provisioning tries to achieve similar goals, however, once a block has been written to, a thinly provisioned block device does not have knowledge if this block contains useful data or not. In order to solve this problem, a file system has to support additional interfaces to the block layer, telling the block layer when blocks become free. Another problem with thin provisioning is in the duplication of file system functionality for accounting and allocating storage blocks. This results in additional levels of indirection and additional I/O for storing persistent allocation information. Conflicting allocation policies may result in less than optimal performance, e.g. blocks that file system allocator tries to allocate continuously might be allocated far from each other by a thin provisioning code. Examples of thinly provisioned block device implementations include Virtual Allocation and dm-thin [5, 12].

Linux provides *fibmap* and *fiemap* ioctl calls, which return information about the physical location of files on the block device to applications [9]. Wires et al. argue that growing number of applications can benefit from the knowledge of the physical location of file data [15]. They present a MapFS interface that allows applications to examine and modify file system mappings between individual files and underlying block devices. Block Register, on the other hand, provides applications and kernel components access to the free space of the file system. Block I/O to files using `bmap()` has been used in Linux to implement a file-backed swap device [7].

Parallel NFS (*pNFS*) is an enhancement of the NFS file system that exposes file layout to the clients in order to improve scalability [10]. While pNFS work is concentrated on large-scale networks, we argue that similar ideas can be successfully applied in a single-machine environment.

9 Conclusion

In this paper, we proposed a novel interface that extends usefulness of the file systems. Our interface is generic and does not rely on a specific file system implementation. It allows kernel storage stack components to share storage space with user files and use the file system to reserve storage space and locate data. On the example of storage snapshotting, we showed how our interface can be used for more flexible and efficient storage utilization without integrating snapshots inside file system's code. In addition, we demonstrate that the new interface may also help optimize storage I/O performance.

10 Acknowledgments

The authors express sincere gratitude to Edward Goggin for his insightful advice regarding this work.

References

[1] BTRFS project. http://btrfs.wiki.kernel.org.

[2] Device-mapper resource page. http://sources.redhat.com/dm/.

[3] dm-loop: Device-mapper loopback target. http://sources.redhat.com/lvm2/wiki/DMLoop.

[4] Linux software RAID. http://linux.die.net/man/4/md.

[5] Thin provisioning. http://lwn.net/Articles/465740/.

[6] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, and M. Shellenbaum. The zettabyte file system. In *Proc. of the 2nd Usenix Conference on File and Storage Technologies*, 2003.

[7] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 3rd edition, 2006.

[8] L. Ellenberg. Drbd 9 and device-mapper: Linux block level storage replication. *Proc. of the 15th International Linux System Technology Conference*, pages 125–130, 2008.

[9] M. Fasheh. Fiemap, an extent mapping ioctl. http://lwn.net/Articles/297696/.

[10] D. Hildebrand and P. Honeyman. Exporting storage systems in a scalable manner with pNFS. In *Proc. of the 22nd IEEE Conference on Mass Storage Systems and Technologies*, pages 18–27. IEEE, 2005.

[11] D. Hitz, J. Lau, and M. A. Malcolm. File system design for an NFS file server appliance. *Proc. of the Winter USENIX Conference*, pages 235âĂŞ–246, 1994.

[12] S. Kang and AL Reddy. An approach to virtual allocation in storage systems. *ACM Transactions on Storage*, 2(4):371 – 399, 2006.

[13] Michael Larabel. Linux 3.3 kernel: BTRFS vs. EXT4. http://www.phoronix.com, 2012.

[14] J. Schindler, S. Shete, and K.A. Smith. Improving throughput for small disk requests with proximal I/O. In *Proc. of the 9th USENIX Conference on File and Storage Technologies*, 2011.

[15] J. Wires, M. Spear, and A. Warfield. Exposing file system mappings with MapFS. In *Proc. of the 3rd USENIX conference on Hot topics in storage and file systems*, 2011.

“Now if we could get a solution to the home directory dotfile hell!”[11]

Making Linux NFS-mounted home directories useful again.

Andrei Warkentin
VMware, Inc.
andreiw@vmware.com

Abstract

Unix environments have traditionally consisted of multi-user and diverse multi-computer configurations, backed by expensive network-attached storage. The recent growth and proliferation of desktop- and single machine- centric GUI environments, however, has made it very difficult to share a network-mounted home directory across multiple machines. This is particularly noticeable in the context of concurrent graphical logins or logins into systems with a different installed software base.The typical offenders are the “modern” bits of software such as desktop environments (e.g. GNOME), services (dbus, PulseAudio), and applications (Firefox), which all abuse dotfiles.

Frequent changes to configuration format prevents the same set of configuration files from being easily used across even close versions of the same software. And whereas dotfiles historically contained read-once configuration, they are now misused for runtime lock files and writeable configuration databases, with no effort to guarantee correctness across concurrent accesses and differently-versioned components. Running such software concurrently, across different machines with a network mounted home directory, results in corruption, data loss, misbehavior and deadlock, as the majority of configuration is system-, machine- and installation- specific, rather than user-specific.

This paper explores a simpler alternative to rewriting all existing broken software, namely, implementing separate host-specific profiles via filesystem redirection of dotfile accesses. Several approaches are discussed and the presented solution, the Host Profile File System, although Linux-centric, can be easily adapted to other similar environments such as OS X, Solaris and the BSDs.

1 Introduction

The title of this paper has been kindly borrowed from a BYU UUG email [11], that originally inspired me to find a solution.

While some systems prefer a centralized approach to storing user-specific program configuration settings, Unix-like systems typically keep user preferences under a number of hidden files and directories kept in the root of the home directory. These hidden files and directories are distinguished by a leading dot in their name, and are thus generally called *dotfiles*. The historical behavior is to store read-once configuration to avoid hard-coded choices, for example `.profile` stores instructions for customizing the shell session, `.emacs` stores EMACS editor settings, and so on. Perhaps due to Unix-like systems being used in extensively networked, remote access and shared storage environments, dotfiles were never meant to be used as a persistent database for runtime-modified preferences, or as a locking mechanism, since that would have compromised the ability to log in concurrently into several machines with the same account stored on the network. There was always some degree of inflexibility, where the same configuration file would not work exactly as expected across different versions of software[1], yet the read-only nature of these files and conservative changes to setting schema meant you could come up with a valid subset of settings for all machines and operating systems in use.

The recent rise of “user-friendly”, graphical user interface-driven and largely PC-centric applications, however, has resulted in a number of popular software packages which are incompatible with the typical university or corporate heterogeneous environment based around network-mounted home directories and network logins. For recent software, such as the GNOME

[1] E.g., `.emacs` or `.vimrc` for the famous editors.

Desktop Environment, there are no simple solutions to making them work in an environment where network mounted home directories are a possibility, largely due to ever evolving and fluid configuration formats. Software distribution-specific minutiae and breaking changes across seemingly compatible major revisions of software, results in an inability to share the same configuration files. In environments where the user can make some guarantees as to software versioning and configuration compatibility, concurrent network logins are largely impossible due to the storage of writeable preferences and runtime data within dotfiles. Given that programs such as web browsers, WYSIWYG editors and desktop environments also manipulate configuration from within the applications, treating their dotfiles as writable databases, you immediately run into write collisions and configuration corruption when running concurrent sessions on several machines. Frequently, such software attempts to protect itself by creating lock files (see Figure 1), which results in denial of service condition during concurrent logins. Finally, the trend towards providing services via remote procedure call mechanisms, seen in software such as IPC buses and audio daemons like Enlightened Sound Daemon or PulseAudio, has resulted in storing named pipes, sockets, authentication cookies, and other unspeakable things within their dotfiles, with largely predictable results.

As an example of all of these, logging in to a Red Hat Linux RHEL 5 and SUSE Linux SLES11 system, concurrently or not, will result in a corrupted desktop on both hosts. Both of these systems use some variation of GNOME 2. A typical file system layout for GNOME 2 configuration can be seen in Figure 2. These issues are not new nor are they the only ones. Configuration file collisions across different versions of software, compounded by fragility and expressive verbosity for default auto-generated settings[2] has also been a bane for upgrading such software and its configuration successfully.

```
$HOME
└─ .mozilla
   └─ firefox
      └─ 2o1nz6r9.default
         ├─ lock
         ├─ .parentlock
         └─ ...
```

Figure 1: Partial structure of Mozilla Firefox configuration.

```
$HOME
├─ .esd_auth
├─ .gconf
│  ├─ apps
│  │  ├─ evolution
│  │  ├─ gnome-terminal
│  │  └─ ...
│  ├─ desktop
│  │  ├─ .gnome
│  │  └─ ...
│  └─ system
│     ├─ http-proxy
│     ├─ proxy
│     └─ ...
├─ .gconfd
│  └─ saved-state
├─ .gnome
└─ .gnome2
```

Figure 2: Typical structure of GNOME 2 configuration.

2 Related Work

The problem space itself is not particularly novel. Roaming User Profiles and Folder Redirection are two similar technologies available to Microsoft Windows users, which specifically deal with networked logins, remote home directories and operating system-specific profiles [8].

Roaming User Profiles (RUP) synchronize the local copy of the user profile, consisting of user directories and a user-specific registry hive holding configuration entries, with the remote server upon login and logout. Conflicts are resolved based on modification time, and the registry hive is treated as an opaque binary object, with no fine grained synchronization. The weakest spot of the entire mechanism is specifically relying on a synchronization mechanism and being largely unaware of what is being synchronized. Copying data back and forth imposes a noticeable penalty, on the order of minutes, for anything but the most trivial profile, forcing users to store their data locally. By not relying on network file access and locking semantics for concurrent access, there is always the potential for silent data loss caused by the "last modified wins" policy or

[2]As seen with GNOME 2, moving into GNOME 3.

by failures during synchronization. Additionally, RUP is not capable of distinguishing between synchronizing settings and synchronizing application data. A lack of fine-grained synchronization of registry keys, and a lack of state separation between user settings and host-specific user settings, results in inconsistent profile behavior across systems configured with a different set of applications or with different versions and revisions of Windows. These limitations are somewhat addressed by using completely separate profiles for Vista and newer versions, and by using the Folder Redirection mechanism to alias certain predefined user profile directories[3] to network locations, bridging the separate profiles to the degree that is possible and reducing the usability "threat" posed by synchronization.

A Roaming User Profile-like approach is not particularly feasible on Linux. Implementing such Windows semantics would mean using a local cache as the real home directory, which would then be synchronized under the user's credentials with the network copy on logging in and logging out. This relies on having a mechanism capable of resolving conflicts, and thus aware of all the possible applications, and having a complex conflict resolution policy. Getting this to work smoothly implies, at the very least, root access to all the machines affected, and some pretty serious source-level hacks[4] to get it all to work in a transparent and fail safe manner.

3 Solutions

Due to the limitations of a synchronization-based design, the proposed solution for the problems described above is dotfile access redirection. By redirecting dotfile accesses to local, host- or operating system- specific copies, we create separate configuration namespaces, thus resolving conflicts arising from concurrent accesses or versioning mismatches. A few approaches involving access redirection were investigated. The benefits of separate configuration namespaces are clear. Whatever the implementation details, a few design goals were kept in mind. In particular:

- All dotfile accesses are redirected.
- Accesses to dotfiles from each host are redirected to a special directory, specific for that environment.

[3]My Documents, etc.

[4]I, of course, mean changes to the authorization and authentication system, PAM.

```
$HOME
|__profiles
|  |__andreiw-lnx ...  Dotfiles
|  |  |                specific to host
|  |  |                andreiw-lnx.
|  |  |__.gnome2
|  |  |__.xyzzy
|  |  |__...
|  |__andreiw-vmw ...  Dotfiles
|  |  |                specific to host
|  |  |                andreiw-vmw.
|  |  |__.gnome2
|  |  |__...
|  |__andreiw-vm1 ...  Dotfiles
|     |                specific to host
|     |                andreiw-vm1.
|     |__.gnome2
|     |__...
|__.gnome2
|__.xyzzy ...  Visible here because
|              we are on host
|              andreiw-lnx.
|__...
```

Figure 3: View of a home directory on host andreiw-lnx, with dotfile redirection enabled.

- Mechanism and policy are separate.
- System configuration changes are minimal, ideally not requiring root access.

With this design there are, by default, no configuration collisions, conflicts and deadlocks. If a few of the dotfiles can indeed be safely shared, then a few strategic symbolic links can be employed. Learning from RUP, which suffers both from a poor mechanism and from mixing both mechanism and policy, the user has full control over how the redirection target directory is picked, whether it is by host name, IP or Ethernet address, or something completely different. The resulting mechanism is very flexible. See Figure 3 for a typical home directory layout with this design. Note that the visible dotfiles in the home directory root are really located inside the environment-specific directory.

One investigated approach was to override standard C library calls by loading a custom library with the `LD_PRELOAD` environment variable, similar to how the `fakeroot` package works [3]. The `LD_PRELOAD` environment variable signals the dynamic linker to load a specified library and attempt resolving symbols be-

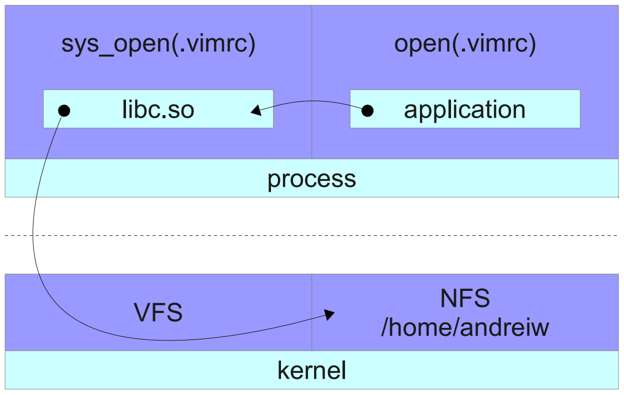


Figure 4: Regular `open()` path for an application.

fore loading all the libraries required by the loaded executable. Conceptually, calls like `open()`, `chmod()`, `unlink()` and the rest could be intercepted and modified to redirect accesses to dotfiles elsewhere (see Figures 4 and 5). In practice, however, this suffers from a few problems. The most significant issue is that it works only for executables dynamically linked to the C library. Statically linked software would bypass the redirection completely. Additionally, the method is very fragile and operating system dependent. The preloaded redirector would need to be tailored for the specific version of the C library in the system, as certain functionality is exposed and implemented differently, like the `stat()` and `mknod` family of routines[5]. The redirector would also need to properly handle both absolute and relative accesses to ensure isolation against malicious activity, and finally, would need to deal with the the `*at()`[6] variants, which operate relative to an opened file descriptor, by maintaining state about every opened file descriptor.

Another alternative implementation relied on `ptrace()`, coupled with process memory patching to intercept and redirect actual system calls, as `fakeroot-ng` does [3]. This solves the C library dependence issues along with being able to redirect calls made by statically-linked executables, yet at a cost of architecture dependence and severe performance penalties [5].

The long-term solution would be the FreeDesktop.org XDG Base Directory Specification [2], which separates

[5]GNU libc, for example, wraps `mknod()` with a versioned call to `xmknod()`, and `stat()` as versioned calls to `xstat()`, `fxstat()`, and `lxstat()`. See `/usr/include/stat.h`.

[6]`openat()`, `faccessat()` and similar calls were added in Linux 2.6.18, and are meant to address race conditions resulting from opening files in directories other than the current one [7].

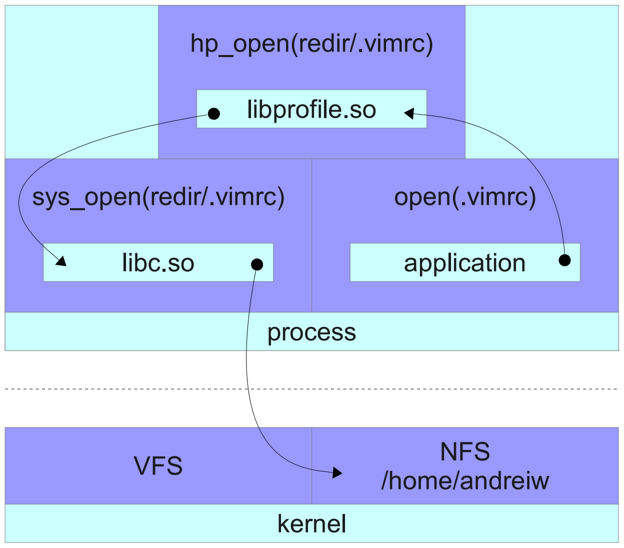


Figure 5: Modified `open()` path for an application with a file system redirector loaded using `LD_PRELOAD`.

application user data, caching-related, configuration and runtime-specific application data across four base directories, specified by environment variables as illustrated in Figure 6. In an environment where concurrent logins are expected, some or all of the base directories can be redirected to locations specific to the OS or host used, thus avoiding conflicts. However, all affected software needs to be rewritten to take this specification into account - a long and dire process with a nebulous future. In defense, many of the heavyweights such as GNOME, LibreOffice and the K Desktop Environment are fully behind the specification. Unfortunately, this does not help at all with older software and existing systems that do not follow the specification.

The solution presented in the remainder of this paper, the Host Profile File System, is built with the Filesystems in Userspace (FUSE) [1] framework and is considered superior to other possible solutions, both described above and not[7], because:

- It is transparent to the system.
- It does not rely on kernel changes.
- It does not require any changes to system services or programs.

[7]Like a redirfs [4]-based solution, with which a dotfile redirector could be implemented, yet would require additional kernel drivers and root access.

```
$HOME
 ├── $XDG_DATA_HOME ...   Application data
 │                        files.
 │    ├── app1
 │    │    └── datafile
 │    └── ...
 ├── $XDG_CONFIG_HOME ...  Application
 │                         settings.
 │    ├── app1
 │    │    └── configfile
 │    └── ...
 ├── $XDG_CACHE_HOME ...  Non-essential
 │                        data, cache.
 │    ├── app1
 │    │    └── cachefile
 │    └── ...
 └── $XDG_RUNTIME_DIR ...  Named pipes,
                           sockets, auth
                           cookies, locks.
      ├── app1
      │    └── lockfile
      └── ...
```

Figure 6: Simplified view of application data and settings under the XDG Base Directory Specification.

- It does not even require root access on the machine to enable.
- It is not based on fragile interfaces, or have machine dependence.
- The performance penalty is minimal.
- It is reasonably portable.

4 FUSE

FUSE is a Linux kernel driver that provides the necessary glue to have a fully user space implementation of a file system. This takes much of the complexity out of implementing a file system, due to not having to worry about complex locking and memory management interactions. FUSE also has a stable and OS-agnostic interface with consideration for backwards compatibility, meaning that maintainability is not an issue. FUSE has been ported to other Unix systems such as Solaris and OS X, which makes a FUSE-based solution viable in heterogeneous environments. Because a FUSE-based file system looks just like any other kind of VFS-provided file system, applications access it transparently. And most importantly, FUSE allows a user to mount their own private file system, meaning that neither system changes nor root access is necessary.

There is some overhead associated with the lack of direct `mmap()` semantics and with copying data between the FUSE driver and FUSE daemons. Tests with a pass-through FUSE file system have shown a 2% overhead [6]. A likely more realistic local test run, involving copying 22GiB of various software repositories, has shown a difference of under 9%[8] in total time spent, which is reasonable, given the end goal of accessing an NFS-mounted home directory, which wouldn't be used for I/O intensive workloads or large files anyway. FUSE overhead and performance has been critically analyzed elsewhere [9] with similar results.

5 The Host Profile File System

The Host Profile File System (HPFS) implements the previously described dotfile redirection design as a filter file system, mounted over the user's home directory. HPFS is implemented as a FUSE file system, and runs as a daemon with user privileges. The daemon forwards all file and directory accesses to the overlaid file system, and is able to do so by capturing the file descriptor of the user's home directory prior to the actual mount operation and by leveraging the `*at()` series of system calls to perform file I/O relative to an open file descriptor (See Figure 7). Without the `*at()` series of system calls this would not have been possible.

Incidentally, in a FUSE-based design we lose most of the complexity arising in a `LD_PRELOAD`-based solution, as all paths passed by FUSE are absolute (see Figure 8). Handling `readdir()` is slightly tricky, due to the need to hide existing dotfiles in the home directory root, and the need to account for dotfiles inside the redirection target directory. The current implementation does not virtualize the `structdirent` offset field, so applications relying on caching directory entries returned by the `readdir()` system call may see unexpected behavior. This limitation is not FUSE-specific, and would need to be addressed even if HPFS functionality were to be implemented as a VFS extension similar to GoboLinux GoboHide [10], which allows hiding files and directories from `readdir()`.

The redirection directory is passed to the HPFS daemon as a command line parameter, and is meant to

[8]31m51s versus 34m56s.

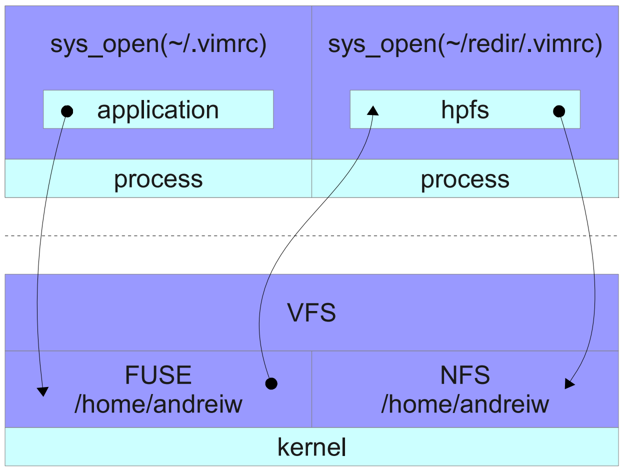


Figure 7: HPFS in action. HPFS is mounted over the already NFS-mounted /home/andreiw, hiding the original files from the user.

```
static int hp_open(const char *path,
  struct fuse_file_info *fi)
{
  /*
   * priv.fd contains the real $HOME
   * priv.redir_fd points to where
   * dotfiles are redirected to.
   */
  int fd = priv.fd;

  if (*path == '/')
    path++;

  if (!*path)
    path = ".";
  else if(*path == '.')
    fd = priv.redir_fd;

  fd = openat(fd, path, fi->flags);
  if (fd == -1)
    return -errno;

  fi->fh = fd;
  return 0;
}
```

Figure 8: `open()` handler for HPFS.

be derived by the support scripts. Two support scripts have been developed, one for the Bourne-Again Shell (`.bash_profile`), and one for the X11 Window System (`.xprofile`), to support redirection on both console and GUI logins. The scripts figure out the redirection path based on the host name, and enable the HPFS daemon if need be, while avoiding race conditions. The current version expects the system and CPU architecture to be the same everywhere, and a more complete version could thus be more intelligent in the choice of HPFS binary to run.

6 Conclusion

HPFS is fully functional and transparently usable in a real environment, and has been in active use for the past seven months across several machines. Further improvements would be improving `readdir()` virtualization, extended attributes support, filtering redirection by effective user ID (EUID), porting to other Unices and improving the surrounding ecosystem of scripts and helpers. Additionally, further performance impact measurements need to be done with HPFS and NFS. HPFS is open source, and the sources are freely available [12].

7 Acknowledgments

Many thanks to Alexandre Depoutovitch for his feedback on the paper, as well as Edward Goggin and Aju John for their support.

References

[1] Filesystem in userspace, 2011. http://fuse.sourceforge.net/.

[2] W. Bastian, R. Lortie, and L. Poettering. XDG Base Directory Specification. http://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html.

[3] Fakeroot-NG. Ptrace ld_preload comparison, 2009. http://fakeroot-ng.lingnu.com/index.php/PTRACE_LD_PRELOAD_comparison.

[4] F. Hrbata. RedirFS. 2007. http://www.redirfs.org/docs/linuxalt_2007/paper.pdf.

[5] Jörg Zinke. System call tracing overhead, 2009. http://www.linux-kongress.org/2009/slides/system_call_tracing_overhead_joerg_zinke.pdf.

[6] S. A. Kiswany, M. Ripeanu, S. S. Vazhkudai, and A. Gharaibeh. stdchk: A Checkpoint Storage System for Desktop Grid Computing. 2008. http://arxiv.org/pdf/0706.3546.pdf.

[7] Linux Programmer's Manual. openat(2), 2009. http://man7.org/linux/man-pages/man2/openat.2.html.

[8] Microsoft Corporation. Managing roaming user data deployment guide. August 2006. http://technet2.microsoft.com/WindowsVista/en/library/fb3681b2-da39-4944-93ad-dd3b6e8ca4dc1033.mspx?mfr=true.

[9] A. Rajgarhia and A. Gehani. Performance and extension of user space file systems. 2010. http://www.csl.sri.com/users/gehani/papers/SAC-2010.FUSE.pdf.

[10] L. C. V. Real. Gobohide: surviving aside the legacy tree, 2006. http://www.gobolinux.org/?page=doc/articles/gobohide.

[11] M. Torrie. [uug] Gnome vs KDE, 2009. http://uug.byu.edu/pipermail/uug-list/2009-March/002134.html.

[12] A. Warkentin. Host Profile File System source repository, 2012. https://github.com/andreiw/HPFS.

Improving RAID1 Synchronization Performance Using File System Metadata

Reducing MD RAID1 volume rebuild time.

Hariharan Subramanian
VMware, Inc.
hari@vmware.com

Andrei Warkentin
VMware, Inc.
andreiw@vmware.com

Alexandre Depoutovitch
VMware, Inc.
aldep@vmware.com

Abstract

Linux MD software RAID1 is used ubiquitously by end users, corporations and as a core technology component of other software products and solutions, such as the VMware vSphere® Storage Appliance™ (vSA). MD RAID1 mode provides data persistence and availability in face of hard drive failures by maintaining two or more copies (mirrors) of the same data. vSA makes data available even in the event of a failure of other hardware and software components, e.g. storage adapter, network, or the entire vSphere® server. For recovery from a failure, MD has a mechanism for change tracking and mirror synchronization.

However, data synchronization can consume a significant amount of time and resources. In the worst case scenario, when one of the mirrors has to be replaced with a new one, it may take up to a few days to synchronize the data on a large multi-terabyte disk volume. During this time, the MD RAID1 volume and contained user data are vulnerable to failures and MD operates below optimal performance. Because disk sizes continue to grow at a much faster pace compared to disk speeds, this problem is only going to become worse in the near future.

This paper presents a solution for improving the synchronization of MD RAID1 volumes by leveraging information already tracked by file systems about disk utilization. We describe and compare three different implementations that tap into the file system and assist the MD RAID1 synchronization algorithm to avoid copying unused data. With real-life average disk utilization of 43% [3], we expect that our method will halve the full synchronization time of a typical MD RAID1 volume compared to the existing synchronization mechanism.

1 Introduction

RAID arrays have gained a wide popularity over the last decade. By maintaining data redundancy, they provide a cheap solution for data availability, fault tolerance, and scalability in the event of hardware and software failures [2]. Some of the popular RAID implementations include RAID1, which maintains two or more identical copies of the data over physically separate storage devices, and RAID10 which augments RAID1 with data striping. RAID arrays can be implemented at hardware level, e.g., RAID hardware adapters, as a software product, e.g., Linux MD RAID driver, or as a part of more robust and complex applications, e.g., VMware vSphere® Storage Appliance™ (vSA).

In RAID1, the loss of one copy of the data due to a component failure (e.g., hard drive) is typically followed by an administrative operation, that replaces the failed component with a new one. As part of this, all data needs to be copied (synchronized) to the newly added component. This restores the data redundancy and fault tolerance characteristics. However, storage size has grown exponentially over the recent years, while data access latency and bandwidth improvement rate is significantly smaller. For large arrays, this results in hours during which the array functions below its optimal performance and reliability. Before the synchronization is complete, additional failures may result in data loss and/or unavailability. Therefore, it is very important to minimize synchronization time. In our work, we advocate a new, easy to implement method that reduces the amount of data that needs to be synchronized and consequently decreases the synchronization time.

We implemented our method in the Linux MD software RAID1 driver and integrated it with the VMware vSA product. The key observation behind our method

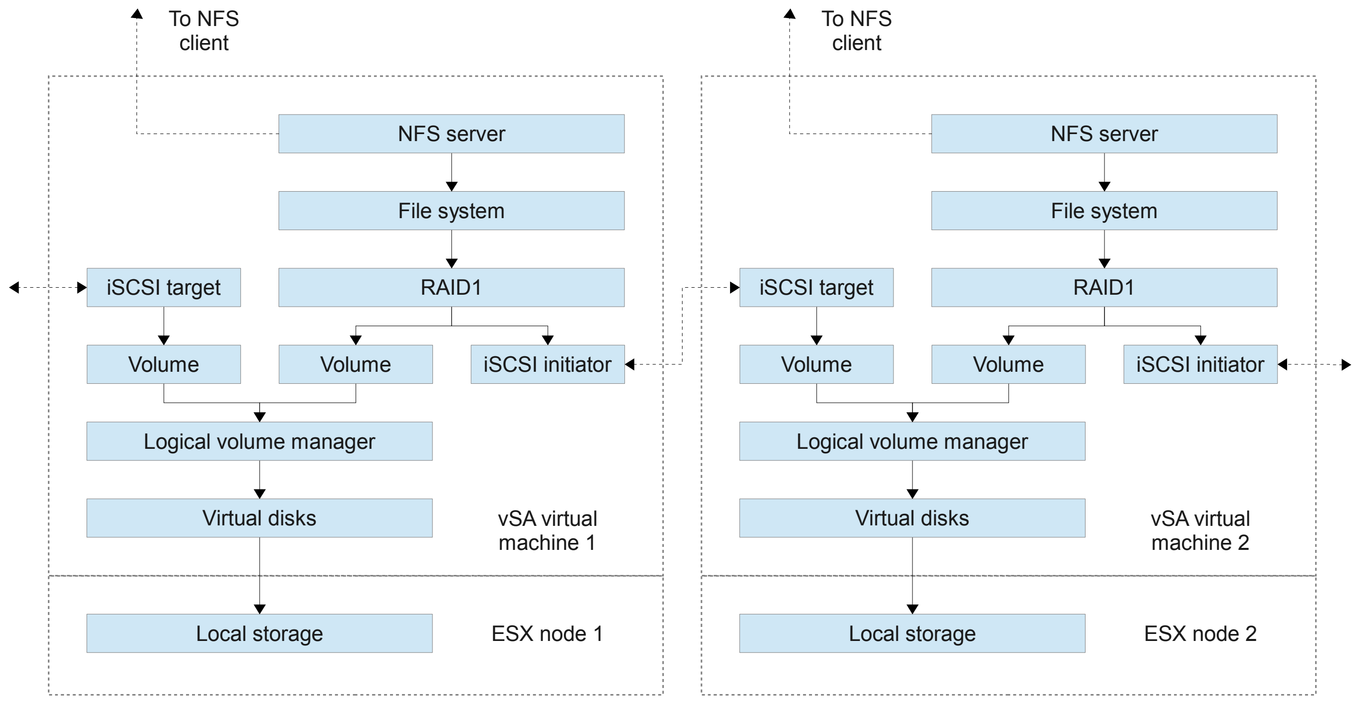


Figure 1: vSA architecture overview.

is that, since the synchronization is in the block device layer, the MD RAID1 mirroring driver needlessly synchronizes unallocated file system blocks, in addition to blocks containing useful data. We investigated three different approaches of transferring the unused block list from the file system to the synchronization algorithm. All three approaches populate the unused block list and change synchronization logic to take a list of unallocated blocks into account. They require minimal changes to the existing MD RAID1 control flow.

The approaches differ in how the unused data blocks are obtained. The first approach leverages user space file system utilities (specifically, `e2fsprogs` in an EXT4 file system) to obtain a list of unused file system blocks. A user space helper application uses the list to construct a bitmap representing the blocks that are currently in use by the file system. It then passes the bitmap to the MD RAID1 kernel driver, where it is used to skip copying the disk blocks that are not in use by the file system. The second approach continuously tracks the blocks not in use by the file system by intercepting discard requests to the block device (called `REQ_DISCARD` in the Linux kernel). The final, hybrid approach, avoids the overhead of maintaining in-memory unused block state of the previous approach, while also taking advantage of a user space helper, albeit in a way which is much simpler and independent of file-system implementation. It utilizes the Linux kernel `FITRIM ioctl` to force the file system to send `REQ_DISCARD` requests for unused blocks.

2 vSphereStorage Appliance

Linux MD RAID is not only widely used by the end users, it is also an important building block for larger and more complex software products. One such example is the vSphere® Storage Appliance™ (vSA) developed by VMware and released as a commercial product last year [1]. This software provides shared storage benefits such as reliability, availability, and shared access without needing to buy complex and expensive specialized shared storage hardware. The high level architecture of the vSA storage stack is presented at Figure 1. vSA consists of two or more hardware nodes with ESX server installed, running a virtual machine with Linux and vSA software.

The vSA software exports data to clients through the NFSv3 protocol. EXT4 file system is used to store the user data. In order to provide reliability and availability, the Linux MD RAID driver is utilized to duplicate data between the hardware nodes. Access to the remote node storage is done through the iSCSI protocol over the network. Data synchronization speed between nodes is often limited by the 1Gbit network link. Because vSA targets the small and medium business (SMB) market, cost savings is an important criteria. Therefore, an upgrade to a 10Gbit network is often undesirable due to the high cost involved in replacing not only network cards but routers, switches, and other infrastructure components.

In this environment, optimization of data transfers is very important. Without our mechanism, it takes approximately 5 hours to synchronize a typical 2TiB vSA data volume after one of the nodes was replaced. During this time, the vSA functions in degraded mode with decreased performance and without fault tolerance. With our mechanism in place and an average storage utilization of 43% [3], this critical time is reduced to slightly more than 2 hours. We believe that not only vSA, but other projects and products involving storage replication would benefit from our mechanism.

3 Control flow

Our method introduces changes to the MD RAID1 array algorithm only during array synchronization. Therefore, there is no additional run-time overhead for any array functions in regular or degraded modes compared to vanilla Linux MD driver. Our mechanism comes into play whenever RAID synchronization is triggered automatically or upon user request.

Upon receiving such request, depending on whether a full or incremental synchronization is required, the vanilla Linux MD driver copies either all blocks to the new block device, or only those blocks marked in the internal write-intent MD bitmap as changed since the time the array was healthy and fully synchronized. If a write request arrives while synchronization is in progress, the MD RAID driver pauses the synchronization process until the write operation is complete on all disks. This prevents race conditions between synchronization and regular writes.

In all of our approaches, any IO error encountered by the RAID block devices used by MD is reported to a user mode helper program. In case of the vSA, this is a Java-based application referred to as the vSA business logic software. This helper application is responsible for handling detected IO errors, detecting that a previous hardware failure has been rectified and re-introducing previously failed block devices back into a degraded MD RAID volume. Thereby, this helper program controls the point at which MD synchronization process starts.

In our mechanism, upon receiving a synchronization request, the MD driver starts synchronizing blocks marked in the write-intent bitmap to all degraded devices just as described above, but it queries an additional in-use/unused block list. Depending on the actual implementation, this list might come from a user-space helper program or from handling an `FITRIM ioctl` issued to the file system layered on top of the RAID volume. With the list of unused blocks, the MD driver can proceed to synchronize only blocks that are both marked as in-use and marked in the write-intent bitmap.

While synchronization is in progress, previously unused blocks may become allocated again, but this is equivalent to the concurrent writes and synchronization case above. In this case, before any attempt to read such a block, a write must be issued to initialize the block data. This write will always be propagated to all devices during synchronization, as previously described. If no write was issued to a previously unused block before reading, an application cannot depend on the read contents of such a block, thus there should be no need to synchronize it.

There is one important difference in the behavior of the vanilla Linux MD driver and our modified MD driver. If a block has been reported as unused to our synchronization mechanism, is subsequently allocated and read from several times without being written to, the vanilla Linux MD driver would return the same data on every read. With our mechanism in place, different reads to uninitialized data might return different data depending on the device the read operation was dispatched to. However, all POSIX-compliant file system returns zeroes for reads to unwritten parts of a file. We can not think of any correctly written code that depends on such behavior, but if such software exist, it should not be used with our mechanism.

4 Metadata snapshot

The first approach introduces no run-time overhead to the I/O and data synchronization paths of the RAID driver, and is only involved after a hardware failure is detected and rectified. The approach involves the following steps,

- Obtaining list of unused blocks from the file system.

- Representing the data in an in-use bitmap.
- Injecting the in-use bitmap to MD using a new `ioctl` call.

The list of unused blocks is obtained out-of-band by examining on-disk file system structures. For vSA, this involves examining the EXT4 file system on the MD RAID device. `dumpe2fs`, a common system utility from the `e2fsprogs` package, is used to query file system metadata and statistics for the EXT2, EXT3 and EXT4 file systems. The vSA business logic software, which controls disk creation, failure detection and other management operations, parses `dumpe2fs` output to generate a list of unused blocks from the vSA file system. The list contains ranges of blocks in units of file system block size. The information is presented for each file system block group as a comma separated list of unused block ranges.

The control flow for performing MD re-synchronization is presented at Figure 2.

The data obtained is used to populate a bitmap representing blocks that are currently used by the file system. Instead of extending the existing in-memory write intent state, a separate bitmap was used. The write-intent bitmap divides the disk into chunks, and keeps track of which disk chunks have modifications that need to be synchronized to disks that are currently unavailable. A separate bitmap enables us to pick a different granularity for tracking used blocks, with the intent of investigating optimum granularity for tracking such information. This flexibility is potentially worth the additional complexity and memory overhead of maintaining a new bitmap.

The in-use bitmap divides the MD device into equal sized chunks. A chunk is always larger than the size of a file system block, and would ideally match the granularity of an individual synchronization I/O. The bitmap comprises a series of pages, each covering 32768 chunks. With the additional in-use bitmap, the modified MD RAID1 synchronization algorithm determines if the synchronization I/O being performed can be skipped. The actual changes to the main routine involved are minimal.

The in-use bitmap is injected into the MD RAID1 driver using a new `ioctl` mechanism. The MD RAID1 driver is modified to accept a bitmap solely while synchronization operation is ongoing. Once the synchronization operation is complete, the bitmap is automatically cleared.

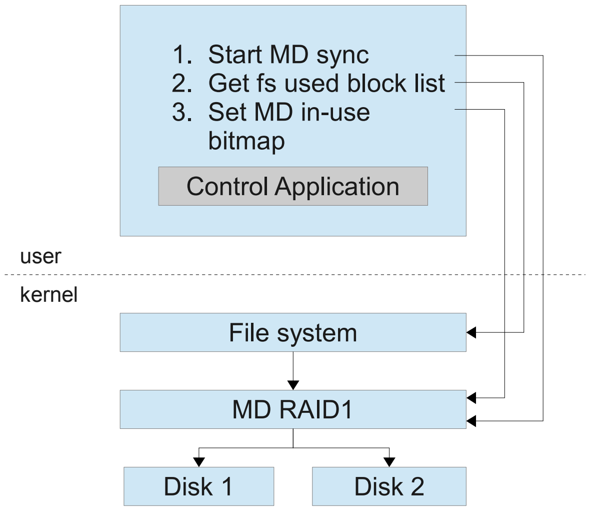


Figure 2: Control flow for MD resync with the metadata snapshot approach.

This avoids any data consistency issues resulting from possible malicious use of the interface, and follows good security practices. Since the vSA business logic software completely controls when an MD synchronization operation occurs, it would not be possible for an out-of-date bitmap to be applied, avoiding possible data corruption issues.

5 Discard request tracking

The second approach of exposing file system unused block information to the MD driver relies on an already existing set of functionality present in the Linux kernel. The Linux block I/O subsystem provides a way to notify hardware that a range of blocks is not in use anymore by upper layers. A file system might thus send `REQ_DISCARD` requests when a file is deleted. The original aim of this functionality was to enable more intelligent wear-leveling mechanisms for solid-state storage, yet it is used in implementing thin-provisioned SCSI LUNs and provides the data we need to avoid synchronizing unused blocks. The method by which unused block ranges are sent into the block layer is a `REQ_DISCARD` I/O request. Just like any other I/O operation, it consists of a start block and the number of blocks affected, and arrives at the same MD I/O dispatch routine handling regular accesses. This implies that the MD driver has to keep track of blocks being marked as used and unused. A block is marked as being unused when a `REQ_DISCARD` request for it arrives, and is marked as being in use on a write request. In practice, there are real-life restrictions that limit the usefulness of an approach based purely on live tracking of `REQ_DISCARD` requests, as we shall see.

