JYVÄSKYLÄ STUDIES IN COMPUTING
153

Nezer Jacob Zaidenberg

Applications of Virtualization in Systems Design

JYVÄSKYLÄN YLIOPISTO

JYVÄSKYLÄ STUDIES IN COMPUTING 153

Nezer Jacob Zaidenberg

Applications of Virtualization in Systems Design

Esitetään Jyväskylän yliopiston informaatioteknologian tiedekunnan suostumuksella
julkisesti tarkastettavaksi yliopiston Ylistönrinteellä, salissa YlistöKem4
kesäkuun 15. päivänä 2012 kello 12.

Academic dissertation to be publicly discussed, by permission of
the Faculty of Information Technology of the University of Jyväskylä,
in Ylistönrinne, hall YlistöKem4, on June 15, 2012 at 12 o'clock noon.

UNIVERSITY OF JYVÄSKYLÄ

JYVÄSKYLÄ 2012

Applications of Virtualization in Systems Design

JYVÄSKYLÄ STUDIES IN COMPUTING 153

Nezer Jacob Zaidenberg

Applications of Virtualization in Systems Design

UNIVERSITY OF JYVÄSKYLÄ

JYVÄSKYLÄ 2012

Editor
Timo Männikkö
Department of Mathematical Information Technology, University of Jyväskylä
Pekka Olsbo, Ville Korkiakangas
Publishing Unit, University Library of Jyväskylä

URN:ISBN:978-951-39-4763-7
ISBN 978-951-39-4763-7 (PDF)

ISBN 978-951-39-4762-0 (nid.)
ISSN 1456-5390

Jyväskylä University Printing House, Jyväskylä 2012

ABSTRACT

Zaidenberg, Nezer Jacob
Applications of Virtualization in Systems Design
Jyväskylä: University of Jyväskylä, 2012, 288 p.
(Jyväskylä Studies in Computing
ISSN 1456-5390; 153)
ISBN 978-951-39-4762-0 (nid.)
ISBN 978-951-39-4763-7 (PDF)
Finnish summary
Diss.

In recent years, the field of virtualization has generated a lot of attention and has demonstrated massive growth in usage and applications.

Recent modification to the underlying hardware such as Intel VT-x instructions and AMD-v instructions have made system virtualization much more efficient. Furthermore, recent advances in compiler technology have led the process virtual machine to dominate not only modern programming languages such as C# and Java but also in “ancient" programming languages such as C, C++ and Objective-C.

As a consequence of this trend, virtual services and applications using virtualization have started to spread. The rise of technologies such as storage virtualization, virtualization on clouds and network virtualization in clusters provide system designers with new capabilities that were once impossible without vast investment in development time and infrastructure.

This work describes several problems from the fields of Internet streaming, kernel development, scientific computation, disaster recovery protection and trusted computing. All of these problems were solved using virtualization technologies.

The described systems are state-of-the-art and enable new capabilities that were impossible or difficult to implement without virtualization support. We describe the architecture and system implementations as well as provide open source software implementations.

The dissertation provides an understanding of both the strengths and the weaknesses that are connected by applying virtualization methods to system design. Virtualization methods, as described in this dissertation, can be applicable to future applications development.

Keywords: Virtualization, Storage Virtualization, Cloud Computing, Lguest, KVM, QEMU, LLVM, Trusted computing, Asynchronous mirroring, Replication, Kernel Development, Prefetching, Pre-execution, Internet Streaming, Deduplication

Author

Nezer J. Zaidenberg
Department of Mathematical Information Technology
University of Jyväskylä
Finland

Supervisors

Professor Pekka Neittaanmäki
Department of Mathematical Information Technology
University of Jyväskylä
Finland

Professor Amir Averbuch
Department of Computer Science
Tel Aviv University
Israel

Reviewers

Professor Carmi Merimovich
Department of Computer Science
Tel Aviv-Jaffa Academic Collage
Israel

Professor Jalil Boukhobza
Department of Computer Science
Université de Bretagne Occidentale
France

Opponent

Professor Juha Röning
Department of Electrical and Information Engineering
University of Oulu
Finland

ACKNOWLEDGEMENTS

Without the knowledge and assistance of the mentioned below, this work would not have been possible. I would like to take this opportunity to recognize thank, and acknowledge the following people and Institutions.

First and Foremost, it was a privilege to have Prof. Amir Averbuch as supervisor. I wish to extend my gratitude to Amir for his support and assistance. I have been working with Amir since 2001, when Amir tutored me during my M.Sc thesis. Amir's support throughout the years has made this work possible.

I wish to express my gratitude and warmest thanks toward my second supervisor Prof. Pekka Neittaanmäki for his review, guidance and assistance with all things big and small.

Chapter 3 presents LgDb, joint work with Eviatar Khen, a co-operation that was very stimulating for both of us.

Chapter 4 presents AMirror, joint work with Tomer Margalit. Working with Tomer was interesting and mutually beneficial.

Chapter 5 presents Truly-Protect, joint work with Michael Kiperberg. The cooperation with Michael was very diligent and will also lead to more future work.

Chapter 6 presents a LLVM-Prefetch, joint work with Michael Kiperberg and Gosha Goldberg.

Chapter 7 presents PPPC system, joint work with Prof. Amir Averbuch and Prof. Yehuda Roditi. The work is inspired by systems developed by vTrails. A start-up I founded.

I would also like to thank the other collaborators I have worked with, that is described in chapter 8, Tamir Erez, Limor Gavish and Yuval Meir.

Furthermore, this work wouldn't have been complete without insightful, helpful and often frustrating comments by my reviewers Prof. Jalil Boukhobza, Prof. Carmi Merimovich and the comments of Prof. Tapani Ristaniemi as well as countless other anonymous reviewers of our peer-reviewed papers and posters.

I would like to express my gratitude to the staff of the Department of Mathematical Information Technology for their help and assistance with solving everyday problems. Particularly Timo Männikkö for his review of the manuscript and Kati Valpe and Rantalainen Marja-Leena for all their help.

Finally, I wish to thank my parents and my family and all my friends for their moral support and interest in my work, especially my dearest partner Vered Gatt for her patience, encouragement and review of the work and my brother Dor Zaidenberg for his insightful comments.

I greatly appreciate the support of the COMAS graduate school, STRIMM consortium and Artturi and Ellen Nyyssönen foundation that provided funding for the research.

I owe a debt of gratitude to my faithful proof-reader, Ms. Shanin Bugeja. If this monograph is reasonably free from error it is due in no small measure to her meticulous and ruthless application of the red pen.

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

CONTENTS

1 INTRODUCTION

Virtualization is an innovative method for system development that has attracted a lot of buzz in recent years. In this dissertation, we present five challenging problems that are hard to solve without utilizing virtualization methodologies. These problems which originated from trusted computing, high availability, kernel development, efficient computation and peer-2-peer streaming, have very little in common. However, these problems share common core infrastructure in their proposed solutions. While all the solutions make use of virtualization, the exact virtualization method varies from solution to solution.

We introduce five systems, which are classified as system, process, storage and network virtualization, that solve these problems. In each case, virtualization either increase the solution capabilities or reduced the development cycle time.

While we describe the virtualization approach that is being used for each problem, the developed methodologies can be employed to tackle other problems as well.

1.1 Virtualization Methods for System Design

Virtualization provides an abstraction layer or an indirection layer between the running software and the hardware.

Virtualization abstraction provides the ability to insert operations that in regular computing environment (without virtualization) are almost impossible to perform.

We demonstrate how the power of indirection provided by the virtualization layer is critical in providing constructive solution to each system that is described in this dissertation.

1.2 The Main Problems that are Solved

In this section, we present the problems, The reasons why the problems are important and motivates their solution and finally the proposed solutions are briefly described.

1.2.1 System 1: LgDb: Kernel Profiling and Code Coverage

User space developers are able to use many tools to profile and analyze their code. Unfortunately, user space tools do not work on kernel development leaving kernel developers without such gadgetry.

LgDb, described in chapter 3, is a tool that is based on Lguest. It allows kernel developers to profile their code and use code coloring techniques for kernel modules. System virtualization provides the ability to develop such a tool.

Lguest, which is the underlying technology that we used, is described in Appendix 6.

1.2.2 System 2: Disaster Recovery Protection Using Asynchronous Mirroring of Block Devices and Virtual Machines

Disaster recovery protection addresses the problem of losing as little data as possible while being capable of recovering the data as soon as possible in case of disaster. Disaster can be defined as a wide variety of events ranging from server thefts to fire or flood. In all cases of disaster, we can observe unexpected, permanent or temporal lack of access to the servers in the primary site.

There are multiple "off-the-shelf" solutions to achieve disaster recovery for the enterprise market. Such solutions include VMWare VMotion[Gle11] or Remus for Xen[CLM$^+$08], but there is no solution for disaster recovery protection for home users or small businesses.

We developed the AMirror system discussed in chapter 4 that provides high availability environment for block devices and virtual machines.

The system is unique compared to "off-the-shelf" systems because it can work on a relatively low upload link and can provide reliability to home users and small businesses.

AMirror focuses on the use of storage virtualization and system virtualization for high availability and disaster recovery protection.

AMirror is implemented in the Linux block device layer discussed in Appendix 5. It is capable of replicating virtual machines running on Lguest and QEMU/KVM. Lguest and QEMU/KVM are described in Appendixes 6 and 3.

1.2.3 System 3: Truly-Protect: Using Process VM for Trusted Computing

The "Trusted Computing" concept means using trusted hardware for providing conditional access to a software system or digital media. Unfortunately, there are no physical trusted computing environments available today, especially in home user environments.

Chapter 5 describes the Truly-Protect system. Truly-Protect is a framework that incorporates a process virtual machine that masks the running software internals from a malicious user.

Truly-Protect can serve as a framework for a conditional access or digital rights management system. The use of a process virtual machine as a framework for trusted computing is not new but Truly-Protect is unique compared to other industrial products as it relies on cryptographic foundation instead of relying on obscurity.

Truly-Protect relies on the cryptographic measures and assumptions about the system itself, specifically it assumes that part of the CPU internal state (registers' content) is hidden from the user. These assumptions allow Truly-Protect to act as a VM-based conditional access/trusted computing environment.

Providing a trusted computing environment without virtual machine or hardware support is impossible. Truly-protect attempts to bridge that gap. It demonstrates the power of process virtual machines.

Appendix 1 describes the current status of trusted computing. Appendix 2 briefly describes the cryptographic methods and protocols used by Truly-Protect.

1.2.4 System 4: LLVM-prefetch: Prefetching and Pre-execution

Modern computational environments tend to have I/O-bound and CPU-bound segments in the code. This observation leads developers to attempt to run in parallel I/O-bound and CPU-bound code segments.

Unfortunately, there is no tool that can automatically merge CPU-bound and I/O-bound segments. While programmers can usually do the job manually it is impossible to redevelop and redesign all the programs for this goal. This fact motivates the development of LLVM-prefetch.

In LLVM-prefetch, we use the power of the VM environment to construct a pre-execution thread for data prefetching. Developing similar systems without a VM requires a programmer time investment for manually creation of the pre-execution threads. The use of virtual machines enables us to automate this process.

LLVM-prefetch is another example that demonstrates the power of the process virtual machine.

LLVM-prefetch is described in chapter 6. The Low Level Virtual Machine (LLVM), which is the process virtual machine that is used as underlying infrastructure, is described in chapter 4.