The first idea that comes to mind is to track the in-use/unused state in the same memory and disk structures already used to keep track of write intent state. Using the same memory structures has the implication of no extra memory overhead[1], and the bit twiddling is done at the same code location and under the same locks, meaning that the run-time overhead is the least of other possible approaches. The new state is persisted across reboots by extending the on-disk write intent bitmap with an in-use/unused bit. Of course, the RAID1 I/O dispatch routine also needs to be changed to handle `REQ_DISCARD` block I/O by marking the affected blocks as unused and finishing the I/O.

Unfortunately, in a real life setting, where each bitmap chunk is significantly larger than the typical I/O size[2], this approach gives poor results. A typical discard I/O request acts on a range of kibibytes, usually with a granularity of 4KiB or so, and such requests are basically lost without keeping track of discard requests at a finer granularity than the bitmap chunk. We could maintain a separate bitmap, say at 4KiB granularity, but the memory overhead of such a bitmap are enormous at typical capacities[3]. The disk persistence is an additional problem. Extending the write-intent bitmap carries the same granularity issues present with reusing the in-memory state, while maintaining a separate more granular bitmap would result in additional disk I/Os, lowering the write performance. Additionally, the change in internal RAID metadata brings upgradeability implications where the existing structures need to be replaced by larger ones.

Support of upgrade from an earlier version of MD metadata to a newer one aware of the in-use/unused blocks, means we need to have some method of registering blocks not in use by the file system with MD. Fortunately, there is no need for a new interface to achieve this. The `FITRIM` file system `ioctl` causes the file system to send `REQ_DISCARD` I/O requests for all unused blocks. At the current moment, file systems do not persist in-use/unused knowledge across remounts, thus `FITRIM` is a sufficient method to initialize our MD in-use/unused state. Given the issues with on-disk persistence, we might as well rely on `FITRIM` to initialize our in-memory state after mounting the file system.

To address the issues surrounding tracking `REQ_DISCARD` requests, we can switch from a bitmap to a data structure that makes it more convenient to store unused block ranges. Such a structure is a special interval tree that coalesces overlapping and sequential ranges, implemented using a red-black tree. The time complexity is $O(\log n)$, especially if you assume the total number of intervals to be pretty low. The changes to MD to enable the use of discard ranges as an optimization are minimal, and the synchronization overhead can be mitigated by employing a relativistic red-black tree algorithm [4] instead of the default Linux rb-tree implementation. The Achilles heel of this approach, however, is the worst-case memory overhead. An access pattern of small discards, such as interleaved 4KiB accesses, will result in an overhead 32 times worse than the equivalent overhead of using a bitmap with 4KiB granularity. Some of this can be mitigated by enforcing a minimum granularity and pruning the range tree based on memory pressure, but all of this added complexity basically nullifies the original advantages of storing ranges.

6 Hybrid Approach

The previous approach to tracking `REQ_DISCARD` requests relies on the assumption that it is typical to expect discard requests over the normal lifetime of an on-line RAID array with a mounted file system. However, that is not the case. While a file system like EXT4 certainly could be mounted in a way that will generate discard requests for every file erase, that was generally avoided in older kernels due to discard requests being processed synchronously and acting as barriers, impacting I/O performance. If we consider that we always issue `FITRIM` during RAID synchronization, then we just have to handle `REQ_DISCARD` requests during synchronization time. At synchronization time we can employ simple logic to track consecutive discards, marking the affected chunks in the separate bitmap as not in use. This is effectively the combination of the first approach with first idea considered in the previous section, without the on-disk persistence changes and with a more intelligent and restrictive algorithm for marking chunks as being not in use.

[1]We end up stealing a bit from the per bitmap-chunk field to describe the new in-use/unused state, which has has the largely irrelevant implication of reducing the number of concurrent I/O per block chunk from 16383 to 8191.

[2]For a 100MiB disk, the default bitmap chunk is 4KiB, while for a 10TiB disk, the bitmap chunk size is usually 64MiB.

[3]A 10TiB disk would need a 320MiB bitmap.

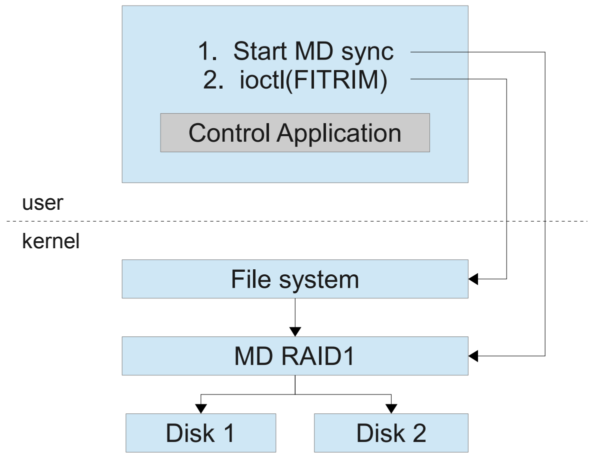


Figure 3: Control flow for MD resync with the hybrid approach.

It is better than the previous approach to tracking discards because it covers the case of out-of-order small discards that ultimately add up to a larger unused block—we let the file system handle coalescing and ordering these at `FITRIM` time. Using a separate bitmap, with a granularity smaller than the write intent bitmap, improves the case where the write intent chunk size is too large to effectively use this algorithm.

The control flow for performing MD re-synchronization is presented at Figure 3.

7 Discussion

Each of the above approaches to obtain the list of unused blocks has its own advantages and disadvantages. The main advantage of using a file system metadata is its independence on the Linux kernel version deployed on the system. Certain distributions, such as all versions of Red Hat Enterprise Linux or SUSE Enterprise Linux, except the latest SLES11SP2, do not support `REQ_DISCARD` or `FITRIM` functionality. This makes usage of a user space helper the only way to get the list of unused blocks, given that back porting changes to the kernel block I/O subsystem and file systems is no trivial matter. This approach does not depend on kernel version, which makes it applicable to most of currently deployed systems. Another advantage is that more functionality remains in user space, which makes the code more reliable and easier to debug. The disadvantage of this approach is its dependence on user mode utilities which are not standardized, which are file system-dependent, and which may change their output format in future versions.

The advantage of discard request tracking is in the transparency of the approach. There is no need for anything special to occur to make use of this functionality to improve synchronization, other than ensuring that the file system generates discard requests for erased files. The disadvantage of live tracking is in its memory consumption. The bitmap based approach, with a sufficiently small granularity to achieve effectiveness, would consume an additional 300MiB of kernel memory. A range-based approach would scale the memory usage, but become effectively unbounded with severe file system fragmentation. Therefore, we would prefer the hybrid approach.

The advantage of obtaining the list of unused blocks through `FITRIM` `ioctl` command is in the use of a standard interface, recommended for use by all general purpose file systems. In current kernels, the `FITRIM` interface supported by EXT3, EXT4, btrfs, xfs, ocfs, and others. We would, however, like to see better defined documentation on `FITRIM` behavior. For example, to the best of our knowledge, the ordering of blocks to be discarded is not described, nor are any guarantees regarding the blocks reported for file systems that persist discarded block information. This allows some file system implementations to report unused blocks out-of-order, or to only report changes since the last `FITRIM` even across remounts, both of which will negatively affect the hybrid approach.

8 Conclusion

In this paper, we proposed a new method for RAID array synchronization. This method requires minimal changes to the existing Linux kernel code. It reduces synchronization time by a factor of two in the common case, thus improving reliability and performance of the RAID array.

We investigated and compared different implementation methods and highlighted their strong and weak points. The optimal granularity of the unused block bitmap needs to be further investigated, and we are planning to extend the synchronization improvements to other RAID levels provided by the MD driver. Furthermore, the `FITRIM` kernel interface needs to be better defined to address the ordering and persistence concerns noted in the discussion section above.

9 Acknowledgments

Many thanks to Edward Goggin and Aju John for the peer review and support of this research.

References

[1] VMWare vSphere Storage Appliance Technical Deep Dive, 2011.

[2] Conference on File and Storage Technologies. *RAID: high-performance, reliable secondary storage*, 1994.

[3] Conference on File and Storage Technologies. *A study of practical deduplication*, 2011.

[4] P. W. Howard and J. Walpole. Relativistic red-black trees. December 2010.

Out of band Systems Management in enterprise Computing Environment

Divyanshu Verma
Dell India R&D Centre.
Divyanshu_Verma@Dell.com

Srinivas G Gowda
Dell India R&D Centre.
Srinivas_G_Gowda@Dell.com

Ashokan Vellimalai
Dell India R&D Centre.
Ashokan_Vellimalai@Dell.com

Surya Prabhakar
Dell India R&D Centre.
Surya_Prabhakar@Dell.com

Technical Reviewers: {Ramkrishna_Rama, Jagadeesh_Raju, Neti_Prasad}@Dell.com

Abstract

Out of band systems management provides an innovative mechanism to keep the digital ecosystem inside data centers in shape even when the parent system goes down. This is an upcoming trend where monitoring and safeguarding of servers is offloaded to another embedded system which is most likely an embedded Linux implementation.

In today's context, where virtualized servers/workloads are the most prevalent compute nodes inside a data center, it is important to evaluate systems management and associated challenges in that perspective. This paper explains how to leverage Out Of Band systems management infrastructure in virtualized environment.

1 Introduction

Out Of Band systems management is the de-facto capability in enterprise computing world to manage physical servers inside a data center. It provides remote administrators the ability to connect, gather server information and at the same time control the servers even in non-OS environment.

Today's enterprise computing environment is dominated by virtualization technology which allows one single server to be used by many virtual machines. In this paper we look at how we can make Out Of Band System management utility scale up to this new challenge of virtualization. We talk about the ways in which an existing systems management utility can be used to handle virtual machines. Later on we also discuss some of the security challenges that may be posed while trying to implement this method.

1.1 Acronyms and Abbreviations

OOB - Out Of Band Management

VMS - Virtual Management Software

VM - Virtual Machine

VMM-I - Virtual Machine Management Interface

BMC - Base Board Management Controller

IPMI - Intelligent Management Platform Controller

LAN - Local Area Network

OOB-I - Out Of Band Management Interface

RMCP+ - Remote Management Control Protocol.

TOPT - Time-Based One-Time Password Algorithm

DES - Data Encryptions Standard

2 Evolution and Design of Managing Virtual Machines using Out Of Band Channel

In a typical virtualization setup, Virtual Machines and Physical servers are managed and controlled using management software. In this paper we refer to this management software as Virtual Management Software (VMS). This VMS provides advanced features such as High Availability and Live Migration. All the physical servers in data center are connected to VMS over Ethernet. In summary, VMS at an application level manages Virtual Machines (VMs) using a hypervisor that is deployed on each of the physical systems. VMS usually dedicates a

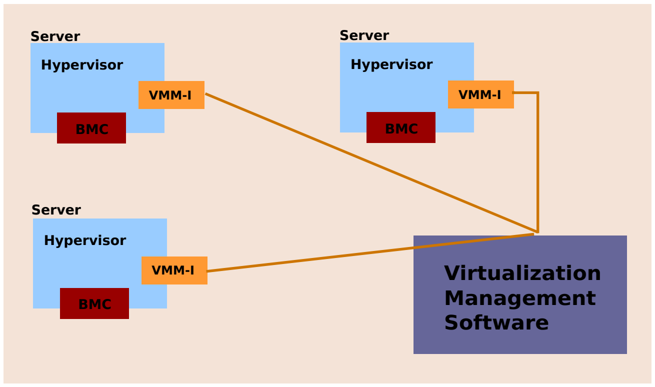


Figure 1: Current Architecture

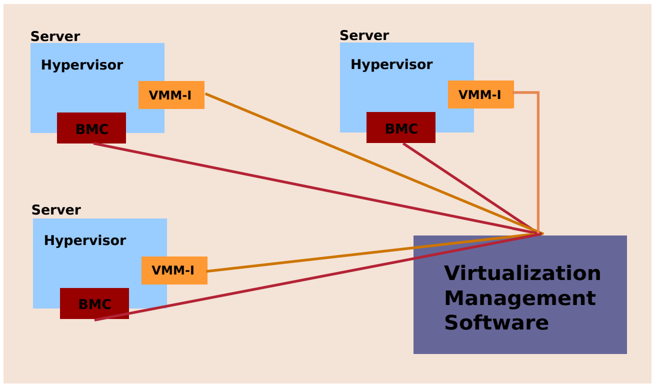


Figure 2: Proposed Architecture

system interface on each of the physical servers to communicate with the hypervisor. This interface is called as the VM Management Interface (VMM-I) and is shown in Figure 1.

The physical servers used in data centers today are mostly Enterprise grade equipped with Out Of Band Systems Management capability. One of the widely used OOB Management implementations has a Baseboard Management Controller (BMC) embedded inside the physical server. BMC supports the industry-standard Intelligent Platform Management Interface (IPMI) specification, which enables users to configure, monitor, and recover systems remotely. Each of these physical servers hosts a base hypervisor and VMs on top of it. Each of these hypervisors are connected to VMS through VMM-I.

2.1 Proposed Solution

Figure 2 depicts the proposed model, wherein we are using the Out Of Band Management Interface (OOB-I) as a secondary interface to manage the VMs using the hypervisors. Similar to VMM-I, each OOB-I is also connected to VMS. VMS uses OOB-I channel to communicate with BMC by using IPMI over LAN. IPMI OverLAN is a functionality that provides remote machines the ability to send IPMI messages over Network to BMC using UDP protocol (IPv4). The UDP packets are formatted to contain IPMI request/response messages with IPMI session headers and Remote Management control Protocol (RMCP+) headers (IPMI v2.0 Spec). For IPMI OverLAN to work, BMC needs to have a dedicated Management Network interface associated to it.

2.2 Architecture of Proposed Data Path to establish OOB communication channel between VMS and hypervisor

When VMS intends to send a message to hypervisor, it will encode the message in an IPMI message format and send this message using RPMCP+ protocol over OOB-I channel. Once sent to BMC, these messages are picked up by the hypervisor and decoded back to the original format (VMS message). Similarly when the hypervisor needs to send data to VMS, it will send it to BMC and VMS reads these messages over OOB-I.

In this solution we implemented two Buffer queues in BMC, one to hold the data that VMS sends to hypervisor and the other to temporarily store data sent from hypervisor to VMS. To access these buffers we need four new sets of IPMI commands to read and write the respective queues. Figure 3 gives an overview of the stack

Figure 3 shows the different modules that are involved in enabling the OOB-I channel between VMS and hypervisor. When VMS needs to communicate with the hypervisor over OOB-I, VMS would encode the message into an IPMI packet as payload and send it over the OOB-I. In this solution the four IPMI commands associated with the BMC Buffer Queues are implemented as OEM Commands. The idea is to carry these VMS/hypervisor messages as payload using IPMI Commands, so the encoding and decoding logic would be confined to the payload and would keep the IPMI/BMC changes at minimal. This design avoids decoding of every message VMS/hypervisor sends as a separate IPMI message.

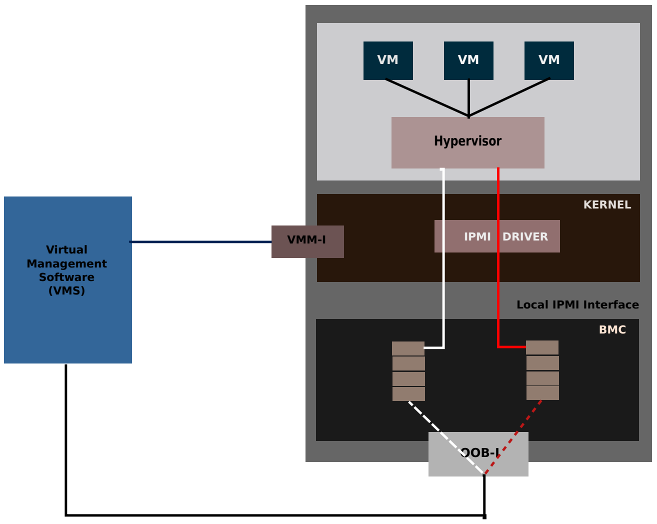


Figure 3: Proposed stacks overview

2.3 Low-level Details of the Implementation

Consider a scenario when VMS decides to communicate with a hypervisor. VMS would encode the message that needs to be sent to the hypervisor in IPMI format and use the new IPMI commands to send this message to the appropriate BMC. For VMS to communicate with hypervisor it needs to know the IP address of OOB-I apart from VMM-I IP. VMS message structure that needs to be sent to hypervisor.

```
struct vms_message {
cmd_id  cmd,   //actual command id
vm_id vm_name, //name of virtual machine
ip_addr oob_i, //IP address of OOB-I
ip_addr vmm_i, //IP address of VMM-I
.              //meta data
.
};
```

API to convert vms_message into raw IPMI format

```
char *ipmi_payload  convert_to_raw_ipmi\
_data(struct vms_message *vms_data) {
// returns *ipmi_payload - VMS message
// converted into IPMI format
};
```

API to send the vms_message to BMC over OOB-I using RMCP+ protocol. IPMI over LAN requires user authentication.

```
char ipmi_send_message (char *ipmi_\
payload, structipmi_system_interfac\
e_addr bmc_addr )\
{
  .
  .
 char *user_name,
 char *passwd,

  Request message
  [NetFn]
  [CMD ]   -  [COPY_VMS_TO_BMC_BUFF_1]
  [Payload ] - [ipmi_payload]
  char *ipmi_payload
  .
  .
};
```

BMC on receiving the IPMI message from VMS, saves the payload in Buffer Queue1 (Figure 3). hypervisor periodically check for any VMS requests in Buffer Queue1. Any messages in this Queue are picked up by hypervisor using the new IPMI command. The next task is to decode the payload message from BMC, which is exactly the reverse of the encoding mechanism that was carried out in VMS.

IPMI command to read message from Buffer Queue1

Request

```
[NetFn] [Cmd] [OEM_ID] [READ_BMC_BUFF\
_QUEUE1]
```

Response

```
[Completion Code ] [ Payload ]
```

API to read the vms_message to BMC (Queue1) over local IPMI interface.

```
char *ipmi_payload  ipmi_get_message (\
struct ipmi_system_interface_addr bmc_\
addr )   {
  .
  .
  Read Payload from BMC-Queue1
  [NetFn]
  [CMD ]   -  [READ_BMC_BUFF_ QUEUE1]
   .
   .
   Response message
   [Completion Code ]
   [ IPMI_Payload ]
   return ipmi_payload
};
```

API to decode the ipmi_payload into original VMS message format

```
struct vms_message *vms_data  \
convert_to_vms_format (ch\
ar *ipmi_payload  )
```

3 Out of Band Systems Management with Virtualization - Challenges

Out of band management of VMs brings additional challenges to the table, some of which are discussed below. In section 3.1 we discuss how to secure the OOB-I channel end to end, i.e. starting from VMS to hypervisor. Then in section 3.2 we discuss how to ensure that only a rightful authority can initiate state changes to the virtual machines. Both scenarios will be explored using a Linux/hypervisor case study.

3.1 Securing the OOB-I communication Channel

The complete communication channel for the proposed solution comprises of multitude of components, as illustrated in Figure 4.

- OOB-I
- BMC layer
- Linux Kernel Layer
- Exposed Userspace IPMI device
- Userspace hypervisor software

In the above mentioned components, OOB-I is generally secured using RMCP+ protocol. BMC access is typically controlled by a password authentication. Linux kernel space is off-limits to user space processes and hence is considered secure. In this implementation the modified hypervisor software uses the IPMI Device interface (`/dev/ipmi`) to obtain the information from BMC. So any stray read/write to BMC through this device can reveal the payload.

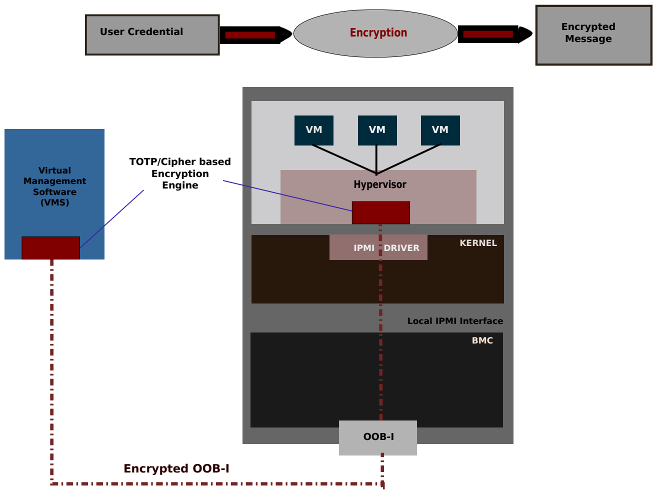


Figure 4: TOTP-Cipher Encryption

We used “time-based one-time password Algorithm” (TOPT, RFC6238) signature engine to generate a secret key within VMS. This secret key is used as auxiliary input to a symmetric key algorithm based cipher (DES/AES) to encrypt the outgoing payload. This encrypted payload is then transferred through OOBI. A stray read/write to BMC cannot decode the payload since it does not have the TOPT signature.

The hypervisor layer also has the TOPT signature engine with same algorithm, which it uses internally to decrypt and verify the payload it received from BMC. Once the hypervisor has decrypted the payload, it separates the user credential, associated VM-ID and control message.

3.2 Protection against accidental state change

In a traditional virtualization setup, the hypervisor has all the authority to carry out management tasks on all virtual machines. This allows the hypervisor to shut down, close or change the state of VMs without any restriction. However, with the growing level of (mission-critical) work load running on VMs, it is important that there should be additional security layer to help avoid any accidental state change in VMs. We investigated this problem and proposed a *request-challenge-verify* interface between hypervisor and VMs.

In the solution, whenever a state change request is given to VM, the request is verified against an authorization list. Here it checks that if state change request coming from an authorized process or not. If the request is valid,

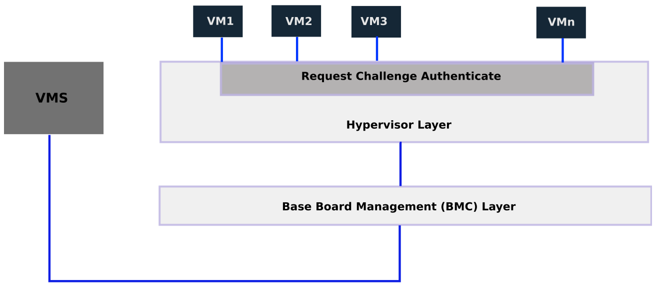

VM1
VM2
VM3
VMn
VMS
Request Challenge Authenticate
Hypervisor Layer
Base Board Management (BMC) Layer

Figure 5: Request-challenge-authenticate

then it is accepted and state change activity is carried out, otherwise the hypervisor rejects the state change request and sends an alert to a registered user to inform about unauthorized attempt to change the state of a VM. This is illustrated in Figure 5.

4 Known Constraints

This implementation needs changes in VMS, hypervisor and BMC. In the absence of VMM-I, the scope of network critical tasks such as Live migration over OOB-I is limited.

5 Conclusion

We explored a method in which Systems management capability can be used to handle virtual machines on the servers and perform various tasks in the same way it would be done on a physical server. We also explored use cases in terms of handling security situations which are evolving in a virtualized data center environment. The finding here is that in a virtualized environment were each VM can be running different tasks, it is unsafe to have unquestioned authority resting with VMS, since any error can prove very costly. The other aspect is the protection of data that travels from VMS to hypervisor. These challenges are unique and they need special attention in the virtualized data centers. One important finding is that, we need further explore what are the repercussions of creating a request-challenge-authenticate framework since VMS does not enjoy super user status any more.

6 References / Additional Resources

IPMI Home Page http://www.intel.com/design/servers/ipmi/index.htm

IPMI SPEC http://download.intel.com/design/servers/ipmi/IPMI2_0E4_Markup_061209.pdf

LibVirt Virtualization http://libvirt.org

Kernel Based Virtual Machine http://www.linux-kvm.org/page/Main_Page

TOTP: Time-Based One-Time Password Algorithm http://tools.ietf.org/html/rfc6238

ClusterShell, a scalable execution framework for parallel tasks

Stéphane Thiell, Aurélien Degrémont, Henri Doreau, Aurélien Cedeyn
Commissariat à l'Energie Atomique et aux Energies Alternatives (CEA)
`{stephane.thiell,aurelien.degremont,henri.doreau,aurelien.cedeyn}@cea.fr`

Abstract

Cluster-wide administrative tasks and other distributed jobs are often executed by administrators using locally developed tools and do not rely on a solid, common and efficient execution framework. This document covers this subject by giving an overview of ClusterShell, an open source Python middleware framework developed to improve the administration of HPC Linux clusters or server farms.

ClusterShell provides an event-driven library interface that eases the management of parallel system tasks, such as copying files, executing shell commands and gathering results. By default, remote shell commands rely on SSH, a standard and secure network protocol. Based on a scalable, distributed execution model using asynchronous and non-blocking I/O, the library has shown very good performance on petaflop systems. Furthermore, by providing efficient support for node sets and more particularly node groups bindings, the library and its associated tools can ease cluster installations and daily tasks performed by administrators.

In addition to the library interface, this document addresses resiliency and topology changes in homogeneous or heterogeneous environments. It also focuses on scalability challenges encountered during software development and on the lessons learned to achieve maximum performance from a Python software engineering point of view.

1 Introduction

From a logical perspective, cluster system software is what differentiates a cluster from a collection of individual nodes. Having a scalable and resilient cluster system management toolkit is essential to the successful operation of clusters. According to the TOP500 [1] list, more than 80% of installed HPC systems are running Linux, and open source software is now used as the foundation of most general-purpose[1] supercomputers. But even the simplest cluster-wide administrative task can become a nightmare when executed on a petaflop supercomputer of thousands of nodes. Often, tools or services are tuned to scale on a case-by-case basis. As a result, clusters often rely on a fragile administration software layer, suffering from the lack of robustness, usability and from management complexity. ClusterShell answers this by providing an open source, robust and scalable framework for cluster management and administration, that can be used by both system administrators and software developers. Indeed, benefiting from full-featured and scalable tools can save a lot of time for administrators, resulting in more efficient daily operations and reduced downtime during scheduled maintenance.

ClusterShell is available as a Free Software product under the terms of the CeCILL-C license [5], a French transposition of the GNU LGPL, and is fully LGPL-compatible. It consists in a Python (v2.4 to 2.7) library and a small set of command-line tools. It takes care of common administration issues encountered on clusters, such as operating on groups of nodes, running distributed commands using optimized execution algorithms, as well as helping result analysis by gathering and merging identical command outputs, or retrieving return codes. It takes advantage of existing remote shell facilities already installed on most systems, such as SSH. The command-line tools, `clush`, `clubak` and `nodeset` are efficient building blocks for administrators that also allow traditional shell scripts to benefit from some of the library features. Figure 1 shows an overview of the ClusterShell framework.

Primarily, the ClusterShell Python library implements an efficient event-based mechanism for parallel administrative tasks, whether they be local or distant. The ClusterShell Python API[2] provides an event-driven

[1] *General purpose* as defined in [7]

[2] http://packages.python.org/ClusterShell/

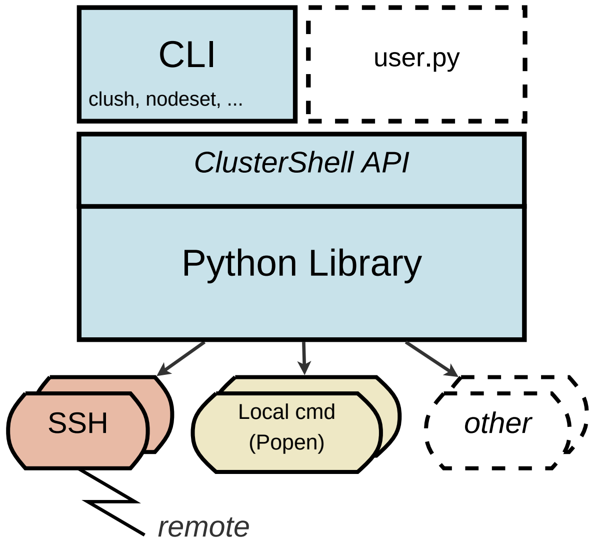


Figure 1: ClusterShell overview

and object-oriented interface that allows application to schedule actions, like running shell commands or copying a file, and to register for specific events from these local or remote tasks. Several helper methods are then available to analyze results during the execution or afterwards.

2 Cluster naming scheme

Computer clusters often use a systematic naming scheme, such as a common node prefix conforming to RFC 1178 *"Choosing a name for your computer"* [14] plus a numbering scheme. For example, high-availability clusters [10], database clusters and Hadoop clusters (for HDFS DataNodes [19]) frequently use simple serial naming procedures for individual servers. The naming policy of high-performance computing cluster nodes is often as simple, but can also be more complex by adopting a multi-dimensional numbering scheme. A numbering system that matches physical locations in a server room is sometimes adopted (eg. nodes are named according to the rack and slot number in [20, 6, 18] like `r01n03`). Another example could be to use logical positions in a multi-dimensional interconnection network (eg. `torus-1-2-3`). In this section, we define the *nodeset* and node group notations and present associated features available in ClusterShell to easily and efficiently manage these cluster naming schemes.

```
curie0
curie0,hwm0
stor[02-08]
curie[50,100-120,1500-6000]
curie[200-249]-ipmi
curie[200-249],gaia[20-59]
da[10-19]c[1-2]
```

Figure 2: Common *nodeset* examples

curie[2-8/2] $\Longleftrightarrow$ curie[2,4,6,8]
stor[01-10/3] $\Longleftrightarrow$ stor[01,04,07,10]
curie[50,1500-2000/2,3000-6000/4]-ipmi

Figure 3: Stepped *nodeset* examples

2.1 The *nodeset* notation

The *nodeset* notation presented here is a syntax for specifying a set of cluster nodes or host names. The comma (`,`) is used to separate different naming patterns (eg., to address multiple clusters). This notation is an extension of the one already used in cluster tools, such as SLURM resource manager [11], pdsh [16] or kanif [9]. Indeed with clusters growing, a commonly adopted notation has emerged mainly to allow the use of ranges whenever possible. A continuous range is specified by starting and an ending number, separated by a dash (e.g. `0-9`). Numbers may be expressed with a fixed length padding of leading zeros (`0`). Each discontinuous range, or single number, is separated by a comma (`,`). This makes a set or ranges, or *rangeset*. Within a *nodeset*, square brackets are used to signal a *rangeset*. Figure 2 shows this naturally compact cluster nodes naming scheme.

As an extension to the traditional *rangeset* notation, the stepping syntax already seen for Lustre™ networking [2] appends to a continuous range a slash character (`/`) and a step size. For example, the `2-8/2` format indicates a range of `2-8` stepped by `2`; that is `2,4,6,8`. Figure 3 shows some examples of this `nodeset` notation extension.

Moreover, our practical experience in cluster administration has shown that being able to embody some basic set operations in *nodeset* can be very convenient, and thus even more when working with node groups (discussed below). We define as a valid *nodeset* notation the following special operators:

- `,` as the union operator,

- `!` as the difference operator,
- `&` as the intersection operator,
- `^` as the symmetric difference operator.

nodeset patterns are read from left to right, and character operators are processed as they are met. Figure 4 shows a simple example.

`curie[0-50]!curie5` $\Longleftrightarrow$ `curie[0-4,6-50]`

`curie[0-10]&curie[8-20]` $\Longleftrightarrow$ `curie[8-10]`

Figure 4: *nodeset* character operator examples

2.2 Node groups

A node group represents a collection of nodes. Working with node groups is much more convenient and much safer when administrating large compute clusters or server farms. For example, a node group can correspond to nodes using the same set of resources or a specific type of hardware. Node groups can be used for a variety of reasons. In most cases, cluster software already provides several sources of static or dynamic node groups (eg., from a cluster database, genders [15], SLURM *nodes*, *partitions* or *jobs* [11], etc.). ClusterShell is able to bind to these node group sources and to provide unified information to the cluster management software. Node group provisioning is done through user-defined shell commands or through library extensions in Python. That is, ClusterShell itself doesn't manage node group definitions. Still, binding to a node group source based on flat files is straightforward[3].

The unified node group string notation introduced with ClusterShell is invariably prefixed by the arobase character (`@`) and constructed from the node group source followed by a separator character (`:`) and a node group name, the latter being freely expressed by the source. The notation can be further simplified using relative naming by omitting the node group source. In this case, the node group source configured by default is used to resolve the group. Figure 5 illustrates this syntax.

Moreover, when node group names are themselves adopting a systematic naming scheme as seen in section 2 for node names, we are able to represent a set

[3]A node group source example, based on a flat file, is provided by default.

$$@\underbrace{\text{compute}}_{\text{group name in default group source}}$$

$$@\underbrace{\text{slurm}}_{\text{explicit group source name}}:\underbrace{\text{bigmem}}_{\text{group name}}$$

Figure 5: Overview of *nodeset* syntax for node groups

of node groups in a similar fashion. The following example illustrates how to represent twenty Scalable Storage Units in a storage cluster: `@ssu[00-19]`, that is `@ssu00`, `@ssu01`, etc., up to node group `@ssu19`, each one corresponding to a set a nodes.

Evaluating node groups in a *nodeset* notation is quite straightforward, they are simply substituted by their corresponding nodes when needed. As for a regular set of nodes, the special operator characters seen in section 2.1 are supported with node groups. Figure 6 shows an example.

> `@slurm:standard&@ethswnode:sw[0-6/2]`
> stands for nodes from the SLURM partition named *standard* which are also connected to even-numbered switches (*sw0*, *sw2*, *sw4* and *sw6*)

Figure 6: Example of *nodeset* notation using node groups and the intersection operator

Using a node group explicitly indicates a grouping intention so operations are computed on the whole group, but also on the whole set of groups when brackets are used to designate a set of ranges. Otherwise, the operator-separated list of elements is evaluated from left to right. Intentionally, there is no support for parentheses or other ways to explicitly indicate precedence by grouping specific parts. Indeed, we tried to keep the syntax simple enough, focusing on the wide variety of tasks that cluster administrators perform.

2.3 Working with *nodesets*

nodeset objects are omnipresent in the ClusterShell framework within the `NodeSet` Python class. Two user-interfaces are available to manipulate *nodeset* strings whose syntax is described in section 2.1: one is

```
$ nodeset -f da1c1 da1c2 da3c1 da3c2
da[1,3]c[1-2]
```

Figure 7: Example of multi-dimensional *nodeset* folding using the `nodeset` command-line tool

the `NodeSet` Python class and the second one is the `nodeset` command-line tool.

For instance, `nodeset` provides optional switches to count the number of nodes within a *nodeset* (`-c`), to expand it (`-e`), to fold nodes into a *nodeset* (`-f`), to access node groups information, etc. It has become for us an essential command for daily cluster administration and an integral part of our shell scripts. All of its features are described in the documentation and on the ClusterShell Wiki[4].

The rest of this section describes some implementation aspects of different *nodeset* features.

2.3.1 *nodeset* folding

To fold a *nodeset*, we need a way to fold a set of ranges (a *rangeset*), as seen on section 2.1. `RangeSet` is the Python class that manages a *rangeset*. The latest implementation uses a standard Python `set` object to store the set of indices. We discuss in section 3.2 performance issues encountered on this topic. The folding implementation uses an iterator[5] on *slice* found objects, each one representing a set of indices specified by a range, plus a possible step. This is called, for example, when displaying a *nodeset* as a string.

Uni-dimensional *nodeset* is thus mainly solved by having a way to fold a *rangeset*. Multi-dimensional *nodeset* folding is more complicated. While expanding a multi-dimensional *nodeset* is easily achieved through a Cartesian product of all dimensions (we use Python's `itertools.product()`), folding is achieved by comparing *rangeset* vectors two by two, and to merge these vectors if they differ only by one item. Figure 7 shows an example of this multi-dimensional folding feature, available starting with ClusterShell version 1.7.

[4]https://github.com/cea-hpc/clustershell/wiki

[5]`RangeSet._folded_slices()`

2.3.2 Node groups regrouping

Another interesting ClusterShell feature is the ability to find fully matching node groups for a specified *nodeset*. This is called the *regroup* functionality. A simple heuristic implementation determines whether to use the `list` (list all groups) plus `map` (group to nodes) external commands, or to use `reverse` (node to groups). It then resolves node groups, returning largest groups first.

3 Scalability challenges with CPython

As a system software, ClusterShell is relying on CPython, the most-widely used implementation of the Python programming language. It is also the default of all Linux distributions used for clustering that we know of. This sections addresses performance challenges we faced in order to use CPython at scale.

3.1 Parallel programming

Because of its *Global Interpreter Lock* (or GIL), the standard CPython interpreter is unable to achieve actual concurrency with multithreaded programming [3]. Nevertheless, modules from the Python standard library can be leveraged to bypass this limitation and write high performance parallel code.

ClusterShell uses a combination of non-blocking I/O management and multiprocessing. The event-based I/O notification infrastructure is described in section 4.1. For CPU-intensive operations such as SSH connections, ClusterShell spawns external processes via the `fastsuprocess` module (see section 3.3). It therefore delegates scheduling operations to the OS, removing GIL-based contention constraints.

3.2 `RangeSet` performance

`RangeSet` is the Python class that manages a set of ranges as seen in section 2.1. In its first implementation[6] used in-memory *slice* objects representing the set of indices specified by a range, plus an optional step value (≥ 1). We first thought that direct access to ranges and operations done on these objects for 10k nodes (eg., on a range like `1-10000`) would be optimal with limited

[6]up to ClusterShell 1.5

memory footprint. But performance issues were quickly encountered when running on thousand node HPC clusters. The complexity of most related algorithms being in $O(R)$ with a number of discontinuous ranges of R, the bottleneck was then the high number of discontinuous ranges seen on these clusters. These sparse *nodesets* are commonly seen on large clusters (the way nodes are replying, in a random fashion, can create such "holes").

We then developed an intermediate implementation in Python using a `bintrees`-based AVL tree [13] to operate on ranges in $O(log(n))$. While it significantly outperformed the first implementation, we still did not achieve the performance we aimed for in all cases, probably because of the CPython overhead when creating a large number of objects. As a comparison, `bintrees` benchmarks using the *pypy* interpreter[7] show a 10 to 40 times speedup over CPython[8].

In the current implementation, the `RangeSet` class finally uses a Python `set`. Ranges are expanded as numeric indices in the set and a folding algorithm is used in case it needs to display a *rangeset*. It probably looks less elegant than using a balanced tree of ranges, but it is significantly faster than the AVL-tree implementation, mainly because sorting and set-like operations are very efficient in CPython.

3.3 `fastsubprocess`

Early versions of ClusterShell used the `subprocess` Python module to spawn new processes and connect to their input, output or error pipes. When using a large *fanout* value (> 128), that is, the number of child processes allowed to run at a time, we noticed a significant overhead localized in `subprocess.Popen`, even on medium size clusters. We found out that the parent Python process spends its time in a blocking `read(2)` operation, waiting for its children, leading to a serialization of all forked processes. Indeed, a pipe allows exceptions raised in the child process before the new program has started to execute, to be re-raised in the parent for convenience. This problem has been discussed on Python issue #11314[9] and the choice of feature *vs.* performance has been kept for now.

[7]http://pypy.org/
[8]http://pypi.python.org/pypi/bintrees/
[9]http://bugs.python.org/issue11314

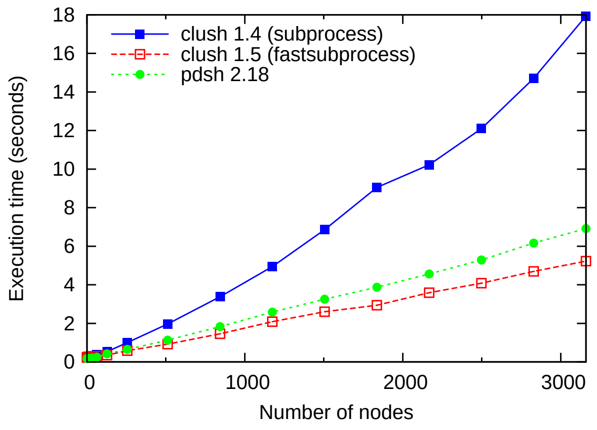


Figure 8: Performance comparison between ClusterShell *engines* based on `subprocess` and `fastsubprocess` and with C-based `pdsh` using a *fanout* value of 128

To work around this issue, we decided to adapt the `subprocess` module to make a faster, performance oriented version of the module for ClusterShell, that we named `fastsubprocess`. We removed the pipe used to transfer potential execution failures from the child to its parent, thus avoiding the blocking `read(2)` call. A child process returns a status code of 255 on `execv(3)` failure, which is handled by `Popen.wait()` in the ClusterShell library on proper event. We now also return file descriptors instead of file objects to avoid calling `fdopen()`. The only drawback of `fastsubprocess` is that it is not able to distinguish between an explicit return code of 255 from the child and an `execv(3)` failure, which we considered being an acceptable shortcoming considering the performance gain presented below.

Experiment: We evaluated the performance of the `fastsubprocess` module on Tera-100, CEA largest HPC Linux cluster, composed of a four-socket eight-core Intel® Xeon Nehalem EX (X7560) head node[10] running at 2.27 GHz with 64 GB of RAM, and more than 3000 compute nodes[11] each also four-socket X7560 nodes with 64 GB of RAM. ClusterShell version 1.4 was implemented using the regular Python `subprocess` module. Figure 8 clearly illustrates the scalability problem of this module when used intensively. As of version 1.5, we switched to our own `fastsubprocess` optimized module. `Pdsh` and ClusterShell 1.5 produced very similar results. How-

[10]S6030 bullx node
[11]S6010 bullx nodes

ever, ClusterShell execution times were slightly lower.

4 Scalable execution framework

In order to make ClusterShell production-ready on 10k-nodes clusters, we focused on both vertical and horizontal scalability aspects.

Numerous optimizations spread over the whole codebase brought scale-up improvements. Low memory and CPU footprint, as well as high performance I/O management have been achieved by leveraging efficient I/O notification facilities provided by the operating system.

Starting with ClusterShell version 1.6, the library is shipped with a major horizontal scalability improvement, allowing commands to be propagated to the targets through a tree of gateways (or proxies).

4.1 Vertical scalability

Dealing with the I/O streams from the multiple SSH instances that ClusterShell spawns can be a performance bottleneck. ClusterShell addresses this issue with a specific I/O management layer. Basically, massively parallel applications such as ClusterShell face the same problems as heavily loaded servers handling thousands of clients. In this regard, ClusterShell uses non-blocking I/Os and the most efficient I/O management paradigms [12].

Within a library instance, I/O management is done by a backend module, referred to as the *engine*. Several *Engines* are implemented. Each one relies on an non-blocking I/O demultiplexing system call (such as `select(2)` or `epoll(7)`) and exports a well defined interface to the upper layers of the library. This is entirely transparent and the other layers are fully *engine-agnostic*.

An engine provides primitives for registering and unregistering read, write or exception events on file descriptors, as well as an event loop entry point.

Each SSH process gets its standard input, output and error pipes registered to the library *engine* when starting. The engine processes the events from each I/O stream, and the potential timers, in a single-threaded loop.

The best available backend is selected at runtime, given that some OS-specific system calls might be unavailable on the running platform. This strategy allows ClusterShell to leverage the most efficient I/O notification subsystem [8] amongst the ones available.

Three backends are currently implemented:

- High performance, Linux-specific `epoll(7)`-based engine.
- Intermediate `poll(2)`-based engine
- Fallback `select(2)`-based engine

The most efficient backends being system-specific, this redundancy allows ClusterShell to achieve high performance while staying significantly portable. ClusterShell is available on a large number of systems, and packaged into several GNU/Linux distributions, including Red Hat® Enterprise Linux (RHEL) through the Fedora Extra Packages for Enterprise Linux (EPEL) repository, Fedora[12], Debian[13] and Arch linux[14].

Because of the system load generated by starting numerous concurrent SSH processes, the performance differences between the `epoll(7)` and `poll(2)`-based engines is hardly measurable. Therefore, further performance and scalability improvements have been done on the horizontal aspects.

4.2 Horizontal scalability

Even though the most efficient engines can handle thousands of I/O streams, the number of concurrent SSH processes is a blocking limitation [9] due to the CPU and memory load generated on the root node (from which commands are issued).

That is why we designed and implemented a new distributed propagation mode within the project. Commands are delivered through a network of gateways and results are sent back to the root node upward the created propagation tree.

The load gets shared between gateways, and the $O(\lambda N)$ propagation time we observe with a flat-tree mode (λ being the unit execution time) becomes $O(\lambda K log_k(N))$, with an arity of K [17] (K being the number of branches a gateway connects to). Figure 9 shows a schematic illustrating this principle.

[12]https://admin.fedoraproject.org/pkgdb/acls/name/clustershell

[13]http://packages.debian.org/fr/sid/clustershell

[14]http://aur.archlinux.org/packages.php?ID=53476

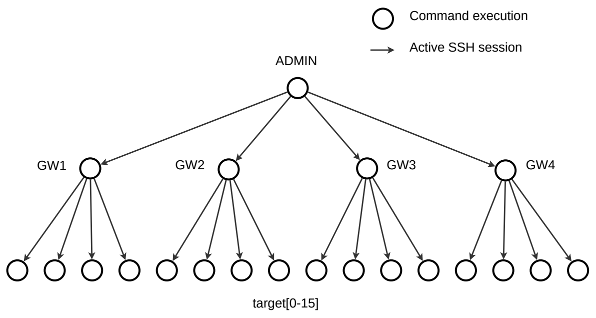


Figure 9: Hierarchical command propagation scheme

We also implemented a *grooming* mode that allows gateways to aggregate responses received within a certain timeframe before transmitting them back to the root node in a batch fashion. This contributes to reducing the load on the root node by delegating the first steps of this CPU intensive task to the gateways.

ClusterShell uses the same command sending techniques it uses in "normal" mode to control the gateways. As a result, the only requirement to setup a propagation tree is to have ClusterShell installed on the nodes that are susceptible to act as gateways, along with running a SSH server. SSH was chosen as a transport channel as it allows the propagation tree subsystem to use already in-place ClusterShell mechanisms, and also because of its reliability and security mechanisms. Nevertheless, the ClusterShell connector manager was designed with modularity in mind to ease support of additional protocols (such as RSH, PDSH or a ClusterShell-specific communication protocol).

A lightweight communication protocol ensures proper exchanges within the tree, using serialized Python objects embedded in a XML stream. As Python has a built-in incremental SAX parser (which is event-based), XML was a natural choice to represent the data and to guide the execution flow of the parser when they are received.

4.2.1 Communication within the tree

Gateways are implemented as ClusterShell-based state machines. Once instantiated from the remote ClusterShell process, gateways receive the topology to use, the targets to reach and the command to execute. Gateways recursively contact the next hop machines, deploying the propagation tree until final targets are reached.

```
# Allow connections from admin nodes
# to gateways
admin[0-2]: gateways[0-20]

# Allow connections from gateways to
# compute nodes
gateways[0-20]: compute[0-5000]
```

Figure 10: Topology syntax

The communication channel between the root node and a gateway (as well as between two gateways) is a single SSH connection that remains open until all results have been collected and sent back to the root node, which is also responsible for closing the channel at the transport layer.

4.2.2 Adaptive propagation

Topology is expressed through a configuration file on the root node as a list of possible connections between source and destination nodesets.

Mechanisms are implemented within ClusterShell to mark a gateway as unreachable and exclude it from the topology. Additionally, a work-stealing mechanism could be interesting, to let gateways adjust the load in real time between each other. The work made by C. Martin in that domain for the TakTuk project [17] stresses how valuable those mechanisms are when dealing with heterogeneous clusters and grids.

4.3 Experiments

In this section, we evaluate the performance of the scalable ClusterShell execution model, as introduced on section 4.2. To perform this experiment, we used Curie, a 2 Petaflop HPC Linux cluster operated by CEA. More precisely, we used the *Thin Nodes* partition of Curie, which consists of 5040 dual-socket nodes[15] each containing two eight-core Intel® Sandy Bridge EP (E5-2680) processors running at 2.7 GHz and 64 GB of RAM. Curie's operating system is Bullx Linux Advanced Edition, based on Red Hat® Enterprise Linux 6.1. The experiments have been done during a scheduled maintenance so no job was running. We avoided any external perturbation (such as the one that could be

[15]B510 bullx nodes

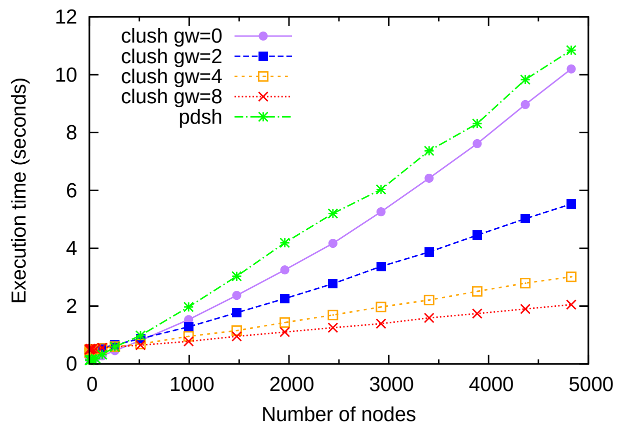


Figure 11: Performance comparison between `clush` v1.6 in basic mode (sliding window), in distributed mode (1 level of *n* gateways) and `pdsh` v2.18 on Curie (using ssh, fanout=128, command="echo ok")

induced by NFS, LDAP, etc.) by using a properly configured *root* superuser.

To measure the command propagation time, we remotely execute a command with the help of the `clush` command-line tool which is part of the ClusterShell framework. A command option allows an easy setup of the topology configuration file as seen in section 4.2.2.

For the experiment, we chose the command `echo ok` which has a negligible execution time and still enables some parsing code to be covered with a lightweight payload. Figure 11 presents the execution time of this command on up to 4828 remote nodes, with different execution models: basic model with a fixed fanout value (sliding window), tree-based propagation model with a varying number of gateways. `pdsh` v2.18 was used as a reference (using a fanout of 128, which we found to be the optimal value).

In basic execution mode (gw=0), `clush`'s curve looks smoother than `pdsh`'s one. Also, execution time is slightly lower, which is probably due to the event-based `epoll(7)`-based *engine*.

In distributed mode, with a single level of gateways, `clush` induces a constant overhead of about 300 ms, which is slightly noticeable on this figure at the leftmost part of the graph. This overhead is rapidly hidden by the gain of using a distributed command propagation (at about 250 nodes). The performance gain of the tree-based propagation is significant when increasing the number of gateways.

5 Related works

Several solutions exist to distribute administration tasks on parallel systems.

In terms of integration, these approaches can be classified in two categories: those providing a library API, like `func`[16] or `fabric`[17], and standalone applications like `pdsh` [16] or `gexec`[18]. ClusterShell combines both approaches by providing a library and tools built on top of it. Also, unlike other tools like `gexec`, ClusterShell does not require installation of an additional daemon on remote nodes.

In terms of scalability, existing solutions can also be classified in two categories, those that streamline direct commands, like `capistrano`[19], and the ones that propagate commands through a scalable (eg. hierarchical) scheme like `taktuk` [17].

Developed to facilitate production on large-scale systems, ClusterShell leverages the best of both approaches. Indeed, ClusterShell provides a convenient and scalable Python library along with efficient administration tools, especially designed for HPC clusters.

6 Conclusion

In this paper, we have presented ClusterShell, a lightweight Python framework used daily in production on the largest CEA HPC Linux clusters. System administrators and developers at CEA are working very closely, and this cooperation allowed us to improve the ClusterShell library to address the wide area of needs that administrators express for compute clusters as well as storage, post-processing clusters and even server farms.

From a Python performance perspective, limitations we faced were not the ones we initially expected. Also, by using original and creative techniques, we managed to circumvent common pitfalls.

Today, ClusterShell is used as a building block for other HPC software projects, such as Shine [4], an open source solution designed to setup and manage the

[16]`https://fedorahosted.org/func/`
[17]`http://fabfile.org/`
[18]`http://www.theether.org/gexec/`
[19]`https://github.com/capistrano/capistrano`

Lustre™ file system on a cluster, or Sequencer [21], an open source tool to efficiently control hardware and software components in HPC clusters.

We also presented the scalable execution engine of ClusterShell and the performance experiments we conducted, reflecting the success of our approach on large homogeneous clusters.

References

[1] Top 500 supercomputer sites. http://www.top500.org/, 2012.

[2] Oracle and/or its affiliates. Lustre™ 2.0 Operations Manual, 2011.

[3] David Beazley. Inside the Python GIL, 2009.

[4] CEA. Shine, Open Source Lustre management tool. http://lustre-shine.sourceforge.net/.

[5] CEA, CNRS and INRIA. CeCILL and Free Software. http://www.cecill.info/index.en.html.

[6] Brooks Davis, Michael AuYeung, Matt Clark, Craig Lee, Mark Thomas, James Palko, and Robert Varney. Lessons learned building a general purpose cluster. In *Proceedings of the 2nd IEEE International Conference on Space Mission Challenges for Information Technology*, SMC-IT '06, pages 226–234, Washington, DC, USA, 2006. IEEE Computer Society.

[7] Allan R. Hoffman et. al National Academies. *Supercomputers: Directions in Technology and Applications*. The National Academies Press, 1989.

[8] L. Gammo, T. Brecht, A. Shukla, and D. Pariag. Comparing and evaluating epoll, select, and poll event mechanisms. In *Proceedings of the 6th Annual Ottawa Linux Symposium*, volume 19, 2004.

[9] Guillaume Huard. Kanif, a TakTuk wrapper for cluster management and administration. http://taktuk.gforge.inria.fr/kanif/, 2007.

[10] Red Hat Inc. et al. Red Hat Enterprise Linux 6 cluster administration, configuring and managing the high availability add-on, 2011.

[11] Morris A. Jette, Andy B. Yoo, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*, pages 44–60. Springer-Verlag, 2002.

[12] Dan Kegel. The C10K problem. http://www.kegel.com/c10k.html, 2003.

[13] Donald Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching, Third Edition*. Addison-Wesley, 1997.

[14] D. Libes. Choosing a name for your computer. RFC 1178 (Informational), August 1990.

[15] LLNL. Genders. https://computing.llnl.gov/linux/genders.html, 2007.

[16] LLNL. Pdsh. https://computing.llnl.gov/linux/pdsh.html, 2007.

[17] Cyrille Martin. *Déploiement et contrôle d'applications parallèles sur grappes de grandes tailles*. PhD thesis, Institut National Polytechnique de Grenoble, France, 2004.

[18] Hiroyuki Mishima and Jun Ni. Rocks Cluster Installation Memo. Technical report, Medical Imaging HPC & Informatics Lab, 2008.

[19] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[20] Simon Fraser University. ClusterAdmin - HPC wiki. https://wiki.cs.sfu.ca/HPC/ClusterAdmin, 2010.

[21] Pierre Vignéras. Sequencer: smart control of hardware and software components in clusters (and beyond). In *Proceedings of the 25th international conference on Large Installation*

System Administration, LISA'11, pages 4–4, Berkeley, CA, USA, 2011. USENIX Association.

DEXT3: Block Level Inline Deduplication for EXT3 File System

Amar More
M.A.E. Alandi, Pune, India
ahmore@comp.maepune.ac.in

Zishan Shaikh
M.A.E. Alandi, Pune, India
zishan366shaikh@gmail.com

Vishal Salve
M.A.E. Alandi, Pune, India
vishaluttamsalve@gmail.com

Abstract

Deduplication is basically an intelligent storage and compression technique that avoids saving redundant data onto the disk. Solid State Disk (SSD) media have gained popularity these days owing to their low power demands, resistance to natural shocks and vibrations and a high quality random access performance. However, these media come with limitations such as high cost, small capacity and a limited erase-write cycle lifespan. Inline deduplication helps alleviate these problems by avoiding redundant writes to the disk and making efficient use of disk space. In this paper, a block level inline deduplication layer for EXT3 file system named the DEXT3 layer is proposed. This layer identifies the possibility of writing redundant data to the disk by maintaining an in-core metadata structure of the previously written data. The metadata structure is made persistent to the disk, ensuring that the deduplication process does not crumble owing to a system shutdown or reboot. The DEXT3 layer also takes care of the modification and the deletion a file whose blocks have been referred by other files, which otherwise would have created data loss issues for the referred files.

1 Introduction

While data redundancy was once an acceptable part of the operational backup process, the rapid growth of digital content storage has made organizations approach this issue with a new thought process and look for other ways to optimize storage utilization. Data deduplication would make disk more affordable by avoiding backing up redundant data.

Data deduplication is basically a single-instance storage method which helps in reducing the storage needs by eliminating redundant data. Only one instance of the data is actually stored and the redundant data, which will not exist physically, is just a pointer to the unique data. For example consider an example of an email system which may contain 100 instances of a 10 MB attachment. If the entire system was to be backed up, all 100 instances would be saved requiring 1000 MB of disk space. With deduplication in place, only 10 MB of one instance would actually exist and the other redundant instances would just be a pointer to this unique instance saving precious disk space.

Data deduplication generally works at the file or block level. The latter outperforms the former because file level deduplication would work for files that are same as a whole; whereas block level deduplication would work for blocks (a constituting part of the file) as a whole. Each chunk or block of data, that is about to be written to the disk, is subjected to hash algorithm such as MD5. This process generates a unique hash value for each block, which is stored in a database for further referral. If a file is updated, then only the added or changed data is stored, and disk storage is allocated only for these parts. In our work, we present an implementation of an inline block level deduplication layer added to the EXT3 file system, named as the DEXT3 layer.

2 Related Work

2.1 Types of Deduplication

Deduplication has various implementation approaches which can be significantly classified by their resiliency level (file vs. block) or by when they perform deduplication (inline vs. post-process) or by their method of duplicate data detection (hash vs. byte wise comparison). As stated earlier, inline deduplication is a strategy in which the duplicate data is identified before it hits the disk. Post-process deduplication is a strategy in which the data is first written to the disk and then de-duplication processing occurs in the background. In hash-based strategies, the process uses cryptographic hashes to identify duplicate data whereas byte wise strategies compare the data itself directly.

2.2 Deduplication Targets

Venti [1] and Foundation [2] both perform deduplication with respect to a fixed block size. Venti is basically an archival system. Foundation is a system that stores snapshots of various virtual machines and also uses Bloom filters to detect potential duplicates on the system.

The NetApp deduplication function [3] for file servers is used in integration with WAFL and FlexVol [4] and makes use of hashes to find duplicate data blocks. In this system, hash collisions are resolved through byte by byte comparison. This process runs in the background, therefore making it a post-process deduplication system.

The LBFS [5], Data Domain [6], HYDRAstor [7], REBL [8] and TAPER [9] identify and detect potential duplicates using content-defined chunks.

2.3 Deduplication Performance and Resources

The Data Domain [6] is a deduplication system that makes use of the spatial locality of data in a given backup stream to improve the performance of searching for hash metadata of the same data stream.

Sparse indexing [10] is a technique of deduplication that reduces the size of the data information kept in the RAM. It achieves this by sampling data hashes of data chunks.

These processes work fine with large data sets in a provided locality; however, we have to assume that the access patterns have small or no locality in a primary storage system.

2.4 Performance With Solid State Disks (SSD)

It is generally efficient to use fast devices like Solid State Disks for frequently used data. Chunkstash [12] and dedupv1 [11] make use of solid state disks for metadata. SSD and metadata is a good combination, mainly because I/O operations for metadata are always small and random.

3 DEXT3 design

The proposed design works on the following two principles:

1. Provide a working file system which tries to save space and avoid redundant disk writes.
2. Organize the in-core data structure efficiently and make it persistent to the disk.

In order to find the potential duplicates, the write system call invoked by the kernel is intercepted in the VFS itself. The data that is about to be written to the disk is available in a buffer in the write system call. This buffer is then broken into chunks of 4 kB each, owing to the 4 kB block size design of the kernel. These 4 kB blocks are compared with the data that has been previously written. If the new data block matches any previously written block, then instead of writing out the new block, the file's metadata is updated to point to the existing block on disk.

In order to identify potential duplicate blocks efficiently, an in-memory mapping of data hashes and the corresponding block numbers of the data is maintained. This mapping is created whenever there is a successful write to the disk. When control stays within the VFS, the hash value of block data is inserted into the data structure, and when the file system allocates a block to this data, its block number is stored in the data structure. In order maintain correct file system behaviour, another field is added in the data structure which maintains the reference count of the block, which indicates how many times a particular block has been used. This count is used whenever the kernel is about to release a block.

Deduplication could be implemented in a block layer device driver, which sits in between the file system layer and the actual underlying block device. This approach has the advantage that it is general and can be used for any file system. However, this extra layer of indirection incurs a performance penalty. The proposed implementation simply uses the existing block pointers in file system metadata, regardless of whether the pointer points to a regular block or a deduplicated block.

The system may be shut down or rebooted at any time. Being held in memory, the entire data structure would be lost, and after the system restarts again, the kernel would be unaware of the deduplicated blocks. Deleting a file whose blocks have been deduplicated would cause data loss issues to the referring files. The user might modify the file whose blocks have been referred by other files. This will still cause issues to the files which point to the blocks contained by the file being modified. All these issues would be handled in the proposed design.

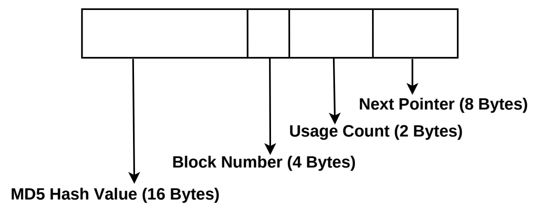


Figure 1: Node Structure of Dedupe Database

4 Implementation Details

DEXT3, a version of the EXT3 filesystem is implemented for Linux to provide on-the-fly inline block-level deduplication. DEXT3 is written for the Linux kernel version 2.6.35. The occurrence of duplicate data is detected in the write system call itself and then controls block allocation when the control of the kernel flows to the file system. The block de-allocation process of the file system is also modified in order to avoid freeing deduplicated blocks. The coming sections explain the design in detail.

4.1 Dedupe Database and Working of Deduplication Mechanism

The dedupe database is basically a data structure that maintains metadata with respect to the chunk of data that was previously written to the disk. The data structure that has been implemented is hash table with linear chaining. The size of the table is static, but the size of the chain is dynamic and grows as entries are added to the structure.

Figure 1 shows the structure of a node in this chain. The node occupies a total 30 bytes in size. The size of this node is fixed. The fields of this node describe the data chunk written to the disk in detail, that is, its hash value, the block number allocated by the kernel and the usage count which shows how many times the block with this block number has been referred by files. This field is useful in handling block de-allocation.

Following two hash functions are used by DEXT3:

MD5 Message Digest 5

FNV Fowler Nollvo hash

MD5 is used to generate hash value of the 4 kB data chunk which is intercepted in the write system call. This hash value is then used for checking redundant and duplicate data in the write system call.

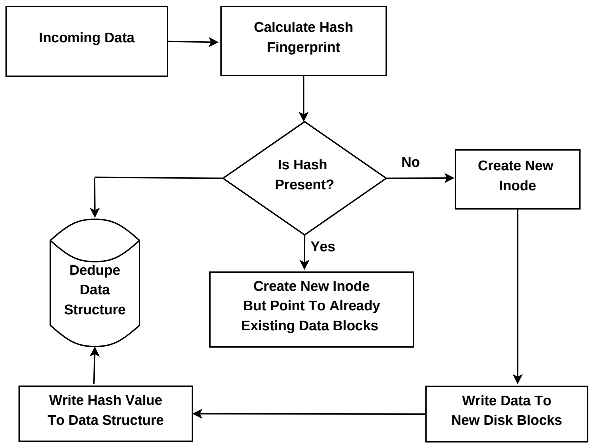

Incoming Data
Calculate Hash Fingerprint
Is Hash Present?
No
Create New Inode
Yes
Dedupe Data Structure
Create New Inode But Point To Already Existing Data Blocks
Write Hash Value To Data Structure
Write Data To New Disk Blocks

Figure 2: Working of the DEXT3 layer

The FNV hash is used to build the data structure itself. This function returns an integer hash value of 32-bits. This value is further truncated to 21-bits to keep the memory requirements optimal. So with a 21-bit FNV hash, a total 2097152 indices could be used to build the table. At every index one linear chain is maintained. The structure of a node in this chain is as stated in Figure 1.

First the 16-byte hash value of the 4 kB data chunk is obtained. Then the same chunk is subjected to FNV hash to obtain an index in the linear chain. The MD5 hash is inserted in this chain, and at a later stage when the kernel allocates a block to this data chunk, the newly allocated block number is stored in the same node and the usage count of this node is initialized to 1.

Before inserting the hash value in the chain, the chain lookup is performed. If the lookup fails, then the data is new and therefore not redundant, and the kernel is allowed to follow its normal allocation routines. However if the lookup succeeds, then the data is redundant. The usage count of the matching node is incremented, and the kernel does not allocate a new block for this data. This entire process is carried out before the data chunk is actually allocated a block on to the disk. Figure 2 explains the above stated design.

4.2 Main File Deletion and Modification

The deletion of a file whose blocks are referred by other files would cause data loss issues. To prevent this unlikely event, another data structure called the *character* *bitmap* is introduced. This bitmap is a character array that maintains the deduplicated status of all the blocks available in the file system. The bitmap status of a block is set to deduplicated (i.e. 1), whenever we end up performing a successful lookup in the linear chain for that block number. It would imply that the block has now been referred by more than one files. The status of a block remains 0 in the bitmap if it has never been deduplicated. The bitmap is efficient in terms of both memory and lookup. Each byte in the array can hold the status of eight blocks at once. Looking up the status of a block involves a single logical AND operation.

Before the kernel goes deeper into file deletion, the DEXT3 layer first decrements the usage count of all the blocks held by the target file. If the usage count of a block reaches zero, it means that the file about to be deleted is the last file to ever refer this block. So the layer:

1. Deletes its node from the chain
2. Updates its bitmap status to zero

If the usage count does not reach to zero, the DEXT3 layer simply allows the kernel to proceed to the next step.

The next step in file deletion is releasing the blocks held by the file. When the kernel is about to de-allocate and release a specific block, the DEXT3 layer first checks the deduplicated status of that block in the bitmap. If the status returned is 1, it means that there are files in the file system those still refer to this data block and releasing this block would be problematic. In this case the DEXT3 layer does not allow the kernel to release this block. Owing to this strategy, even if the main file inode has been deleted, if it holds any deduplicated block, then those blocks would still be available for use by the deduplicated files. Figure 3 and 4 explain this design.

The next challenge is to handle the modification of a file whose blocks have been deduplicated. Once again, this would cause data loss issues to the deduplicated blocks. Whenever a file is modified, such as via an editor, the kernel is asked to do the following things:

1. De-allocate the current inode
2. Allocate a new inode

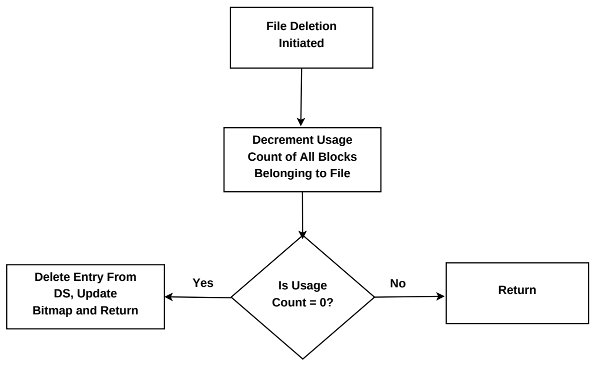

File Deletion Initiated
Decrement Usage Count of All Blocks Belonging to File
Is Usage Count = 0?
Yes
Delete Entry From DS, Update Bitmap and Return
No
Return

Figure 3: File Deletion Phase - I

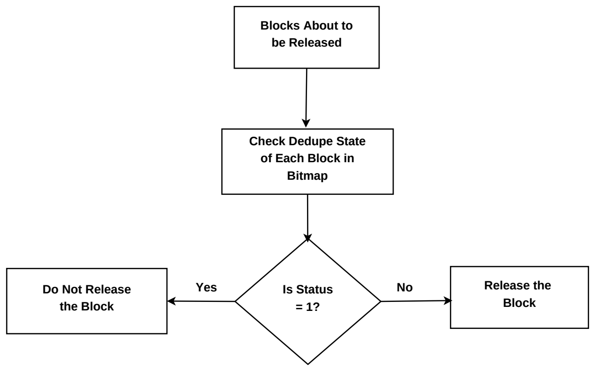

Blocks About to be Released
Check Dedupe State of Each Block in Bitmap
Is Status = 1?
Yes
Do Not Release the Block
No
Release the Block

Figure 4: File Deletion Phase - II

When the kernel deletes the current inode, it de-allocates the blocks that are currently held by the file. When a block is about to be allocated, the control is as shown in Figures 3 and 4.

When the kernel allocates a new inode to the file, the control flows through the write system call again. As the DEXT3 layer exists in the write system call, the control flow is as shown in Figure 2.

4.3 Permanent Data Structure

The entire DEXT3 database resides in the memory and is therefore volatile. Whenever the system is shutdown or rebooted, this highly precious information is lost. After the system boots again, the kernel is unaware of which block has been deduplicated. If it were about to release a block, then it would have directly released that block, even it was deduplicated earlier. To prevent this catastrophic event, the data structure is made persistent to the disk. When the system boots and the deduplication process starts again, data structure is rebuilt from this saved information. The bitmap is not flushed to the disk. Instead when the data structure is rebuilt, at the

% of Deduplication	Partition Capacity (GB)	Saved Disk Space (GB)
0	136	0
10	150	14
20	164	28
30	177	41
40	190	54
50	204	68
100	272	136

Table 1: Statistics with respect to the data structure

same time the bitmap is updated. This strategy keeps the kernel informed about the deduplicated status of each block even if the system is booted any number of times. This also helps to maintain and start the deduplication process from the point where it had stopped due to system shutdown.

5 Statistics for Disk Space Saving using DEXT3

5.1 Statistics With Respect to the Data Structure

Considering the number of nodes in a linear chain to be 17, we calculate the following statistics: With 17 nodes present at each of the 2097152 indices in the table, the size of the data structure goes up to nearly 1 GB. With this 1 GB of metadata, the layer manages 136 GB of data stored on to the disk, without any possibility of deduplication.

In the entire structure, if 10% of blocks have been deduplicated, then the partition capacity rises virtually to 150 GB saving nearly 14 GB of disk space. With 100% deduplication, partition capacity doubles to 272 GB, saving complete 136 GB disk space.

5.2 Statistics With Respect to File Size

Considering the size of a file to be 4 GB, the total number of blocks that would be allocated to this file would be 1049612.

% of Deduplicated Blocks	Total Blocks Saved (GB)	Saved Disk Space (MB)
10	104961	410
20	209922	820
30	314884	1230
40	419845	1640
50	524806	2050
100	104961	4100

Table 2: Statistics with respect to the File Size

6 Conclusion

The statistics and results were encouraging and indicate how DEXT3 is able to save significant amount of disk space and reduce the number of disk writes. The memory requirements remain optimal. Including deduplication in the kernel introduces a little overhead in the system performance. Though on one hand we introduce this overhead, on the other hand DEXT3 provides significant cost, space and energy savings. We view this as acceptable since performance gain is not the primary goal, rather our goal is to avoid writes and achieve space savings. The DEXT3 layer does not modify any other filesystem metadata other than the inode. The inode is updated at run time itself, so there is no need to update the file system metadata explicitly. The persistent data structure strategy makes it possible to rebuild the data structure after system boot. Almost all the applications that use solid state disks for primary storage make use of the existing and standard file systems. Providing a simple DEXT3 layer is crucial in order to promote real world use. The original FFS in UNIX added disk awareness to an otherwise hardware oblivious filesystem. We find block level inline deduplication i.e. DEXT3 as a crucial and important layer of SSD limitations. More importantly, as the industry transitions away from the spinning disks towards solid state devices, this kind of approach, as we see, will become increasingly critical.

References

[1] S. Quinlan and S. Dorward, *Venti: a new approach to archival storage*. The First USENIX conference on File and Storage Technologies (Fast '02), January 2002.

[2] S. Rhea, R. Cox and A. Pesterev, *Fast, inexpensive content-addressed storage in Foundation*. The 2008 USENIX Annual Technical Conference, June 2008.

[3] C. Alvarez, *NetApp deduplication for FAS and V-Series deployment and implementation guide*. Technical ReportTR-3505, January 2010.

[4] S. Rhea, R. Cox and A. Pesterev, *Fast, inexpensive content-addressed storage in Foundation*. The 2008 USENIX Annual Technical Conference, June 2008.

[5] A. Muthitacharoen, B. Chen, and D. Mazieres, *A low-bandwidth network file system*. The 18th ACM Symposium on Operating Systems Principles (SOSP), Banff, Alberta, Canada, October 2001.

[6] B. Zhu, K.Li, and H. Patterson, *Avoiding the disk bottleneck in the Data Domain deduplication file system*. The 6th USENIX Conference on File and Storage Technologies (FAST '08), February 2008.

[7] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki, *HYDRAstor: a scalable secondary storage*. The 7th USENIX Conference on File and Storage Technologies (FAST '09), February 2009.

[8] P. Kulkarni, F. Douglis, J. Lavoie, and J. M. Tracey, *Redundancy elimination within large collections of files*. The 2004 Usenix Annual Technical Conference, June-July 2004.

[9] N. Jain, M. Dahlin, and R. Tewari, *TAPER: tiered approach for eliminating redundancy in replica synchronization*. The 4th USENIX Conference on File and Storage Technologies (FAST '05), December 2005

[10] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise and P. Camble, *Sparse indexing: large scale, inline deduplication using sampling and locality*. The 7th USENIX Conference on File and Storage Technologies (FAST '09), February 2009.

[11] D. Meister and A. Brinkmann, *dedupv1: improving deduplication throughput using solid state drives (SSD)*. IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), May 2010.

[12] B. Debnath, S. Sengupta and J. Li, *ChunkStash: speeding up inline storage deduplication using flash memory*. The 2010 USENIX Annual Technical Conference, June 2010.

[13] Keren Jin, Ethan L. Miller, *The Effectiveness of Deduplication on Virtual Machine Disk Images*. SYSTOR 2009 May 2009, Haifa, Israel.

ARMvisor: System Virtualization for ARM

Jiun-Hung Ding
National Tsing Hua University
adjunhon@sslab.cs.nthu.edu.tw

Chang-Jung Lin
National Tsing Hua University
chucklin@sslab.cs.nthu.edu.tw

Ping-Hao Chang
National Tsing Hua University
phchang@sslab.cs.nthu.edu.tw

Chieh-Hao Tsang
National Tsing Hua University
jeffzaza@sslab.cs.nthu.edu.tw

Wei-Chung Hsu
National Chiao Tung University
hsu@cs.nctu.edu.tw

Yeh-Ching Chung
National Tsing Hua University
ychung@cs.nthu.edu.tw

Abstract

In recent years, system virtualization technology has gradually shifted its focus from data centers to embedded systems for enhancing security, simplifying the process of application porting as well as increasing system robustness and reliability. In traditional servers, which are mostly based on x86 or PowerPC processors, Kernel-based Virtual Machine (KVM) is a commonly adopted virtual machine monitor. However, there are no such KVM implementations available for the ARM architecture which dominates modern embedded systems. In order to understand the challenges of system virtualization for embedded systems, we have implemented a hypervisor, called ARMvisor, which is based on KVM for the ARM architecture.

In a typical hypervisor, there are three major components: CPU virtualization, memory virtualization, and I/O virtualization. For CPU virtualization, ARMvisor uses traditional "trap and emulate" to deal with sensitive instructions. Since there is no hardware support for virtualization in ARM architecture V6 and earlier, we have to patch the guest OS to force critical instructions to trap. For memory virtualization, the functionality of the MMU, which translates a guest virtual address to host physical address, is emulated. In ARMvisor, a shadow page table is dynamically allocated to avoid the inefficiency and inflexibility of static allocation for the guest OSes. In addition, ARMvisor uses R-Map to take care of protecting the memory space of the guest OS. For I/O virtualization, ARMvisor relies on QEMU to emulate I/O devices. We have implemented KVM on ARM-based Linux kernel for all three components in ARMvisor. At this time, we can successfully run a guest Ubuntu system on an Ubuntu host OS with ARMvisor on the ARM-based TI BeagleBoard.

1 Introduction

Virtualization has been a hot topic and is widely employed in data centers and server farms for enterprise usage. Today's mobile devices are equipped with GHz CPU, gigabytes of memory and high-speed network. With the advancement of computing power and Internet connection in embedded devices, system virtualization also assists to address security challenges, and reduces software development cost for the mobile and embedded space. For instance, the technique of consolidation is capable of running multiple operating systems concurrently on a single computer and the technique of sandboxing enhances security to prevent a secure system from destruction by an un-trusted system. The benefits and the new opportunities have prompted several companies to put virtualization into mobile and embedded devices. Open Kernel Labs announced that the OKL4 embedded hypervisor has been deployed on more than 1.1 billion mobile phones worldwide to date. Hence, research into the internal design and implementation of embedded hypervisors also attact more attention in academic communities. In this paper, we have worked on the construction of ARM based hypervisor without any hardware virtualization support.

Essentially, the ARM architecture was not initially designed with system virtualization support. In the latest variant, the ARMv7-A extension which was announced in the middle of 2010, ARM will start to support hardware virtualization and allow up to 40-bit physical address space. However, the widely used variants of ARM processors, such as ARMv5, ARMv6 and ARMv7, lack any hardware extension for virtualization, making it difficult to design an efficient hypervisor when well-known full virtualization method is being applied. In fact, according to the virtualization requirements proposed by Popek and Goldberg in 1974 [2], ARM is not considered as a virtualizable architecture [1]. There exist numerous non-privileged sensitive instructions which behave unpredictably when guest operating system is running in a de-privileged mode. These critical instructions increase the complexity of full virtualization implementation, and some of them, such as LDRT/STRT/LDRBT/STRBT instructions, will cause huge performance degradations. Traditionally, techniques like trap-and-emulation and dynamic binary translation (DBT) are adopted to handle the sensitive instructions as well as critical instructions for CPU virtualization. In ARMvisor, a lightweight paravirtualization method takes the place of DBT. Merely hundreds of codes are necessarily patched into guest operating system source codes.

For memory virtualization, the shadow paging mechanism is also hard to be manipulated. First, the address translations, access permissions and access attributes described in guest page tables and other system registers must be correctly emulated by the underlying hypervisor. Second, the hypervisor must maintain coherences between guest page tables and the shadow ones. Third, the hypervisor needs to protect itself from destruction by guest, as well as to forbid user mode guest to access kernel memory pages. According to the above three requirements, the ARM hypervisor still suffers performance degradation, especially during the sequence of process creation. To reduce the overhead, a lightweight memory trace mechanism for keeping shadow page tables synchronized with the page tables of the guest OS is proposed by ARMvisor.

The experimental results of ARMvisor have shown leaping performance improvement with up to 6.63 times speedup in average on LMBench. Additionally, in embedded benchmark suite such as MiBench, the virtualization overhead is minimal, meaning that performance is fairly close to native execution.

The rest of the paper will be organized as follows. Related works will be discussed in Section 2. Software architecture of our embedded hypervisor is presented in Section 3. A cost model for hypervisor are formed in Section 4. Section 5 describes the optimization methodologies for CPU and memory virtualization. Experimental results and analysis are covered in Section 6. At last, we conclude the paper in Section 7.

2 Related work

A variety of virtualization platforms for ARM have been developed in the past few years as a result of the long leap in computing power of ARM processor. Past work [3] mentioned several benefits of virtualizing ARM architecture in embedded systems. It was noted that the hypervisor provides the solution to embedded system including security issues in online embedded devices due to the downloading of third party applications by isolating hardware resource by virtual machines. An embedded hypervisor also enables heterogeneous operating system platform to deal with conflicting of API in different operating systems.

Others [1] have analyzed the requirement for designing a hypervisor for embedded system. For instance, the hypervisor must be equipped with scheduling mechanisms for latency critical drivers to meet the real-time requirements, and must also support various peripheral assignment policies such as directly assigned, shared assigned devices or run-time peripheral assignment. Other important factors include accelerating the boot time of the guest OS, utilizing the utmost performance of embedded hardware as well as reducing the code size of hypervisor to prevent from potential threat.