1.2.5 System 5: Peer-2-Peer Streaming

The high cost of bandwidth and the lack of broadcast up link capacity is a major problem faced by Internet broadcasters. Furthermore, the high cost of bandwidth is the main reason that we do not see high quality live TV broadcast wide spread on the Internet.

Peer-2-Peer streaming is a method to harness the power of cloud computing for media streaming. In peer-2-peer streaming, the broadcasting servers stream to selected number of top tier peers who deliver the stream to other peers. The process repeats itself until massive number of peers receive the stream.

The Peer-2-Peer streaming system discussed in Chapter 7 is an excellent example of using cloud computing and network virtualization for streaming. Appendix 7 describes low level design details and state machines of the peer-2-peer streaming system.

The system design also maintains trusted computing and protection of digital rights for the broadcaster. Trusted computing is described in Appendix 1.

Chapter 8 describes the future work and systems that are currently under development but are not ready yet for publication exposure, with some initial results.

1.3 The Underlying Concepts

We provide a detailed description for some of the underlying concepts and technologies that are used in the individual systems. With the exception of the virtualization chapter, the underlying concepts are described in the appendixes.

1.3.1 Virtualization

Virtualization is the common methodology for all the systems that we developed. Chapter 2 discusses the different virtualization environments types and the different virtual machines. We also discuss how process and system virtual machines are implemented and survey common implementations.

1.3.2 Trusted Computing

Trusted computing is a concept that originated from information security. It applies to scenarios where a content (software, media) owner wants to ensure proper use of the contents. Truly-Protect provides a trusted computing environment based on process virtual machine.

Trusted computing is discussed in Appendix 1.

1.3.3 Cryptography

Truly-Protect relies on several cryptographic algorithms, which are described in detail in Appendix 2.

1.3.4 Linux Block Layer

Linux Block Layer is the common interface for all disk drives under Linux. It is described in detail in chapter 5. The Linux Block Layer is the underlying technology for the AMirror system discussed in Appendix 4

1.3.5 LLVM

LLVM is an innovative compiler and run-time environment. The run-time environment is based on a low-level virtual machine, a process virtual machine that executes at near assembler level and provides plenty of compile-time, link-time and run-time performance improvements.

LLVM is described in Appendix 4.

1.3.6 Lguest and Virt I/O

Lguest is a semi-educational Linux-on-Linux hypervisor developed initially in [Rus] as a test bed for Virt I/O.

Virt I/O is the abstract interface for driver development under Linux.

We used Lguest as the underlying technology for our kernel development tool set (LgDb). We also implemented our asynchronous mirroring for virtual machines for Lguest. Lguest is described in Appendix 6.

1.3.7 KVM and QEMU

KVM [Kiv07] is a fully fledged, industrial grade virtualization environment that resides in the Linux kernel.

QEMU [Bel05] is an emulation environment for virtual machines. Its latest versions are closely coupled with KVM.

KVM and QEMU were used in our AMirror asynchronous mirror for replication of virtual machines across networks.

KVM and QEMU are described in Appendix 3.

1.4 Dissertation Structure

Chapter 2 provides a detailed introduction to virtualization technology as well as virtual machines. The rest of the dissertation describes the innovative systems 1-5 that we developed. The known underlying technologies, which are required by systems 1-5, are organized and described in the Appendixes.

Figure 1 presents the flow and functional connections between the various chapters in the dissertation.

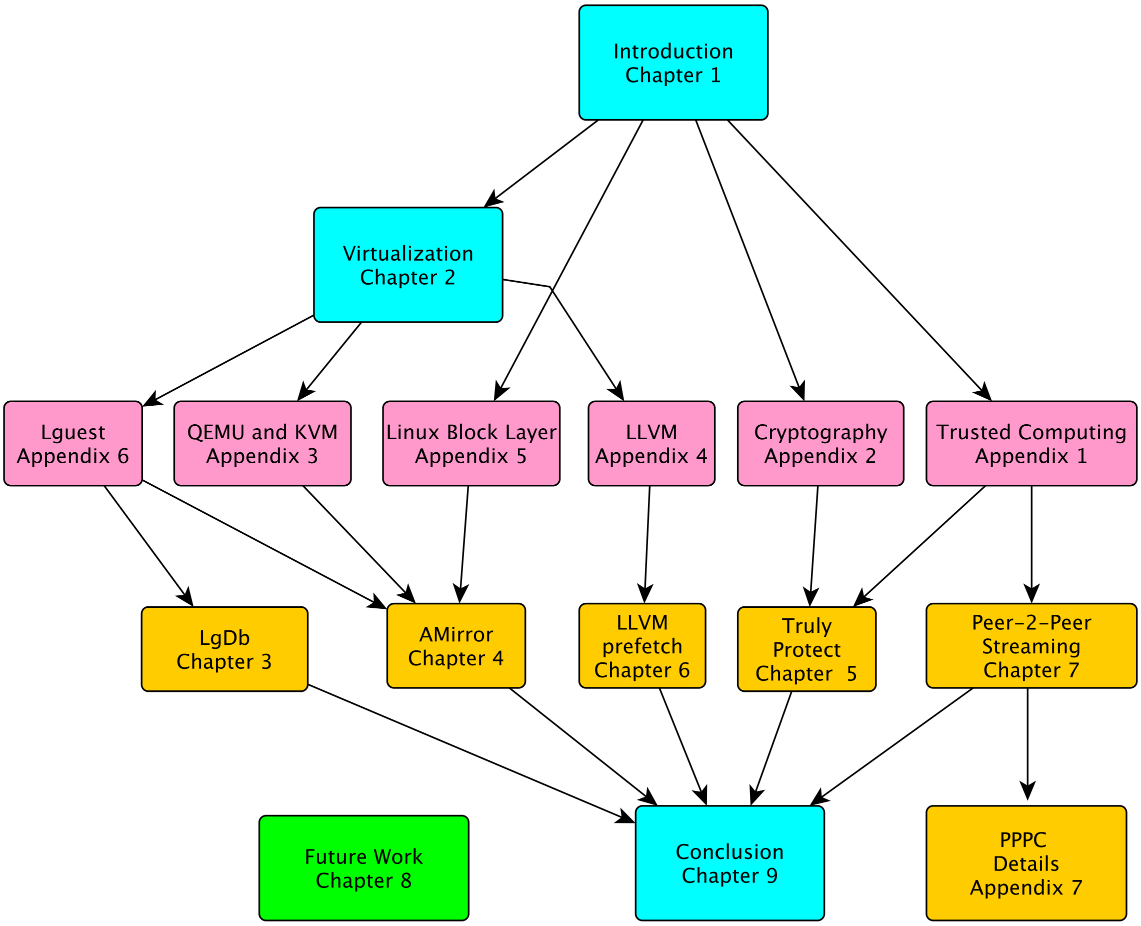


FIGURE 1 The flow and the functional connections among the chapters in the dissertation

1.5 Contributions and papers from the dissertation

The technologies and systems, which were described in this dissertation, have been submitted to peer-reviewed conferences. The personal contributions to the systems, conferences and journals in which they were published are described below.

LgDb The initial paper was published in SCS SPECTS 2011 [AKZ11a] and a poster in IBM Systor 2011. Our second paper in the field which describes LgDb 2.0 is in a final preparation stage. In LgDb 2.0 we have removed the dependency on code injections and used a kernel debugger connected over virtual serial port instead of hypervisor. My contributions include the concept and the design of the system, debugging critical parts of the software, authoring of the paper and its presentation in the SPECTS and Systor conferences. Participation in the design of LgDb 2.0

Asynchronous mirror published as [AMZK11] in IEEE NAS 2011 with a Poster demonstrating the work appearing in IBM Systor 2011. My contributions included the system concept and internal design, debugging of critical parts and authoring of the paper and its presentation in IBM Systor 2011.

Truly-Protect The initial paper was published in IEEE NSS [AKZ11b] and a second paper presents additional system developments that was submitted to IEEE System journal[AKZ12]. My contributions include the system concept and design, debugging of critical parts, the GPGPU-enhancement for Truly protect, authoring the paper and the presentation of Truly Protect in the NSS conference.

LLVM-Prefetch This system concept was suggested by Michael Kiperberg, an M.Sc student under my supervision. I participated in the system design and development as well as in the system benchmarking. design. I authored the system description document. This system has not been published yet.

PPPC The initial version of the system was presented in CAO 2011 as [ARZ11] and a detailed system description will appear in P. Neittaanmäki's 60th jubilee publication[ARZ12]. My contributions include the system concept, the system design, participating in the development of the algorithm, the development of the system software and authoring of the papers.

2 VIRTUALIZATION

The term "virtualization" refers to the usage of a virtual rather than actual (physical) computer resource.

In this chapter, we will describe various forms of virtualization, virtualization concepts, taxonomy of virtual machine types and modern implementations.

We differentiate between system virtual machines such as VMWare ESX server [VMW] or Lguest [Rus] from process virtual machines such as the Java Virtual Machine [LY99] or LLVM [Lat11]. Virtualization aspects in modern operating system such as storage virtualization and network virtualization or virtualization in clouds are described here as well.

2.1 Introduction to Virtualization

To begin our discussion on virtual machines, we will begin with a definition of physical (non-virtual) machines. For a process, a machine is the physical hardware it runs on coupled with operating systems with all the resources it provides. For operating system (OS) components, we must distinguish between physical resources such as main system memory or file systems on disk drives and virtual resources. OS provides running processes with virtual resources such as virtual memory or the virtual file systems. At the process level, the virtual memory or files and paths on the virtual file system are as close as we can get to the physical resources. Furthermore, normally a process has no access to "none-virtual" file systems or memory.

Furthermore, we can distinguish between physical hardware that can be located on physical devices and virtual hardware that runs on a hypervisor.

Of course, some hardware components may themselves be virtual such as a virtual tape library which is a typical disk based system that emulates a tape drive or a solid state disk posing as a mechanical magnetic disk drive with heads and sectors.

In this dissertation, we will use the practical definition where a virtual machine is an environment that is used to execute a particular system (either a user process or a complete system) in which at least some components are used via an indirect abstraction layer.

The virtual machine will execute the system like the normal system it was designed for. The virtual machine is implemented as a combination of a real machine with physical resources and virtualizing software. The virtual machine may have resources that differs from the real machine. For example, a different number of disk drives, memory or processors. These processors may even execute a different instruction set than the instruction set of the real machine. This is common in a emulator as well as in process virtual machines executing for example Java byte-code instead of the machine instruction set.

The virtual machine does not guarantee and usually cannot provide identical performance to a physical system. Virtualization does not pose requirements on the performance but merely on the ability to run software.

Since the virtualization environment is different for different types of systems such as process environment and system environment, there are many types of virtual environment systems. Common virtualization environments include:

1. System virtual machine
   (a) Emulation;
   (b) Hardware virtualization;
   (c) Desktop virtualization;
   (d) Software virtualization.
2. Process virtual machine
   (a) High level virtual machine;
   (b) Low level virtual machine;
   (c) Stack based virtual machine;
   (d) Register based virtual machine.
3. Virtualization in grid and clouds
   (a) Infrastructure as a service;
   (b) Platform as a service;
   (c) Software as a service.
4. Virtualization of hardware in OS components
   (a) Memory virtualization;
   (b) Storage virtualization;
   (c) Data virtualization;
   (d) Network virtualization.