Numerous hypervisors have been designed for newer capabilities for embedded ARM platforms. OKLab has developed OKL4 Microvisor [4] based on L4 microkernel for ARM, catering to the merits of embedded virtualization. They claim that the hypervisor has been ported to millions of mobile handsets and supports system such as Windows and Android to run atop of OKL4 Microvisor. They claim that the OKL4 Microvisor supports a secure embedded platform. Other commercial VMMs include VMware's MVP [5], Trango [6] and VirtuaLogix [7]. Nevertheless, none of these solutions, including the OKL4 Microvisor, are open-sourced and thus the insights of their design are not available.

On the other hand, Xen [8] is a well-known hypervisor for system virtualization, and has been successfully ported to ARM architecture in Xen version 3.02 [9]. "Xen for ARM" required the guest code to be para-virtualized by adding hyper-calls for system events such as page table modification. However, code that needs to be para-virtualized is not revealed in the paper. The cost of maintenance with generations of different guest operating system soars higher as heavier as the guest's code being modified for virtualization.

"KVM for ARM" [1] also implemented an embedded VMM under KVM in ARMv5. They proposed a lightweight script-based approach to para-virtualize the kernel source code of the guest OS automatically, by switching various kinds of non-privileged sensitive instructions with pre-encoded hyper-calls that trap to hypervisor for later emulation. Nonetheless, they applied a costly memory virtualization model when de-privileging guest system in the user mode of ARM architecture. First, they did not apply the "reverse map" mechanism in memory virtualization for keeping the coherency of guest's and shadow page table. In their model, each modification results in a page table flush since the hypervisor is unaware of the correspondence of guest's page table and shadow page table. Furthermore, the benchmarking or profiling results are not yet revealed, so it is hard to evaluate the performance results of running virtual machines on their work.

In contrast, ARMvisor introduces a lightweight memory virtualization model mechanism to synchronize the guest page table, which is more suitable for use in embedded system due to the performance and power consumption concern. Detailed measurement and analysis of system and application level benchmarks will be reported in the following section. We proposed a cost model to measure overhead due to virtualization in general use cases. According to the profiling results, we can design several optimization methodologies.

3 Overview of ARMvisor

The proposed hypervisor is developed based on the open source project KVM (Kernel-based Virtual Machine) which was originally designed for hardware virtualization extension of x86 architecture (Intel VT, AMD SVM) to support full-virtualization on Linux kernel. It has been included in the mainline Linux since kernel version 2.6.20. KVM is composed numerous loadable kernel modules which provide the core functions of the virtualization. A modified QEMU is used to create the

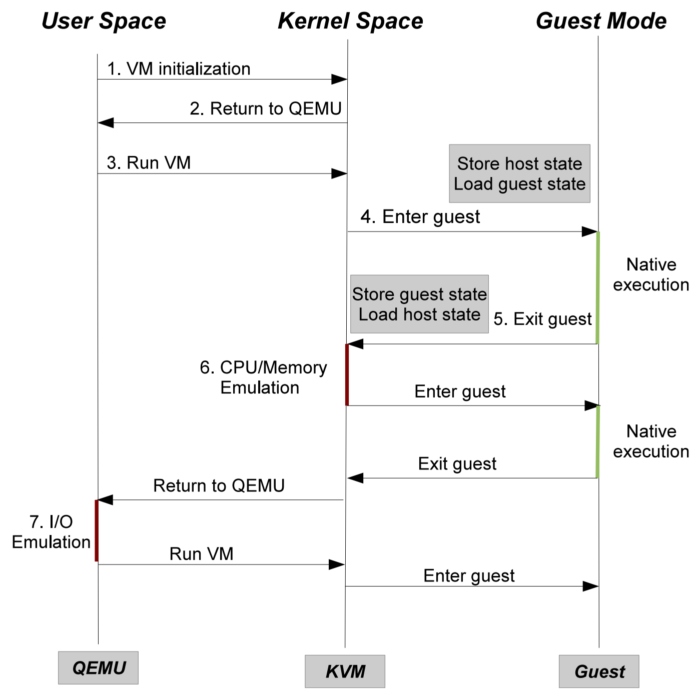


Figure 1: KVM execution path

guest virtual machine and to emulate the I/O devices. Figure 1 illustrates the execution flow when providing a virtualization environment for the guest using KVM. A virtual machine (VM) is initiated in QEMU by using the system call interface (`ioctl`) provided by the modules of KVM. After the VM has finished its initialization procedures, KVM changes the execution context to emulated state of the guest OS and starts to execute natively. Guest exits its execution context whenever an exception occurs. There are two kinds of traps in KVM: *lightweight* and *heavyweight* traps. In lightweight traps, the emulation is handled in internal functions of KVM, implemented in the kernel space of Linux. In contrast, heavyweight traps that include I/O accesses and certain CPU functions will be handled by QEMU in user space, so context switches are required. Hence, the cost of a single heavyweight trap is higher than a lightweight one.

Numerous hardware virtualization extension primitives of the x86 architecture are leveraged by KVM to provide fast and stable workloads for virtual machines. A new execution *guest mode* is designed to allow direct execution of non-privileged yet sensitive x86 instructions. Instead of unconditionally trapping on every privilege or sensitive guest instruction, a subset of those instructions can execute natively in the virtual ring 0 provided by the guest mode. They modify the shadowed CPU states indicated on *VMCB* (Virtual Machine Control Block) to emulate the desired effect as is natively executed. VMCB is an in-memory hardware structure which contains privileged registers of the guest OS context and the control state for VMM. VMM can fetch guest's running status directly as well as setup the behavior of exceptions from the VMCB. Furthermore, functions are defined for the switch between guest and host mode. For instance, after VMM finishes its emulation tasks like page table filling or emulating I/O instruction, `vmrun` is called to reload the state of virtual machine from the modified VMCB to physical registers and resume its execution in guest mode. Aside from the hardware assistance for CPU virtualization, specific features were proposed to aid MMU virtualization on x86. Intel's *EPT* and AMD's *nested paging* were designed to tackle the issue of "shadow paging" mechanism traditionally applied in MMU virtualization.

However, due to the lack of hardware assistance in our experimental ARM architecture, KVM's performance suffers in various aspects on ARM. In CPU virtualization, software techniques are adopted due to the lack of hardware support, such as the x86 *guest mode* for virtualization. Lightweight traps are generated by adding hyper-calls for non-privileged sensitive instruction emulation. To apply the trap and emulation model for emulating instruction, guest system is de-privileged to execute in user mode. Besides, without the help of VMCB, a context switch interface is needed for KVM/guest switch. The state save/recovery for both lightweight and heavyweight traps results in extra overhead and thus largely degrades the system performance. Details of the methodologies we took and the optimization applied will be illustrated in the following sections.

3.1 CPU Virtualization

Guest Operating Systems finish critical tasks that access systems resources by executing sensitive instructions. According to the definition [2], there are two categories of sensitive instructions: Control Sensitive and Behavior Sensitive instructions. Control Sensitive instructions are those that attempt to change the amount of resource and the configuration available, while Behavior Sensitive instructions are those that behave depending on the configuration of resources. For example, "CPS" in ARM modifies CPU's status register, a.k.a. CPSR, to change the execution mode or to enable/disable interrupt and has control sensitivity. Moreover, executing CPS in user mode results to NOP effect thus it is also behavior sensitive. Such instructions must be handled properly to keep the guest being correctly executed.

Actually, in order to prevent guest from ruining the system by controlling hardware resource with privilege and allow hypervisor to manage the resource of system, guest operating system is de-privileged to execute in non-privilege mode while hypervisor is located in privilege level for resource management. Pure virtualization that executes guest OS directly in user mode is proposed under the premise that the architecture is virtualizable, i.e. all the sensitive instructions trap when executing in non-privilege mode. Hypervisor intercepts such traps and emulates the instruction in various fashion. The approach requires no modification of guest OS to run in a virtual machine, so the engineering cost for porting and maintaining multiple versions of guest OS is minimal. However, pure virtualization is not feasible in contemporary architectures since most of them including x86 and ARM are not virtualizable and there exist some non-privilege sensitive instructions, called *critical instructions*, which behave abnormally when being de-privileged in user mode.

Solutions have been proposed to ensure the exactness of system execution in non-virtualizable architecture like x86 without hardware extension. Past work [13] imports dynamic binary translation (DBT) techniques to overcome the obstacles in virtualizing x86 architecture to support full virtualization. Guest binary is transformed into numerous compiled code fragments (CCF) and chained together by the VMM. In essence, most of the guest code is identical except those sensitive instructions. Sensitive instructions are either translated into non-sensitive ones, or into jumps to a callout function for correct handling. Nonetheless, DBT seems prohibitive in embedded system since the translated code accounts for large portion of memory and RAM size is comparatively smaller than in large server or work station.

Besides, paravirtualization [8] methodology that replaces the non-privilege sensitive instructions with sets of pre-defined hyper-calls for x86 is also introduced. During execution, the hyper-call traps to the hypervisor for corresponding handling for events such as page table pointer modification. In spite that the performance presents promising run-time results, engineering cost is expensive for porting a guest OS to the virtual machine in Xen and thus adds difficulties for maintaining new versions of guest OS distribution for performance issues. To solve such limitations of paravirtualization, new technique called pre-virtualiztion [14] has been proposed. They presented a semi-automatic sensitive instruction re-writing mechanism in the assembler stage, and assert that compared with paravirtualization, the approach does not only require orders of magnitude fewer modification of guest OS, but also achieves almost the same performance result as paravirtualization.

Given that ARM's ISA is non-virtualizable, ARMvisor chooses paravirtualization techniques to handle guest's non-privilege sensitive instruction. Guest kernel and applications are de-privileged to execute in ARM's user mode, while ARMvisor executes in ARM's supervisor mode to avoid guest from crashing the host system. ARM's SWI, accompanied with a dedicated number, is inserted manually before each sensitive instruction as hyper-calls for instruction emulation. Traps will be triggered and sent to the Dispatcher in the hypervisor. ARMvisor acknowledges the software interrupt triggered with a specific number and then decodes and emulates the sensitive instructions by effectively modifying the "Virtual Register File", which represents the virtual CPU context of guest system.

In practice, trapping on each sensitive results in huge degradation in performance, due to the high cost of traps in modern computer design. As stated earlier, hardware extension of x86 architecture provides extra mode for virtualization to address such issue. Many sensitive instructions can directly execute in guest mode rather than trapping to hypervisor for later handling thus improving performance. Vowing to lower the considerable overhead results from "trap and emulation" without hardware extension in contemporary ARM architecture, we further proposed two optimizing technique to accelerate the performance. Insights of each optimizing heuristic will be discussed in later section.

3.1.1 Instruction emulation

ARMv6 defines 31 sensitive instructions, of which 7 are privileged instructions and will trap when guest system is being de-privileged. The rest of them are critical instructions, which required code patching to prevent non-deterministic behaviour in user mode. Table 1 lists the counts of various types of critical instructions that need to be modified in Linux kernel 2.6.32 to boot

Instruction_type	Count
Data Processing (movs)	4
Status Register Access (msr/mrs, cps)	34
Load/Store multiple (ldm(2,3), stm(2,3))	8

Table 1: Sensitive instruction count

Cache/TLB's invalidation and clean
BTB Flush
TTBR Modification
Domain configuration
Context ID
Processor status (ex: CPU ID, Page Fault Info)

Table 2: ARM co-processor operations

on ARMvisor. We figured that the critical instructions exist in files for three separate purposes in Linux kernel: kernel boot-up, exception handling and interrupt enable/disable. Macros composed of instructions that setup ARM Status Register for interrupt controlling are defined in header files, and are widely deployed in large portion of kernel code. Moreover, numerous critical instructions actually account for large portion of the code in ARM Linux's exception handling entry and return. In our measurement, we found that merely forcing those non-privilege sensitive instructions to trap for instruction emulation in ARMvisor brings about intolerable performance loss in guest system. In fact, these instructions only involve virtual state change and no hardware modification is needed. Consequently, we eliminate the traps by replacing the instructions with Shadow Register File Access (SRFA) in guest's address space.

In contrast to the critical instructions, a few sensitive instructions like ARM co-processor's operation (MCR/MRC) are privileged, and numerous essential operations are accomplished through the execution of those privilege instructions. Table 2 lists several functions achieved by co-processor operations. Traps will be invoked when executing these instructions in ARM's user space and certain hardware alteration is performed by ARMvisor to correctly emulate guest's desired behavior.

To relieve the overhead suffered in emulating those instructions, we concisely analyze all of the operations and propos several refinement methodologies to achieve performance improvement. First, guest's TLB operations and BTB flush is dynamically replaced with NOP's by ARMvisor during execution, since TLB and BTB will be thoroughly flushed during every context switche between guest and the hypervisor. Secondly, operations that read information in co-processors, such as memory abort status and cache type, can also be replaced with SRFA since only virtual state is fetched. Finally, cache operations involving certain hardware configuration must be trapped and finished in privilege level. To minimize the overhead of emulation, technique called *Fast Instruction Trap* (FIT) is applied to reduce costs of context switching and handle those operations in guest address space. Implementation details of SRFA and FIT will be narrated in later section. TTBR and Domain Register modification is comparatively complicated so they are emulated in ARMvisor through lightweight traps.

We conclude that many software optimizations can be applied with paravirtualization techniques to effectively mitigate the run-time overhead for virtualization at the expense of higher engineering cost. It depends on the VM distributor to decide the pros and cons of performance elevation with the trade-off of the cost to maintain versions of guest OS.

3.1.2 CPU virtualization model

The hypervisor gathers system exceptions such as interrupts, page faults and system calls, and properly handles them for each virtual machine. Hardware extensions in the x86 architecture enable guest OS exceptions to be handled by its operating system natively without the interference of hypervisor by setting typical bits in VMCB.

In the ARM architecture, which lacks such hardware assistance, the hypervisor is responsible for distinguishing and virtualizing exceptions of the guest OS by delivering traps to each virtual machine. Synchronous exceptions such as memory access abort, system calls as well as undefined access have higher priority and should be injected to guest virtual machine immediately. Asynchronous exceptions such as interrupt could be delivered later when the pending queue for synchronous exceptions of that virtual machine is empty.

Additionally, exceptions may be invoked for virtualization events such as hyper-calls for instruction emulation and extra shadow page table miss, which are non-existent when executing the OS on bare hardware.

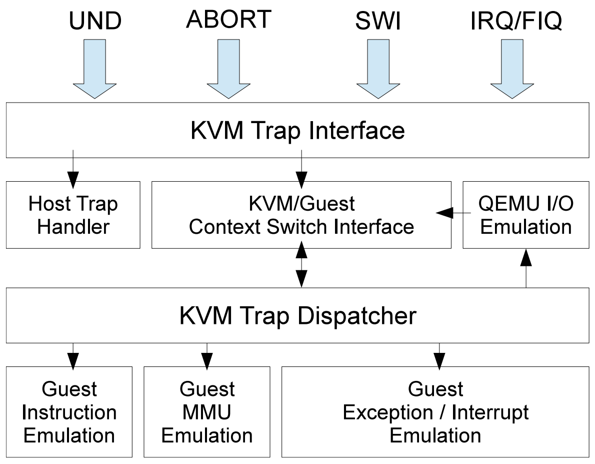


Figure 2: VCPU virtualization flow

These traps are handled internally by the hypervisor and the guest is actually unaware of such events.

As depicted in Figure 2, exceptions are routed to ARMvisor for unified administration by replacing host kernel's exception vector with *KVM Vector* when KVM module is loaded. Therefore, system's exceptions are re-directed to the Trap Interface inside ARMvisor for later handling. The interface verifies the host and guest exceptions and branch for separate handling path. Whenever guest exceptions are discovered, KVM/Guest Switch Interface saves the guest interrupt state and switches to KVM's context for later handling. In fact, since KVM executes in a different address space from the guest, page tables must be changed for consequent trap handling. However, unlike the hardware extension of x86 architecture that restores the page table pointer register automatically in VMCB for exception exits, KVM/Guest Switch Interface must be contained in a shared page which is accessible in both guest and KVM address space, otherwise the behavior would be undefined after the modification of TTBR (Translation Table Base Register) in ARM. As shown in Figure 3, the interface is contained in a page between the address of `0xffff0000` to `0xffff1000` (if high vectors are used). This page is write-protected to prevent from malicious attacks crashing the system.

The trap dispatcher in ARMvisor forwards lightweight and heavyweight traps to different emulation blocks. Several traps, such as hyper-calls, memory access abort and interrupt, are handled by emulation blocks in ARMvisor respectively as illustrated in Figure 2. After all KVM's internal emulation blocks finish their emula-

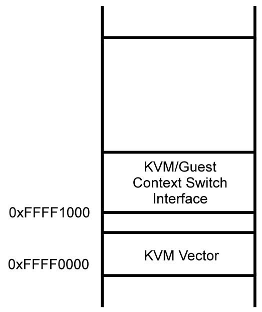


Figure 3: KVM vector implementation in ARMvisor

tion, KVM/Guest Switch Interface verifies if the trap is lightweight, and if so, restores the guest interrupted context to continue its previous execution. Otherwise, the trap is heavyweight, so a context switch is required since subsequent emulation tasks are to be finished in QEMU. After a heavyweight trap finishes its emulation, such as I/O access and CPU related operations through the interfaces provided by QEMU, another context switch is taken to resume the guest's previous execution.

3.2 Memory virtualization

Generally, the VMM provides a virtualized memory system for a guest virtual machine. When the user-level applications and operating system run in a VM, their memory access will be precisely controlled and remapped to host physical memory. For security, the VMM necessarily protects itself from illegal access by any guest and isolates one VM from another one. In practice, implementing memory virtualization is relatively more complex than CPU virtualization. The efforts involve working on guest physical memory allocation, shadow paging mechanism, and virtual Memory Management Unit (vMMU) emulation. Our implementation and the considerations for ARM memory architecture will be discussed later.

3.2.1 Guest physical memory allocation

The guest physical memory can be allocated in two ways: static or dynamic allocation. A static allocation will reserve continuous memory pages from host physical memory. The allocated memory pages are occupied and unlikely to be shared with VMM or other VMs, thus the host memory resources are not being well utilized. In contrast, the dynamic allocation method

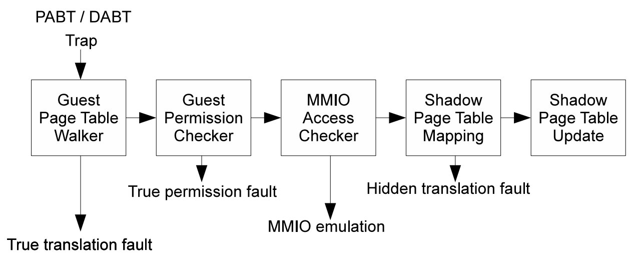


Figure 4: The emulation flow of shadow paging

maps a region of host virtual memory as guest physical memory. The host physical memory pages are allocated dynamically, as the technique of demand paging. In [10], VMware ESX server further provides a ballooning driver for guest to reclaim unused memory pages. Current KVM manages guest memory resources using the existing functionalities within the Linux kernel including buddy allocator, slab/slub allocator, virtual memory subsystem. Once a VM is created, KVM considers the guest physical memory as part of the user space memory allocated in the VM process.

While executing guest binary code, all the memory accesses will be remapped to host memory by a series of address translation processes. First, a guest virtual address (GVA) can be translated to a guest physical address (GPA) by walking through guest page tables. Then the host virtual address (HVA) is generated by the information stored in the GPA-HVA mapping table. Eventually the HVA will be translated to the host physical address (HPA) by host page tables. It has no efficiency if every guest memory access is forced to the surplus translation. The general solution of memory virtualization is to use a mechanism called shadow paging, which maintains a shadow of the VM's memory-management data structure to directly translate GVA to HPA.

3.2.2 Shadow paging

The overall design of shadow paging shown in Figure 4. When the hardware Prefetch Abort (PABT) trap or Data Abort (DABT) trap is caught by the ARMvisor, the trap will be handled following the shadow paging mechanism. The first step is to obtain the mapping information of the trapped GVA by walking through the guest page table. If the mapping does not exist, ARMvisor will deliver a true translation fault to guest. Otherwise, ARMvisor will translate the GVA to GPA and will check whether the access permission is allowed by interpreting the guest page table entries. If the memory access is illegal, ARMvisor will inject a true permission fault. When first two steps are finished and no true guest fault is generated, the trap would be handled as a certain hidden faults such as MMIO emulation fault, hidden translation fault or hidden protection fault in the following steps. MMIO access checker will examine whether the accessed memory region is located in the guest I/O address space. If it is a MMIO access, the trap will be handled by the I/O emulation model in QEMU. Otherwise, the trap is forwarded to the last two steps with respect to shadow page table (SPT) mapping and update.

Modern ARM processors use hardware-defined page tables to map virtual address space to physical address space. The translation table held in main memory has two levels: the first-level table holds section, super section translations and pointers to second-level tables. The second-level table holds large and small page translations. Four types of mapping size are defined as supersection (16MB), section (4MB), large page (64KB) and small page (4KB). For each guest page table (GPT) of guest process, ARMvisor will allocate one SPT to map it. All of the SPTs are cached and are searched while switching guest processes. Initially, the SPT loaded into host translation table base register (TTBR) is empty when running guest. As a result, any memory access will trigger a hidden translation fault. The fault GVA is used by ARMvisor to walk through the current first-level SPT and second-level SPT to fill new entries for the translation miss. The filled entries contain the mapping to host physical memory and the permission setting in guest's page table. ARMvisor will restore the execution context of the trapped instruction afterwards, then the memory access will natively be handled by the host MMU. Generally, the maintenance of SPTs has two considerations: One is how to emulate guest's permission under de-privileged mode; the other is how to keep the coherence between GPTs and SPTs. They will be explained in the next two subsections.

3.2.3 Permission model & Synchronization model

In the ARMv6 architecture, access to a memory region is controlled by the current processor mode, access domain and access permission bits. The current processor mode will be either non-privileged mode (User mode) or privileged modes (FIQ, IRQ, Supervisor, Abort, Undefined and System modes). The type of access domain is specified by means of a domain field of the page table entry and a Domain Access Control Register (DACR). Three kinds of domain access are supported: The type of NA (No access) will generate a domain fault for any memory access. The type of Client will check access permission bits to guard any memory access. The type of Manager will not check any permission, so no permission fault can be generated. The access permission bits are encoded in the page table entries with several fields, APX, AP, and XN. Table 3 shows the encoding of the access permissions by APX and AP. The XN bit acts as an additional permission check. If set to 1, the memory region is not executable. Otherwise, if XN clear to 0, code can execute from the memory region.

APX:AP[1:0]	Privileged mode	User mode
0:00	NA	NA
0:01	RW	NA
0:10	RW	RO
0:11	RW	RW
1:01	RO	NA
1:10	RO	RO

Table 3: The encoding of permission bits

Guest OS will use the permission model to separate kernel space from user spaces. Any user process can only access its user space. Access to kernel space or other user spaces is not allowed. Additionally, guest OS may use Copy-On-Write mechanism to speed up the process creation. Hence, hypervisor is obliged to maintain the equivalent memory access permission to reflect the desired guest behavior correctly. However, fully virtualizing guest's access permission is difficult since guest is executed in de-privileged mode for security issues. For example, once the permission of a memory region is set as (RW, NA) by guest OS, it means the memory region can be arbitrarily accessed in the guest kernel mode, while the access is forbidden in the guest user mode. In such case, since guest is physically executed in ARM's non-privilege mode (user mode), the access permission is remapped to RW/RW to emulate the exact memory access behavior in virtual privilege mode. Therefore, a thorough flush of SPTs is required for any mode switch between virtual privilege and non-privilege level because the access permission set in SPT differs based on the virtual privilege level of guest. Moreover, LDRBT/LDRT/STRBT/STRT in-

GPT Privilege mode	GPT User mode	SPT Privilege mode	SPT User mode
NA	NA	NA	NA
RW	NA	RW	RW
RW	RO	RW	RW
RW	RW	RW	RW
PO	NA	RO	RO
RO	RO	RO	RO

Table 4: Access Permission bits remapping in Shadow Page Table for kernel space region

structions must be treated as sensitive instructions since they access memory with user permission while the processor is in the privileged mode. Trapping each of these instructions results to huge slowdown in program execution, since they are frequent used by operating system to copy data from/to user space. Previous work [1] actually proposed a memory access permission model using permission remapping mechanism in SPT mentioned above and suffered great performance loss due to large amount of SPT flush and sensitive instruction traps.

In view of cutting down the overhead results from permission transformation, we proposed a methodology called double shadow paging in ARMvisor. We allocate two SPTs for one corresponding GPT: kernel SPT (K-SPT) and user SPT (U-SPT). Whenever translation misses occur, each newly allocated entry which maps the kernel space region in K-SPT is translated from GPT entry by ARMvisor to emulate guest's privilege level access, while entries in U-SPT have the same permission as in GPT. Table 4 demonstrates mapping of the access permission bits in kernel space region from GPT to SPT. ARMvisor loads either K-SPT or U-SPT to ARM's TTBR (Translation Table Base Register) depending on whether the virtual CPU mode it is switching to is privileged or non-privileged.

The mechanism of double shadow paging eliminates the need for SPT flushing after virtual mode switch, as was required in the previous single-table permission remapping methodology. However, the maintenance of double PTs adds complexity to the design of the synchronization model for GPT and SPT for shadow paging mechanism in ARMvisor. Memory usage in the system will also grow since more pages are allocated as first/second level of U-SPT/K-SPT during the lifecycle of a process comparing with the single SPT model.

	GUD	GKD
Virtual User Space	Client	No access
Virtual Kernel Space	Client	Client

Table 5: Domain configuration in Shadow Page Table

To reduce memory usage in the double shadow paging model, we utilize ARM's Domain control mechanism and map one GPT to only one SPT. Permission bits of the SPT entries that map to kernel space region are translated by ARMvisor as in the double shadow paging model. However, the Domain Access Control Register (DACR) is configured by ARMvisor to reflect the same protection level on the corresponding SPT. Table 5 lists the Domain configuration of ARMvisor. GUD and GKD represent the domain bits of SPT that maps to "Guest User space" and "Guest Kernel space" respectively. In Virtual User Space, GUD is set as client and GKD is set as No access to protect the guest kernel space from invalid access by its user space. In the Virtual Kernel Space region, both GUD and GKD are set as client. DACR is modified by ARMvisor for events that involve in virtual CPU mode switch such as system call, interrupts and return to user space.

ARMvisor uses memory trace to maintain the coherence between GPT and SPT. Once a SPT is allocated to map a GPT, the memory page of this GPT is write protected. This allows ARMvisor to determine when the guest OS tries to modify the GPT entries. The synchronization model is implemented by using a RMAP data structure which records the reverse mapping from guest physical pages to SPT entries. When a guest physical page is identified as GPT, the SPT entries pointing to the page will have their write permission bits disabled. Therefore, later modification of GPT will trigger a protection fault and ARMvisor will update the SPT to prevent it from becoming inconsistent with GPT.

3.2.4 Virtual MMU emulation

Other than guest physical memory allocation and shadow paging mechanism, ARMvisor must emulate a virtual MMU for guest OS to access. The ARM processor provides coprocessor, CP15, to control the behavior of MMU. The controls include enabling/disabling MMU, reloading TTBR, resetting Domain register, accessing FSR/FAR, changing ASID as well as operating

Cache and TLB. All of them are handled in the memory virtualization.

4 Cost model

In a virtualized environment, guest applications and operating systems inevitably suffer certain degree of performance degradation, which correlates to the design of hypervisor. For hypervisor developers, it is crucial to have a cost model to assist them on analyzing performance bottlenecks before adopting optimization. In this section, we will propose a cost model to formulate hypervisor overheads so as to discover the deficiencies of hypervisor.

The cost model defines *Guest Performance Ratio* (GPR) to evaluate a hypervisor design. As shown in (1), GPR is defined as the ratio of the execution time for an application running natively on the host (T_{host}) versus the time taken in the virtual machine (T_{guest}). The range of values is $0 < GPR <= 1$. Theoretically, T_{guest} is greater than or equal to T_{host}. If GPR is close to 1, this means the guest application runs in native speed. Otherwise, if GPR is close to 0, this means the guest application suffer huge overheads from virtualization.

$$GPR = T_{host}/T_{guest} \tag{1}$$

In (2), T_{guest} is divided into two parts: T_{native} represents the instruction streams of a guest application, which can be natively executed on the host, while T_{virt} represents the virtualization overheads. In practice, the value of T_{native} s usually constant and relats to the type of guest application. Meanwhile, the value of T_{virt} is related to both guest application type and hypervisor design. For example, in matrix manipulation applications, most time is spent on non-sensitive instructions, and as a result, the value of T_{native} is close to T_{guest}. If the application has many sensitive instructions, the value of T_{virt} will be higher.

$$T_{guest} = T_{native} + T_{virt} \tag{2}$$

Developers will optimize the T_{virt} which is decomposed into five parts, as shown in 3.

$$T_{virt} = T_{cpu} + T_{mem} + T_{suspend} + T_{idle} + \varepsilon \tag{3}$$

The components are described as follows.

T_{cpu} represents the cost of emulating sensitive instructions and exception generating instructions, such as software interrupt. It can be formed as following formula:

$$T_{cpu} = \sum Cs(i) \cdot T_{inst}(i) + \sum Ce(j) \cdot T_{excpt}(j)$$

T_{mem} represents memory virtualization overheads including emulating guest memory abort, handling shadow paging, and maintaining consistency between guest page table and shadow page table. It can be formed as following formula:

$$T_{mem} = Cm(0) \cdot T_{abt} + Cm(1) \cdot T_{shadow} + Cm(2) \cdot T_{sync}$$

T_{io} represents the cost of I/O virtualization, including memory mapped I/O and port mapped I/O. Both are emulated using the device model provided by the hypervisor. It can be formed as following formula:

$$T_{io} = \sum Ca(i) \cdot T_{mmio}(i) + \sum Cb(j) \cdot T_{portio}(j)$$

$T_{suspend}$ represents time during which the guest is temporally suspended by the hypervisor. For example, when an interrupt is coming, guest is trapped into hypervisor, and then hypervisor will handle this interrupt by its ISR. Switching to other virtual machines or host thread, the running guest will be suspended for a while.

ε is used to represent as the side effect from virtualization. For instance, cache or TLB flushing may cause guest application suffer the penalty of cache miss or TLB miss.

According to the cost model, we can figure out the optimization approaches in three directions:

1. Reducing the virtualization trap counts: try to reduce Cs, Ce, Cm, Ca, Cb variables
2. Reduce the emulation time : try to reduce T_{inst}, T_{excpt}, T_{abt}, T_{miss}, T_{sync}, T_{mmio}, T_{portio}, $T_{suspend}$, ε
3. Changing virtualization model by paravirtualization or hardware assist. This can eliminate some variables.

In the following section, we will follow these guidelines to optimize the CPU and Memory virtualization. For CPU virtualization, we proposed Shadow Register file

(SRF) to reduce traps from sensitive instructions. Additionally, Fast Instruction Trap (FIT) is used to reduce the emulation time of some sensitive instructions. In virtualizing ARM's MMU, we use paravirtualization technique to eliminate the synchronization overhead; this will be further illustrated in later sections.

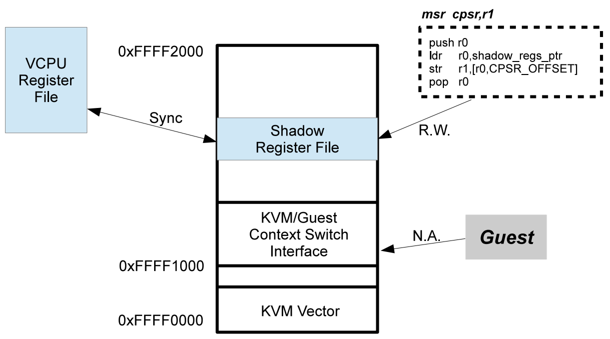
5 Optimization

5.1 CPU Optimization

Frequent lightweight traps for instruction emulation result in significant performance loss of the guest system. As a result, software techniques are important to minimize the frequency of "trap and emulation" count for sensitive instructions. Beside the lightweight and heavyweight traps mentioned before, two abstractions *Direct Register File Access* (DRFA) and *Fast Instruction Trap* (FIT) are proposed to accelerate the procedure of instruction emulation and reduce the overhead in CPU virtualization. The optimizing results showed great promise and will be demonstrated in later section.

We proposed *Shadow Register File* (SRF), which maps virtual CPU shadow states of the Register File into a memory region accessible by both the VMM and guest with read/write permission. Rather than unconditionally trapping on every sensitive instruction, DRFA speeds up the execution by *replacing* SWIs and sensitive instructions with a sequence of load or store instructions that access the SRF in guest address space. The methodology can be applied to instructions that only read or write the SRF and do not need privilege permission for subsequent emulation. Furthermore, VMM security is ensured because the SRF contains only the virtual state of guest, which if corrupted, would not affect the operation of the VMM.

Currently, DRFA is successfully employed to read and write the ARM PSR (Program Status Register), LDM/STM (2) and CP15 c1, c5 and c6 access in ARMvisor. To illustrate, pseudo-code for replacing a `mcr` instruction with DRFA is shown in Figure 5. The instruction is substituted by loading and effectively modifying the register copy in SRF to ensure the desired action. However, since SRF only contains subset of the VCPU state, it must be coherent with the VCPU Register File copy which ARMvisor acknowledges and without which the guest system's behavior would be unpre-



```
msr  cpsr,r1

push r0
ldr   r0,shadow_regs_ptr
str   r1,[r0,CPSR_OFFSET]
pop   r0
```

Figure 5: Shared memory mapping between KVM and Guest

dictable. In fact, to reduce cost of keeping them coherent, the synchronization is only held on demand and the overhead is actually low.

Unlike the previous instructions which can be replaced with DRFA, there are other instructions which require extra emulation and which must be finished in privileged mode. As mentioned in Section 3.1.1, several sensitive instructions that relate to ARM co-processor operations (including cache operations) require higher privilege for correct emulation. Vowing to simplify the emulation path for such cases, we replaced those instructions with *Fast Instruction Trap* (FIT). This consists of a series of pre-defined macros which encode information of the replaced instructions. ARMvisor's FIT handler is actually mapped in guest address space in high memory sections because the decoding process of instructions emulation is not necessary for FIT's. Thus, in contrast to Lightweight instruction traps, the emulation overhead of FIT is considerably lower since instructions can be handled without a context switch to ARMvisor.

5.2 Memory Optimization

To reduce the overhead of memory virtualization, ARMvisor paravirtualizes the guest operating system to support shadow paging. We found that many protection faults happened during guest process creation when applying the synchronization model mentioned previously to ARMvisor. Second-level guest page table is usually modified by guest OS for Copy-On-Write or remapping usage. After analyzing such behavior in detail, we simply add two hyper-calls in the guest source code to hint ARMvisor that modifications of second-level GPT were made by guest OS. One hyper-call is added as the guest OS sets the page table entry; the other one is added to

Device	Description
CPU	ARM Cortex-A8
CPU Clock Frequency	720 MHz
Cache hierarchy	16KB of L1-Instruction Cache 16KB ofL1-Data Cache 256KB of L2 Unified cache
RAM	256MB
Board	TI BeagleBoard

Table 6: Hardware configuration

notify ARMvisor when guest OS frees a second-level page table. These notifications will free the synchronization overhead for guest second-level page tables.

6 Evaluation

We currently support ARMv7 for host and ARMv6 for guest. For the host, we used ARM TI BeagleBoard with ARM Cortex-A8 as our platform. The detailed hardware configuration is showen in Table 6. The software parts consists of the ARMvisor (Linux 2.6.32), QEMU-ARM (0.11.1) as well as para-virtualized Guest (Linux 2.6.31.5). In the guest, we use Realview-eb ARM11 as our hardware environment for guest OS.

To demonstrate the overhead of virtualization on ARMvisor, we first measured the slowdown of ARMvisor compared with native performance when running the micro-benchmarks LMBench [11]. Then we analyze the overhead in depth by an internal profiling tool and we develop a trap cost model to explain what other overheads exist. Finally, groups of application benchmarks will also be evaluated the performance of ARMvisor.

6.1 Profiling counter

Table 7 shows the trap counts when executing a simple program in the guest that does nothing but returned immediately. Column kvm_orig shows trap counts without CPU and Memory optimization, while in column kvm_opt column all optimizations are enabled. This do-nothing program actually measures the overhead of process creation. During the procedure of fork and exec, page tables are modified frequently for mapping libraries or data into the address space. Common Operating Systems use *Demand Paging* to manage the memory usage. Nonetheless, the approach generates great overhead in Memory Virtualization due to the design of our page table synchronization model. In our synchronization model, the same page table is repeatedly being modified when forking and loading program for execution. Since we write-protect the page table when the corresponding shadow page table is found, repeated filling of entries in the page table generates consequent permission fault, and causes the corresponding shadow page being zapped often. We provide a solution to such use case by adding hyper-calls for page table modification. Page table protection traps disappear because the synchronization model is not necessary anymore. Furthermore, the subsequent page fault is eliminated since we map the memory address in shadow page in the hyper-call for page table entry modification before the guest accesses the page.

	kvm_orig	kvm_opt
sen_inst_trap	12825	813
irq	6	3
fast_trap	5034	5562
dat_trans	1301	25
protection	299	0
mmio	95	56
Dabt_true	178	186
Pabt_kvm (inst_trans)	758	6
Pabt_true	106	106
mem_pv	0	714
Total_trap	20602	7471

Table 7: Profiling count for nothing

As illustrated above, kvm_opt has far less sensitive instruction traps since abundant portion of traps were replaced with direct access to SRF in the guest address space. On the other hand, the translation misses for both data and instruction access are reduced tremendously in kvm_opt, since we map the corresponding memory address in shadow page table when the hyper-call for guest page table modification is triggered. The protection trap count for page table modification also comes down to zero since any attempts by guests to modify the second level page table is acknowledged by KVM through hyper-calls and the protection model is removed. Even hyper-calls for guest page table set/free introduce certain overhead to the system; the total trap count of Memory Virtualization does decline enormously. In conclusion, after the optimization in CPU and Memory Virtualization are applied, the total trap count is only about 36% compared to original version.

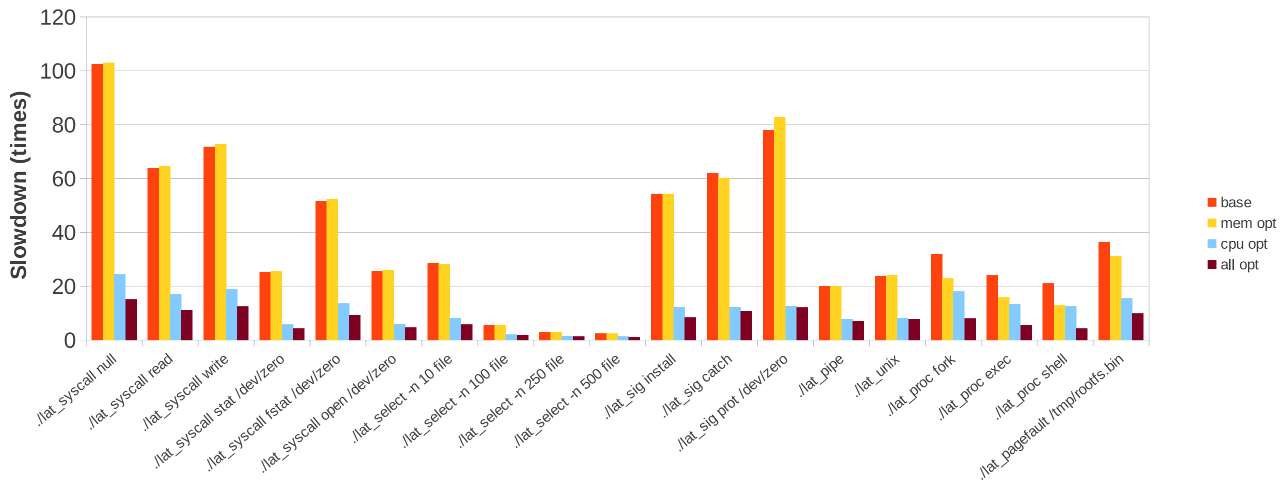


Figure 6: Ratio of performance slowdown on LMBench

6.2 LMBench

To evaluate the performance when applying different optimization on ARMvisor in guest system's basic operation, LMBench is measured to help us understand the bottleneck of our current design. LMBench contains suites of benchmarks that are designed to measure numerous system operations such as system call, IPC, process creation and signal handling. We refer to the paper [12] to choose sets of benchmarks in LMBench and measured them in various environments and compare them with native performance.

Figure 6 presents the ratio of slowdown on 3 different extensions of features comparing to native Linux kernel. The *simple* version means no optimization applied in the ARMvisor, all sensitive instructions will be trapped into hypervisor and the guest OS does not inform ARMvisor for creating or modifying a page table so hypervisor needs to trace the guest page table. The *cpu-opt* version improves the performance of ARMvisor in three aspects: firstly it use SRF technique to reduce the trap overheads from several sensitive instructions for instructions as MSR, MRS and CPS. Secondly, it uses binary translation to change guest TLB/Cache operations on-the-fly. All of the TLB operations are translated into NOP operations since the guest no longer needs to maintain hardware TLB, and the hypervisor will assist the guest to maintain TLB according to the shadow page tables. And finally, the overhead of exception/interrupt handling in guest OS is reduced by finishing the combination logic of guest code in ARMvisror to largely reduce the traps of critical instructions. The *full-opt* version further comprise the memory para-virtualization which uses hyper-calls to inform ARMvisor about guest page table modification and finalization, and ises SWI fast trap to reduce the memory tracing overhead and SWI delivery overhead.

As can be seen in Figure 6, system call and signal handling related benchmarks suffer great performance loss in ARMvisor. Performance slowdown is less significant in `lat_select` benchmark, typically when the number of selected fd increases since the accounted time portion for native execution increases. Nonetheless, the slowdown still reaches 30 times when selecting 10 files. After measuring those benchmarks using the proposed cost model, we figured that the performance gap could be mainly attributed to the frequent lightweight traps for instruction emulation during the execution path. Each trap includes a pair of context switches between guest and VMM, which is time consuming. Decoding individual sensitive instructions for correct emulation also generate latencies for the total execution. Moreover, since the SWI exception is indirectly generated by ARMvisor's virtual exception emulator, the cost of exception delivery adds additional slowdown to the system call operation. However, after applying CPU-opt in ARMvisor, performance improves largely since the times of lightweight trap for instruction emulations reduces. Mem-opt hardly contributes any performance improvement for system call and signal han-

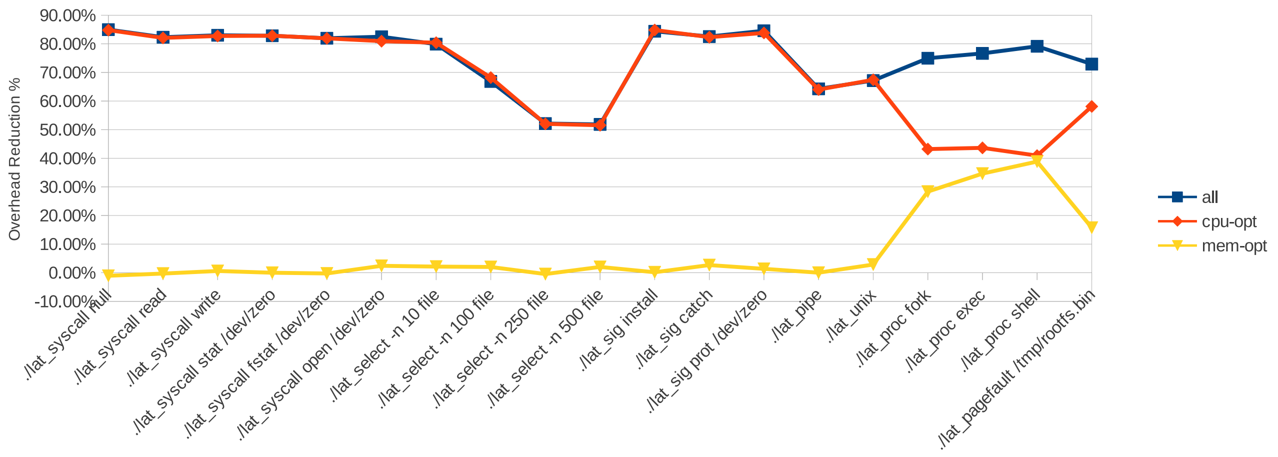


Figure 7: Ratio of performance improvement on LMBench

dling since those operations also would not require large amounts of page table modification. Figure 7 demonstrates that All-opt improves benchmarks' performance in categories of system call, signal handling and select approximately over 80% on average.

The All-opt version has less profound improvements in IPC related benchmarks including `lat_pipe` and `lat_unix`. According to our profiling results, I/O access rate in IPC operation is much higher contrast to the three types of benchmarks mentioned above. The behaviors are quite different in pipe and Unix Socket operation. In the scenario of communicating through pipe, the guest kernel scheduler is often called to switch the process for inter-process message sending and receiving. Linux Kernel scheduler fetches hardware clock counter as TSC (Time Stamp Counter) for updating the run queue reference clock for its scheduling policy. Since we currently focus on minimizing CPU and Memory virtualization overhead, All-opt improve less significantly in pipe operation, since the overhead results in frequent heavyweight traps exists for I/O emulation.

Process creation benchmarks like `lat_procfork`, `lat_procexec` and `lat_procshell` have similar behavior as the do-nothing program. As previously mentioned, synchronization for page tables in Memory Virtualization results in considerably larger overhead for process operations than in the other benchmarks. As shown in Figure 7, the performance of process creation operation improves about 45% on average in Mem-opt solely.

Benchmark `lat_pagefault` tests the latency of page fault handling in operating system. Even though we map the page in shadow on the PTE write hyper-calls to prevent further translation miss; performance improvement in Mem-opt is smaller than in process creation benchmarks. Profiling by our cost model, we found that sensitive instruction traps account for larger proportion of total traps than those of process creation. As a result, the optimization for CPU virtualization has more notable effect on performance improvement.

7 Conclusion and future works

In this paper, we investigated the challenges of constructing virtualization environments for embedded systems by the implementation of an ARM based hypervisor, ARMvisor. ARMvisor assumes that ARM processor has no hardware virtualization extension. The experimental results show that ARMvisor suffers huge performance degradation when techniques such as trap-and-emulation and shadow paging are being adopted. As a result, we formulized a cost model for discovering the performance bottlenecks of hypervisor in depth. Furthermore, based on the cost model, several optimization methodologies are proposed to reduce the overheads of CPU and Memory virtualization. The experimental results have shown leaping performance improvement with up to 4.65 times speedup in average on LMBench by comparing with our original design. We conclude that to virtualize embedded ARM platforms without hardware virtualization extension in current architecture, the cost model is crucial to assist developers to further optimize their hypervisors.

References

[1] Christoffer Dall and Jason Nieh Columbia University, "KVM for ARM," in *the proceeding of Linux Symposium 2011*

[2] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, vol. 17, no. 7, pp. 412-421, 1974.

[3] Gernot Heiser, "The role of virtualization in Embedded Systems", in *IIES '08 Proceedings of the 1st workshop on Isolation and integration in embedded systems*

[4] Gernot Heiser, Ben Leslie, "The OKL4 microvisor: convergence point of microkernels and hypervisors", in *APSys '10 Proceedings of the first ACM asia-pacific workshop on Workshop on systems*

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

[6] Trango. "Trango: secured virtualization on ARM", http://www.trango-vp.com

[7] VirtualLogix. "VirtualLogix Real-Time Virtualization and VLX", http://www.osware.com

[8] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, Andrew Warfield, "Xen and the Art of Virtualization", in *SOSP '03 Proceedings of the nineteenth ACM symposium on Operating systems principles*

[9] Joo-Young Hwang; Sang-Bum Suh; Sung-Kwan Heo; Chan-Ju Park; Jae-Min Ryu; Seong-Yeol Park; Chul-Ryun Kim; Samsung Electron. Co. Ltd., Suwon , "Xen on ARM: System Virtualization using Xen Hypervisor for ARM-based Secure Mobile Phones", in *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*

[10] Carl A. Waldspurger VMware, Inc., Palo Alto, CA, "Memory Resource Management in VMware ESX Server", in *ACM SIGOPS Operating Systems Review - OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*

[11] McVoy, L. W., & Staelin, C. (1996). "lmbench: Portable tools for performance analysis," In *USENIX annual technical conference*. Barkely (pp. 279-294), Jan. 1996.

[12] Yang Xu, Felix Bruns, Elizabeth Gonzalez, Shadi Traboulsi, Klaus Mott, Attila Bilgic. "Performance Evaluation of Para- virtualization on Modern Mobile Phone Platform", in *Proceedings of International Conference on Electrical, and System Science and Engineering*

[13] Keith Adams , Ole Agesen, "A comparison of software and hardware techniques for x86 virtualization", *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, October 21- 25, 2006, San Jose, California, USA

[14] Joshua, LeVasseur, et al. "Pre-Virtualization: Slashing the Cost of Virtualization", http://l4ka.org/publications/2005/previrtualization-techreport.pdf, 2005.

Clustering the Kernel

Alexandre Lissy
Mandriva S.A.
alissy@mandriva.com

Jean Parpaillon
Mandriva S.A.
jparpaillon@mandriva.com

Patrick Martineau
University François Rabelais Tours, Laboratory of Computer Science (EA 2101)
Team Scheduling and Control (ERL CNRS 6305)
patrick.martineau@univ-tours.fr

Abstract

Model-checking techniques are limited in the number of states that can be handled, even with new optimizations to increase capacity. To be able to apply these techniques on very large code base such as the Linux Kernel, we propose to slice the problem into parts that are manageable for model-checking. A first step toward this goal is to study the current topology of internal dependencies in the kernel.

1 Introduction

As a general goal of "applying model-checking techniques to the Linux Kernel", we studied the literature around this topic in [9]. One major conclusion is that despite the use of model-checking *under the hood* in some tools (such as Coccinelle [4]), direct application of existing model-checking tools and algorithms seems impossible because the whole kernel is too big, i.e. contains too much state.

To circumvent this limitation, we propose to work at the source code level instead of working on the model-checking algorithm itself. By breaking down the code base into smaller chunks, the number of states that must be analyzed is reduced. Before working on the slicing itself, we propose to study the current internal dependency topology of the kernel.

In the remainder of this paper, we will first make a brief presentation in section 2 of the topic discussed in [9]; then in section 3 we describe how we built the graph representing the kernel. Section 4 will focused on the analysis of what is inside the graph and what it means for us, before we conclude.

2 Model-Checking and Kernel

We are interested in answering the question: "is it possible to directly apply model-checking to the Linux Kernel code base". A literature survey for kernel-related verification, including but not limited to Linux, that might be linked to model-checking, revealed at least two major projects:

- SLAM, initiated by Microsoft
- Coccinelle

A third major project to be cited is the effort conducted by Engler et al.

2.1 A first step: compiler extensions to check rules

Using compiler extensions to check system rules has been the first major attempt to check system code. This has been performed as a fork of the GCC compiler with a matching language called METAL, which allows definition of a state machine and patterns to match in source code. Thanks to this tool, called `xgcc` [6], a first major empirical study of errors in kernel source code has been performed [5] against Linux, OpenBSD and the FLASH embedded system. Results showed that device drivers were the main hot point in term of bugs.

Those results were corroborated ten years later by the Coccinelle team as part of a similar updated survey [12] using their own tools. Newer kernels showed a real improvement since the first study.

2.2 Verification at Microsoft: SLAM

With the introduction of boolean programs [2], work started on verification of code at Microsoft. The goal was to verify correct usage of interfaces, i.e. APIs [3]. Boolean programs are an abstraction of source code where all variables are substituted with booleans: this allows for state evaluation to take place. This is code using a CEGAR (Counter-Example Guided Abstraction Refinement) loop, allowing to determine feasible paths in the source code. It allows, *in fine*, to find code-paths that are not good and hence bugs.

Results are good enough (in term of bug finding and false positive) that the tool has been included in the Windows 7 DDK as the Static Driver Verifier [1].

2.3 Coccinelle, tracking Linux bugs

The Coccinelle tool was designed to apply massive changes to APIs, defined as "collateral evolutions" [8, 10] and has been rapidly used to hunt bugs in the code. In order to apply API evolutions, the tool must work at a higher level than only source code: it is a "semantic patch" tool, i.e. instead of replacing a line by another, the code is manipulated at the level of adding or removing a parameter to a function. This is internally done by transforming the patch into a temporal logic formula, which is matched against the source code: model-checking.

Once the tool could match source code, its developers thought of using it to find bugs: the Semantic Patch Language is used to describe what to change and how; with a couple of new operators introduced, it can be used to find bugs. So naturally Coccinelle has been used to track them. It is now being used for discovering protocols, i.e. how an API should be used [7], and to track bug life cycles [11].

3 Kernel graph

The goal is to be able to study dependencies between "modules" in the source code of the kernel. We consider those modules to be the `.o` files, produced during the compilation. This assumption is done because:

- it is easy to extract symbols usages from object files using elf parsing (`readelf` for example)
- it can be matched to C source files

The directed graph is then defined as:

- Nodes are the objects files that have been analyzed
- Edges are symbols used/exported by object files
- Edges are directed.

Edge directions are defined with the source being the object file (A) that is uses a symbol and the target being the object file (B) that exports this symbol: this symbolize a dependency: $A \rightarrow B$.

The analyzed object files are limited to the architecture that the code has been built for. To date, all the processing has only been done on an AMD64 system. Also, since we limit ourselves to build the kernel using the `defconfig` and `allyesconfig` configurations, we are sensitive to changes of those configurations. The first one will only build with a subset of modules that fits for the system, while the second will enable much more code, nearly everything.

Code used to extract all the information is available at `git://git.mandriva.com/users/alissy/callgraph.git`, and is written in Python using SQLAlchemy, PyLibELF and Tulip modules.

4 Graph analysis

In this section we explain what is measured on the graph, and why. Then we present, explain and try to interpret those results.

Linux versions considered were 3.0 through 3.4, covering a time span of nearly one year.[1]

4.1 Measures

We will look at occurrence of edges in the graph: they are labeled with the symbols corresponding to the relation, so it allows us to see how much a symbol is used. From this we can derive which symbols are the most important in term of usage.

[1]3.0 released July 22nd 2011; while 3.4 released May 20th 2012.

We will see how dense the graph is. Graph density is defined as follows, for a given graph $G = (E, N)$:

$$d_G = \frac{|E|}{|N| \times (|N| - 1)}$$

We will check the average path length inside the graph, that is how "far" apart two nodes are. It is computed directly thanks to the `Tulip`[2] library, which states:

> Returns the average path length of a graph, that is the sum of the shortest distances for all pair of distinct nodes in that graph divided by the number of those pairs. For a pair of non connected nodes, the shorted distance is set to 0.

We will have a look at the degrees of the graph, in and out. As a reminder, in-degree of a node is the number of exported symbols used, and out-degree is the number of imported used ones. By used, it means we can have duplicates: a symbol is used by several other ones.

To have a better view of the dependency, we propose to produce "heatmap" of the kernel dependency, with subdirectory granularity, available in section 4.7.

4.2 Graph size

Before we present the specific measures, we can already have a look at the general graph size: number of nodes, number of edges. The raw values are available in Figure 1. Variations are presented in Figure 2 and are computed using the previous one, considering the first version as a basis. Each version is compared to its first ancestor, e.g. `v3.1` against `v3.0`. In the second table, positive values indicate increases, while negative values indicate decreases.

We also measured the size of the code base using SLOCCount (v2.26) to compare the size evolution of the graph and the related code base. This information is presented in Figure 3. The evolution is computed the same way than previously explained.

A first observation we can make is that, looking at Figures 3 and 2, while the size of the code base evolution is similar between `defconfig` and `allyesconfig`, and raw numbers shows that the difference is very small, it seems not to be correlated with the evolution of nodes nor edges.

[2] http://tulip.labri.fr

	Nodes	
Version	`defconfig`	`allyesconfig`
v3.0	1836	9593
v3.1	1842	9764
v3.2	1861	9897
v3.3	1874	10044
v3.4	1871	10172
	Edges	
Version	`defconfig`	`allyesconfig`
v3.0	51700	321463
v3.1	52390	332865
v3.2	53005	337717
v3.3	53418	344314
v3.4	53646	349271

Figure 1: Number of nodes and edges

	Nodes	
Version	`defconfig`	`allyesconfig`
v3.0	-	-
v3.1	+0.33%	+1.78%
v3.2	+1.03%	+1.36%
v3.3	+0.70%	+1.49%
v3.4	−0.16%	+1.27%
	Edges	
Version	`defconfig`	`allyesconfig`
v3.0	-	-
v3.1	+1.33%	+3.55%
v3.2	+1.17%	+1.46%
v3.3	+0.78%	+1.95%
v3.4	+0.43%	+1.44%

Figure 2: Variations in nodes and edges

	SLOCCount	
Version	`defconfig`	`allyesconfig`
v3.0	9614824	9612505
v3.1	9704743	9702470
v3.2	9862036	9860466
v3.3	9977312	9976172
v3.4	10120350	10119606
	Evolution	
Version	`defconfig`	`allyesconfig`
v3.0	-	-
v3.1	+0.94%	+0.94%
v3.2	+1.62%	+1.63%
v3.3	+1.17%	+1.17%
v3.4	+1.43%	+1.44%

Figure 3: Code base size evolution

4.3 Measure: Symbols occurrences

Symbols occurrences are computed simply by counting how many times a symbol is used, i.e. how many edges with this symbol exists in the graph. Raw values for kernel v3.0 are available in Figure 4; values for other versions (v3.1 to v3.4) are not provided but they are close. The table is limited to top 10. In the most used edges, throughout the versions, we can derive three categories:

- String manipulations, with `printk()` being one of the most used symbols
- Memory management, for example functions `kfree()`, `kmalloc_caches()`, `kmem_cache_alloc_trace()` and `__kmalloc()`
- Locking primitives case, including `mutex_lock()` (in the `defconfig`), `mutex_lock_nested()` (in the `allyesconfig`) and `mutex_unlock()`

Looking at the 50 most used symbols, there is not much change in the categories involved: strings see more symbols used (`sprintf()`, `str*()`); memory management sees its space extended with for example `memset()`, `memcpy()`, `_copy_from_user()` and `_copy_to_user()`; locking primitives are also in the top 50.

Extending our view from top 10 to the top 50, however, shows a difference when looking at the results on `allyesconfig`: strings functions are less present in the hall of fame, and driver-related symbols appear: `drv_get_drvdata()`, `drv_set_drvdata()`. Also, workqueues symbols (especially `__init_work()`) appear in the top 50 when looking at `allyesconfig` but not in `defconfig`.

Symbol	Occurrences
`defconfig`	
_raw_spin_lock	782
_cond_resched	806
__kmalloc	846
current_task	864
mutex_lock	912
mutex_unlock	936
kmem_cache_alloc_trace	1254
kmalloc_caches	1270
printk	1658
kfree	1706
`allyesconfig`	
mutex_lock_nested	4614
mutex_unlock	4898
__kmalloc	5156
__stack_chk_fail	6258
kmem_cache_alloc_trace	6922
kmalloc_caches	6950
kfree	10152
printk	11336
__gcov_init	19014
__gcov_merge_add	19014

Figure 4: Symbols occurrences in the graph, kernel v3.0

	Density	
Version	`defconfig`	`allyesconfig`
v3.0	0.015346	0.003494
v3.1	0.015449	0.003492
v3.2	0.015313	0.003448
v3.3	0.015219	0.003413
v3.4	0.015333	0.003376

Figure 5: Graph density

4.4 Measure: Graph density

The raw values are available in Figures 5 and 6.

Figure 5 shows that the density for `allyesconfig` is slightly lower than that of `defconfig`, which is not a surprise considering the definition of both. A fact that is not that obvious, however, is that in `defconfig`, on the set of versions studied, density is quite stable; while in `allyesconfig` we can see that it is constantly dropping, even though the decrease is slow.

To have a better understanding we looked more closely at the density inside kernel v3.0. For each subdirectory containing source code, we compare density between `defconfig` and `allyesconfig`; the values are available in Figure 6. Some directories are highly impacted: `crypto`, `drivers`, `fs`, `net`, `security`, `sound` while some other are not, or only slightly: `arch`, `block`, `init`, `ipc`, `kernel`, `lib`, `mm`.

4.5 Measure: Average path length

The raw values are available in Figure 7. A more detailed per-subdirectory overview on kernel v3.0 to v3.4 is available in Figure 8 for the `defconfig` build and in Figure 9 for the `allyesconfig` build.

In Figure 7, we observe that in both `defconfig` and

Linux v3.0	Density	
Subdir	`defconfig`	`allyesconfig`
arch	0.039320	0.035418
block	0.268398	0.281667
crypto	0.241935	0.073537
drivers	0.021583	0.002376
fs	0.063002	0.018673
init	0.291667	0.291667
ipc	0.712121	0.719697
kernel	0.122087	0.126854
lib	0.019572	0.016000
mm	0.309949	0.299454
net	0.060322	0.015070
security	0.288762	0.103541
sound	0.173263	0.024607

Figure 6: Graph density per subdirectory, v3.0

	Average Path Length	
Version	`defconfig`	`allyesconfig`
v3.0	2.410770	2.013618
v3.1	2.413076	2.013085
v3.2	2.415017	2.013867
v3.3	2.417895	2.014709
v3.4	2.429603	2.014612

Figure 7: Graph average path length

`allyesconfig` the average path length slowly increases, at least between versions 3.0 and 3.4; moreover, there is an order of magnitude of difference in the increase between both build configurations: the increment for `defconfig` is around 0.002 (although going from 3.3 to 3.4 shows an increment of 0.012), while in `allyesconfig` it is around 0.0006 with two majors points: going from 3.0 to 3.1, we have a decrease of about 0.0005 and from 3.3 to 3.4 it also decreases but only 0.00009.

Since the major difference between both consists of more drivers (not only the `drivers` subdirectory), it is trivial to assume that the reason is inside those. We propose to have a closer look at this in the table available in Figure 9, in Section 4.5.1. To have a better understanding of the "big" increase between 3.3 and 3.4 in `defconfig`, we will have a look at the details in Figure 8 in Section 4.5.2.

4.5.1 Detailed Average Path Length, kernel 3.0 to 3.4, `defconfig`

In Figure 8, we can notice:

- The `arch` subdirectory is nearly constantly decreasing, apart from the 3.3 to 3.4 evolution which shows a slight increase.
- The `block` subdirectory shows a constant average path length, with a step between 3.2 and 3.3, going from 1.80 to 2.17, before and after it is strictly the same values.
- The `crypto` and `drivers` subdirectories are evolving together, especially with 3.2 showing a light decrease on both, while they increase the rest of the time.
- For `fs` we can observe a constant decrease, although kernel 3.4 shows a noticeable increase. This is probably to be linked with the number of commits concerning `cifs` (54), `xfs` (57), `ext4` (59), `nfsd` (61), `proc` (64), `btrfs` (118), and `nfs` (181).
- While `init` is slowly increasing, `ipc` and `mm` are much more stable.
- The `lib` subdirectory shows a decrease, dropping from 1.15 to 0.96.
- Security-related subdirectory, `security`, is having a rough time, alternating between increase, decrease and stability (3.2 and 3.4 shows the same values
- The `net` subdirectory also shows an alternating behavior.
- Main part of the kernel, in the `kernel` subdirectory, is decreasing in a quite stable way, dropping from 2.0607 to 2.0417 over the studied versions.
- The `sound` part is also slowly decreasing over versions.

So, generally speaking, some parts of the kernel are "shrinking", i.e. each sub-part is getting closer to its neighbors: this is when average path length decreases. Some other parts are in expansion, with a good example being the `drivers` part. Finally, the global increase between 3.3 and 3.4 observed in the previous table can be explained by the changes in `crypto`, `drivers`, `fs` and `net`.

	Average Path Length – `defconfig`				
Subdir	v3.0	v3.1	v3.2	v3.3	v3.4
arch	2.140192	2.132726	2.118681	2.090058	2.096498
block	1.809524	1.809524	1.809524	2.169355	2.169355
crypto	1.790323	1.817204	1.802151	1.881048	1.989919
drivers	2.910024	2.911436	2.897755	2.919029	3.008717
fs	2.570742	2.532482	2.525733	2.491692	2.680926
init	1.805556	1.833333	1.833333	1.944444	1.944444
ipc	1.363636	1.348485	1.348485	1.348485	1.348485
kernel	2.060689	2.067626	2.059235	2.049458	2.041655
lib	1.155375	1.140758	1.125184	1.002295	0.964706
mm	1.914116	1.914116	1.917551	1.911837	1.921633
net	2.432165	2.359474	2.475096	2.350066	2.490345
security	2.613087	2.563300	2.832659	2.834008	2.832659
sound	2.191919	2.191287	2.181980	2.181420	2.181420

Figure 8: Graph average path length per subdirectories, v3.0 to v3.4, `defconfig`

	Average Path Length – `allyesconfig`				
Subdir	v3.0	v3.1	v3.2	v3.3	v3.4
arch	2.079778	2.073589	1.959125	1.969219	2.192387
block	1.930000	1.907692	1.907692	2.278049	2.278049
crypto	1.897335	1.897335	1.921848	1.906866	1.920599
drivers	3.005424	3.011469	3.028173	3.009538	3.000608
fs	3.037271	3.044064	3.048455	3.016527	3.007562
init	1.861111	1.861111	1.861111	1.861111	1.861111
ipc	1.363636	1.348485	1.348485	1.348485	1.348485
kernel	1.915090	1.914885	1.914908	1.914107	1.914962
lib	1.026194	0.956759	0.953852	1.248667	1.254550
mm	1.934973	1.938251	1.942359	1.958333	1.960813
net	2.855261	2.843975	2.854679	2.840436	2.836698
security	3.212210	3.227209	3.187363	3.187960	3.177200
sound	2.546933	2.495413	2.533996	2.530829	2.530232

Figure 9: Graph average path length per subdirectories, v3.0 to v3.4, `allyesconfig`

4.5.2 Detailed Average Path Length, kernel 3.0 to 3.4, `allyesconfig`

Figure 9, shows that the behavior for `arch` is not exactly the same as in `defconfig` build configuration. `crypto` increases slightly. Meanwhile `drivers` starts higher than in `defconfig`, increases slightly in 3.1, and then decreases in v3.3 to finish at a similar level than in `defconfig`. The `fs` subdirectory, however, shows an inverse behavior in `allyesconfig` than in `defconfig`, with higher average path length, increasing from roughly 2.5 to 3.0.

The `init` directory remains stable, with similar values than in `defconfig` configuration. Similarly `ipc` remains unchanged, as does `kernel`: the values are nearly constant along versions for `allyesconfig`, and shows a light difference (1.91 versus 2.05) from `defconfig`. The `lib` subdirectory also shows an inverse behavior in `allyesconfig`. The `mm` subdirectory shows a slight increase, while in `defconfig` there was a slight decrease.

The alternating behavior observed for `net` is confirmed with values ranging from 2.83 to 2.85. In `allyesconfig`, the `security` subdirectory shows a constant decrease and an important delta, ranging from 3.21 to 3.17 while it was between 2.56 and 2.83 for `defconfig`. Finally, the `sound` subdirectory is quite stable around 2.53 with a light decrease at 2.49 for kernel v3.1, exposing a similar behavior than in `defconfig` build configuration, the only difference being the values: around 2.18.

Major differences are `fs`, `security`, `sound` and `lib`. One could have expected that `drivers` showed a much more bigger difference.

4.6 Measure: Degrees

The raw values, from an aggregated subdirectory point of view are available in Figure 10 for kernel v3.0 and 11 for kernel v3.4. Those values are another point of view of the heatmap available, for example, in Figure 12. As a reminder, in-degree of a node in the graph we use maps to *exported* symbols, i.e. they are used by other nodes.

A first look at values shows that:

- Between successive versions of the kernel, there is an increase in degrees, both in and out. This is consistent with the expansion we already exposed.
- The top three consuming subdirectories are `drivers`, `net` and `fs`, in that order for `defconfig` and `drivers`, `fs` and `net` for `allyesconfig`.
- The top five consumed subdirectories are `kernel`, `drivers`, `net`, `fs` and `mm` in `defconfig`. The `allyesconfig` mode shows the same results, apart from `mm` being replaced by `lib`.

Those results can be generalized to versions from 3.0 to 3.4, even though we can notice a decrease in out-degree for kernel 3.4 in `defconfig` build configuration for the `arch` subdirectory. A closer look at this specific subdirectory shows that:

- In-degree is constantly growing, meaning more and more symbols exported.

Linux v3.0	Degrees in	
Subdir	`defconfig`	`allyesconfig`
arch	4540	16112
block	541	1621
crypto	258	907
drivers	8803	87986
fs	6097	28262
init	85	135
ipc	103	104
kernel	12876	92789
lib	4006	26397
mm	5418	24879
net	6504	29760
security	721	1403
sound	1681	11054

	Degrees out	
Subdir	`defconfig`	`allyesconfig`
arch	3489	6831
block	602	906
crypto	497	1322
drivers	17081	191946
fs	7751	42208
init	316	357
ipc	354	431
kernel	4298	6904
lib	396	1001
mm	1721	2798
net	10668	37958
security	1252	3176
sound	3155	25304

Figure 10: Graph degrees per subdirectory, v3.0

Linux v3.4	Degrees in	
Subdir	`defconfig`	`allyesconfig`
arch	4983	19822
block	633	1816
crypto	258	1087
drivers	9106	94828
fs	6224	30152
init	85	138
ipc	105	106
kernel	13336	101674
lib	4010	29268
mm	5551	25978
net	6704	31429
security	747	1580
sound	1863	11337

	Degrees out	
Subdir	`defconfig`	`allyesconfig`
arch	3421	7594
block	733	1192
crypto	515	1382
drivers	17696	207607
fs	8080	46375
init	325	374
ipc	360	464
kernel	4599	7767
lib	399	1149
mm	1776	3105
net	10960	41552
security	1271	3888
sound	3430	26482

Figure 11: Graph degrees per subdirectory, v3.4

- Out-degree is increasing-decreasing: 3489 for v3.0, 3421 for v3.1, 3608 for v3.2, 3343 for v3.3 and finally 3421 for v3.4.

4.7 Measure: Heatmaps

Heatmaps are generated from the previously presented dependency graph. It allows to more easily visualize how things are organized:

- First, we merge together nodes at a defined depth (in term of subdirectories), while keeping edges as they were originally: hence, we get the same dependencies but with a bigger granularity, more human-readable
- Then, we process all the newly-created nodes, and we count the number of edges between each pairs of nodes
- Finally, to be able to compare between versions of the kernel, we normalize things

A first look at two heatmaps, Figures 12 and 13 which are Linux v3.0 kernel's root, respectively in `defconfig` and `allyesconfig` builds. Note that color scale maximums differ at 0.16 and 0.3; we can see the same results as those presented in Section 4.4.

A closer look at the `drivers` subdirectory is available in Figure 14. A first observation is that dependencies are mainly contained inside each subdirectory: there is a thin line with variable value, but nearly always maximum, that runs for each subdirectories' intersection with itself. We can also note that there are three other lines, yet lighter: `base`, `pci` and `usb`. Those directories contains generic stuff for all drivers, or `PCI/USB` stack, hence it is normal that they are being used by a lot of other sub-directories. Other versions of the kernel (e.g. v3.4 in Figure 15) shows nearly the same behavior, only the range of values changes.

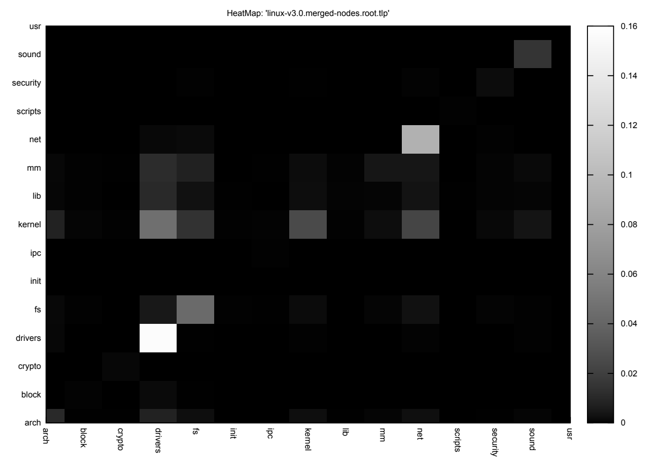


Figure 12: HeapMap of Linux v3.0, `defconfig`

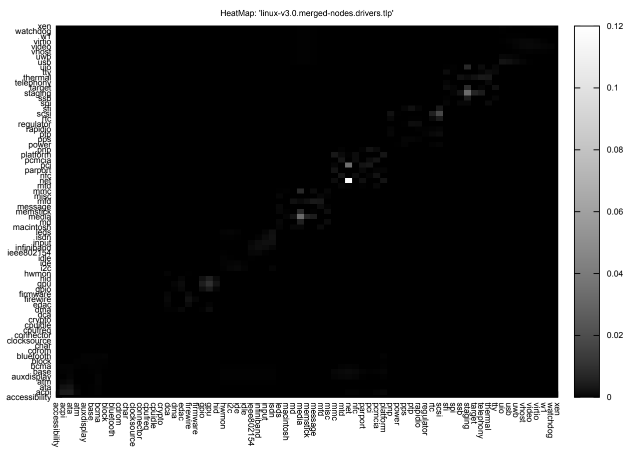


Figure 14: HeapMap of Linux v3.0, `drivers` subdirectory

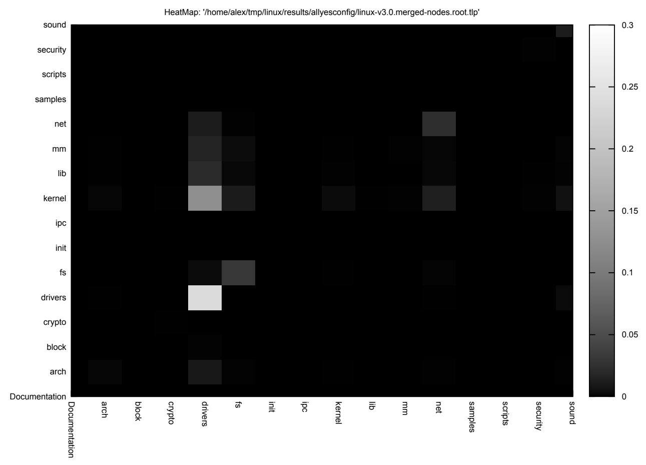


Figure 13: HeapMap of Linux v3.0, `allyesconfig`

Another kind of "heatmap" could be created, that we could call *binary heatmap*, in which we do not count the number of edges between two nodes, but only the existence of an edge between those nodes: it is an adjacency matrix.

5 Conclusion

Those few metrics allows us to have a better look at the kernel: a first interesting fact, easily readable in the heatmaps and not that surprising, is that dependencies are rather located in spots. Drivers depend mainly on stuff that concern the specific driver, a bit on generic stacks such as `base`, `pci` or `usb`, plus a couple of generic libraries in the kernel such as `mm` for memory management, `lib` for things like strings handling and `kernel` for basic stuff.

A second result, which is also not surprising, is that the kernel is expanding, not only in term of code base size (this fact is well known), but we can see it also from a density and average path length: over time, at least within the time span studied, density is decreasing and average path length is increasing. This, only from a global point of view: the details in average path length and density shows that each subsystem evolves in a specific way.

There are some limitations to the present overview of the kernel. First, even if we extract all kind of supported symbols (by the terms of `libelf`) and store them with the correct type in the database, we do not (yet) make use of this kind of meta-data: is the symbol a function, a variable? When trying to cluster the kernel, this might become a good point.

Another limitation, due to the current implementation, is that we do not reproduce the "full tree" in the graph, we only assign nodes a label with the full path: this could ease a more generic and deeper analysis, especially interesting for `drivers` or `fs` subdirectories.

The current study only covers the kernel over one year, ranging from 3.0 to 3.4: it is clearly not enough to draw very generic conclusions. An attempt has been made to run the current symbols extraction over kernel ranging from 2.6.20 to 3.4; it has not been possible due to time constraints: building old kernel with recent GCC

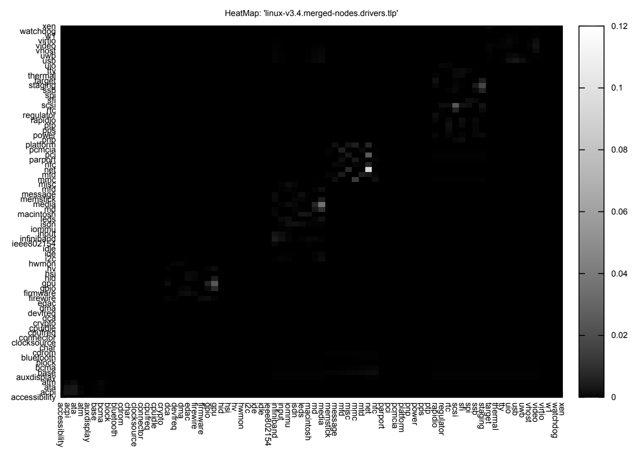


Figure 15: HeapMap of Linux v3.4, `drivers` subdirectory

seems not to be trivial, and symbols extraction for so many kernel would have required much more time than available. This is, however, an issue that must be addressed to be able to confirm the current observation over a wider sample of kernel.

A new measure that could enhance this study is clustering coefficient: we have not been able to perform this one due to time constraints. Applying the same analysis to other (big) code bases, such as Mozilla (Firefox) or LibreOffice, would be interesting.

References

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

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

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

[4] Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. A foundation for flow-based program matching using temporal logic and model checking. In *The 36th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*, pages 114–126, Savannah, GA, USA, January 2009.

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

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

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

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

[9] Alexandre Lissy, Stéphane Laurière, and Patrick Martineau. Verifications around the linux kernel. In *Ottawa Linux Symposium (OLS 2011)*, Ottawa, Canada, June 2011.

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

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

[12] Nicolas Palix, Suman Saha, Gaël Thomas, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in Linux: Ten Years Later. Research Report RR-7357, INRIA, 08 2010.

Non-scalable locks are dangerous

Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich
MIT CSAIL

Abstract

Several operating systems rely on non-scalable spin locks for serialization. For example, the Linux kernel uses ticket spin locks, even though scalable locks have better theoretical properties. Using Linux on a 48-core machine, this paper shows that non-scalable locks can cause dramatic collapse in the performance of real workloads, even for very short critical sections. The nature and sudden onset of collapse are explained with a new Markov-based performance model. Replacing the offending non-scalable spin locks with scalable spin locks avoids the collapse and requires modest changes to source code.

1 Introduction

It is well known that non-scalable locks, such as simple spin locks, have poor performance when highly contended [1, 7, 9]. It is also the case that many systems nevertheless use non-scalable locks. However, we have run into multiple situations in which system throughput collapses suddenly due to non-scalable locks: for example, a system that performs well with 25 cores completely collapses with 30. Equally surprising, the offending critical sections are often tiny. This paper argues that non-scalable locks are dangerous. For concreteness, it focuses on locks in the Linux kernel.

One piece of the argument is that non-scalable locks can seriously degrade overall performance, and that the situations in which they may do so are likely to occur in real systems. We exhibit a number of situations in which the performance of plausible activities collapses dramatically with more than a few cores' worth of concurrency; the cause is a rapid growth in locking cost as the number of contending cores grows.

Another piece of the argument is that the onset of performance collapse can be sudden as cores are added. A system may have good measured performance with N cores, but far lower total performance with just a few more cores. The paper presents a predictive model of non-scalable lock performance that explains this phenomenon.

A third element of the argument is that critical sections which appear to the eye to be very short, perhaps only a few instructions, can nevertheless trigger performance collapses. The paper's model explains this phenomenon as well.

Naturally we argue that one should use scalable locks [1, 7, 9], particularly in operating system kernels where the workloads and levels of contention are hard to control. As a demonstration, we replaced Linux's spin locks with scalable MCS locks [9] and re-ran the software that caused performance collapse. For 3 of the 4 benchmarks the changes in the kernel were simple. For the 4th case the changes were more involved because the directory cache uses a complicated locking plan and the directory cache in general is complicated. The MCS lock improves scalability dramatically, because it avoids the performance collapse, as expected. We experimented with other scalable locks, including hierarchical ones [8], and observe that the improvements are negligible or small—the big win is going from non-scalable locks to scalable locks.