A system virtual machine emulates a set of hardware instructions that allows users to run their own operating systems and processes on a virtual environment. A system virtual machine software is called hypervisor or Virtual Machine Monitor (VMM). We must distinguish between Type-1 hypervisors, which act as operating systems and run directly on the hardware, and Type-2 hypervisors, which

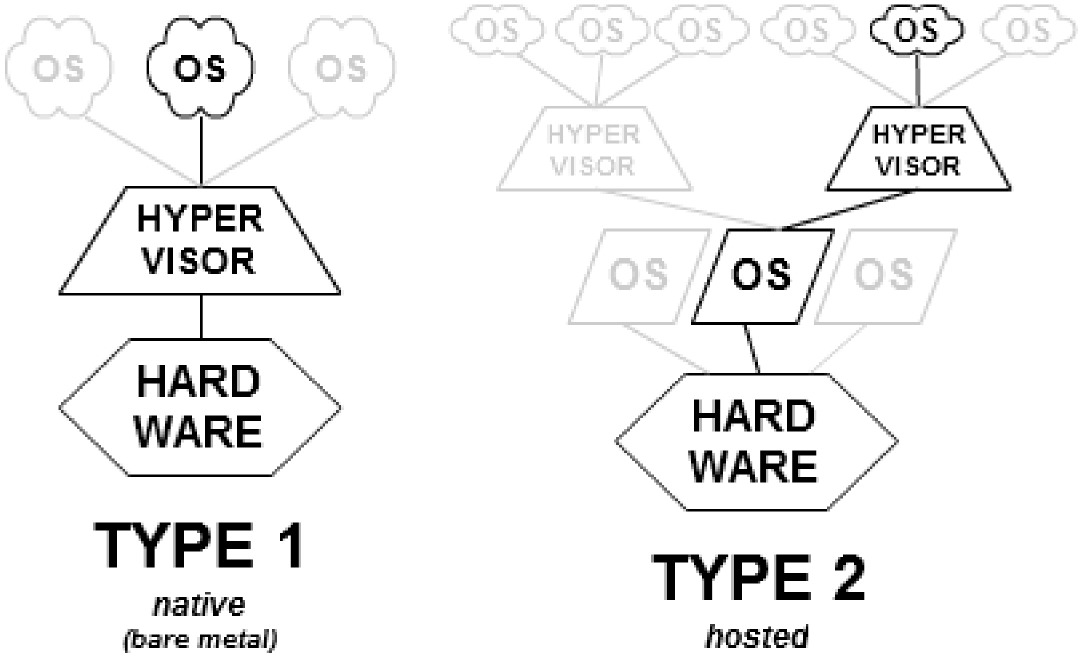


FIGURE 2 Type I and Type II hypervisors

run as user processes under the host operating system. Industrial Type-1 hypervisors include VMWare ESX server, IBM VM operating system for mainframe and Sony Cell level-1 OS on the PlayStation 3. Industrial Type-2 hypervisors include VMWare Workstation, VirtualBox, KVM, etc.

A special case for type-2 hypervisors are the emulators. An emulator is hardware and/or software that duplicates (or emulates) the functions of one computer system on another computer system so that the behavior of the second system is closely resemble the behavior of the first system. Emulators also run as user processes. Bochs [DM08] and the original versions of QEMU [Bel05] are examples of open source emulators.

Section 2.2 describes system virtual machines in greater detail and section 2.3.1 describes emulators. Appendix 6 describes the Lguest hypervisor, which can serve as a reference for building hypervisors from scratch. Appendix 3 describes the KVM and QEMU hypervisors that are used in AMirror, which is our asynchronous replication system and described in chapter 4. Type I and Type II hypervisors are illustrated in Fig. 2.

As opposed to system virtual machines, in process virtual machines, the virtualization software is placed on top of a running operating system and hardware. It provides a complete execution environment for a process instead of an operating system. The process virtual machine is different from standard library or interpreter as it emulates both the user function calls as well as the operating system interface such as system calls. A process virtual machine is illustrated in Fig. 3.

The operating system itself provides a mesh of virtual interfaces to the hardware ranging from virtual memory to virtual file systems. Most of these compo-

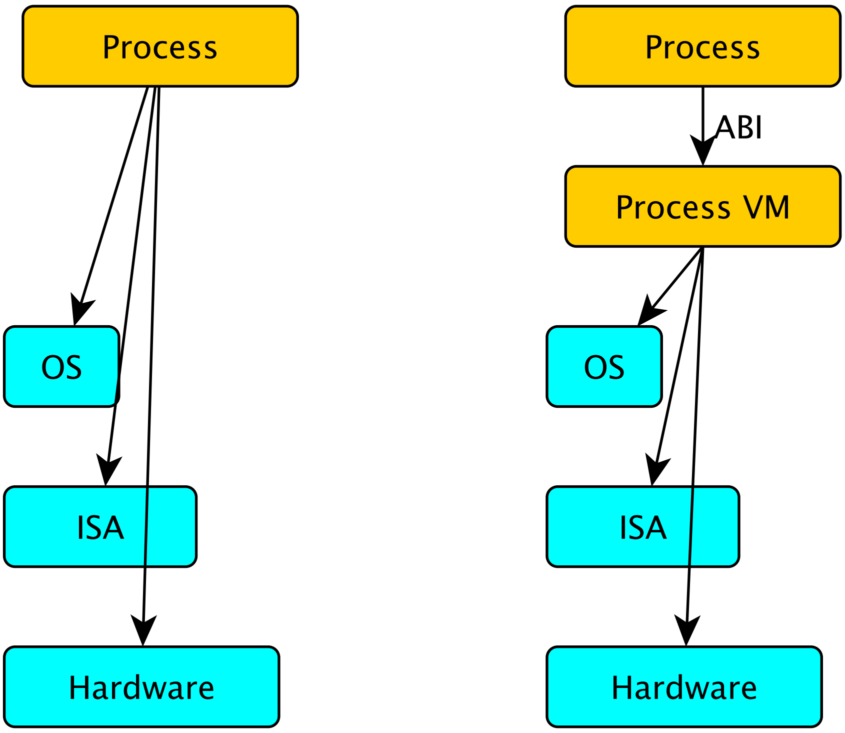


FIGURE 3 Process Virtualization

nents are provided for similar reasons we use process and system virtualization: For example the virtual file system(VFS) provides an indirection layer between disk file systems and the operating systems. The VFS allows multiple file systems support to be provided by the OS and merged transparently by the virtual file system. Virtual memory allows each process to have its own independent memory space. Virtual memory allows the user to use more memory then their machine have by using swap space.

Virtual memory and file systems belong to the operating system scope. The reader is referred to [Lov10] for a good reference on the virtual memory implementation in Linux and [RWS05] for a good reference on the virtual file system on UNIX.

Two other parts of the operating system virtualization capabilities that are not as well known are storage virtualization and network virtualization. Storage virtualization is often implemented by the operating system and other storage devices. When Storage virtualization is implemented by the operating system using a component called Logical Volume Management (LVM). The LVM provides a logical method for managing space on mass-storage devices. "Logical" here means managing the storage logically where user code does not access the physical storage directly and therefore, the user process is not restricted by it but instead the storage is accessed through an abstraction layer. Using the LVM abstraction layer allows the system administrator to define logical (virtual) volumes instead of physical partition), and thus perform logical operations to control each of them by changing its size or location, group a few logical volumes under the same name for managing them together, provide features such as RAID or deduplication, use cloud storage and more. The name "LVM" was first published by HP-UX OS and it is interpreted as two separate concepts:

1. Storage managing administrative unit.
2. OS component that practically manages the storage.

Storage virtualization is discussed in detail in section 2.5. AMirror, which is described in chapter 4, is a logical volume manager that is focused on replication.

Network virtualization is a process that combines hardware and software network resources and network functionality into a single software-based administrative entity which is a virtual network. Network virtualization ranges from the simple acquirement of more than one IP using a single network adapter to virtual private networks and creation of shared network interface for a cluster for load balancing and high availability. Section 2.6 describes in greater detail several network virtualization use cases.

Cloud computing is the next phase in the evolution of the Internet. The "cloud" in cloud computing provides the user with everything. As opposed to the standard Internet module in which the Internet provides the user with content that is saved and processed on the user machine in a cloud computing environment, the "cloud" provides the user with infrastructure, computation power, storage, software and anything else that the user may require.

Since the user needs to receive the power of the cloud in a medium that the user can use, cloud computing usually provides virtual resource. Thus, virtualization goes hand in hand with cloud computing. We use virtualization to provide an abstraction layer for services provided by the cloud. Virtualization in clouds is described in section 2.7. The PPPC system, which is described in chapter 7, is an example of a cloud based system.

2.2 System Virtual Machines

System virtual machines provide a complete system environment in which multiple operating systems, possibly belonging to multiple users, can coexist.

These virtual machines were originally developed during the 1960s and early 1970s, and those machines are the origin of the term *virtual machine* widely used today. Around that time, the requirement for building system virtual machines were set by Popek and Goldberg in [PG74].

By using system virtual machines, a single host hardware platform can support multiple guest OS environments simultaneously. At the time they were first developed, mainframe computer systems were very large and expensive and computers were almost always shared among a large number of users.

Different groups of users sometimes wanted different operating systems to run on the shared hardware. Virtual machines allowed them to do so.

Alternatively, a multiplicity of single-user operating systems allowed a convenient way of implementing time-sharing among many users. As hardware became much cheaper and more accessible, many users deserted the traditional mainframe environment for workstations and desktops, and interest in these classic System Virtual Machines declined and faded.

Today, however, System Virtual Machines are enjoying renewed popularity. Modern-day motivation for using system Virtual machines are related to the motives of old fashioned System Virtual Machines The large, expensive mainframe systems of the past are comparable to servers in server farms.

While the cost of hardware has greatly decreased, the cost of cooling, power (electricity) and man power for monitoring the servers has greatly increased.

Thus servers hosted in server farms are expensive and these servers may be shared by a number of users or user groups. Perhaps the most important feature of today's System Virtual Machines is that they provide a secure way of partitioning major software systems that run concurrently on the same hardware platform.

Software running on one guest system is isolated from software running on other guest systems.

Furthermore, if security on one guest system is compromised or if the guest OS suffers a failure, the software running on other guest systems is not affected. Thus, multiple users can run on a virtual environment on one server and each user applications will run autonomously, unaffected by other users. Naturally, running multiple users in a virtual environment on a single server allows mul-

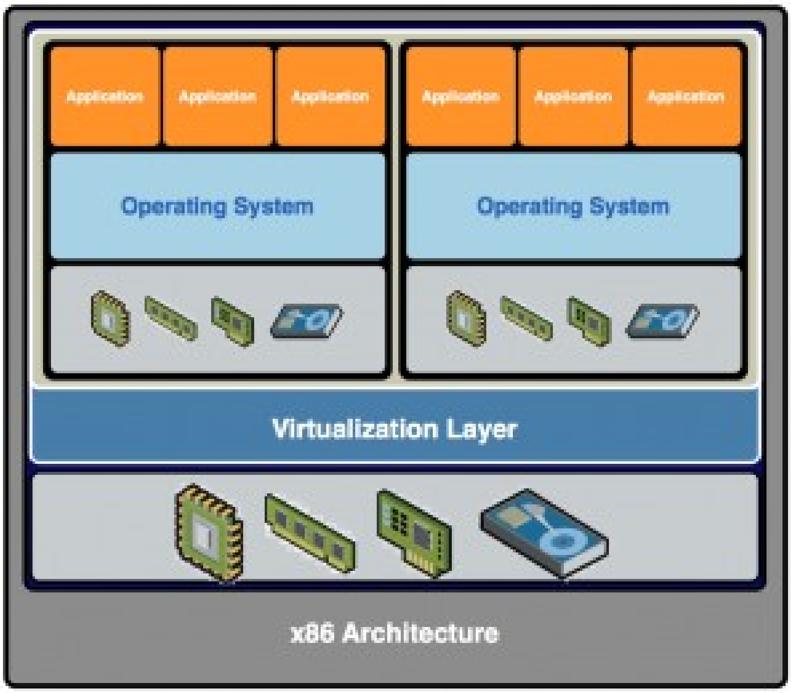


FIGURE 4 Multiple operating systems running on virtual environments on a single system

tiple customers to share the cost thus greatly reducing the maintenance cost of servers in a server farm.