An objection to this approach is that non-scalable behavior should be fixed by modifying software to eliminate serialization bottlenecks, and that scalable locks merely defer the need for such modification. That observation is correct. However, in practice it is not possible to eliminate all potential points of contention in the kernel all at once. Even if a kernel is very scalable at some point in time, the same kernel is likely to have scaling bottlenecks on subsequent generations of hardware. One way to view scalable locks is as a way to relax the time-criticality of applying more fundamental scaling improvements to the kernel.

The main contribution of this paper is amplifying the conclusion from previous work that non-scalable locks have risks: not only do they have poor performance, but they can cause collapse of overall system performance. More specifically, this paper makes three contributions. First, we demonstrate that the poor performance of non-scalable locks can cause performance collapse for real workloads, even if the spin lock is protecting a very short critical section in the kernel. Second, we propose a single comprehensive model for the behavior of non-scalable spin locks that fully captures all regimes of operation, unlike previous models [6]. Third, we confirm on modern $x86$-based multicore processors that MCS locks can improve maximum scalability without decreasing performance, and conclude that the scaling and performance benefits of the different types of scalable locks is small.

The rest of the paper is organized as follows. Section 2 demonstrates that non-scalable locks can cause performance collapse for real workloads. Section 3 introduces a Markov-based model that explains why non-scalable locks can cause this collapse to happen, even for short critical sections. Section 4 evaluates several scalable locks on modern $x86$-based multicore processors to decide which scalable lock to use to replace the offending non-scalable locks. Section 5 reports on the results of using MCS locks to replace the non-scalable locks in Linux that caused performance collapse. Section 6 relates our findings and modeling to previous work. Section 7 summarizes our conclusions.

2 Are non-scalable locks a problem?

This section demonstrates that non-scalable spin locks cause performance collapse for some kernel-intensive workloads. We present performance results from four benchmarks demonstrating that critical sections that consume less than 1% of the CPU cycles on one core can cause performance to collapse on a 48-core $x86$ machine.

2.1 How non-scalable locks work

For concreteness we discuss the ticket lock used in the Linux kernel, but any type of non-scalable lock will exhibit the problems shown in this section. Figure 1 presents simplified C code from Linux. The ticket lock is the default lock since kernel version 2.6.25 (released in April 2008).

An acquiring core obtains a ticket and spins until its turn is up. The lock has two fields: the number of the ticket that is holding the lock (`current_ticket`) and the number of the next unused ticket (`next_ticket`). To obtain a ticket number, a core uses an atomic increment instruction on `next_ticket`. The core then spins until its ticket number is current. To release the lock, a core increments `current_ticket`, which causes the lock to be handed to the core that is waiting for the next ticket number.

```
struct spinlock_t {
  int current_ticket;
  int next_ticket;
}

void spin_lock(spinlock_t *lock)
{
  int t =
    atomic_fetch_and_inc(&lock->next_ticket);
  while (t != lock->current_ticket)
    ;   /* spin */
}

void spin_unlock(spinlock_t *lock)
{
  lock->current_ticket++;
}
```

Figure 1: Pseudocode for ticket locks in Linux.

If many cores are waiting for a lock, they will all have the lock variables cached. An unlock will invalidate those cache entries. All of the cores will then read the cache line. In most architectures, the reads are serialized (either by a shared bus or at the cache line's home or directory node), and thus completing them all takes time proportional to the number of cores. The core that is next in line for the lock can expect to receive its copy of the cache line midway through this process. Thus the cost of each lock handoff increases in proportion to the number of waiting cores. Each inter-core operation takes on the order of a hundred cycles, so a single release can take many thousands of cycles if dozens of cores are waiting. Simple test-and-set spin locks incur a similar $O(N)$ cost per release.

2.2 Benchmarks

We exercised spin locks in the Linux kernel with four benchmarks: FOPS, MEMPOP, PFIND, and EXIM. Two are microbenchmarks and two represent application workloads. None of the benchmarks involve disk I/O (the file-system cache is pre-warmed). We ran the benchmarks on a 48-core machine (eight 6-core 2.4 GHz AMD

Opteron chips) running Linux kernel 2.6.39 (released in May 2011).

FOPS creates a single file and starts one process on each core. Each thread repeatedly `open`s and `close`s the file.

MEMPOP creates one process per core. Each process repeatedly `mmap`s 64 kB of memory with the `MAP_POPULATE` flag, then `munmap`s the memory. `MAP_POPULATE` instructs the kernel to allocate pages and populate the process page table immediately, instead of doing so on demand when the process accesses the page.

PFIND searches for a file by executing several instances of the GNU find utility. PFIND takes a directory and filename as input, evenly divides the directories in the first level of input directory into per-core inputs, and executes one instance of find per core, passing in the input directories. Before we execute the PFIND, we create a balanced directory tree so that each instance of find searches the same number of directories.

EXIM is a mail server. A single master process listens for incoming SMTP connections via TCP and forks a new process for each connection, which accepts the incoming message. We use the version of EXIM from MOSBENCH [3].

2.3 Results

Figure 2 shows the results for all benchmarks. One might expect total throughput to rise in proportion to the number of cores for a while, then level off to a flat line due to some serial section. Throughput does increase with more cores for a while, but instead of leveling off, the throughput *decreases* after some number of cores. The decreases are sudden; good performance with *N* cores is often followed by dramatically lower performance with one or two more cores.

FOPS. Figure 2(a) shows the total throughput of FOPS as a function of the number of cores concurrently running the benchmark. The performance peaks with two cores. With 48 cores, the total throughput is about 3% of the throughput on one core.

The performance collapse in Figure 2(a) is caused by a per-entry lock in the file system name/inode cache. The kernel acquires the lock when a file is closed in order to decrement a reference count and possibly perform clean-up actions. On average, the code protected by the lock executes in only 92 cycles.

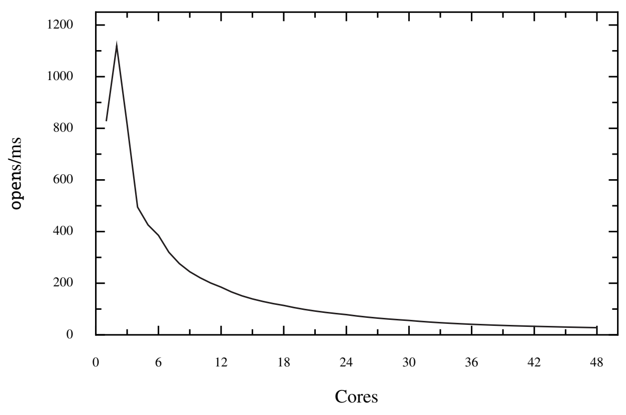
MEMPOP. Figure 2(b) shows the throughput of MEMPOP. Throughput peaks at nine cores, at which point it is 4.7× higher than with one core. Throughput decreases rapidly at first with more than nine cores, then more gradually. At 48 cores throughput is about 35% of the throughput achieved on one core. The performance collapse in Figure 2(b) is caused by a non-scalable lock that protects the data structure mapping physical pages to virtual memory regions.

PFIND. Figure 2(c) shows the throughput of PFIND, measured as the number of find processes that complete every second. The throughput peaks with 14 cores, then declines rapidly. The throughput with 48 cores is approximately equal to the throughput on one core. A non-scalable lock protecting the block buffer cache causes PFIND's performance collapse.

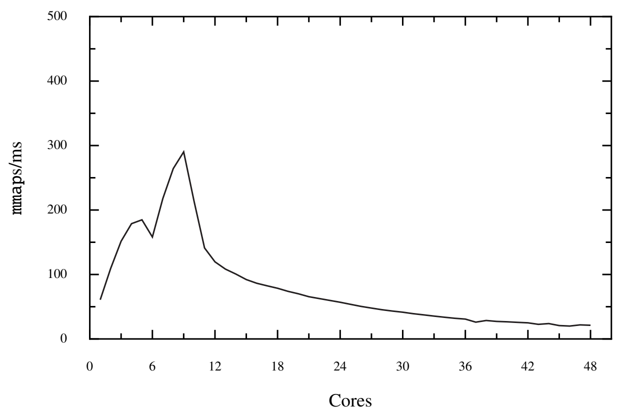
EXIM. Figure 2(d) shows EXIM's performance as a function of the number of cores. The performance collapse is caused by locks that protect the data structure mapping physical pages to virtual memory regions. The 3.0.0 kernel (released in Aug 2011) fixes this collapse by acquiring the locks involved in the bottlenecked operation together, and then running with a larger critical section.

Figure 3 shows measurements related to the most contended lock for each benchmark, taken on one core. The "Operation time" column indicates the total number of cycles required to complete one benchmark operation (opening a file, delivering a message, etc). The "Acquires per operation" column shows how many times the most contended lock was acquired per operation. The "Average critical section time" column shows how long the lock was held each time it was acquired. The "% of operation in critical section" reflects the ratio of total time per operation spent in the critical section to the total time for each operation.

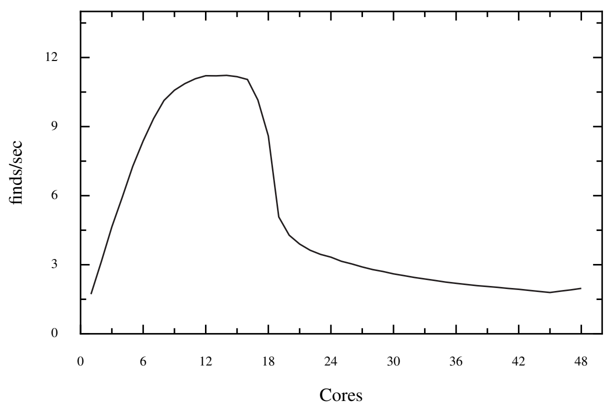
The last column of Figure 3 helps explain the point in each graph at which collapse starts. For example, MEMPOP spends 7% of its time in the bottleneck critical section. Once 14 (i.e., 1.0/0.07) cores are active, one would expect that critical section's lock to be held by some core at all times, and thus that cores would start to contend for the lock. In fact MEMPOP starts to collapse somewhat before that point, a phenomenon explained in the next section.



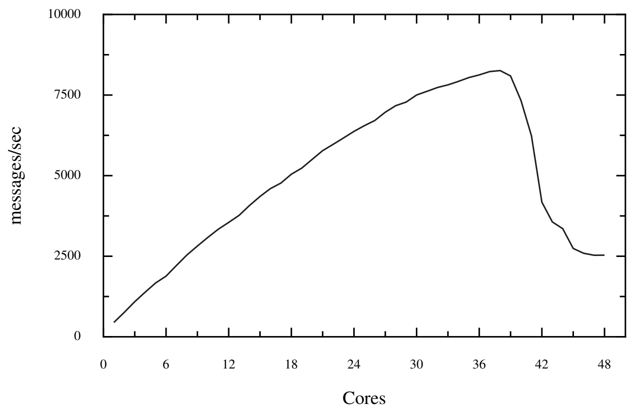
(a) Collapse for FOPS.



(b) Collapse for MEMPOP.



(c) Collapse for PFIND.



(d) Collapse for EXIM.

Figure 2: Sudden performance collapse with ticket locks.

Benchmark	Operation time (cycles)	Top lock instance name	Acquires per operation	Average critical section time (cycles)	% of operation in critical section
FOPS	503	`d_entry`	4	92	73%
MEMPOP	6852	`anon_vma`	4	121	7%
PFIND	2099 M	`address_space`	70 K	350	7%
EXIM	1156 K	`anon_vma`	58	165	0.8%

Figure 3: The most contended critical sections for each Linux microbenchmark, on a single core.

As an example of a critical section that causes non-scalability, Figure 4 shows the most contended critical sections for EXIM. They involve adding and deleting an element from a list, and consist of a handful of inlined instructions.

2.4 Questions

The four graphs have common features which raise some questions.

- Why does collapse start as early as it does? One would expect collapse to start when there is a significant chance that many cores need the same lock at the same time. Thus one might expect MEMPOP to start to see decline at around 14 cores (1.0/0.07). But the onset occurs earlier, at nine cores.
- Why does performance ultimately fall so far?
- Why does performance collapse so rapidly? One might expect a gradual decrease with added cores, since each new core should cause each release of the bottleneck lock to take a little more time. Instead, adding just a few more cores causes a sharp drop in total throughput. This is worrisome; it suggests that a system that has been tested to perform well with N cores might perform far worse with $N+2$ cores.

3 Model

This section presents a performance model for non-scalable locks that answers the questions raised in the previous section. It first describes the hardware cache coherence protocol at a high level, which is representative of a typical $x86$ system, and then builds on the basic properties of this protocol to construct a model for understanding performance of ticket spin locks. The model closely predicts the observed collapse behavior.

3.1 Hardware cache coherence

Our model assumes a directory-based cache coherence protocol. All directories are directly connected by an inter-directory network. The cache coherence protocol is a simplification of, but similar to, the implementation in AMD Opteron [4] and Intel Xeon CPUs.

```
static void
anon_vma_chain_link(
   struct anon_vma_chain *avc,
   struct anon_vma *anon_vma)
{
  spin_lock(&anon_vma->lock);
  list_add_tail(&avc->same_anon_vma,
                &anon_vma->head);
  spin_unlock(&anon_vma->lock);
}

static void
anon_vma_unlink(
   struct anon_vma_chain *avc,
   struct anon_vma *anon_vma)
{
  spin_lock(&anon_vma->lock);
  list_del(&avc->same_anon_vma);
  spin_unlock(&anon_vma->lock);
}
```

Figure 4: The most contended critical sections from EXIM. This compiler inlines the code for the list manipulations, each of which are less than 10 instructions.

3.1.1 The directory

Each core has a cache directory. The hardware (e.g., the BIOS) assigns evenly sized regions of DRAM to each directory. Each directory maintains an entry for each cache line in its local DRAM:

```
[ tag | state | core ID ]
```

The possible states are:

1. invalid (I) – the cache line is not cached;
2. shared (S) – the cache line is held in one or more caches and matches DRAM;
3. modified (M) – the cache line is held in one cache and does not match DRAM.

For modified cache lines the directory records the cache that holds the dirty cache line.

Figure 5 presents the directory state transitions for loads and stores. For example, when a core issues a load request for a cache line in the invalid state, the directory sets the cache line state to shared.

	I	S	M
Load	S	S	S
Store	M	M	M

Figure 5: Directory transitions for loads and stores.

	I	S	M
Load	–	–	DP
Store	–	BI	DI

Figure 6: Probe messages for loads and stores.

3.1.2 Network messages

When a core begins accessing an uncached cache line, it will send a load or store request to the cache line's home directory. Depending on the type of request and the state of the cache line in the home directory, the home directory may need to send probe messages to all directories that hold the cache line.

Figure 6 shows the probe messages a directory sends based on request type and state of the cache line. "BI" stands for broadcast invalidate. "DP" stands for direct probe. "DI" stands for direct invalidate. For example, when a source cache issues a load request for a modified cache line, the home directory sends a directed probe to the cache holding the modified cache line. That cache responds to the source cache with the contents of the modified cache line.

3.2 Performance model for ticket locks

To understand the collapse observed in ticket-based spin locks, we construct a model. One of the challenging aspects of constructing an accurate model of spin lock behavior is that there are two regimes of operation: when not contended, the spin lock can be acquired quickly, but when many cores try to acquire the lock at the same time, the time taken to transfer lock ownership increases significantly. Moreover, the exact point at which the behavior of the lock changes is dependent on the lock usage pattern, and the length of the critical section, among other parameters. Recent work [6] attempts to model this behavior by combining two models—one for contention and one for uncontended locks—into a single model, by simply taking the max of the two models' predictions. However, this fails to precisely model the point of collapse, and does not explain the phenomenon causing the collapse.

$$\begin{tikzcd}
0 \arrow[r, bend left, "a_0"] & \cdots \arrow[l, bend left, "s_0"] \arrow[r, bend left, "a_{k-1}"] & k \arrow[l, bend left, "s_{k-1}"] \arrow[r, bend left, "a_k"] & k+1 \arrow[l, bend left, "s_k"] \arrow[r, bend left, "a_{k+1}"] & \cdots \arrow[l, bend left, "s_{k+1}"] \arrow[r, bend left, "a_{n-1}"] & n \arrow[l, bend left, "s_{n-1}"]
\end{tikzcd}$$

Figure 7: Markov model for a ticket spin lock for n cores. State i represents i cores holding or waiting for the lock. a_i is the arrival rate of new cores when there are already i cores contending for the lock. s_i is the service rate when $i+1$ cores are contending.

To build a precise model of ticket lock behavior, we build on queueing theory to model the ticket lock as a Markov chain. Different states in the chain represent different numbers of cores queued up waiting for a lock, as shown in Figure 7. There are $n+1$ states in our model, representing the fact that our system has a fixed number of cores (n).

Arrival and service rates between different states represent lock acquisition and lock release. These rates are different for each pair of states, modeling the non-scalable performance of the ticket lock, as well as the fact that our system is closed (only a finite number of cores exist). In particular, the arrival rate from k to $k+1$ waiters, a_k, should be proportional to the number of remaining cores that are not already waiting for the lock (i.e., $n-k$). Conversely, the service rate from $k+1$ to k, s_k, should be inversely proportional to k, reflecting the fact that transferring ownership of a ticket lock to the next core takes linear time in the number of waiters.

To compute the arrival rate, we define a to be the average time between consecutive lock acquisitions on a single core. The rate at which a single core will try to acquire the lock, in the absence of contention, is $1/a$. Thus, if k cores are already waiting for the lock, the arrival rate of new contenders is $a_k = (n-k)/a$, since we need not consider any cores that are already waiting for the lock.

To compute the service rate, we define two more parameters: s, the time spent in the serial section, and c, the time taken by the home directory to respond to a cache line request. In the cache coherence protocol, the home directory of a cache line responds to each cache line request in turn. Thus, if there are k requests from different cores to fetch the lock's cache line, the time until the winner (pre-determined by ticket numbers) receives the cache line will be on average $c \cdot k/2$. As a result, processing the serial section and transferring the lock to the next holder when k cores are contending takes $s + ck/2$, and the service rate is $s_k = \frac{1}{s+ck/2}$.

Unfortunately, while this Markov model accurately represents the behavior of a ticket lock, it does not match any of the standard queueing theory that provides a simple formula for the behavior of the queueing model. In particular, the system is closed (unlike most open-system queueing models), and the service times vary with the size of the queue.

To compute a formula, we derive it from first principles. Let $P_0, \ldots, P_n$ be the steady-state probabilities of the lock being in states 0 through n respectively. Steady state means that the transition rates balance: $P_k \cdot a_k = P_{k+1} \cdot s_k$. From this, we derive that $P_k = P_0 \cdot \frac{n!}{a^k(n-k)!} \cdot \prod_{i=1}^{k}(s+ic)$. Since $\sum_{i=0}^{n} P_i = 1$, we get $P_0 = 1/\left(\sum_{i=0}^{n}\left(\frac{n!}{a^i(n-i)!}\prod_{j=1}^{i}(s+jc)\right)\right)$, and thus:

$$P_k = \frac{\frac{1}{a^k(n-k)!} \cdot \prod_{i=1}^{k}(s+ic)}{\sum_{i=0}^{n}\left(\frac{1}{a^i(n-i)!}\prod_{j=1}^{i}(s+jc)\right)} \qquad (1)$$

Given the steady-state probability for each number of cores contending for the lock, we can compute the average number of waiting (idle) cores as the expected value of that distribution, $w = \sum_{i=0}^{n} i \cdot P_i$. The speedup achieved in the presence of this lock and serial section can be computed as $n - w$, since on average that many cores are doing useful work, while w cores are spinning.

3.3 Validating the model

To validate our model, Figures 8 and 9 show the predicted and actual speedup of a microbenchmark with a single lock, which spends a fixed number of cycles inside of a serial section protected by the lock, and a fixed number of cycles outside of that serial section. Figure 8 shows the predicted and actual speedup when the serial section always takes 400 cycles to execute, but the non-serial section varies from 12.5k to 200k cycles. As we can see, the model closely matches the real hardware speedup for all configurations.

In Figure 9, we also present the predicted and actual speedup of the microbenchmark when the serial section is always 2% of the overall execution time (on one core),

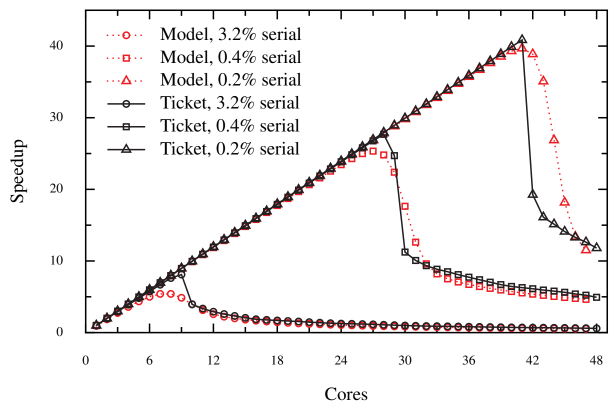


Figure 8: Predicted and actual performance of ticket spin locks with a 400-cycle serial section, for a microbenchmark where the serial section accounts for a range of fractions of the overall execution time on one core.

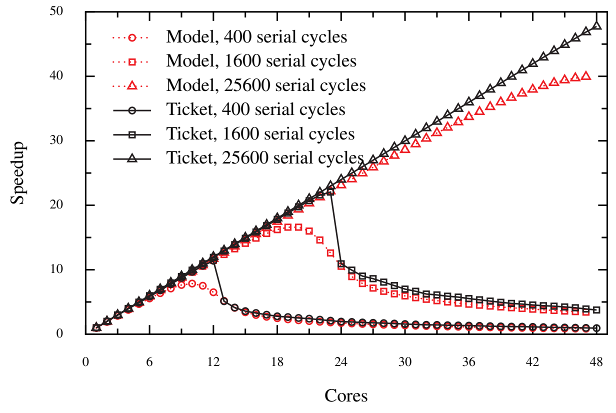


Figure 9: Predicted and actual performance for a microbenchmark where the critical section accounts for 2% of the execution time on one core, but with varying execution time for each invocation of the serial section.

but the time spent in the serial section varies from 400 to 25,600 cycles. Again, the model closely matches the measured speedup. This gives us confidence that our model accurately captures the relevant factors leading to the performance collapse of ticket locks.

One difference between the predicted and measured speedup is that the predicted collapse is slightly more gradual than the collapse observed on real hardware. This is because the ticket lock's performance is unstable near the collapse point, and the model predicts the average steady-state behavior. Our measured speedup reports the throughput for a relatively short-running microbenchmark, which has not had the time to "catch" the instability.

3.4 Implications of model results

The behavior predicted by our model has several important implications. First, the rapid collapse of ticket locks is an inherent property of their design, rather than a performance problem with our experimental hardware. Any cache-coherent system that matches our basic hardware model will experience similarly sharp performance degradation. The reason behind the rapid collapse can be understood by considering the transition rates in the Markov model from Figure 7. If a lock ever accumulates a large number of waiters (e.g., reaches state n in the Markov model), it will take a long time for the lock to go back down to a small number of waiters, because the service rate s_k rapidly decays as k grows, for short serial sections. Thus, once the lock enters a contended state, it becomes much more likely that more waiters arrive than that the current waiters will make progress in shrinking the lock's wait queue.

A more direct way to understand the collapse is that the time taken to transfer the lock from one core to another increases linearly with the number of contending cores. However, this time effectively increases the length of the serial section. Thus, as more cores are contending for the lock, the serial section grows, increasing the probability that yet another core will start contending for this lock.

The second implication is that the collapse of the ticket lock only occurs for short serial sections, as can be seen from Figure 9. This can be understood by considering how the service rate s_i decays for different lengths of the serial section. For a short serial section time s, $s_k = \frac{1}{s+ck/2}$ is strongly influenced by k, but for large s, s_k is largely unaffected by k. Another way to understand this result is that, with fewer acquire and release operations, the ticket lock's performance contributes less to the overall application throughput.

The third implication is that the collapse of the ticket lock prevents the application from reaching the maximum performance predicted by Amdahl's law (for short serial sections). In particular, Figure 9 shows that a microbenchmark with a 2% serial section, which may be able to scale to 50 cores under Amdahl's law, is able to attain less than 10× scalability when the serial section is 400 cycles long.

4 Which scalable lock?

The literature has many proposals for scalable locks, which avoid the collapse that ticket locks exhibit. Which one should we use to replace contended ticket locks? This section evaluates several scalable locks on modern *x*86-based multicore processors.

4.1 Scalable locks

A common way of making the ticket lock more scalable is to adjust its implementation to use proportional back-off when the lock is contended. The challenge with this approach is what constant to choose to multiply the ticket number with. From our model we can conclude that choosing the constant well is important only for short critical section, because for large critical sections collapse does not occur. For our experiments below, we choose the best value by trying a range of values, and selecting the one that gives the best performance. This choice provides the best result that the proportional lock could achieve.

Another approach is to replace the ticket lock with a truly scalable lock. A scalable lock is one that generates a constant number of cache misses per acquisition and therefore avoids the collapse that non-scalable locks exhibit. All of these locks maintain a queue of waiters and each waiter spins on its own queue entry. The differences in these locks are how the queue is maintained and the changed necessary to the `lock` and `unlock` API.

MCS lock. The MCS lock [9] maintains an explicit queue of `qnode` structures. A core acquiring the lock adds itself with an atomic instruction to the end of the list of waiters by having the lock point to its `qnode`, and then sets the next pointer of the `qnode` of its predecessor to point to its `qnode`. If the core is not at the head of the queue, then it spins on its `qnode`. To avoid dynamically allocating memory on each `lock` invocation, the `qnode` is an argument to `lock` and `unlock`.

K42 lock. A potential downside of the MCS lock is that it involves an API change. The K42 lock [2] is a variant of the MCS lock that requires fewer API changes.

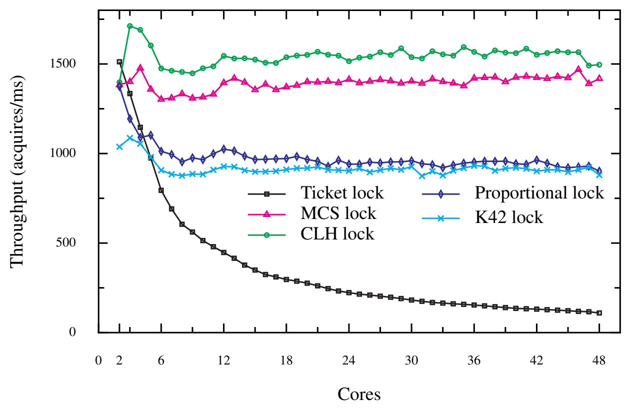


Figure 10: Throughput for cores acquiring and releasing a shared lock. Results start with two cores.

CLH lock. The CLH lock [5] is a variant of an MCS lock where the waiter spins on its predecessor `qnode`, which allows the queue of waiters to be implicit (i.e., the `qnode` next pointer and its updates are unnecessary).

HCLH lock. The HCLH lock [8] is a hierarchical variant of the CLH lock, intended for NUMA machines. The way we use it is to favor lock acquisitions from cores that share an L3 cache with the core that currently holds the lock, with the goal to reduce the cost of cache line transfers between remote cores.

4.2 Results

Figure 10 shows the performance of the ticket lock, proportional lock, MCS lock, K42 lock, and CLH lock on our 48-core AMD machine. The benchmark uses one shared lock. Each core loops, acquires the shared lock, updates 4 shared cache lines, and releases the lock. The time to update the 4 shared cache lines is similar between runs using different locks, and increases gradually from about 800 cycles on 2 cores to 1000 cycles in 48 cores. On our *x*86 multicore machine, the HCLH lock improves performance of the CLH lock by only 2%, and is not shown.

All scalable locks scale better than ticket lock on this benchmark because they avoid collapse. Using the CLH lock results in slightly higher throughput over the MCS lock, but not by much. The K42 lock achieves lower

Lock type	**Single acquire**	**Single release**	**Shared acquire**
MCS lock	25.6	27.4	53
CLH lock	28.8	3.9	517
Ticket lock	21.1	2.4	30
Proportional lock	22.0	2.8	30.2
K42 lock	47.0	23.8	74.9

Figure 11: Performance of acquiring and releasing an MCS lock, a CLH lock, and a ticket lock. Single acquire and release are measurements for one core. Shared acquire is the time for a core to acquire a lock recently released by another core. Numbers are in cycles.

throughput than the MCS lock because it incurs an additional cache miss on acquire. These results indicate that for our *x*86 multicore machine, it does not matter much which scalable lock to choose.

We also ran the benchmarks on a multicore machine with Intel CPUs and measured performance trends similar to those shown in Figure 10.

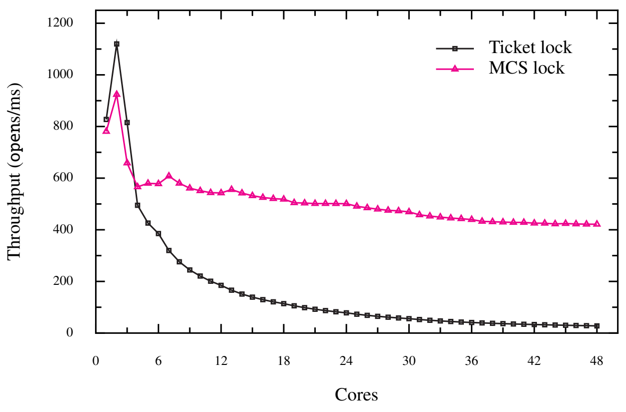
Another concern about different locks is the cost of `lock` and `unlock`. Figure 11 shows the cost for each lock in the uncontended and contended case. All locks are relatively inexpensive to acquire on a single core with no sharing. MCS lock and K42 lock are more expensive to release on a single core, because, unlike the other locks, they use atomic instructions to release the lock. Acquiring a shared but uncontended lock is under 100 cycles for all locks, except the CLH lock. Acquiring the CLH lock is expensive due to the overhead introduced by the `qnode` recycling scheme for multiple cores.

5 Using MCS locks in Linux

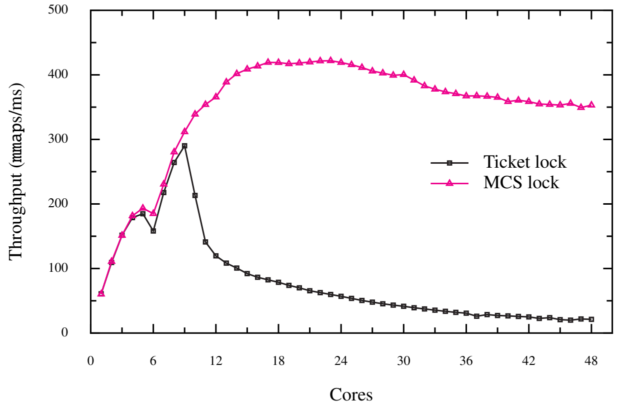
Based on the result of the previous section, we replaced the offending ticket locks with MCS locks. We first describe the kernel changes to use MCS locks, and then measure the resulting scalability for the 4 benchmarks from Section 2.

5.1 Using MCS Locks

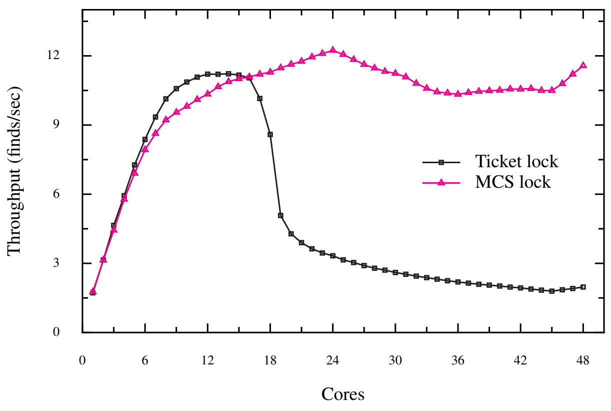
We replaced the three ticket spin locks that limited benchmark performance with MCS locks. We modified about 1,000 lines of the Linux kernel (700 lines for `d_entry`, 150 lines for `anon_vma`, and 150 lines for `address_space`).



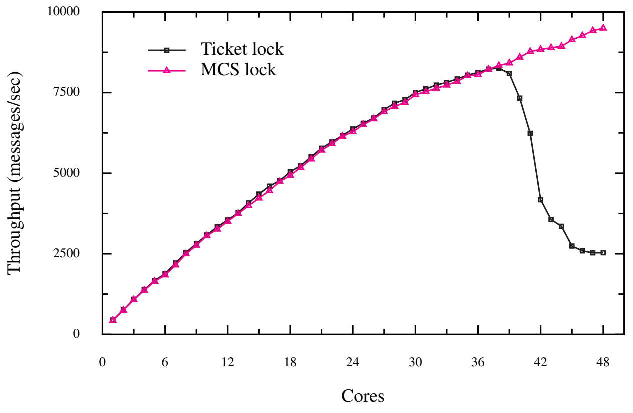
(a) Performance for FOPS.



(b) Performance for MEMPOP.



(c) Performance for PFIND.



(d) Performance for EXIM.

Figure 12: Performance of benchmarks using ticket locks and MCS locks.

As noted earlier, MCS locks have a different API than the Linux ticket spin lock implementation. When acquiring an MCS lock, a core must pass a `qnode` variable into `mcs_lock`, and when releasing that lock the core must pass the same `qnode` variable to `mcs_unlock`. For each lock a core holds, the core must use a unique `qnode`, but it is acceptable to use the same `qnode` for locks held at different times.

Many of our kernel modifications are straightforward. We allocate an MCS `qnode` on the stack, replace `spin_lock` and `spin_unlock` with `mcs_lock` and `mcs_unlock`, and pass the `qnode` to the MCS acquire and release functions.

In some cases, the Linux kernel acquires a lock in one function and releases it in another. For this situation, we stack-allocate a `qnode` on the call frame that is an ancestor of both the call frame that calls `mcs_lock` and the one that calls `mcs_release`. This pattern is common in the directory cache, and partially explains why we made so many modifications for the `d_entry` lock.

Another pattern, which we encountered only in the directory cache code that implements moving directory entries, is changing the value of lock variables. When the kernel moves a `d_entry` between two directories, it acquires the lock of the `d_entry->d_parent` (which is also a `d_entry`) and the target directory `d_entry`, and then sets the value `d_entry->d_parent` to be the target `d_entry`. With MCS, we must make sure to unlock `d_entry->d_parent` with the `qnode` originally used to lock the target `d_entry`, instead the `qnode` original used to lock `d_entry->d_parent`.

5.2 Results

The graphs in Figure 12 show the benchmark results from replacing contended ticket locks with MCS locks. For a large number of cores, using MCS improves performance by at least 3.5× and in one case by more than 16×.

Figure 12(a) shows the performance of FOPS with MCS locks. Going from one to two cores, performance with both ticket locks and MCS locks increases. For more than two cores, performance with the ticket spin lock decreases continuously. Performance with MCS locks initially also decreases from two to four cores, then remains relatively stable. The reason for this decrease in performance is that the time spent executing the critical section increases from 450 cycles on two cores to 852 cycles on four cores. The critical section is executed multiple times per-operation and modifies shared data, which incurs costly cache misses. As more cores are added, it is less likely that a core will have been the last core to execute the critical section, and therefore it will incur more cache misses, which will increase the length of the critical section.

Figure 12(b) shows the performance of MEMPOP with MCS locks. Performance with MCS and ticket locks increases up to 9 cores, at which point the performance with ticket locks collapses and continuously decreases as more cores are used. The length of the critical section is relatively short. It increases from 141 cycles on one core to about 350 cycles on 10 cores. MCS avoids the dramatic collapse from the short critical section and increases maximum performance by 16.6×.

Figure 12(c) shows the performance of PFIND with MCS. The `address_space` ticket spin lock performs well up to 15 cores, then causes a performance drop that continues until 48 cores. The serial section updates some shared data which increases the time spent in the critical section from 350 cycles on one core to about 1100 cycles on more than 44 cores. MCS provides a small increase in maximum performance, but not as much as with MEMPOP since the critical section is much longer.

Figure 12(d) shows the performance of EXIM with MCS. Ticket spin locks perform well up to 39 cores, then cause a performance collapse. The critical section is relatively short (165 cycles on one core), so MCS improves maximum performance by 3.7× on 48 cores.

The results show that using scalable locks is not that much work (e.g., less work than using RCU in the kernel) and avoids the performance collapse that results from non-scalable spin locks. The latter benefit is important because institutions often use the same kernels for several years, even as they upgrade to hardware with more cores and the likelihood of performance cliffs increases.

6 Related Work

The literature on scalable and non-scalable locks is vast, many practitioners are well familiar with the issues, and it is well known that non-scalable locks behave poorly under contention. The main contribution that this paper adds is the observation that non-scalable locks can cause system performance to collapse, as well as a model that nails down why the performance collapse is so dramatic, even for short critical sections.

Anderson [1] observes that the behavior of spin locks can be "degenerative". The MCS paper shows that acquisition of a test-and-set lock increases linearly with processors on the BBN Butterfly and that on the Symmetry this cost manifests itself even with a small number of processors [9]. Many researchers and practitioners have written microbenchmarks that show that non-scalable spin locks exhibit poor performance under contention on modern hardware. This paper shows that non-scalable locks can cause the collapse of system performance under plausible workloads; that is, the locking costs for short critical sections can be very large on the scale of kernel execution time.

The Linux scaling study reports on the performance collapse that ticket locks introduce on the EXIM benchmark, but doesn't explain the collapse [3]. This paper shows the collapse and the suddenness with several workloads, and provides a model that explains the acuteness.

Eyerman and Eeckhout [6] provide closed formulas to reason about the speedup of parallel applications that involve critical sections, pointing out that there is regime in which applications achieve better speedup than Amdahl's law predicts. Unfortunately, their model makes a distinction between the contended and uncontended regime and proposes formula for each regime. In addition, the formulas do not model the insides of locks; instead, they assume scalable locks. This paper contributes a comprehensive model that accurately predicts performance across the whole spectrum from uncontended to contended, and applies it to modeling the inside of locks.

7 Conclusion

This paper raises another warning about non-scalable locks. Although it is well understood that non-scalable spin locks have poor performance, it is less well appreciated that their poor performance can have dramatic effect on overall system performance. This paper shows that non-scalable locks can cause system performance to collapse under real workloads and that the collapse is sudden, and presents a Markov model that explains the sudden collapse. We conclude that using non-scalable locks is dangerous because a system that has been tested with *N* cores may experience performance collapse at a few more cores—or, put differently, if one upgrades a machine in hopes of achieving higher performance, one might run the risk of ending up with a performance collapse. Scalable locking is a stop-gap solution that avoids collapse, but achieving higher performance with additional cores can require new designs using lock-free data structures.

Acknowledgments

We thank Abdul Kabbani for helping us analyze our Markov model. This work was supported by Quanta Computer, by Google, and by NSF awards 0834415 and 0915164.

References

[1] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1:6–16, January 1990.

[2] M. A. Auslander, D. J. Edelsohn, O. Y. Krieger, B. S. Rosenburg, and R. W. Wisniewski. Enhancement to the MCS lock for increased functionality and improved programmability. U.S. patent application 10/128,745, 2003.

[3] S. Boyd-Wickizer, A. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of Linux scalability to many cores. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI 2010)*, Vancouver, Canada, October 2010.

[4] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache hierarchy and memory subsystem of the AMD Opteron processor. *Micro, IEEE*, 30(2):16–29, March–April 2010.

[5] T. S. Craig. Building FIFO and priority-queuing spin locks from atomic swap. Technical Report UW-CSE-93-02-02, University of Washington, Department of Computer Science and Engineering, 1993.

[6] S. Eyerman and L. Eeckhout. Modeling critical sections in Amdahl's law and its implications for multicore design. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*, pages 262–370, Saint-Malo, France, June 2010.

[7] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufman, 2008.

[8] V. Luchangco, D. Nussbaum, and N. Shavit. A hierarchical CLH queue lock. In *Proceedings of the European Conference on Parallel Computing*, pages 801–810, Dresden, Germany, August–September 2006.