The ability to support different operating systems simultaneously, e.g., Windows and Linux as illustrated in Fig. 4, is another reason some users find VM appealing.

Other users use system virtual machines for additional purposes including software development, sand boxing, etc.

2.3 Implementations of System Virtual Machines

2.3.1 Whole System Virtual machines – Emulators

In the conventional System Virtual Machines described earlier, all related system software (both guest and host) and application software use the same ISA as the underlying hardware. In some important cases, the ISA for the host and guest systems is different. For example, the Apple PowerPC-based systems and Windows PCs use different ISAs and even different CPU architectures. For example, Power PC uses IBM Power chips which is a 64-bit RISC processor while PCs use Intel 32bit x86 CISC instruction set. A modern example involves embedded system developers such as iPhone developers that emulate the iPhone ARM processor (32bit RISC processor) on modern Intel x86_64 processors that are 64bit

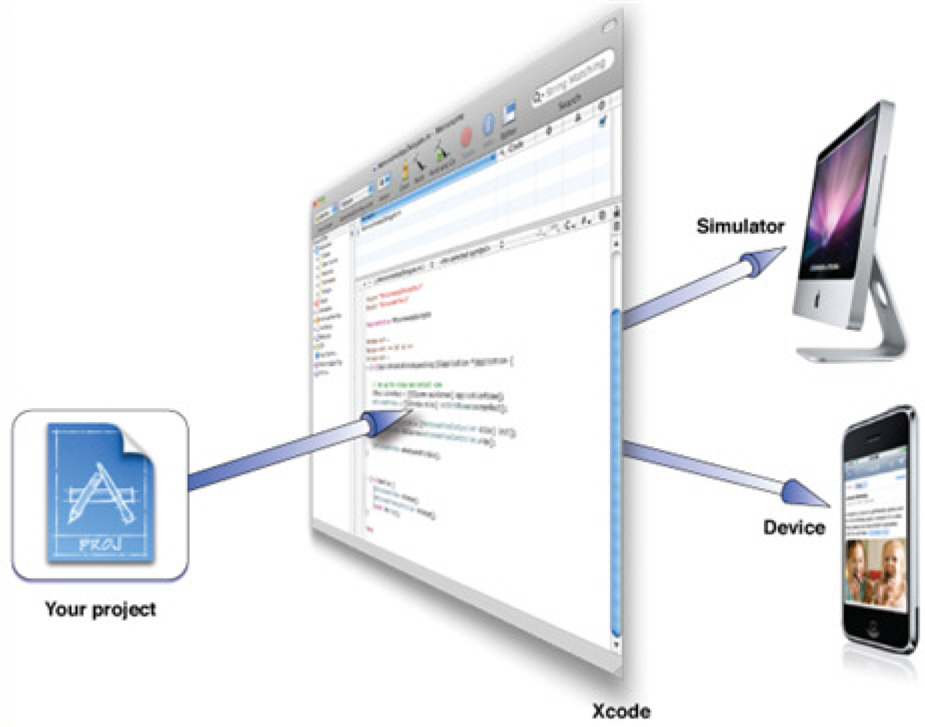


FIGURE 5 iPhone development using device and simulator

CISC processors.

Because software systems are so coupled with the hardware systems without some emulation type, it is necessary to procure another system to run software for both types of hardware. Running multiple systems will complicate the usage and the development which act as a catalyst for an emulation approach to virtualization.

These virtualization solutions are called called *whole-system Virtual machines* because they essentially virtualize all software and hardware components. Because ISAs are different, both applications and OS code require emulation, e.g., via binary translation. For whole-system Virtual machines, the most common implementation method is to place the VMM and guest software on top of a conventional host OS running on the hardware.

An example of this type of virtual machine is the Apple iPhone simulator that runs on top of the standard Apple Mac OS X that allows for the development and the execution of iPhone applications. Figure 5 illustrates the iPhone simulator which is a whole-system VM that is built on top of a conventional operating system (OS X) with its own OS and application programs.

In the iPhone simulator, the VM software is executed as an application program supported by the host OS and uses no system ISA operations. It is as if the VM software, the guest OS and guest application(s) are one very large application implemented on Mac OS X.

To implement a VM system of this type, the VM software must emulate the entire hardware environment. It must control the emulation of all the instructions, and it must convert the ISA guest system operations to equivalent OS calls made to the host OS. Even if binary translation is used, it is tightly constrained because translated code often cannot take advantage of the underlying ISA system features such as virtual memory management and trap handling. In addition, problems can arise if the properties of hardware resources are significantly different in the host and in the guest. Solving these mismatches is a major challenge in the implementation of the whole-system VMs.

2.3.2 Hardware Virtualization

Hardware virtualization refers to the creation of a virtual machine that acts like a real computer with an operating system. Software executed on these virtual machines is separated from the underlying hardware resources. For example, a computer that is running Microsoft Windows may host a virtual machine that looks like a computer with an Ubuntu Linux operating system. Subsequently, Ubuntu-based software can be run on that virtual machine.

Hardware virtualization hides the physical characteristics of a computing platform from users, instead showing another abstract computing platform.

The software that controls the virtualization were originally called to be called a "control program", but nowadays the terms "hypervisor" or "virtual machine monitor" are preferred. (Since the OS is called the supervisor, the virtualization software which sits "above" the OS is the hypervisor.)

In hardware virtualization, the term host machine refers to the actual machine on which the virtualization takes place. The term guest machine, refers to the virtual machine.

Hardware virtualization is optimized with the inclusion of circuits in the CPU and controller chips to enhance the running of multiple operating systems (multiple virtual machines). Hardware virtualization support refers to the inclusion of instructions for saving and restoring the CPU state upon transitions between the guest OS and the hypervisor.

Hardware virtualization support has been available in IBM mainframes since the 70's as well as on Sun high end servers and other machines. Hardware virtualization became popular when Intel first included Intel Virtualization technology in 2004. AMD followed suit in 2006 with AMD virtualization.

2.3.3 Desktop Virtualization

Desktop virtualization is the concept in which the logical desktop is separated from the physical machine. One form of desktop virtualization, called virtual desktop infrastructure (VDI), can be thought of as a more advanced form of hardware virtualization: instead of direct interactions with a host computer via a keyboard, mouse and monitor connected to it, the user interacts with the host computer over a network connection (such as LAN, wireless LAN or even the Internet) using another desktop computer or a mobile device. In addition, the host computer in this scenario becomes a server computer capable of hosting multiple virtual machines at the same time for multiple users. Another form, session virtualization, allows multiple users to connect and log into a shared but powerful computer over the network and use it simultaneously. Each is given a desktop and a personal folder in which they store their files. Thin clients, which are seen in desktop virtualization environments, are simple computers that are designed exclusively to connect to the network and display the virtual desktop system. Originally, thin clients lacked significant storage space, RAM and processing power, but nowadays thin clients do significant processing and even GPU processing and are not significantly inferior to a regular desktop. Desktop virtualization, sometimes called client virtualization, conceptually, separates between a personal computer desktop environment from a physical machine using the client-server computing model. VDI is the server computing model which enables desktop virtualization, encompassing the hardware and software systems required to support the virtualized environment. The term VDI was coined by the company VMWare and it is now an accepted term in the industry adopted by Microsoft, Citrix and other companies.

Many enterprise-level implementations of this technology store the resulting "virtualized" desktop on a remote central server instead of on the local storage of a remote client. Thus, when users work from their local machine, all the programs, applications, processes, and data used are kept on the server and run centrally. This allows users to run an operating system and execute applications from a smart-phone or thin client which can exceed the user hardware's abilities.

Some virtualization platforms allow the user to simultaneously run multiple virtual machines on local hardware such as a laptop, by using hypervisor technology. Virtual machine images are created and maintained on a central server and changes to the desktop VMs are propagated to all user machines through the network, thus combining both the advantages of portability afforded by local hypervisor execution and of central image management.

Running virtual machines locally requires modern thin-clients that posses more powerful hardware that are capable of running the local VM images such as a personal computer or notebook computer and thus it is not as portable as the pure client-server model. This model can also be implemented without the server component, allowing smaller organizations and individuals to take advantage of the flexibility of multiple desktop VMs on a single hardware platform without additional network and server resources.

Figure 6 demonstrates a desktop virtualization environment.

2.3.4 Software virtualization

Software virtualization or application virtualization is an umbrella term that describes software technologies that improve portability, manageability and compatibility for applications by encapsulating them in containers managed by the virtualization abstraction layer instead of the underlying operating system on which they are executed. A fully virtualized application is not installed on the client system hard drive although it is still executed as if it was. During runtime, the application operates as if it was directly interfacing with the original operat-