[9] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

Fine Grained Linux I/O Subsystem Enhancements to Harness Solid State Storage

Suri Brahmaroutu
Suri_Brahmaroutu@Dell.com

Rajesh Patel
Rajesh_Patel@Dell.com

Harikrishnan Rajagopalan
Harikrishnan_Rajagopalan@Dell.com

Sumanth Vidyadhara
Sumanth_Vidyadhara@Dell.com

Ashokan Vellimalai
Ashokan_Vellimalai@Dell.com

Abstract

Enterprise Solid State Storage (SSS) are high performing class devices targeted at business critical applications that can benefit from fast-access storage. While it is exciting to see the improving affordability and applicability of the technology, enterprise software and Operating Systems (OS) have not undergone pertinent design modifications to reap the benefits offered by SSS. This paper investigates the I/O submission path to identify the critical system components that significantly impact SSS performance. Specifically, our analysis focuses on the Linux I/O schedulers on the submission side of the I/O. We demonstrate that the Deadline scheduler offers the best performance under random I/O intensive workloads for the SATA SSS. Further, we establish that all I/O schedulers including Deadline are not optimal for PCIe SSS, quantifying the possible performance improvements with a new design that leverages device level I/O ordering intelligence and other I/O stack enhancements.

1 Introduction

SSS devices have been in use in consumer products for a number of years. Until recently, the devices were sparingly deployed in enterprise products and data centers due to several technical and business limitations.As the performance and endurance continue to improve and prices fall, we believe large scale utilization of these devices in higher demanding environments is imminent. Flash-based SSS can replace mechanical disks in many I/O intensive and latency sensitive applications.

SSS device operations are complex and different from the mechanical disks. While SSS devices make use of parallelism to achieve orders of magnitude in improved read performance over mechanical disks, they employ complex flash management techniques to overcome several critical restrictions with respect to writes.The ability to fine-tune a storage device for optimal utilization is largely dependent on the OS design and its I/O stack. However, most contemporary I/O stacks make several fundamental assumptions that are valid for mechanical disks only. This has opened up a sea of opportunities for designing I/O stacks that can leverage improved performance of the SSS devices. For instance, the OS and file systems can comprehend that SSS contain more intelligence to perform tasks like low level block management and thus expose rich interface to convey more information such as free blocks and I/O priorities down to the device.

In this paper, we first set guidelines for measuring SSS performance based on enterprise relevant workloads. We then discuss the complete life cycle of an I/O request under Linux OS, identifying system software components in the submission path that are not quite tuned to leverage the benefits of SSS. Specifically, we profile the Linux I/O stack by measuring current performance yield under current I/O scheduler schemes and quantifying the performance that can be improved with a better design. For our study, we have used an off-the-shelf SATA SSS as well as a soon-to-be-available Dell PowerEdge Express Flash PCIe SSS to evaluate improvements to Linux I/O stack architecture during the I/O request submission operation.

1.1 Background

SSS performance and device characteristics have been evaluated along several parameters under various OSes and workloads by the industry as well as academia. There has been a specific focus on design tradeoffs [3],

Access Spec	Block Size *KB*	%Reads	%Random
DBOLTP	8	7	100
Messaging	4	67	100
OS Paging	64	90	0
SQL Server Log	64	0	0

Table 1: Configuration Parameters

High Performance Computing [1], throughput performance [2], and so on. However, there is little published work about the detailed analysis on end-to-end I/O request submission or completion operations. Specifically, the focus of the study is the I/O scheduler and request queue management from the request submission perspective. We plan on presenting our findings and recommendations on IRQ balancing and interrupt coalescing from the completion standpoint in subsequent papers in the near future.

All our experiments were conducted on x86-64 bit architecture based Linux server based on 3.2.8 kernel. A dual socket Sandy Bridge motherboard with two 8-core Intel Xeon CPU E5-2665 64-bit processors running at 2.40 GHz, with two hardware threads (with hyper threading) per core, 64GB of DDR III RAM was used. 175GB Dell PowerEdge Express Flash PCIe SSS and LSI MegaRAID SAS 9240 storage controller with a Samsung 100GB SATA SSS connected were used for I/O measurements.

We chose fio [7] as our benchmarking tool mainly due to its asynchronous I/O support. We believe SSS will be deployed in the enterprise primarily for these four usage models: OS paging, Messaging, DB-OLTP and SQL Server Log. Thus, we focused our measurements on workloads generated using these I/O configuration parameters, as shown in Table 1.

2 I/O Request Submission

I/O request traverses through different components of the Linux subsystem as shown above. Most of these layers are optimized to work better for rotational disks. Assumptions such as seek time made considering the mechanical factors of hard drives are not valid for SSS and therefore introduction of SSS poses I/O subsystem architectural challenges. We considered the following components in the I/O submission path for targeted SSS specific optimizations.

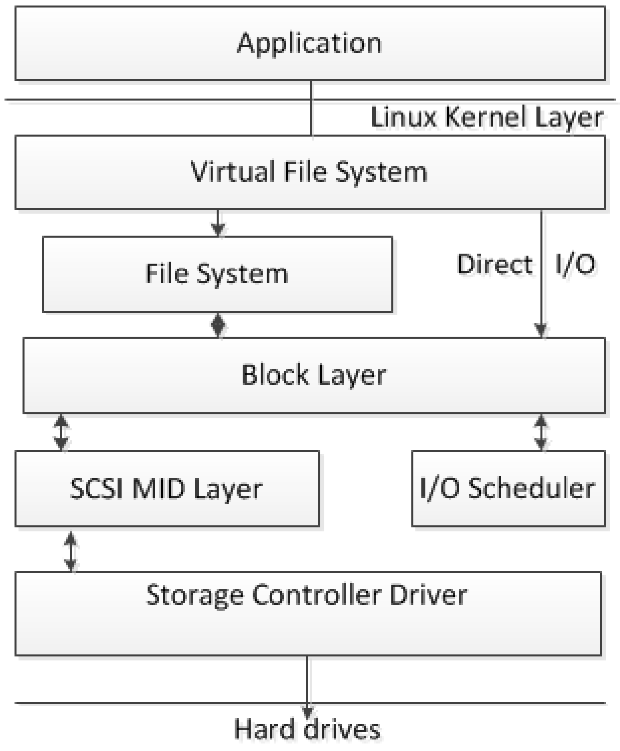


Figure 1: I/O Block Digram

File System and Buffered I/O - When data is accessed through a file system, extra processing and reads are needed to access meta-data to locate the actual data. Furthermore, buffered I/O copies data from SSS to system memory before eventually copying into the user space. This could potentially cause higher latencies. All our empirical data collected and presented in this paper are based on Direct I/O path with no paging or buffering involved.

SCSI Layer Protocol Processing and Abstraction - SCSI subsystems provide enhanced error recovery as well as retry operations for the I/O requests. However, as PCIe interconnect operates in a much more closed system environment, it is fairly safe to assume that transport level errors are rare. Thus, it is more efficient to allow the upper layer protocol (in user space) to perform error handling for PCIe SSS. As the SCSI mid layer contributes about 20% additional latency to the I/O submission path [4], our PCIe SSS design bypasses the SCSI layer in its entirety.

Request Re-ordering and Merging (I/O Scheduler) - The block layer uses one of the elevator (IO scheduler algorithms) to reorder and merge the adjacent requests to reduce the seek time of hard drives and increase per-

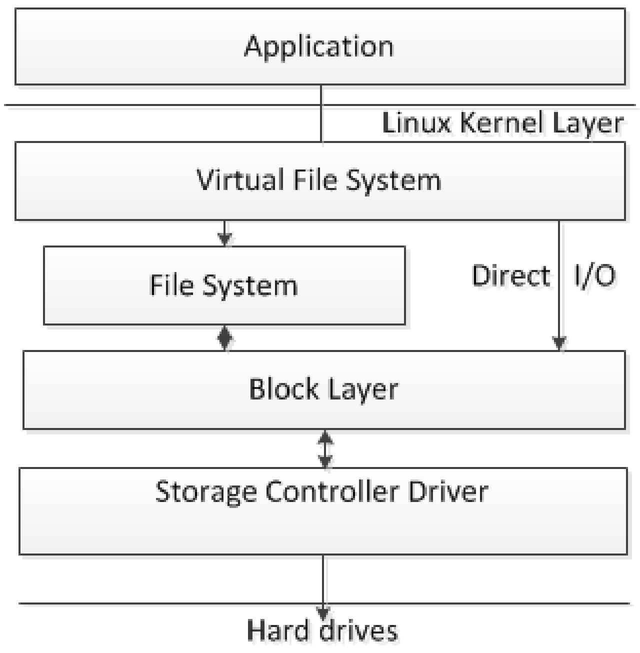


Figure 2: Optimized I/O Block Digram

formance. But in case of SSS with no seek latency involved, using elevator algorithms just adds processing overhead. Section 2.1 presents the taxonomy of various I/O schedulers and further quantifies the performance delivered when the system is configured with each of those scheduler schemes.

Shown above is the IO flow of the new driver model adopted by Dell PowerEdge Express Flash PCIe SSS which avoids generic request queue, IO schedulers and SCSI subsystem. Furthermore, the device utilizes modified AHCI driver with queue depth of 256 elements in order to maximize performance. Section 3 describes our approach further and exhibits the performance gains realized due to these pertinent optimizations.

2.1 I/O Scheduler

The purpose of introduction of these algorithms was to schedule disk I/O requests in such a way that the disk operations will be handled efficiently. Different I/O schedulers were introduced along the way. Selecting the right one matching SSS behavioral strengths and application workload would improve the I/O performance substantially.

Completely Fair Queuing - The Completely Fair Queuing (CFQ) scheduler aims to provide equal amount of I/O bandwidth among all processes issuing I/O requests. Each time a process submits a synchronous I/O request, it is moved to the assigned internal queue. Asynchronous requests from all processes are batched together according to their process's I/O priority. During each cycle, requests are moved from each non-empty internal request queue into one dispatch queue in a round-robin fashion. Our experiments using CFQ scheduler show that the scheme is suited well for large sequential accesses, however, sub-optimal for small or random I/O workloads. Once in the dispatch queue, requests are ordered to minimize disk seeks and serviced accordingly, which may not improve performance in case of SSS, as solid state devices have negligible seek time. Furthermore, we observed that CFQ scheduling caused a noticeable amount of increase in the CPU utilization.

Deadline Scheduler - In Deadline scheduler, an expiration time or "deadline" is associated with each block device request. Once a request reaches its expiration time, it is serviced immediately, regardless of its targeted block device. To maintain efficiency, any other similar requests targeted at nearby locations on disk will also be serviced. The main objective of the Deadline scheduler is to guarantee a response time for each request. The Deadline scheme is well suited in most real world workload scenarios such as messaging as shown here, where SSS are deployed for random I/O performance with deeper queue depths.

Noop Scheduler - Among all I/O scheduler types, the Noop scheduler is the simplest. It moves all requests into a simple unordered queue, where they are processed by the device in a simple FIFO order. The Noop scheduler is suitable for devices where there are no performance penalties for seeks. This characteristic makes Noop scheduler suitable for workloads that demand the least possible latency operating with small queue depths. Also, it may be noted that the SSS firmware has built-in algorithms to optimize the read/write performance, so in many cases it is best suited for the SSS internal algorithms to handle the read/write order. Selecting Noop scheduler will avoid I/O subsystem overhead in rearranging I/O requests.

We further validated our claims by experimenting with SATA SSS. At deeper queue depths, the Deadline scheduler performed better than CFQ by 15% and against the Noop scheduler by around 3%. For sequential writes, we found that CFQ outperforms Deadline as well as Noop schedulers by 30%.We believe most enterprise

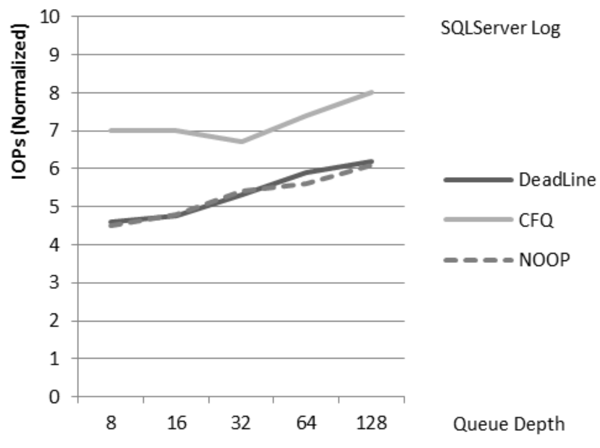


Figure 3: SQL Server

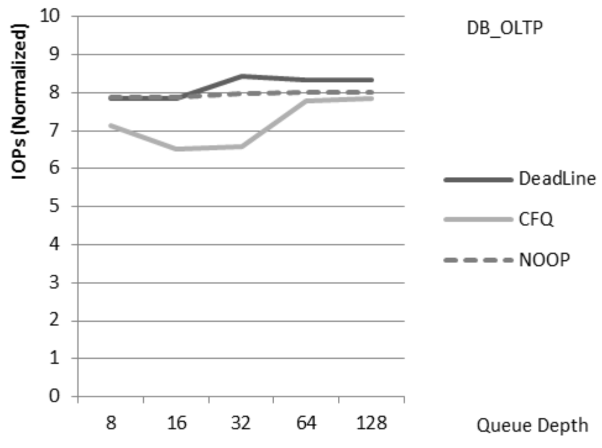


Figure 5: DB_OLTP

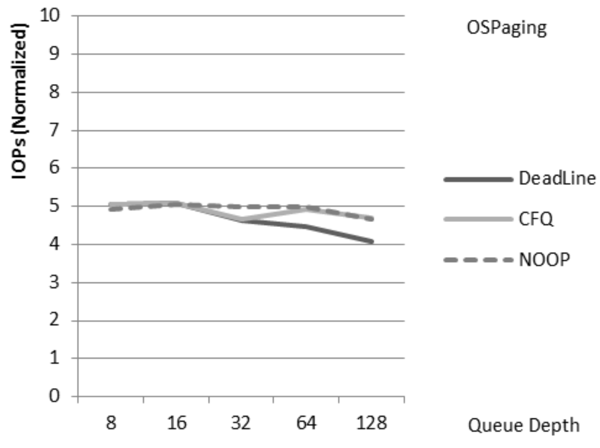


Figure 4: OS Paging

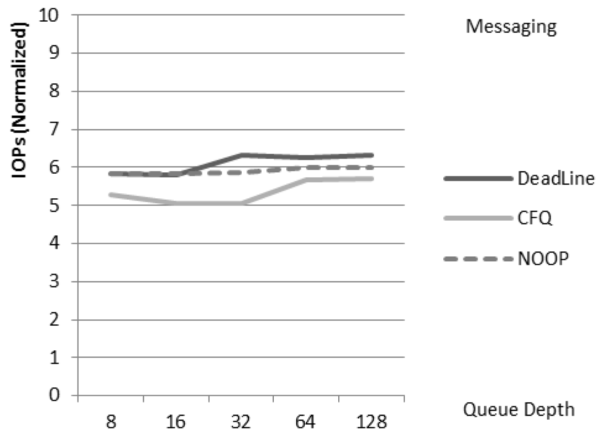


Figure 6: Messaging

end users will invest in SSS devices for random I/O performance benefit and therefore recommend using Deadline scheduler as the default configuration.

3 Optimized Dell Driver Performance

We were able to realize significant performance improvement when the SCSI layer is bypassed. However, the gains were much more substantial when the I/O scheduler logic was avoided as well. Our approach yielded at least 27% better small random read I/O performance at lower queue depths. Another important aspect of our new methodology was to circumvent the limitations imposed by the AHCI interface. The current Linux AHCI driver implementation does not allow the queue depth to increase beyond 31, although most of the enterprise class SSS devices support queue depths up to 256. As a result, we were able to attain up to 200% improvement in small random write I/O performance at deeper queue depths.

Finally, our approach is tuned to offer the best possible SSS performance under most critical and I/O intensive enterprise applications such as DB-OLTP and Messaging without sacrificing performance of other possible usage scenarios such as DB logs or OS Paging. We observed gains of up to 40% and 50% respectively for DB-OLTP and Messaging workloads.

4 Conclusion

We have discussed the current Linux storage I/O stack and its constituents. We also identified specific components that contribute to SSS performance degradation. A new approach for PCIe SSS is presented that bypasses

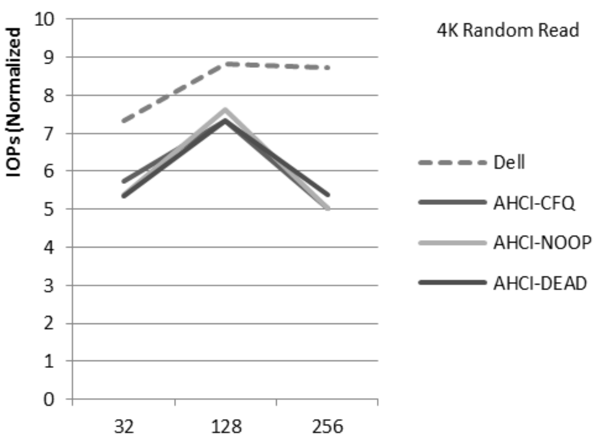


Figure 7: 4K Random Read

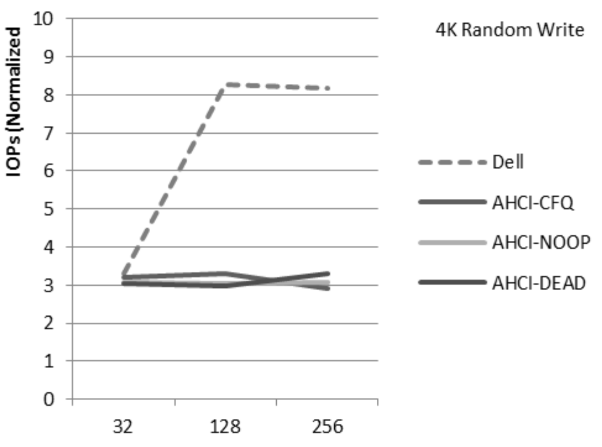


Figure 8: 4K Randon Write

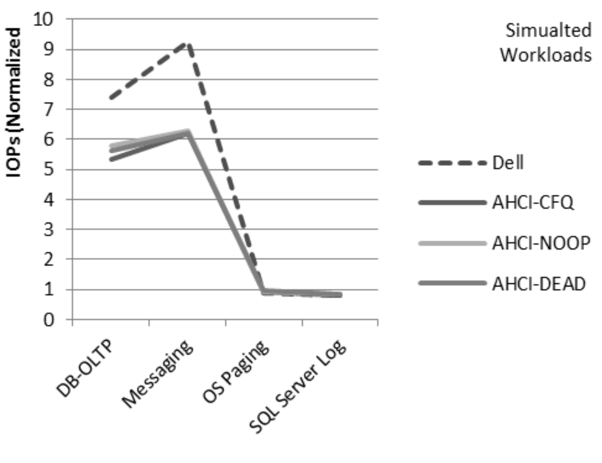


Figure 9: Simulated Workloads

conventional Linux I/O request management and pushes requests out to the device with minimal processing in the software stack. The study quantifies performance gains of up to 200% in some specific I/O patterns and up to 50% in some real world workload scenarios for PCIe SSS.

While it is possible to avoid much of the Linux stack for PCIe SSS, I/O to devices such as SATA SSS can only be serviced by traversing the conventional stack. Although the current I/O schedulers are outdated and imperfect when addressing SSS, they have a critical role to play to optimize the performance of SAS or SATA SSS. We demonstrate that a "one size fits all" scheduler policy does not work well for SATA SSS. Of the current crop of schedulers, the Deadline scheduler offers up to 15% performance advantage over Noop and therefore it is best suited for most I/O intensive enterprise applications. We recommend using Deadline as the default configuration.

As the SSS technology continues to advance, we believe the I/O scheduler methodology should evolve taking the device characteristics into account by working in concert with the intelligence present in the device.

5 Future Work

No system I/O analysis is complete without a thorough investigation of the completion path. We plan on focusing on two aspects: interrupt coalescing opportunities and IRQ balancing schemes that work best for SSS. As we believe vendors continue to support increased number of MSI-X vectors for parallelism, MSI-X configuration especially from the NUMA optimizations standpoint will need to be understood.

It is of paramount importance to understand and optimize the SSS devices from a system perspective at a macro level. For example, real enterprise applications run with file systems and RAID arrays. We believe file systems can gain from organizing meta-data in a way that take into account the special characteristics of SSS. Thus, we intend to study the impact of these and other proposed changes in future on various areas of Linux system design.

6 References / Additional Resources

[1] E. Seppanen, M.T. O£Keefe and D.J. Lilja "High Performance Solid State Storage Under Linux" in Pro-

ceedings of the 26th IEEE (MSST2010) Symposium on Massive Storage Systems and Technologies, 2010

[2] M. Dunn and A.L.N. Reddy, "A New I/O Scheduler for Solid State Devices". Texas A&M University ECE Technical Report TAMU-ECE-2009-02, 2009

[3] N. Agarwal, V. Prabhakaran, T. Wobber,J.D. Davis, M. Manasse and R. Panigrahy, "Design tradeoffs for SSS performance" in Proceedings of the USENIX 2008 Annual Technical Conference, 2008

[4] A. Foong, B. Veal and F. Hady, "Towards SSD-ready Enterprise Platforms" in Proceedings of the 36th International Conference on Very Large Data Bases, 2010

[5] D. P. Bovet and M. Cesati, "Understanding The Linux Kernel", Third Edition, Oreilly & Associates Inc, 2005

[6] A. Rubini, J. Corbet and G. Kroah-Hartman, "Linux Device Drivers", Third Edition, Oreilly & Associates Inc., 2005

[7] J. Axboe, Fio "Flexible IO Tester". `http://freshmeat.net/projects/fio`

Optimizing eCryptfs for better performance and security

Li Wang
School of Computer
National University of Defense Technology
liwang@nudt.edu.cn

Yunchuan Wen
Kylin, Inc.
wenyunchuan@kylinos.com.cn

Jinzhu Kong
School of Computer
National University of Defense Technology
kongjinzhu@163.com

Xiaodong Yi
School of Computer
National University of Defense Technology
xdong_yi@163.com

Abstract

This paper describes the improvements we have done to eCryptfs, a POSIX-compliant enterprise-class stacked cryptographic filesystem for Linux. The major improvements are as follows. First, for stacked filesystems, by default, the Linux VFS framework will maintain page caches for each level of filesystem in the stack, which means that the same part of file data will be cached multiple times. However, in some situations, multiple caching is not needed and wasteful, which motivates us to perform redundant cache elimination, to reduce ideally half of the memory consumption and to avoid unnecessary memory copies between page caches. The benefits are verified by experiments, and this approach is applicable to other stacked filesystems. Second, as a filesystem highlighting security, we equip eCryptfs with HMAC verification, which enables eCryptfs to detect unauthorized data modification and unexpected data corruption, and the experiments demonstrate that the decrease in throughput is modest. Furthermore, two minor optimizations are introduced. One is that we introduce a thread pool, working in a pipeline manner to perform encryption and write down, to fully exploit parallelism, with notable performance improvements. The other is a simple but useful and effective write optimization. In addition, we discuss the ongoing and future works on eCryptfs.

1 Introduction

eCryptfs is a POSIX-compliant enterprise cryptographic filesystem for Linux, included in the mainline Linux kernel since version 2.6.19. eCryptfs is derived from Cryptfs [5], which is part of the FiST framework [6]. eCryptfs provides transparent per file encryption. After mounting a local folder as an eCryptfs folder with identity authentication passed, the file copied into the eCryptfs folder will be automatically encrypted. eCryptfs is widely used, for example, as the basis for Ubuntu's encrypted home directory, natively within Google's ChromeOS, and transparently embedded in several network attached storage (NAS) devices.

eCryptfs is implemented as a stacked filesystem inside Linux kernel, it does not write directly into a block device. Instead, it mounts on top of a directory in a lower filesystem. Most POSIX compliant filesystem can act as a lower filesystem, for example, ext4 [2], XFS [4], even NFS [3]. eCryptfs stores cryptographic metadata in the header of each file written, so that encrypted files can be copied between hosts, and no additional information is needed to decrypt a file, except the ones in the encrypted file itself.

The rest of this paper is organized as follows. For background information, Section 1 introduces the eCryptfs cryptographic filesystem. Section 2 describes the optimizations for eCryptfs performance. Section 3 presents the data integrity enforcement for eCryptfs security. Section 4 discusses our ongoing and future works on eCryptfs. Section 5 concludes the paper.

2 Performance Improvements

2.1 Redundant Cache Elimination

The page cache is a transparent filesystem cache implemented in Linux VFS (Virtual File System) framework.

The fundamental functionality is to manage the memory pages that buffer file data, avoiding frequent slow disk accesses. For eCryptfs (and other stacked file systems in Linux), there exist (at least) two levels of page caches, as shown in Figure 1. The upper level is the eCryptfs page cache, which buffers the plain text data to interact with the user applications. The lower level page cache belongs to the file system on top of which eCryptfs is stacked, and it buffers the cipher text data as well as the eCryptfs file metadata. The eCryptfs file read/write works as follows,

- Read operations result in the call of VFS `vfs_read`, which searches the eCryptfs page cache. If no matching page is found, `vfs_read` calls the file system specific `readpage` call back routine to bring the data in. For eCryptfs this is `eCryptfs_readpage`, which calls `vfs_read` again to cause the lower file system to read the data from the disk into eCryptfs page cache. The data is then decrypted and copied back to the user application.
- Write operations result in the call of VFS `vfs_write`, which copies the data from user space buffer into the eCryptfs page cache, marks the corresponding page dirty, then returns without encryption (unless the system currently has large amount of dirty pages). Encryption is normally performed asynchronously by the dedicated OS kernel thread, during the job of flushing dirty pages into lower page cache, by invoking file system specific `writepage` call back routine, here is `ecryptfs_writepage`. This routine encrypts a whole page of data into a temporary page, then invokes `vfs_write` to copy the encrypted data from the temporary page into the lower page cache.

In real life, eCryptfs is often deployed in archive and backup situations. For the former case, people archive their private documents into an eCryptfs protected directory, consequently, the corresponding eCryptfs files are created and opened for writing, and those files are later opened generally for reading. The latter case is similar, user copies files out from eCryptfs folder, modifies, and copies the revised files back to replace the original ones. In this case, the eCryptfs files are opened for either reading or writing as well, in other words, in the above situations, the files are almost never online edited, i.e., opened for both reading and writing. This is also true for some other situations.

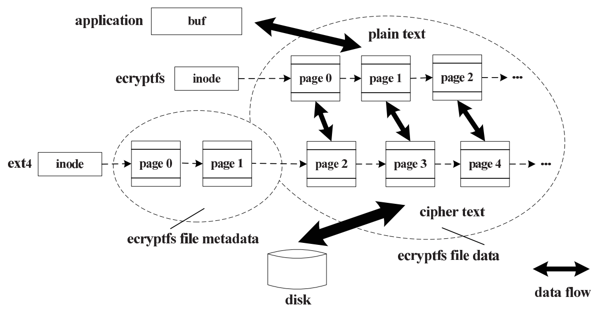


Figure 1: Page caches for eCryptfs on top of ext4 under original implementation.

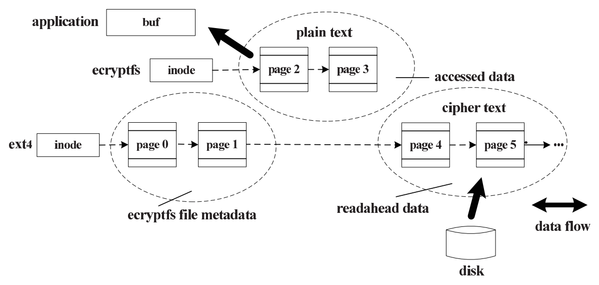


Figure 2: Page caches for eCryptfs on top of ext4 under read optimization for encrypted file.

If the eCryptfs file is opened only for reading, the lower page cache is not needed since the cipher data will not be used after decryption. This motivates us to perform redundant cache elimination optimization, thus reduce ideally half of the memory consumption. The page cache is maintained only at the eCryptfs level. For first read, once the data has been read out from disk, decrypted and copied up to eCryptfs page cache, we free the corresponding pages in the lower page cache immediately by invoking `invalidate_inode_pages2_range`, as shown in Figure 2. A better solution is to modify the VFS framework to avoid allocating the lower page entirely, but that would be much more complicated, and we want to limit our revisions in the scope of eCryptfs codes.

If the eCryptfs file is opened for writing, and the write position is monotonic increasing, which guarantees the same data area will not be repeatedly written, then the eCryptfs page cache is not needed since the plain data will not be used after encryption. It is beneficial to maintain the page cache only at the lower level. Once the data has been encrypted and copied down to lower page cache, the corresponding pages in eCryptfs page cache are freed immediately.

```
enum efsrwstate {
   ECRYPTFS_RW_INIT,
   ECRYPTFS_RW_RDOPT,
   ECRYPTFS_RW_WROPT,
   ECRYPTFS_RW_NOOPT,
};
struct ecryptfs_rw_state {
   struct mutex lock;
   efsrwstate state;
};
struct ecryptfs_inode_info {
   ...
   struct ecryptfs_rw_state rw_state;
 };
```

Figure 3: The data structures for redundant cache elimination for encrypted file.

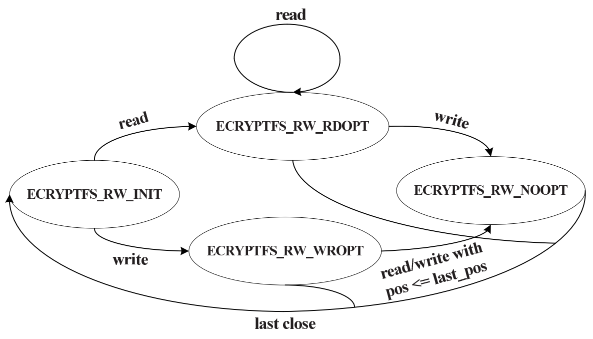


Figure 4: The state machine for redundant cache elimination for encrypted file.

To achieve this optimization, we maintain a simple state machine as an indicator. There are four states as shown in Figure 3, `ECRYPTFS_RW_INIT` indicates an initialized state, with neither readers nor writers. `ECRYPTFS_RW_RDOPT` indicates the file is currently opened only for reading, and the read optimization, i.e, lower page cache early release applies. `ECRYPTFS_RW_WROPT` indicates the file is currently opened only for writing and the write position is monotonic increasing, in which case, the write optimization, i.e, upper page cache early release applies. `ECRYPTFS_RW_NOOPT` applies for remaining cases. As shown in Figure 4, when a `file` data structure is initialized, the state is set to `ECRYPTFS_RW_INIT`. If it is then opened for reading, the state transitions to `ECRYPTFS_RW_RDOPT`, and remains unchanged until a

```
1  static int ecryptfs_readpage(struct file * file,
                                 struct page *page)
2  {
3     ...
4     flags = inode_info->crypt_stat.flags.
5     /* skip non-encrypted file */
6     if (!(flags & ECRYPTFS_ENCRYPTED))
7       goto out;
8     mutex_lock(&inode_info->rw_state.lock);
9     state = inode_info->rw_state.state;
10    if (state == ECRYPTFS_RW_RDOPT) {
11      pgoff_t index;
12      /* calculate the lower page index */
13      index = (ecryptfs_lower_header_size(
          crypt_stat) >> PAGE_CACHE_SHIFT)
                    + page->index;
14      /* release the lower page */
15      invalidate_inode_pages2_range(
          lower_inode->i_mapping, index, index);
16    }
17    mutex_unlock(&inode_info->rw_state.lock);
18    out:
19    ...
20 }
```

Figure 5: The `readpage` routine for eCryptfs for encrypted file.

writer gets involved. In that case, the state changes from `ECRYPTFS_RW_RDOPT` to `ECRYPTFS_RW_NOOPT`. If the file is initially opened for writing, the state becomes `ECRYPTFS_RW_WROPT`, and remains unchanged until the write position decreases or any reader gets involved, in which case, the state becomes `ECRYPTFS_RW_NOOPT`. After the file is closed by the last user, the state returns to `ECRYPTFS_RW_INIT`.

Figure 5 shows the eCryptfs code implementing `readpage`. After the data have been copied into eCryptfs page cache, and redundant cache elimination is applicable, in line 13, the corresponding lower page index is calculated, then in line 15, the lower page is released.

We evaluate redundant cache elimination on a machine with a Pentium Dual-Core E5500 2.8GHz CPU, 4G of memory (DDR3), the kernel version 3.1.0-7 i686 with PAE enabled, the same test system is used for all the experiments of this paper described. We use the command '`iozone -t` *x* `-s` *y* `-r 4M -i 1`' to measure 'Re-read' throughput. Here *x* ranges from 1 to 256, increasing by orders of 2, representing the number of processes, and *y* ranges from 2G to 8M, representing the file size, such

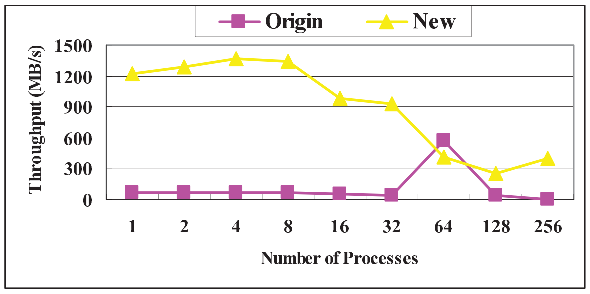


Figure 6: Re-read throughput for redundant cache elimination over original implementation.

that the total operated data set $x*y$ is always equal to 2G.

As shown in Figure 6, the optimized eCryptfs achieves a big speedup under this group of tests. This is because the data set is 2G, if two level of page caches are present, it will consume up to 4G memory, thus lead to a poor re-read throughput due to memory page thrashing. With the redundant cache elimination, the memory is enough to buffer the data set, therefore brings a good throughput.

There is furthermore a special case when the eCryptfs folder is mounted with `ecryptfs_passthrough` flag set, which allows for non-eCryptfs files to be read and written from within an eCryptfs mount. For those non-eCryptfs files, eCryptfs just relays the data between lower file system and user applications without data transformations. In this case, the data will be double cached in each level of page cache, and the eCryptfs page cache is not needed at all. For this case, we directly bypass the eCryptfs page cache by copying data between lower file system and user buffer directly.

For example, Figure 7 shows the codes implementing `read`. In line 14, `vfs_read` is invoked with the lower file and user buffer as input, this will read data from lower file system directly to user buffer. In line 19, the eCryptfs inode attributes are updated according to the ones of the lower inode.

Linux as well as most other modern operation systems provide another method to access file, namely the memory mapping by means of the `mmap` system call. This maps an area of the calling process' virtual memory to files, i.e, reading those areas of memory causes the file to be read. While reading the mapped memory areas for the first time, an exception is triggered and the architecture dependent page fault handler will call file system

```
1   static ssize_t ecryptfs_read(struct file *file,
                char *buf,  size_t count, loff_t *ppos)
2   {
3      int err;
4      struct file *lower_file;
5      loff_t pos_copy = *ppos;

6      /* skip encrypted file */
7      if (ecryptfs_file_to_private(file)->crypt_stat->
              flags & ECRYPTFS_ENCRYPTED)
8        goto out;
9      lower_file = ecryptfs_file_to_lower(file);
10      /*
11       * directly read data into the user buffer from
12       * the lower file, bypass the ecryptfs page cache
13       */
14     err =  vfs_read(lower_file, buf, count, &pos_copy);

15     /* update the ecryptfs inode attributes */
16     lower_file->f_pos = pos_copy;
17     *ppos = pos_copy;
18     if (err >= 0) {
19       fsstack_copy_attr_atime(file->f_dentry->d_inode,
                         lower_file->f_dentry->d_inode);
20     }

21     return err;
22   }
```

Figure 7: The `read` routine for eCryptfs for non-ecrypted file.

```
1   const struct file_operations ecryptfs_main_fops = {
2     ...
3     .mmap = ecryptfs_file_mmap,
4   };

5   static int ecryptfs_file_mmap(struct file *file,
                         struct vm_area_struct *vma)
6   {
7       ...
8       vma->vm_file = lower_file;
9       rc = lower_file->f_op->mmap(lower_file, vma);
10      get_file(lower_file);
11      new_ops = &inode_info->ecryptfs_vm_ops;
12      mutex_lock(&inode_info->lower_file_mutex);
13      if (!inode_info->lower_vm_ops) {
14         inode_info->lower_vm_ops = vma->vm_ops;
15         new_ops.fault = ecryptfs_vma_fault;
16         ... // the assigns to other interfaces omitted
17       }
18      mutex_unlock(&inode_info->lower_file_mutex);
19      vma->vm_ops = new_ops;
20      vma->vm_private_data = file;

21      return rc;
22    }
```

Figure 8: The `mmap` routine for eCryptfs for non-encrypted file.

```
static int ecryptfs_vma_fault(struct vm_area_struct *
                              vma, struct vm_fault *vmf)
{
    file = vma->vm_private_data;
    inode = file->f_dentry->d_inode;
    inode_info = ecryptfs_inode_to_private(inode);

    return inode_info->lower_vm_ops->fault(vma, vmf);
}
```

Figure 9: The `fault` routine for eCryptfs for non-encrypted file.

specific `readpage` routine to read the data from disk into page cache.

With regard to `mmap`, to bypass the eCryptfs page cache, the approach is shown in Figure 8. In line 8, the owner of the `vma` address space is replaced with the lower file. In line 9, the lower filesystem `mmap` implementation is called, this will assign the lower filesystem implemented operation set to the field `vm_ops` of `vma`. In line 14, this set is saved and then `vm_ops` is replaced with the eCryptfs implemented operation set in line 19, which, in most cases, simply call the saved lower operation set, and update some attributes, if necessary. For example, the implementation of `fault` interface is shown in Figure 9. By this approach, the eCryptfs page cache will not be generated.

2.2 Kernel Thread Pool based Encryption

Linux-2.6.32 introduced the feature of per-backing-device based writeback. This feature uses a dedicated kernel thread to flush the dirty memory of each storage device. eCryptfs makes use of this feature by registering an 'eCryptfs' backing device while mounting a eCryptfs folder. Therefore, a dedicated kernel thread will be generated to flush the pages dirtied by eCryptfs from the ecryptfs page cache to the lower page cache. Basically, the kernel thread achieves this goal in two steps, first encrypting the data, then writing encrypted data to lower page cache. Since encryption is CPU intensive task, and the current write back framework supports only one thread per device, it could not exploit the power of multiple CPU cores to perform parallel encryption, and will slow down the speed of submitting pages to lower page cache.

A better way is to use a kernel thread pool to do this job in a pipeline manner. Two groups of kernel threads

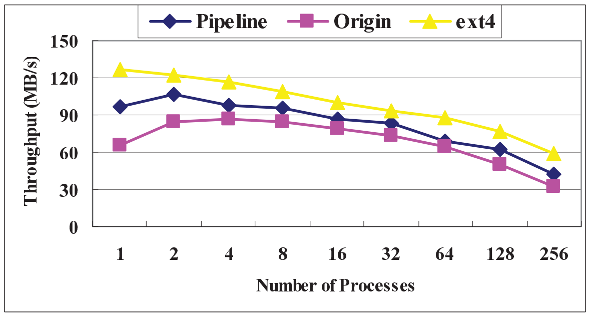


Figure 10: Write throughput for thread pool based encryption over original implementation and ext4.

are generated, one is responsible for encryption, called `e-thread`. the other is for writing data down, called `w-thread`. The number of e-thread is twice of the number of CPU cores. w-threads are spawned on-demand. `ecryptfs_writepage` submits the current dirty page to pipeline, wakes up the e-threads, then returns. The e-threads grab dirty pages from the pipeline, perform encryption, submit the encrypted pages to the next station of the pipeline, and dynamically adjust the number of w-threads according to the number of write pending pages, if necessary. w-threads write the encrypted pages to the lower page cache.

This approach is evaluated by measuring 'Write' throughput using `iozone`. For comparison, the throughput on ext4 is also tested to give an upper bound. The parameters are '`iozone -t` *x* `-s` *y* `-r 4M -i 0 -+n`', where *x* is from 1 to 256, corresponding, *y* from 8G to 32M. As shown in Figure 10, the optimized codes achieve an obviously higher throughput than the original implementation.