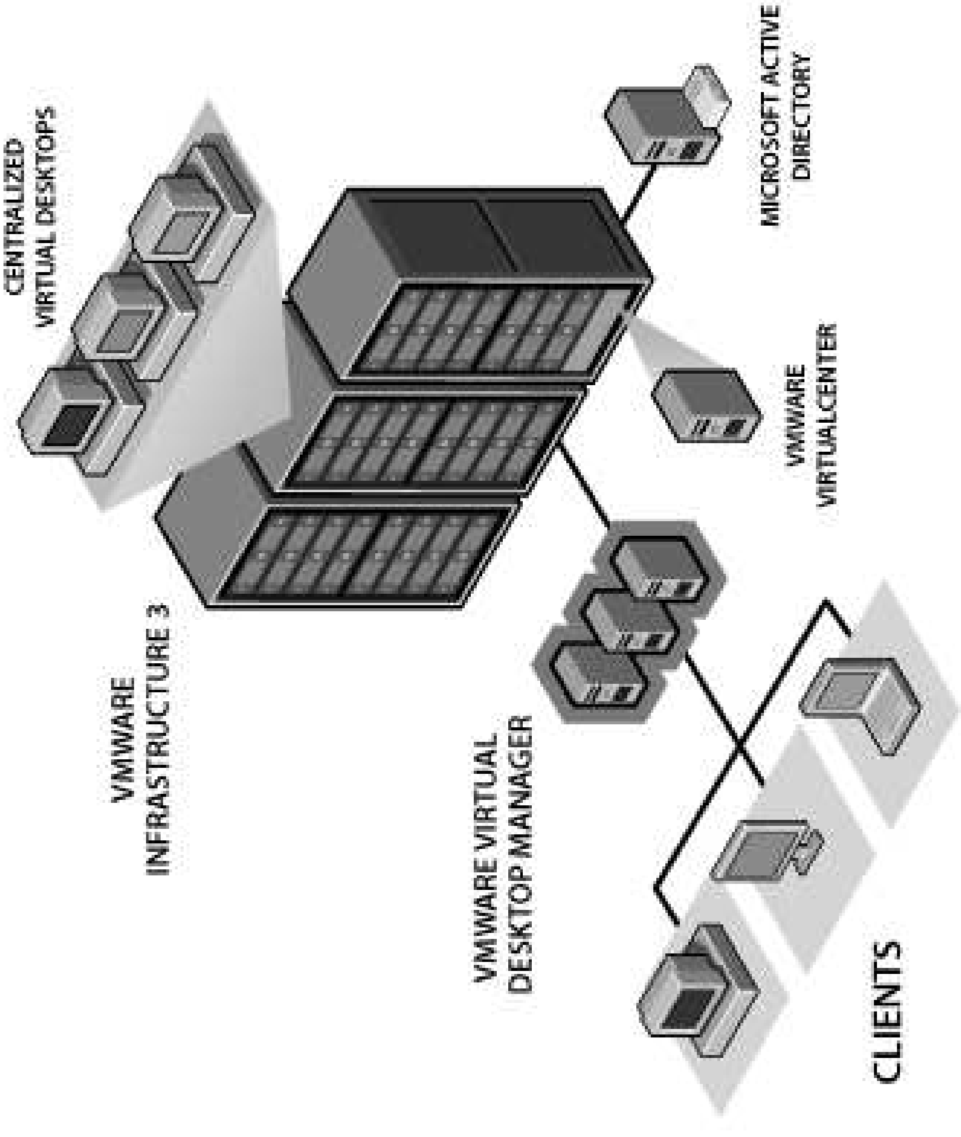


FIGURE 6 Desktop Virtualization

ing system and all the resources managed by it. However, the application is not technically installed on the system. In this context, the term "virtualization" refers to the software being encapsulated in an abstraction layer (application), which is quite different to its meaning in hardware virtualization, where it refers to the physical hardware being virtualized.

The virtualization application is hardly used in modern operating systems such as Microsoft Windows and Linux. For example, INI file mappings were introduced with Windows NT to virtualize, into the registry, the legacy INI files of applications originally written for Windows 3.1. Similarly, Windows 7 and Vista implement a system that applies limited file and registry virtualization so that legacy applications, which attempt to save user data in a read-only system location that was write able by anyone in early Windows, can still work [MR08].

Full application virtualization requires a virtualization layer. Application virtualization layers replace part of the runtime environment normally provided by the operating system. The layer intercepts system calls, for example file and registry operations in Microsoft Windows, of the virtualized applications and transparently redirects them to a virtualized location, often a single file. The application never knows that it's accessing a virtual resource instead of a physical one. Since the application is now working with one file instead of many files and registry entries spread throughout the system, it becomes easy to run the application on a different computer and previously incompatible applications can be run side-by-side. Examples of this technology for the Windows platform are BoxedApp, Cameyo, Ceedo, Evalaze, InstallFree, Citrix XenApp, Novell ZENworks Application Virtualization, Endeavors Technologies Application Jukebox, Microsoft Application Virtualization, Software Virtualization Solution, Spoon (former Xenocode), VMware ThinApp and InstallAware Virtualization.

The Microsoft application virtualization environment is demonstrated in figure 7.

Modern operating systems attempt to keep programs isolated from each other. If one program crashes, the remaining programs generally keep running. However, bugs in the operating system or applications can cause the entire system to come to a screeching halt or, at the very least, impede other operations. This is a major reason why application virtualization, running each application in a safe sandbox, has become desirable.

2.3.5 The Use of System Virtual Machines in Our Work

LgDb [AKZ11a] described in chapter 3 uses the power of system virtual machine as a platform for debugging and proofing Linux kernel modules.

In AMirror system [AMZK11] described in chapter 4 we use the power of system virtualization to replicate a running virtual machine allowing near instantaneous recovery in case of system failure.

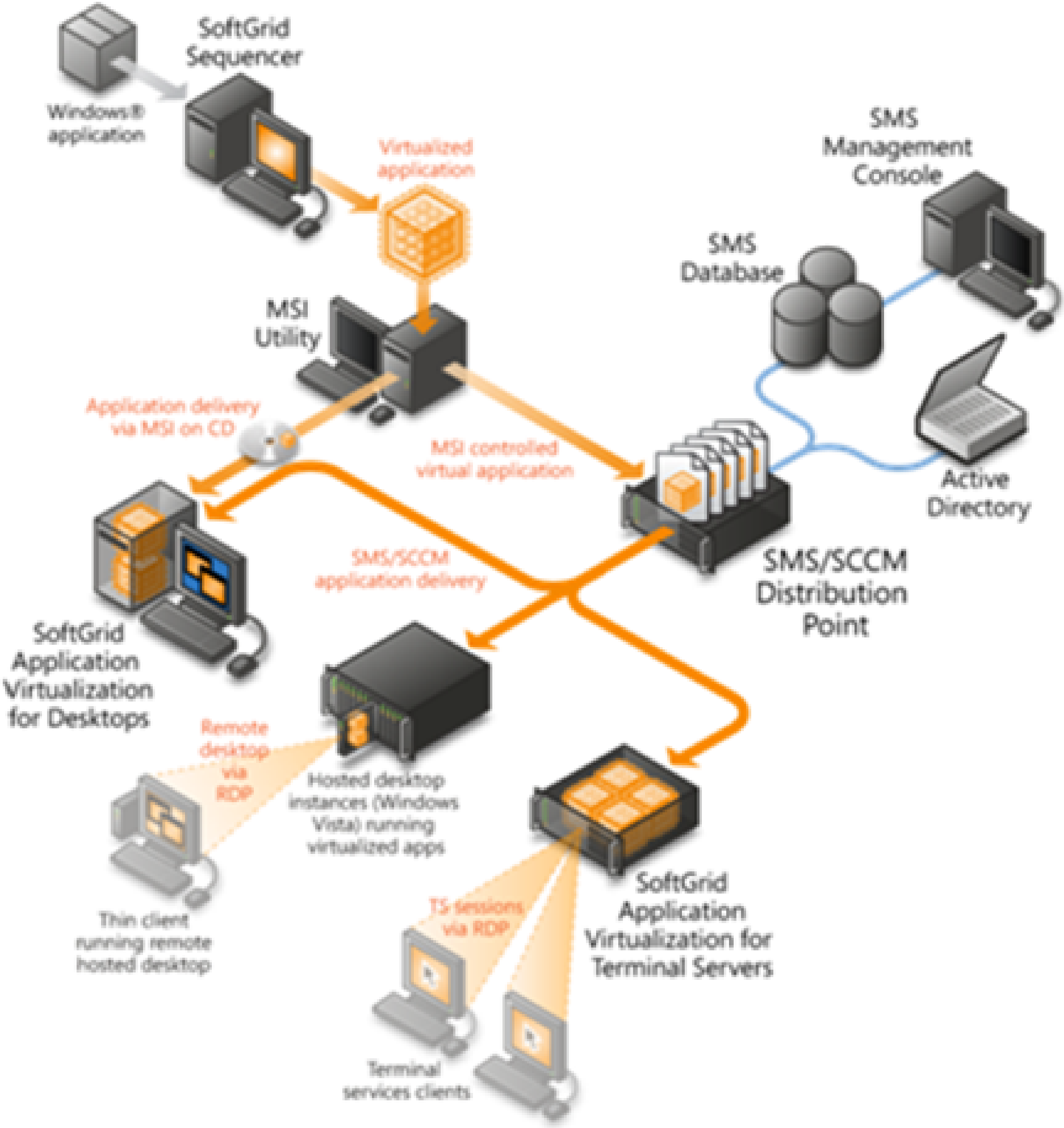

Windows® application
SoftGrid Sequencer
Virtualized application
SMS Management Console
SMS Database
MSI Utility
Application delivery via MSI on CD
MSI controlled virtual application
Active Directory
SMS/SCCM application delivery
SMS/SCCM Distribution Point
SoftGrid Application Virtualization for Desktops
Remote desktop via RDP
Hosted desktop instances (Windows Vista) running virtualized apps
Thin client running remote hosted desktop
TS sessions via RDP
SoftGrid Application Virtualization for Terminal Servers
Terminal services clients

FIGURE 7 Microsoft Application Virtualization

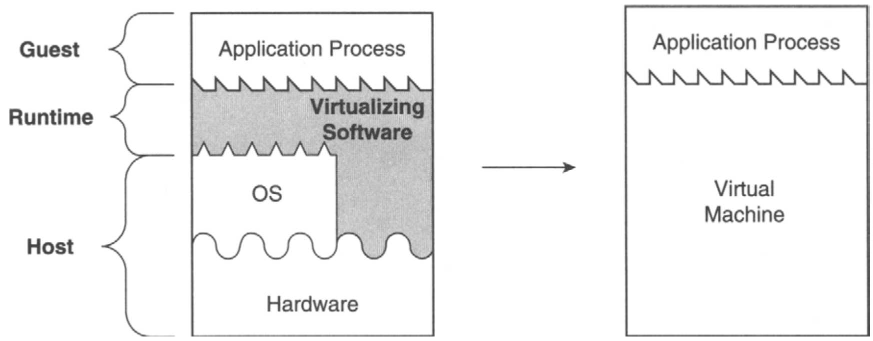


Figure 1.6 A Process Virtual Machine. *Virtualizing software translates a set of OS and user-level instructions composing one platform to another, forming a process virtual machine capable of executing programs developed for a different OS and a different ISA.*

FIGURE 8. Application virtual machines-OS and Application viewpoints

2.4 Process Virtual Machines

A process virtual machine, sometimes called an application virtual machine, runs as a normal application inside an OS and supports a single process. It is created when that process is started and destroyed when it exits.

Unlike a system virtual machine, which emulates a complete system on which we launch an OS and run applications, a process virtual machine is dealing with only one process at a time. It provides the process an ABI alternative to the OS and Hardware ABI and is designed to run processes that use the virtual ABI.

Figure 8 illustrates an application running in a native environment as the OS perceives it (a running process) and as the application running on an application virtualization environment perceives it (as an OS replacement).

Hardware independence can be achieved by running the process VM on multiple platforms and running our own software using the VM ABI. Since we use identical VM ABI across multiple platforms individual platform, API differences do not affect our application.

The process VM is an abstract computing machine. Like a real computing machine, it has an instruction set and manipulates various memory areas at run time. This type of VM has become popular with the Java programming language, which is implemented using the Java Virtual Machine (JVM).[LY99] Other examples include the .NET Framework applications, which runs on a VM called the Common Language Runtime (CLR)[Box02] and Google's Android applications, which runs on a VM called Dalvik Virtual Machine (DVM). [Ehr10]. All the above examples are "high level" virtual machines. By contrast "low level" virtual machines also exist. "Low level" virtual machines are not aware of "high level" programming language context terms such as classes and inheritance and instead focus on portability and execution acceleration. LLVM which is the most popular example of a Low level virtual machine is described in detail in appendix 4.

Figure 9 and figure 10 illustrate the popular CLR and Dalvik (Android) application virtual machines respectively.

real machine being emulated by the virtual machine, is referred to as the *native* machine. The name given to the virtualizing software depends on the type of virtual machine being implemented. In process VMs, virtualizing software is often referred to as the *runtime*, which is short for "runtime software."[2] The runtime is created to support a guest process and runs on top of an operating system. The VM supports the guest process as long as the guest process executes and terminates support when the guest process terminates.

In contrast, a *system virtual machine* provides a complete system environment. This environment can support an operating system along with its potentially many user processes. It provides a guest operating system with access to underlying hardware resources, including networking, I/O, and, on the desktop, a display and graphical user interface. The VM supports the operating system as long as the system environment is alive.

A system virtual machine is illustrated in Figure 1.7; virtualizing software is placed between the underlying hardware machine and conventional software. In this particular example, virtualizing software emulates the hardware ISA so that conventional software "sees" a different ISA than the one supported by hardware. In many system VMs the guest and host run the same ISA; however. In system VMs, the virtualizing software is often referred to as the *virtual machine monitor* (VMM), a term coined when the VM concept was first developed in the late 1960s.

[2] Throughout this book, we will use the single-word form *runtime* as a noun to describe the virtualizing runtime software in a process VM; *run time* (two words) will be used in the more generic sense: the time during which a program is running.

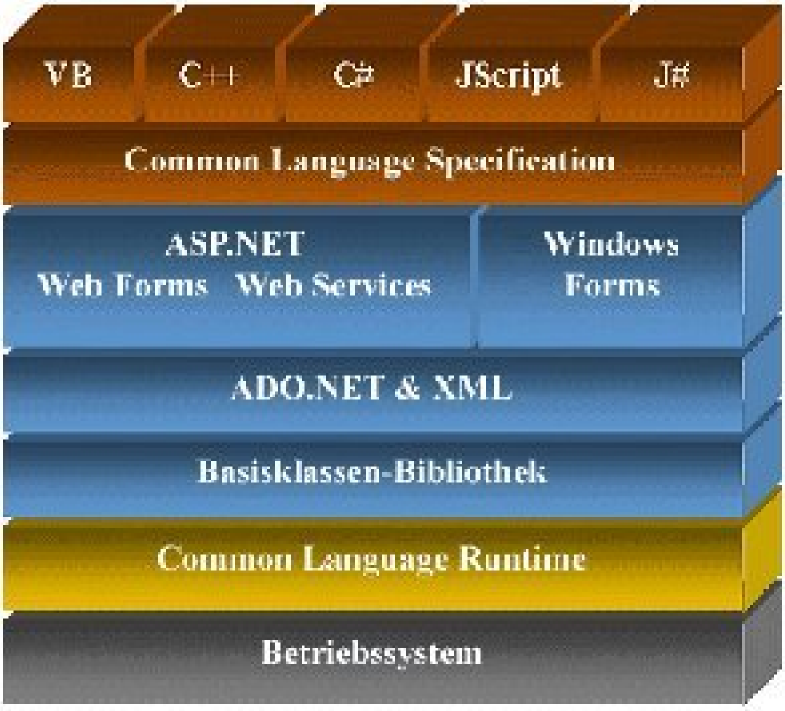


FIGURE 9 Common Language Runtime (CLR) Environment

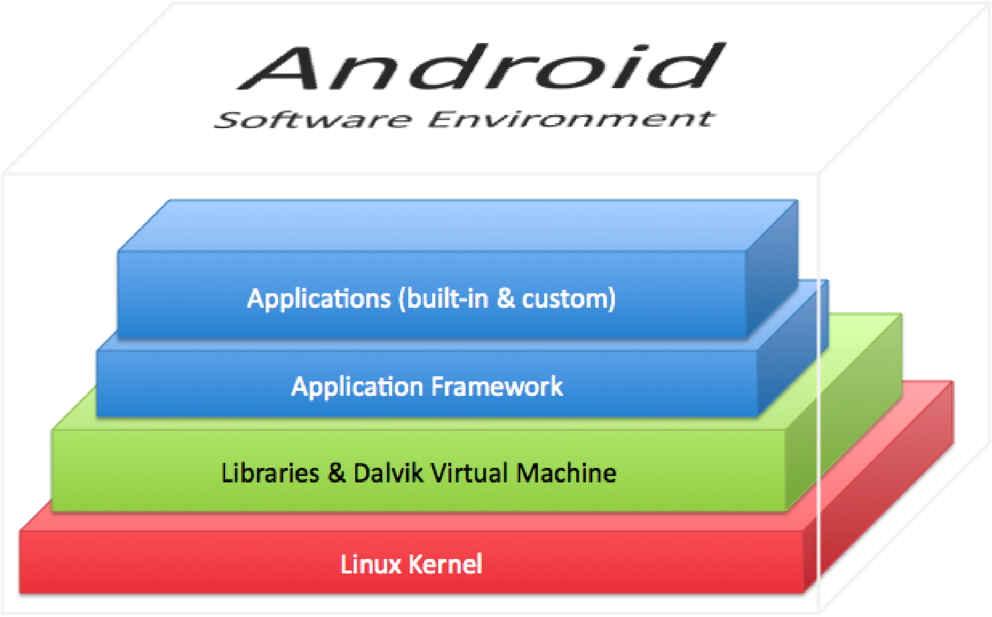

Android
Software Environment
Applications (built-in & custom)
Application Framework
Libraries & Dalvik Virtual Machine
Linux Kernel

FIGURE 10 Android (Dalvik) Environment

2.4.1 The Use of Process Virtual Machines in Our Work

Truly-protect[AKZ11b] is built on top of two "low-level" virtual machines. LLVM-Prefetch(Described in chapter 6) is built on top of LLVM. Both LLVM-Prefetch and Truly-Protect rely on the features of process virtual machines to operate in the background in order to perform the task they were designed for.

2.5 Storage Virtualization

Storage virtualization is both a concept and term used within the realm of computer science. Specifically, storage systems may use virtualization concepts as a tool to enable better functionality and more advanced features within the storage system. For example using Logical Volume Manager (LVM) allows the users to treat multiple storage devices as a single logical entity without regard to the hierarchy of physical media that may be involved or that may change. The LVM enables the applications to read from and write to a single pool of storage rather than individual disks, tapes and optical devices. Also called "file virtualization," it allows excess capacity on a server to be shared, and it can make migration from one file server to another transparent to the applications.

Another commonly used feature of storage synchronization is information life management. Storage replication allows the users to decide on which physical component (new/old) to record the data and how many copies to keep based on the properties of the data.

When a file is moved off the hard disk, it is replaced with a small stub file that indicates where the backup file is located.

2.5.1 The Use of Storage Virtualization in Our Work

The AMirror System[AMZK11] described in chapter 4 uses the power of storage virtualization for disaster recovery purposes.

2.6 Network Virtualization

There are several cases were it is preferable to use virtual rather then physical network resources. One of the most widely used example is IP-based load balancing such as LVS, The Linux Virtual Server[ZZ03]. Linux Virtual Server (LVS) is an advanced load balancing solution for Linux systems. LVS is an open source project started by Wensong Zhang in May 1998. The mission of the project is to build a high-performance and highly available server for Linux using clustering technology. The LVS cluster is industry grade proven solution that provides good scalability, reliability and serviceability.

In LVS scenario multiple computers, all running Linux, are active in a clus-

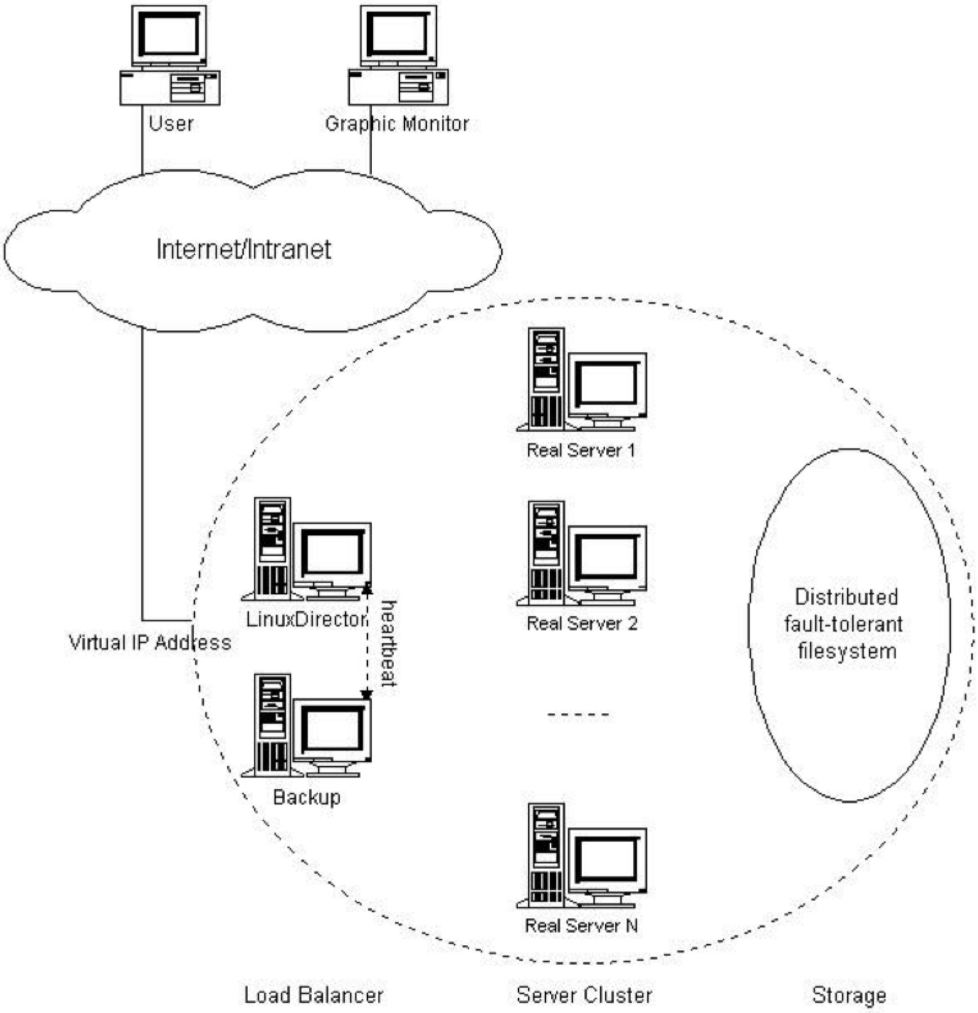


FIGURE 11 Using VIP via LVS.

ter environment. The owner of the cluster would likely use the cluster as a single computer were the load is shared between the computers in the cluster. Furthermore, should one or more of the computers in the cluster stop functioning, while other remain operational there may be a drop in the cluster capacity to serve but the cluster should still be able to serve all clients.

In the case the cluster owner uses LVS, a new VIP, Virtual IP is created for the cluster. The LVS software is a load balancing request sent to the Virtual IP across all active servers in the LVS cluster via the IPVS service (IP Virtual Server). IPVS is major part of LVS software. IPVS is implemented inside the Linux Kernel.

Using VIP for clustering via LVS is illustrated in figure 11.

Another common use for network virtualization is creating VPNs (Virtual Private Network). In this case, we would like to share resources and servers with remote users without the burden of security authentication.