2.3 Write Optimization

The Linux VFS framework performs buffered writes at a page granularity, that is, it copies data from user space buffers into kernel page cache page by page. During the process, VFS will repeatedly invoke the file system specific `write_begin` call back routine, typically once per page, to expect the file system to get the appropriate page in page cache ready to be written into. For eCryptfs, the routine is `ecryptfs_write_begin`, which looks for the page cache, and grabs a page (allocates it if desired) for writing. If the page does not contain valid data, or the data are older than the counterpart on the disk, eCryptfs will read out the corresponding data from the disk into the eCryptfs page cache, decrypt

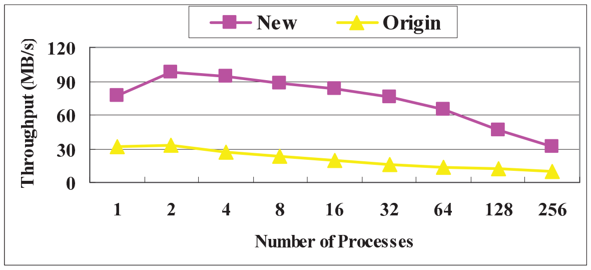


Figure 11: Write throughput for write optimization over original implementation.

them, then perform writing. However, for current page, if the length of the data to be written into is equal to page size, that means the whole page of data will be overwritten, in which case, it does not matter whatever the data were before, it is beneficial to perform writing directly.

This optimization is useful while using eCryptfs in backup situation, user copies file out from eCryptfs folder, modifies, and copies the revised file back to replace the original one. In such situation, the file in eCryptfs folder is overwritten directly, without reading.

Although the idea is simple, there is one more issue to consider related to the Linux VFS implementation. As described above, VFS calls `write_begin` call back routine to expect the file system to prepare a locked updated page for writing, then VFS calls `iov_iter_copy_from_user_atomic` to copy the data from user space buffers into the page, at last, VFS calls `write_end` call back routine, where, typically, the file system marks the page dirty and unlocks the page. `iov_iter_copy_from_user_atomic` may end up with a partial copy, since some of the user space buffers are not present in memory (maybe swapped out to disk). In this case, only part of data in the page are overwritten. Our idea is to let `ecryptfs_write_end` return zero at this case, to give `iov_iter_fault_in_readable` a chance to handle the page fault for the current iovec, then restart the copy operation.

This optimization is evaluated by measuring 'Write' throughput using `iozone`. The command parameters are '`iozone -t` *x* `-s` *y* `-r 4M -i 0 -+n`', where *x* is from 1 to 256, correspondingly, *y* is from 8G to 32M, and the files written into have valid data prior to the experiments. As shown in Figure 11, the optimized codes enjoy around 3x speedup over the original implementation under this group of tests.

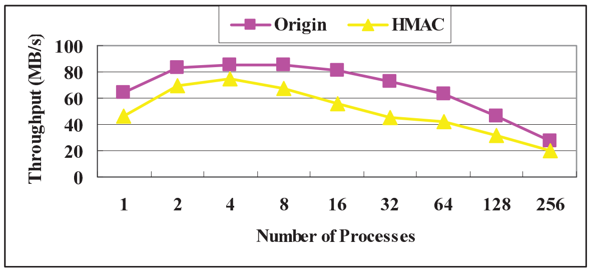


Figure 12: Write throughput for eCryptfs with HMAC-MD5 verification over the one without.

3 Data Integrity Enforcement

In cryptography, HMAC (Hash-based Message Authentication Code) [1] is used to calculate a message authentication code (MAC) on a message involving a cryptographic hash function in combination with a secret key. HMAC can be used to simultaneously verify both the data integrity and the authenticity of a message. Any cryptographic hash function, such as MD5 or SHA-1, may be used in the calculation of an HMAC; the resulting MAC algorithm is termed HMAC-MD5 or HMAC-SHA1 accordingly. The cryptographic strength of the HMAC depends upon the cryptographic strength of the underlying hash function, the size of its hash output length in bits, and on the size and quality of the cryptographic key.

We implemented HMAC verification for eCryptfs, enabling it to detect the following situations,

- Unauthorized modification of file data
- Data corruption introduced by power loss, disk error etc.

Each data extent is associated with a HMAC value. Before an extent is encrypted and written to lower file page cache, its HMAC is calculated, with the file encryption key and the data of the current extent as input, and the HMAC is saved in the lower file as well. When the data is later read out and decrypted, its HMAC is recalculated, and compared with the value saved. If they do not match, it indicates that the extent has been modified or corrupted, eCryptfs will return an error to user application. Since the attacker does not know the file encryption key, the HMAC value cannot be faked.

The HMAC values are grouped into dedicated extents, rather than appended at end of each extent due to a performance issue. According to our test, `vfs_read` and `vfs_write` with non extent-aligned length are much slower than the aligned counterpart. Accordingly, the data extents are split into groups.

The decrease in throughput due to HMAC verification is evaluated by measuring 'Write' throughput using `iozone`, the hash algorithm is MD5. The command parameters are '`iozone -t` *x* `-s` *y* `-r 4M -i 0 -+n`', where *x* is from 1 to 256, corresponding, *y* from 8G to 32M. As shown in Figure 12, the decrease is modest.

4 Future Work

We are implementing per-file policy support for eCryptfs. That is, allow to specify policies at a file granularity. For example, a file should be encrypted or not, what encryption algorithm should be used, what is the length of the key, etc. In addition, we are considering to implement transparent compression support for eCryptfs.

Related to VFS, we are taking into account to modify VFS to give filesystems more flexibilities, to maintain page cache at their decisions.

5 Conclusion

This paper presents the optimizations to eCryptfs for both performance and security. By default, Linux VFS framework maintains multiple page caches for stacked filesystems, which, in some situations, is needless and wasteful, motivating us to develop redundant cache elimination, the benefits of which have been verified experimentally. Furthermore, this approach is applicable to many other stacked filesystems. We enhance the eCryptfs security by introducing HMAC verification. To exploit parallelism, we introduce a thread pool, working in a pipeline manner to do the encryption and write down jobs. We have also presented a simple write optimization, which is very useful while using eCryptfs in backup situation.

References

[1] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pages 1–15, London, UK, 1996. Springer-Verlag.

[2] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Linux Symposium*, 2007.

[3] R. Sandberg, D. Golgberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the sun network filesystem. In *USENIX Conference*, 1985.

[4] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the xfs file system. In *USENIX Conference*, 1996.

[5] E. Zadok, I. Bădulescu, and A. Shender. Cryptfs: A stackable vnode level encryption file system. Technical Report CUCS-021-98, Computer Science Department, Columbia University, 1998.

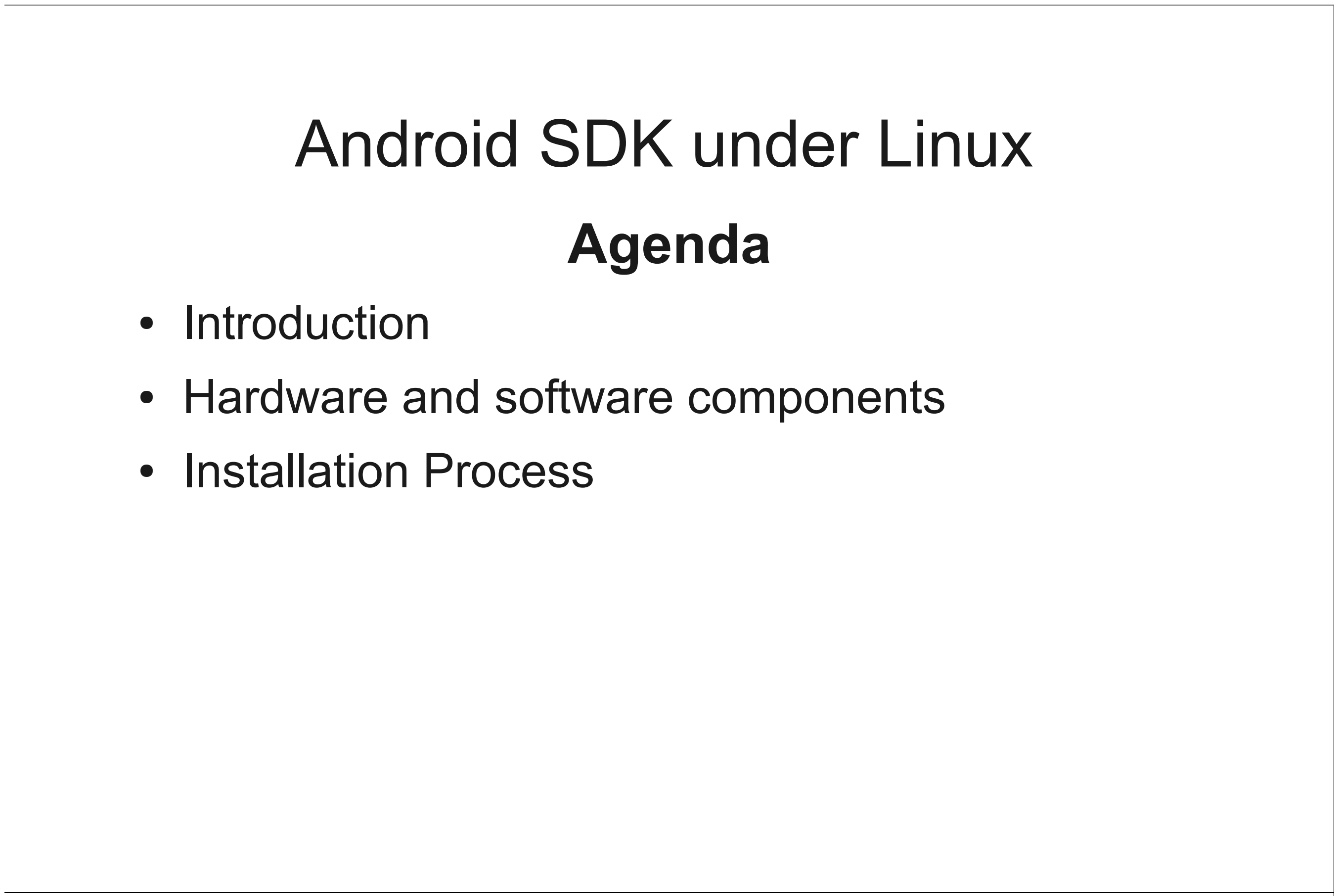
[6] E. Zadok and J. Nieh. Fist: A language for stackable file systems. In *Proc. of the Annual USENIX Technical Conference*, pages 55–70, San Diego, CA, 2000. USENIX Association.

Android SDK under Linux

Jean-Francois Messier
Android Outaouais
jf@messier.ca

Abstract

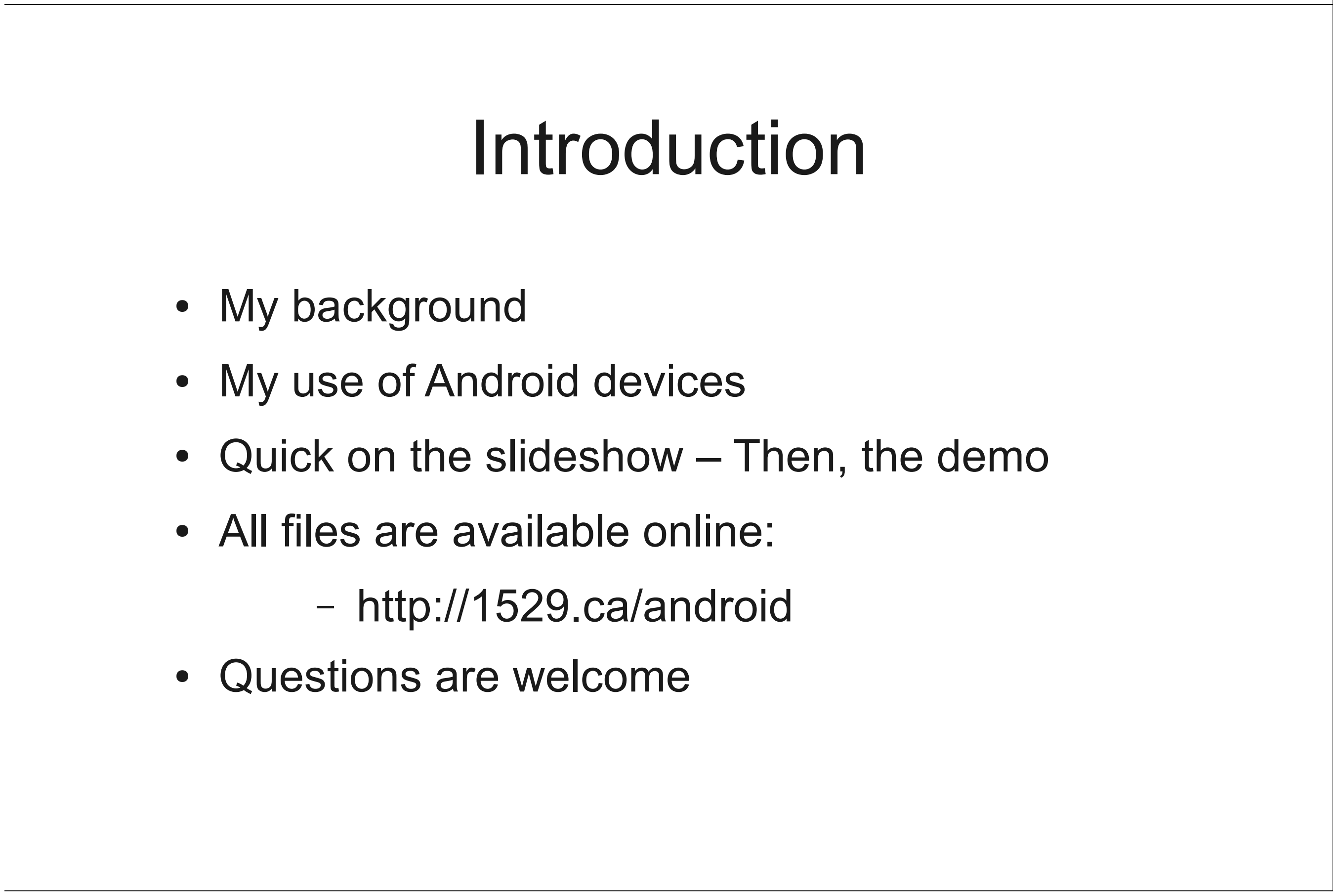
This is a tutorial about installing the various components required to have an actual Android development station under Linux. The commands are simple ones and are written to be as independent as possible of your flavour of Linux. All commands and other scripts are in a set of files that will be available on-line. Some processes that would usually require user attendance have been scripted to run unattended and are pre-downloaded. The entire set of files (a couple of gigs) can be copied after the tutorial for those with a portable USB key or hard disk.



Android SDK under Linux

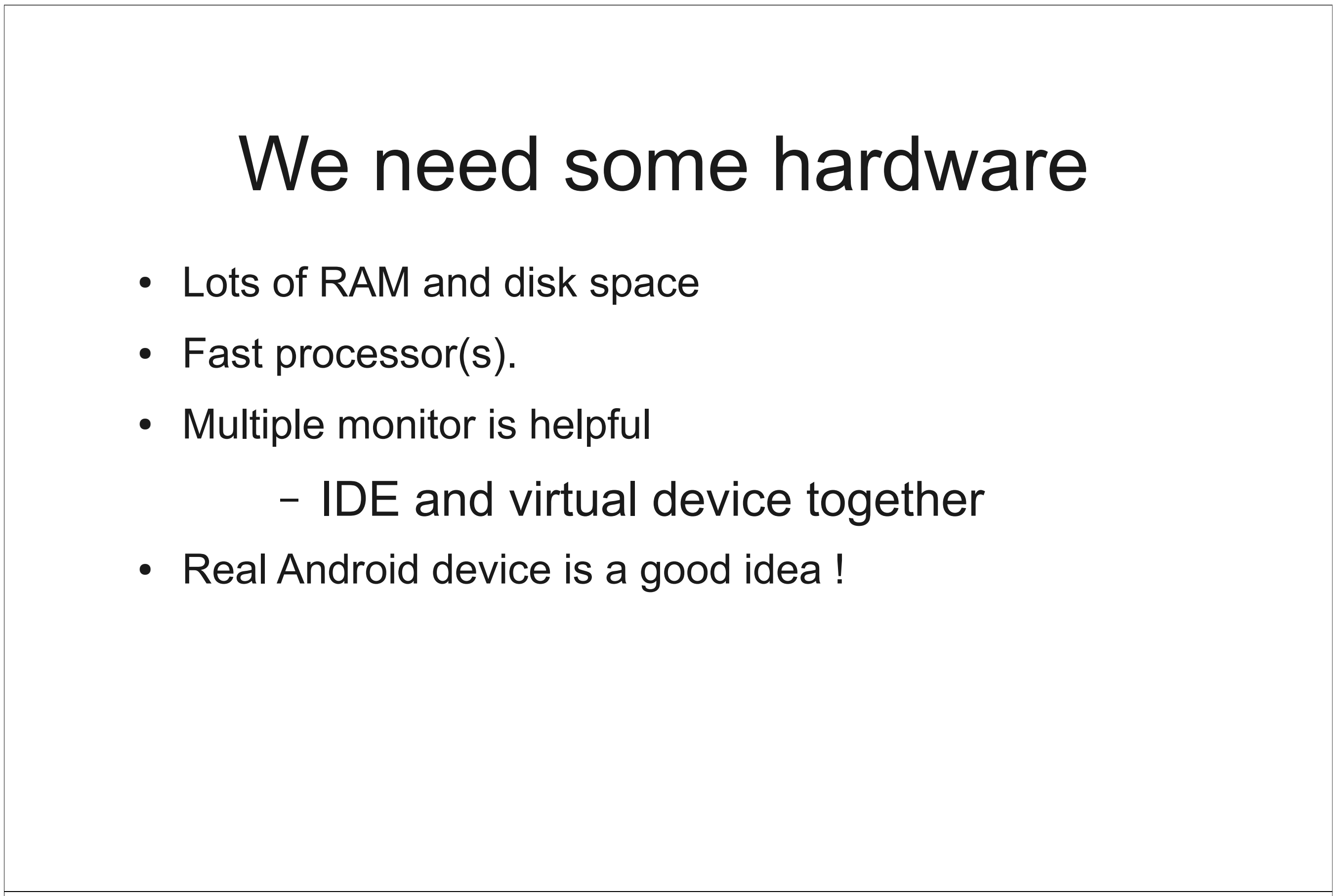
Agenda

- Introduction
- Hardware and software components
- Installation Process



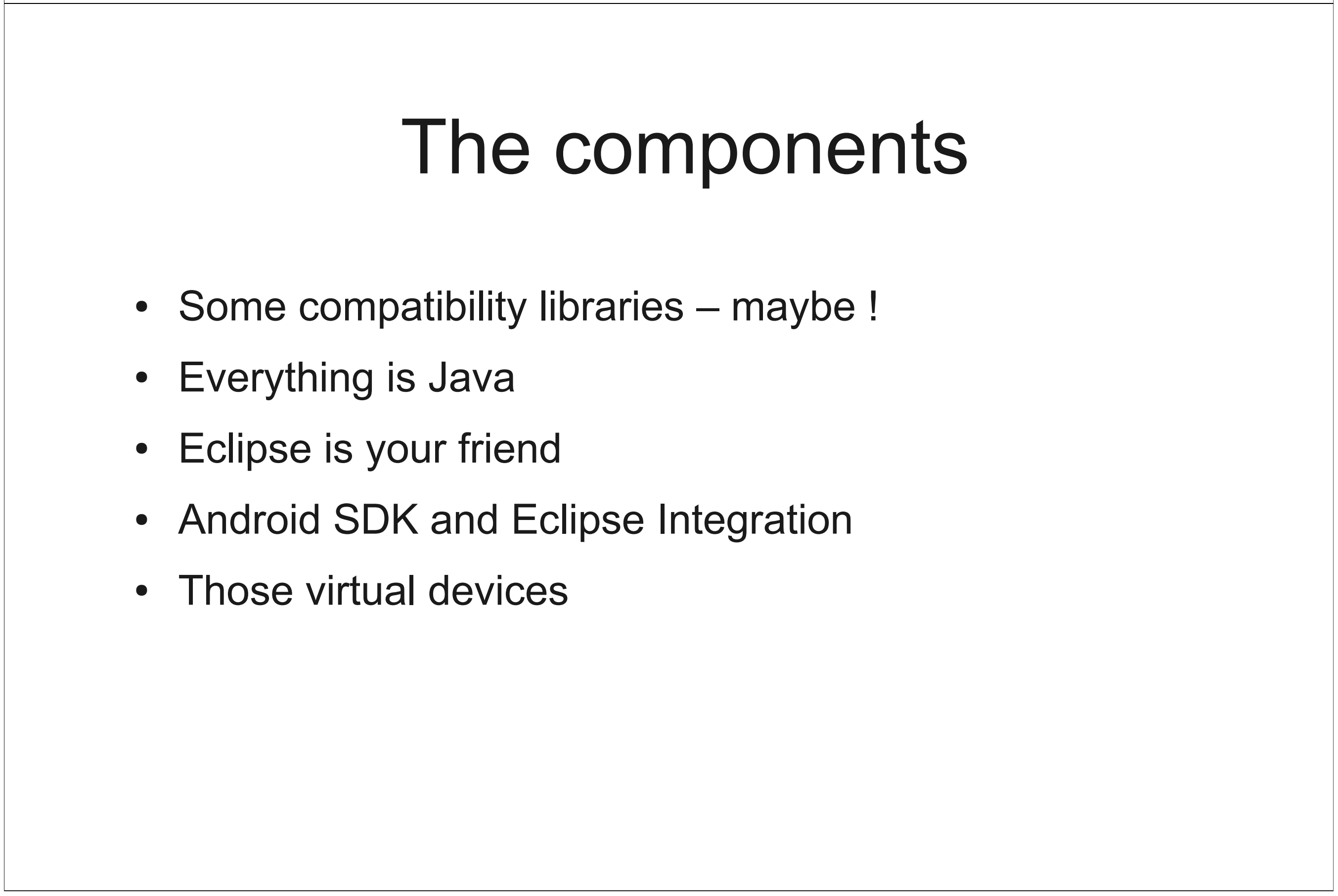
Introduction

- My background
- My use of Android devices
- Quick on the slideshow – Then, the demo
- All files are available online:
  - http://1529.ca/android
- Questions are welcome



We need some hardware

- Lots of RAM and disk space
- Fast processor(s).
- Multiple monitor is helpful
  - IDE and virtual device together
- Real Android device is a good idea !



The components

- Some compatibility libraries – maybe !
- Everything is Java
- Eclipse is your friend
- Android SDK and Eclipse Integration
- Those virtual devices

32-bit compatibility libraries

```
sudo apt-get install ia32-libs
```

or (maybe)

```
yum install glibc.i686
```

Everything is Java

- The tools for Android development are using Java
- The applications use another JVM for Android.
- Should be Oracle Java Development Kit (JDK)
- Get it from

http://www.oracle.com/technetwork/java/javase/downloads

Java (continued)

- My quick script (3 + 2 lines):

```
tar -zxf jdk-7u2-linux-x64.tar.gz
sudo mv ./jdk1.7.0_02 /opt
sudo ln -s /opt/jdk1.7.0_02/bin/java /usr/bin/java

which java
java -version
```

Eclipse is your friend

- In my installations, Eclipse is the main tool to edit and manage the source files. It has the integration for the Android libraries and tools.
- I use *Eclipse Classic*
- Get it here:

`http://www.eclipse.org/downloads/?osType=linux`

My script for Eclipse

- Download the version you need, then:

```
tar -zxf eclipse-SDK-XXX-linux-gtk.tar.gz
sudo mv ./eclipse /opt
sudo ln -s /opt/eclipse/eclipse /usr/bin/eclipse
which eclipse
```

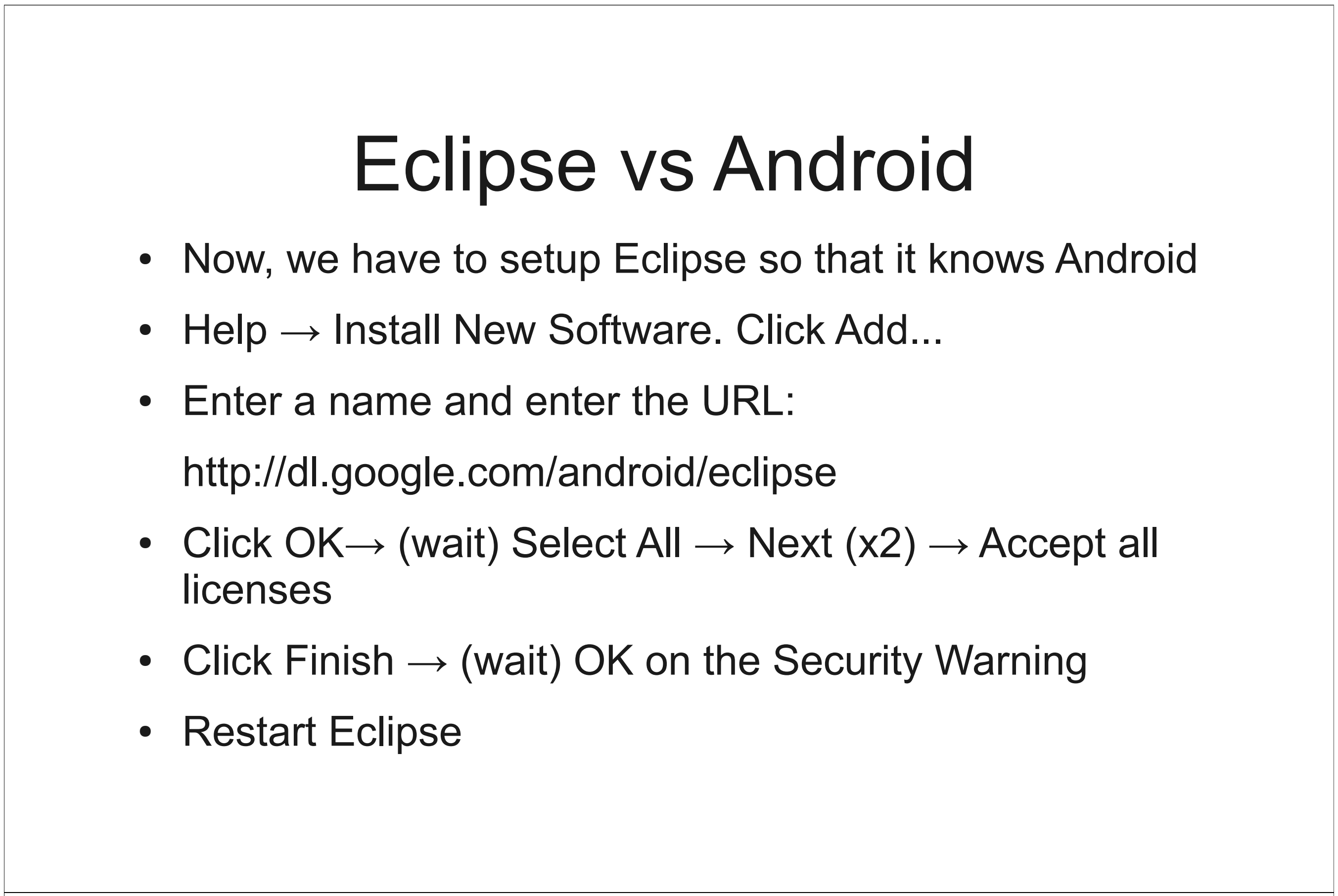
- Try to start from the command line or otherwise
- Create the shortcut as you see fit

```
(ubuntu) sudo apt-get -y install gnome-panel

(ubuntu) gnome-desktop-item-edit ~/Desktop --create-new
```

Setup the Android SDK

- Get the file, unzip and move it:
- The script:

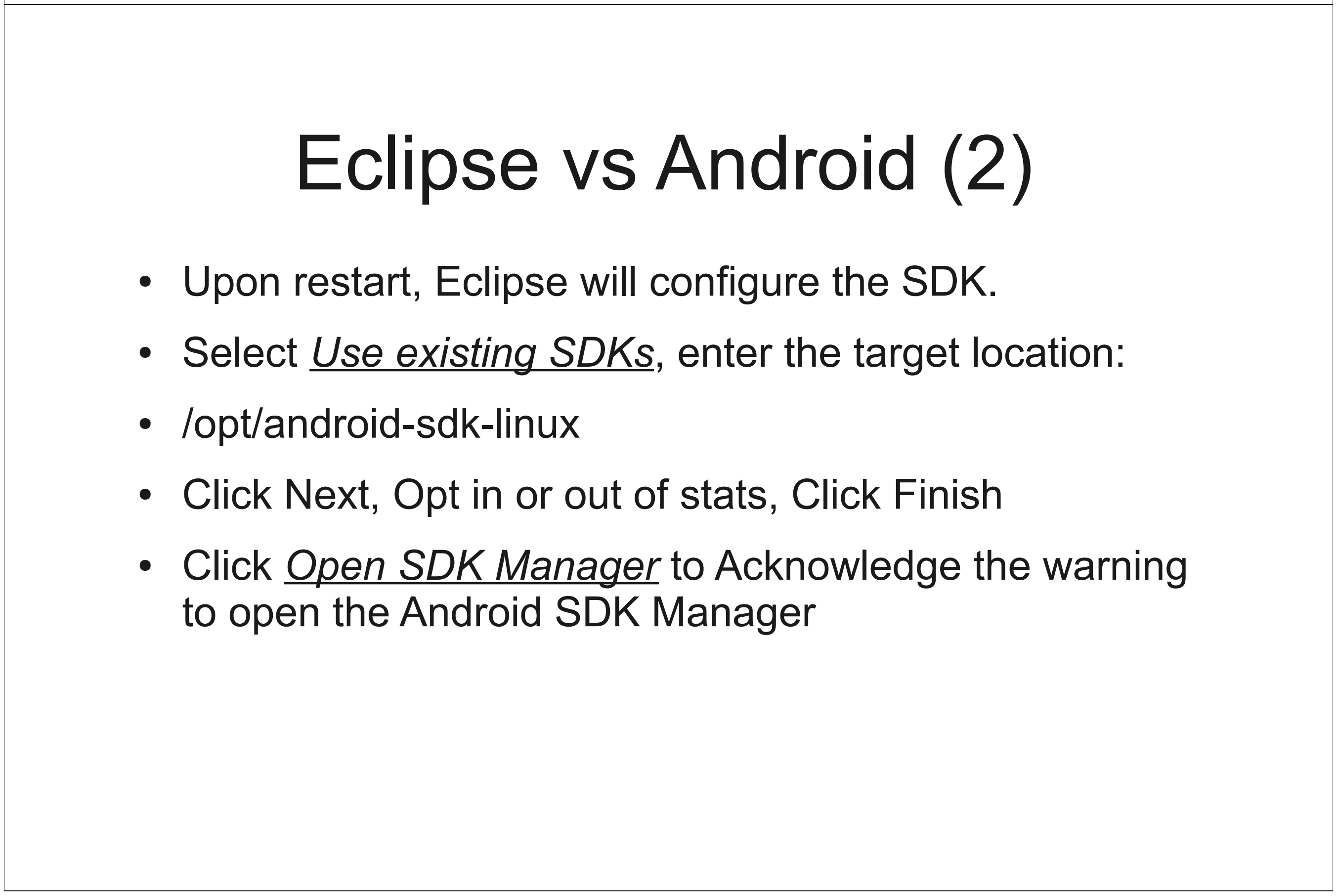
```
wget
http://dl.google.com/android/android-
sdk_r20-linux.tgz

tar -zxf android-sdk_r20-linux.tgz

sudo mv android-sdk-linux/ /opt
```



Eclipse vs Android

- Now, we have to setup Eclipse so that it knows Android
- Help → Install New Software. Click Add...
- Enter a name and enter the URL:

  http://dl.google.com/android/eclipse
- Click OK→ (wait) Select All → Next (x2) → Accept all licenses
- Click Finish → (wait) OK on the Security Warning
- Restart Eclipse



Eclipse vs Android (2)

- Upon restart, Eclipse will configure the SDK.
- Select *Use existing SDKs*, enter the target location:
- /opt/android-sdk-linux
- Click Next, Opt in or out of stats, Click Finish
- Click *Open SDK Manager* to Acknowledge the warning to open the Android SDK Manager

Eclipse vs Android (3)

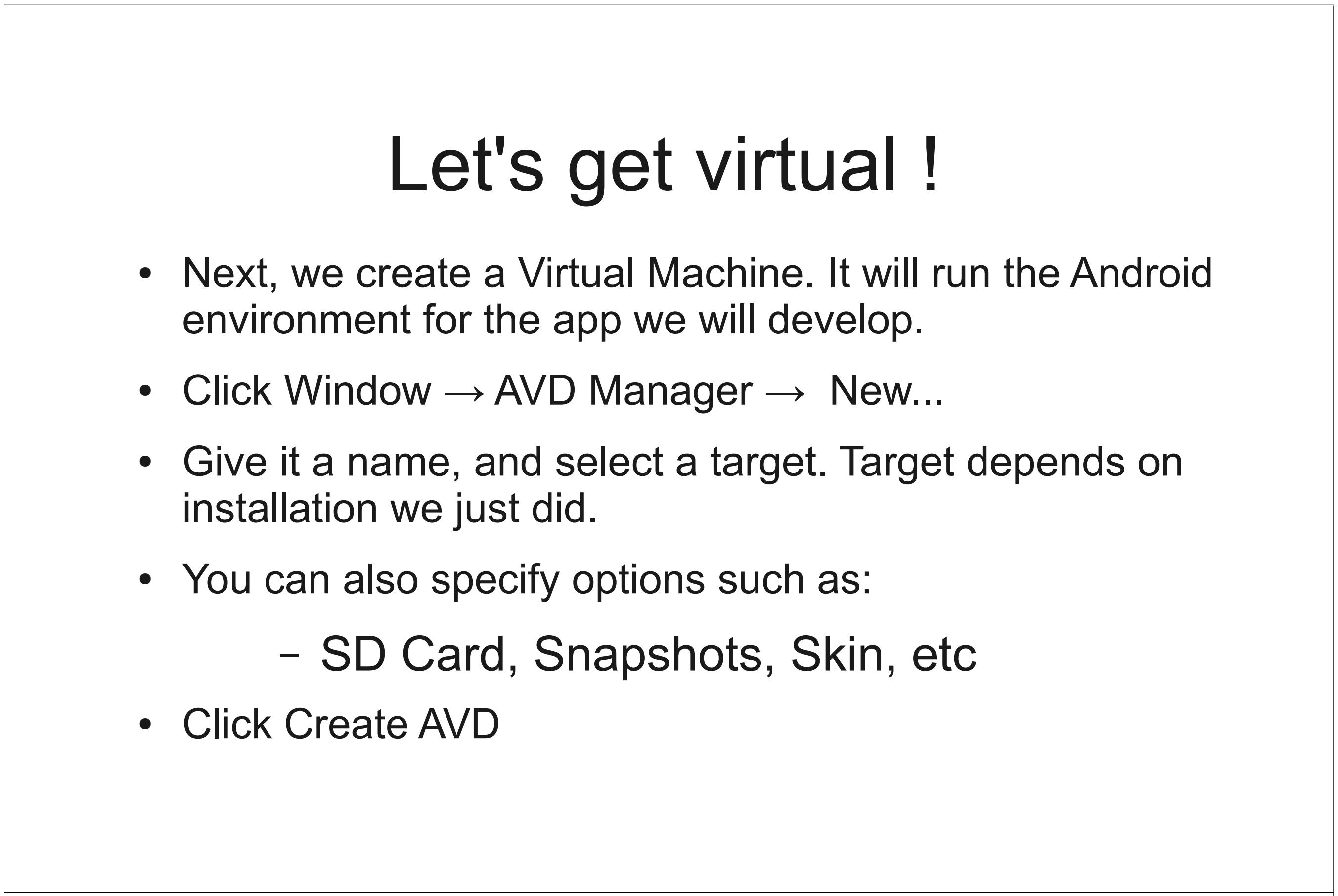
Here, we select the platform(s) where our apps will work.

- Select the platform(s), the tools and the extras.
- The more platforms you select, the longer it takes !
- Only select those you need.
- Accept all licenses, etc...
- Then, we have to run:

  sudo chmod -R 777 /opt/android-sdk/tools

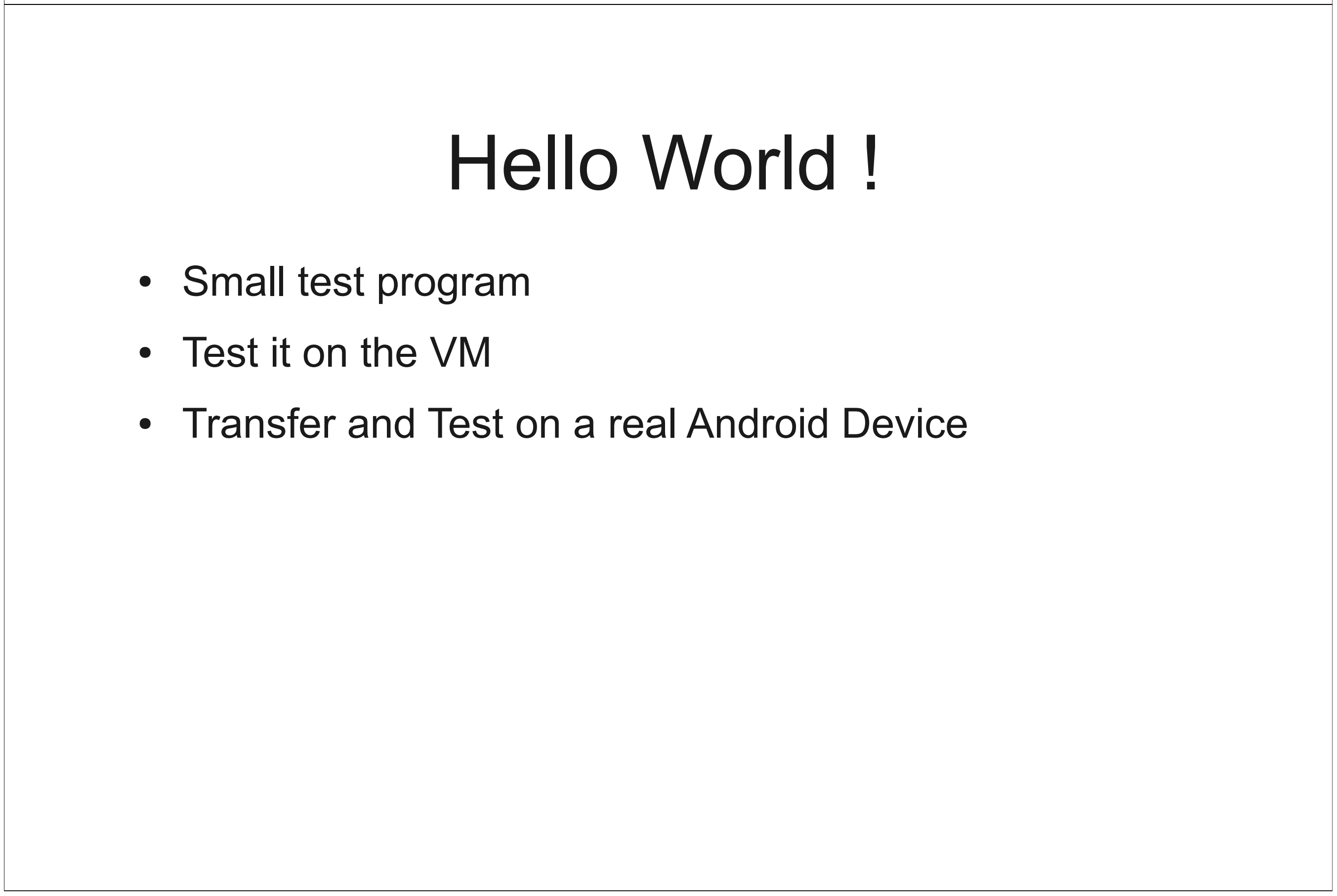
Offline install

- There are scripts to download and install the files directly, instead of downloading through the SDK.
- The whole set of files is over 2GB.
- The ZIPs are to be unzipped in different directories, based on their names.
- Scripts are available at my web site



Let's get virtual !

- Next, we create a Virtual Machine. It will run the Android environment for the app we will develop.
- Click Window → AVD Manager → New...
- Give it a name, and select a target. Target depends on installation we just did.
- You can also specify options such as:
  - SD Card, Snapshots, Skin, etc
- Click Create AVD



Hello World !

- Small test program
- Test it on the VM
- Transfer and Test on a real Android Device

Questions

?

Contact info

Jean-Francois Messier

jf@messier.ca

http://www.android-outaouais.com

http://www.1529.ca/android