By creating a secure tunnel between the local site and remote site, we can create Virtual Private Network. In the VPN environment remote computers look like local computers to our security software. The VPN environment allows ac-

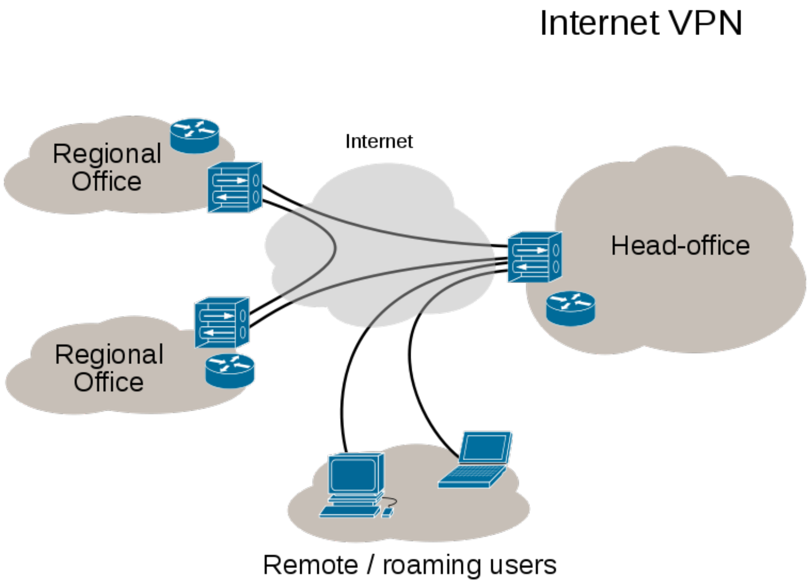


FIGURE 12 Virtual Private Network (VPN).

cess to file and resource sharing with remote computers. VPN is illustrated in figure 12.

In each of the cases described above as well as in other cases of virtual private network. We use an abstract interface to simulate a network component and perform additional functions for the user.

2.6.1 Use of Network Virtualization in our solutions

PPPC, The Peer-2-Peer streaming system that we have developed uses virtual multicast IP to send the stream transparently throughout the network. The virtual multicast provide the abstraction layer for application layer allowing us to hide the actual peer-2-peer streaming.

2.7 Cloud Virtualization

Cloud computing refers to resources (and shared resources), software, and computation power that are provided to computers and other devices as a utility over a The Internet.

Cloud computing is often modeled after a utility network such as a power network, sewage network and other services where resources are subscribed to and paid for based on the user consumption.

As demonstrated in figure 13 cloud computing is often characterized as either

- Software as a Service

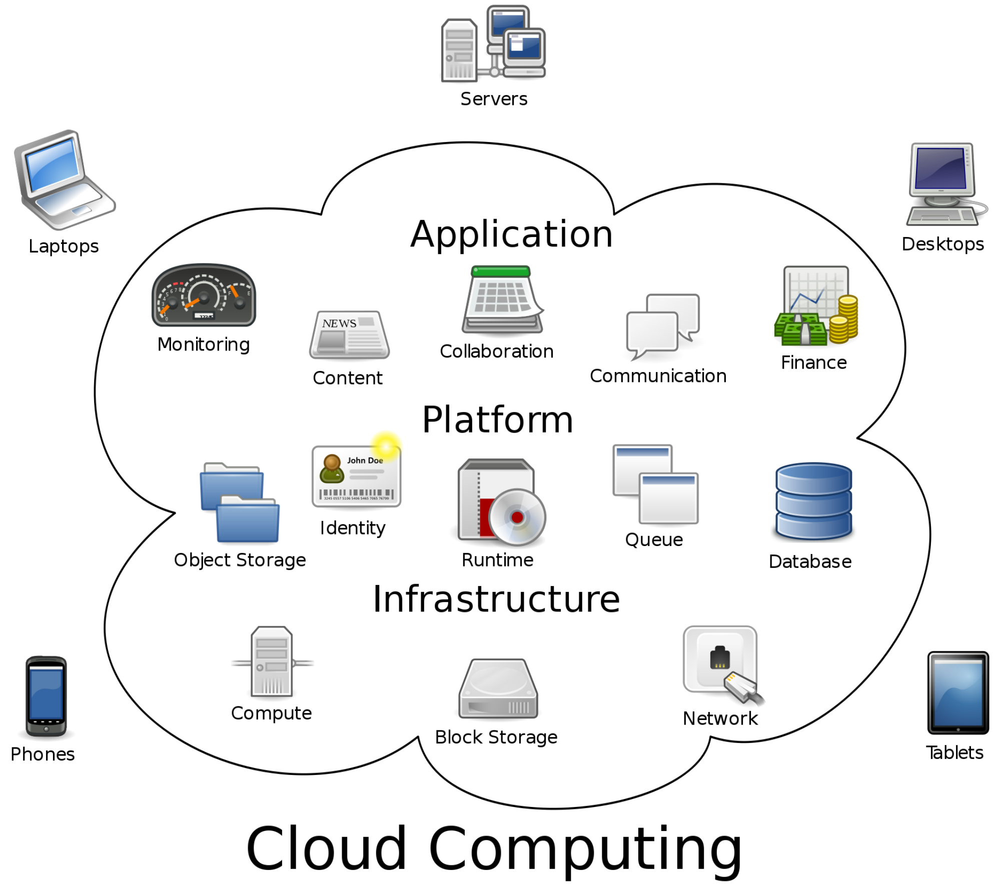


FIGURE 13 Cloud Computing Environments

– Infrastructure as a Service
– Platform as a Service

2.7.1 Software as a Service

Software as a Service (SaaS) is a software delivery model in which software and its associated data are hosted over the Internet and are typically accessed by clients using a web browser.

SaaS has become a common delivery model for most business applications, including accounting, collaboration, customer relationship management (CRM), enterprise resource planning (ERP), invoicing, human resource management (HRM), content management (CM) and service desk management. SaaS has been incorporated into the strategy of mainly all leading enterprise software companies.

2.7.2 Platform as a Service

Platform as a service (PaaS) is the delivery of a computing platform and solution stack as a service.

PaaS offerings facilitate deployment of applications without the cost and complexity of buying and managing the underlying hardware and software. Furthermore, the PaaS environment also handles provisioning hosting capabilities, providing all of the facilities required to support the complete life cycle of building and delivering web applications and services entirely available from the Internet.

PaaS offerings may include facilities for application design, application development, testing and deployment. Paas may also offer services such as hosting or application services such as team collaboration, web service integration and marshalling, database integration, security, scalability, storage persistence, state management, application versioning, application instrumentation, developer community facilitation and disaster recovery.

The peer-2-peer platform described in chapter 7 is an example of a PaaS module. Future versions of the AMirror system, described in chapter 4 that may offer a central mirroring site for multiple users will also provide PaaS.

2.7.3 Infrastructure as a Service

Infrastructure as a service (IaaS) is a method to deliver computer infrastructure, typically a platform virtualization environment, as a service. The infrastructure is provided along with raw (block) storage and networking. This infrastructure can later be used by the user to install a complete computation environment with operating system software and data.

In an IaaS paradigm, clients don't purchase servers, software, data center space or network equipment. Instead clients buy those resources as a fully outsourced service. Suppliers typically bill such services on a utility computing basis; the amount of resources consumed (and therefore the cost) will typically reflect the level of activity or capacity.

3 KERNEL DEBUGGER

3.1 Introduction

In this chapter, we present the LgDb, a system for performance measurement. Unlike many commercial system LgDb can be used for kernel module profiling. Furthermore, LgDb can be used to ensure that all the code is executed during testing for decision coverage testing. LgDb is built on top of Lguest, which is a Linux-on-Linux hypervisor while being a part of the Linux kernel.

3.1.1 Development Tools

Modern user space developers enjoy a variety of tools that assist their developments while shortening the development cycle time. In this chapter we focus on code coverage and profilers.

The *Code coverage* concept was introduced in [MM63]. fifty five percent of the developers use code coverage tools as stated in a recent report [SW] . Automatic tools such as Insure++ [Para] and Jtest [Parb] allow userspace developers and testers to ensure that all the written code is indeed covered by some testing suite. By guaranteeing that all the code was tested including code for exception and error handling, software developers can eliminate dead code. Furthermore, by testing all the exceptional cases, we guarantee that the system operates in a predictable sense even when an exception occurs.

The *Profiling* concept is a subject of an on going active research [SE94]. Solutions such as Rational quantify [IBM] and gprof [GKM] provide developers with insight about code parts where the CPU spends most of its executing time. The basic assumption is that almost every written piece of code can be optimized. Therefore, by employing *Profilers*, developers can employ their time more efficiently by optimizing only the parts that make a significant effect on the total run-time.

3.1.2 Lguest

Lguest [Rus] is a simplified Linux-on-Linux para-virtualization hypervisor that allows to run multiple Linux kernels on top of a system that runs Linux kernel. The system, which runs the Lguest hypervisor, is called "host" throughout this chapter.

Each kernel, which runs on Lguest is running on virtual hardware simulated by the host. Otherwise it is identical to the host kernel. Each system hosted on the host is called "guest" throughout this chapter.

Lguest was integrated into the Linux kernel and it meant to serve as a reference and educational hypervisor. Lguest supports only x86 environments.

3.1.3 State-of-the-Art Solutions

There are plenty of industrial and open source solutions for code coverage and profiling. However, practically all of them deal specifically with user space development.

Currently there is no industrial solution for profiling and testing coverage for Linux kernel modules. Similar solutions in terms of architecture and services provided include:

Valgrind [NS03] is an open source project. Valgrind allows to run a user space code on a virtual environment. In addition, it allows many tools such as memcheck and helgrind to run for code quality proofing. The Valgrind approach is very similar to ours in the sense that the code runs in a virtual environment except that it only works for user space programs.

Oprofile [Opr] is an open source profiling environment that can be used for kernel code. Oprofile is currently in alpha status. It is a statistical tool and therefore uses a very different approach from our tool. Its immediate advantage over our solution is that it supports wider range of architectures.

Gcov [gcc] is part of the Linux Testing Project[LTP]. It is integrated into the gcc compiler and enable checking coverage of both kernel space and user space code. Gcov approach differs from LgDb as Gcov is a compiler based tool and not a hypervisor based. LgDb also has different functionality since gcov does not simulate failure at given points. This leaves the developer with the task of simulating different cases and leaves the developer with The remaining code segments that would rarely on never be covered in normal testing. However, gcov can be used to detect dead code (segments that cannot occur due to conflicting conditions) which LgDb cannot detect.

3.2 Motivation

3.2.1 Code Coverage in the Kernel Space

One basic coverage criteria is the *decision coverage* [Tea]. It verifies that each "edge" in the program was tested (for example, each branch of an IF statement). A result of this criteria is the *path coverage* [MRW06] criteria, which verifies that every possible route through a given part of the code is executed.

The existing coverage tools require the compilation of the code with some special libraries while simulating all cases where a request may fail. These tools do what they are expected to do but fail to simulate across all paths. Some paths cannot be tested in most real life scenarios due to the involved complexity in their simulations. For example, it may be very difficult to cause a certain kmalloc to fail with a certain combination of locked mutexes and other running threads.

Since simulation of all cases usually cannot be achieved within reasonable effort, we are bound to encounter cases where some code cannot be tested for coverage. For example, where we assume the program checks for return values of some hardware operations indicating hardware failure or insufficient resources. In order to simulate these conditions we would need to simulate hardware failure or use all the system resources on specific time. Simulating these conditions may be difficult.

3.2.2 Profiling in Kernel Space

Our goal is to build an instrumentation based profiler that is not statistically based like Oprofile. Construction of an instrumentation based profiler is motivated by the fact that it is more accurate than a statistical profiler while providing more capabilities such as generating call graphs. Because a large portion of the kernel's functions are short such as handlers for instance, there is a fair chance that they will not be executed during the occurrence of a sample event. Oprofile's user manual states this openly:"...sometimes a backtrace sample is truncated, or even partially wrong. Bear this in mind when examining results" [Opr11]. Increasing the samples rate will produce more accurate results while slowing the system.

Unfortunately, without virtualization, building an instrumentation based profiler for the kernel is not a straightforward task.

When profiling a userspace process we use another process for analyzing the "trace". On the other hand, tracing the supervisor with a user process is impossible.

A process that traces the supervisor can only trace the system while the process is in a running state. Since the OS needs to be traced even when the process is not running (in some cases, especially when the process is not running) we cannot use the user space method in the kernel.

Another possible approach is to allow the supervisor to trace itself. This can be done by using various type of messages and shared memory or by using performance counters. The messages can be triggered using code injections all over the kernel.

Allowing the supervisor to trace itself affects the supervisor run and naturally upsets the inspected module run. Furthermore, if multiple threads are running, we have to deal with locking for the profiling which is likely to affect the inspected process as well.

We now present the LgDb and show how it can overcome all of the above difficulties by using virtualization.

3.3 LgDb: The "Virtual" Kernel Debugger

Although *LgDb* stands for Lguest debugger it is not a standard debugger. Although kernel debuggers exist none of them are integrated into the Linux kernel. The reason for this is rather philosophical: kernel development should not be easy [Lin]. Since we agree with this approach, our goal is to move kernel code development to a virtualized environment. This will not make it easier but it will increase the kernel's reliability (code coverage) and efficiency (profiling). LgDb is a proof of concept tool that achieves these objectives.

3.3.1 Architecture

LgDb is based on the Lguest hypervisor. We chose Lguest because Lguest has a simple source code that enables us to extend its hypercalls relatively easy. Lguest is a simple hypervisor that is kept on the kernel for pedagogical purposes. We describe Lguest internal structure in detail in Appendix 6.

Even though LgDb is based on Lguest, LgDb's architecture is simple and can be easily ported to more industrial hypervisors such as Xen, KVM and VMWare.

Running on top of Lguest limits us to supporting only Intel 32bit architecture but has the benefit of providing simple, easy to understand and modify architecture that is stable. Furthermore, unlike industrial hypervisor, Lguest code base is stable and changes are infrequent which eases our development process.

The host kernel, which runs LgDb, has a modified Lguest kernel module. The host kernel is running at privilege level 0, which is the privilege level of the OS in x86 platforms. Thus, the host kernel has full privilege permissions. Furthermore, as per x86 architecture, no other kernel, specifically guest kernels, have full privilege permissions.

The guest kernel is launched using a userspace application called Launcher. From the host's point of view, the Launcher *is* a guest. Every time that the Launcher's process is in "running" state it causes the hypervisor to *switch* to the guest. The switch operation performs a context switch and switches all the registers with the saved guest state. When the launcher is preempted, the guest state (CPU registers) is saved in memory and the host state is restored.

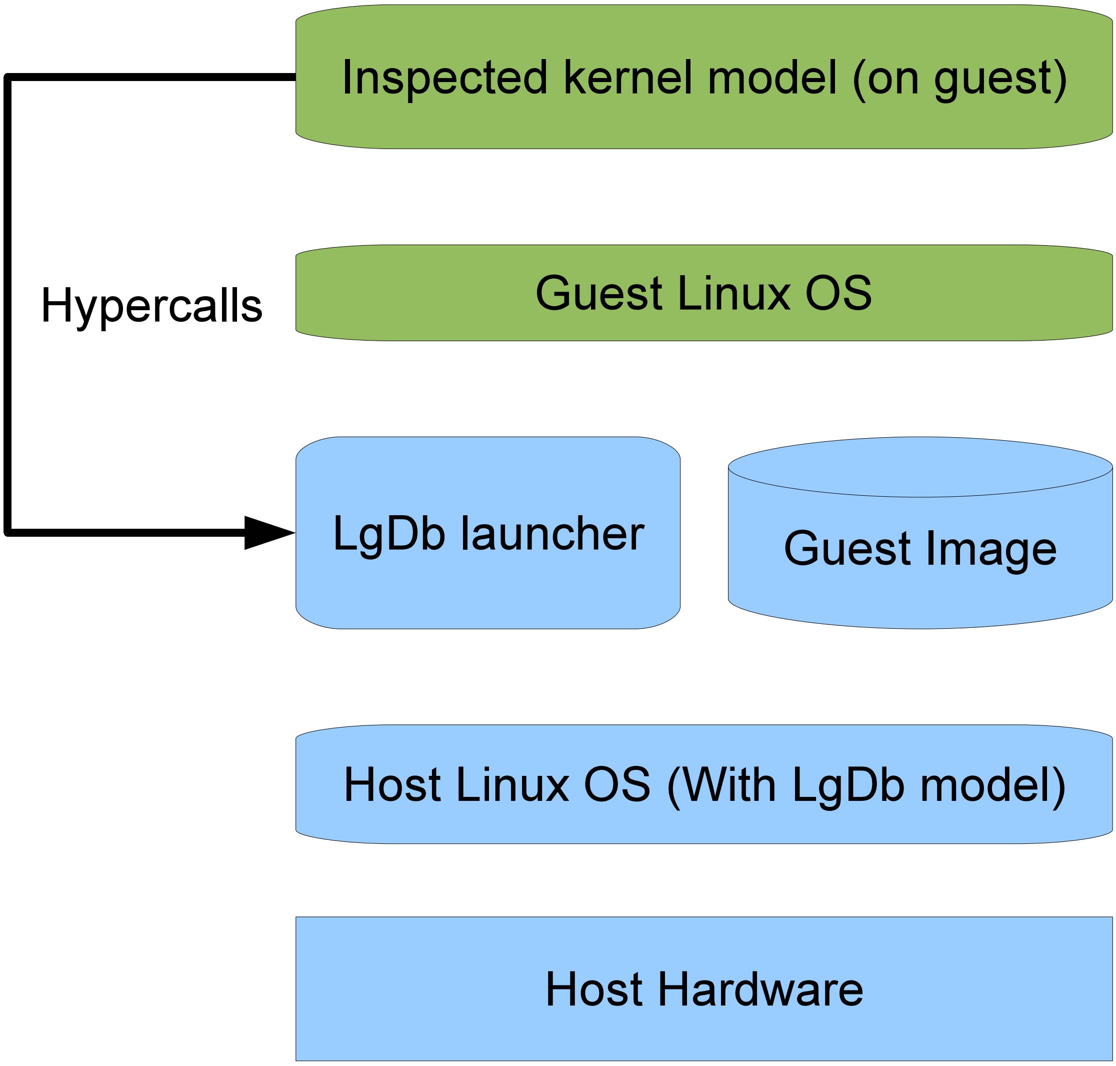

Inspected kernel model (on guest)
Hypercalls
Guest Linux OS
LgDb launcher
Guest Image
Host Linux OS (With LgDb model)
Host Hardware

FIGURE 14 The LgDb System Architecture

In Lguest case, the guest runs at the x86 privilege level 1. A process at privilege level 1 has limited permissions (compared to the OS which runs at privilege level 0). For example a process at privilege level 1 cannot access the hardware. When the the guest kernel needs to perform a privileged instruction (such as hardware access) it generates *Hypercall*. A hypercall is typically a request from the guest to the host instructing the host to perform a privileged operation.

Thus, the hypercall provides a mechanism for a guest to host communication. LgDb is extending the hypercall mechanism to provide profiling and code coverage information.

LgDb capitalizes on the hypercall mechanism and implements new hypercalls to provide code coverage and profiling. Thus, LgDb architecture requires that an API will be add to generate the hypercall in the inspected kernel code.

By running the inspected kernel model on the guest and sending hypercalls we can enlist the host to assist in the inspection of the guest.

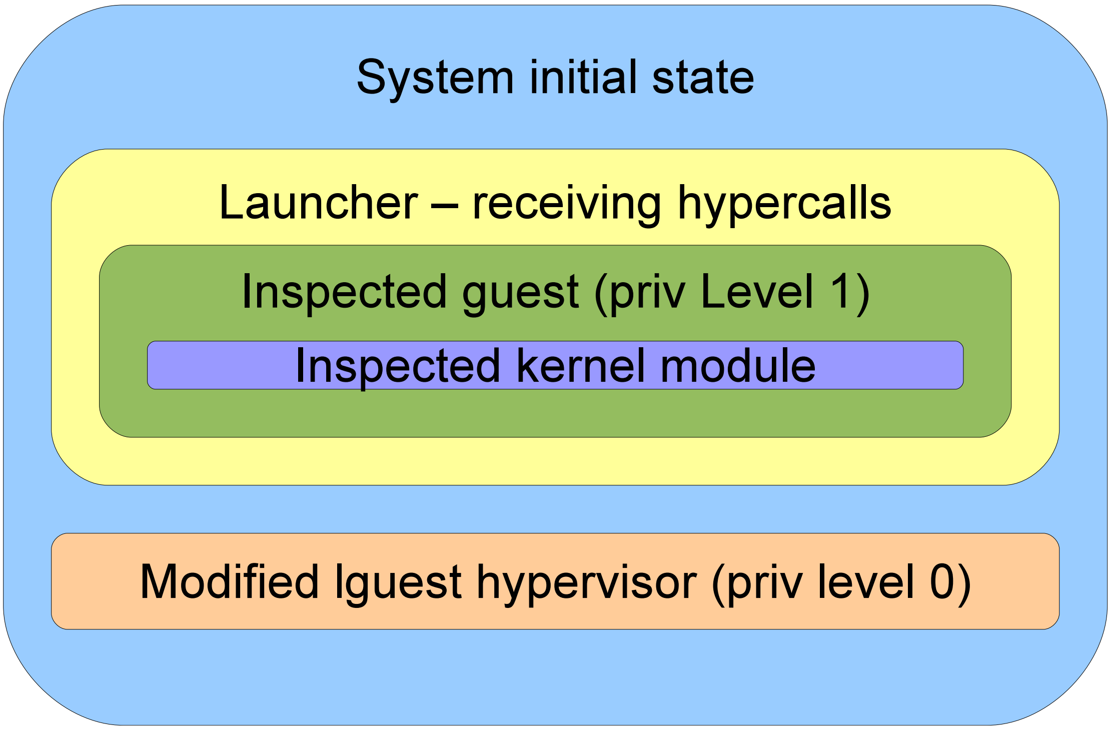

System initial state
Launcher – receiving hypercalls
Inspected guest (priv Level 1)
Inspected kernel module
Modified lguest hypervisor (priv level 0)

FIGURE 15 The LgDb System - x86 Privileges Levels

3.3.2 Forking Guests

In order to support the code coverage tool we implemented a "forking" for the Lguest guest. By forking we mean that at some point (when the hypercall is received) a new identical guest is created. Both guests continue to run from the forking point. The "parent" guest waits until the "child" finishes.

Forking a guest is quite simple. The inspected guest image is a segment in the launcher's memory. Forking the launcher process, with additional actions like cloning I/O threads (called "virtqueues" in the Lguest world), creates an exact copy of the inspected machine. When a hypercall requesting fork is received at the host, the fork(2) system call is being invoked. At this point, the original launcher process goes to sleep. As both guests share the same code, there is a need to indicate to each guest which path to choose (we want the "child" guest will go at a different path from the "parent" guest). This is done by using the hypercall mechanism to return a unique value for each guest (at this case, a boolean value is sufficient). The forking procedure does not end at user space as described. The host kernel holds for each guest (i.e for each launcher process) its virtual hardware state information. The information the host saves for each guest includes the guest virtual CPU registers (instruction pointer, stack pointer) values, interrupt and page tables and the kernel stack. When calling fork(2) in a launcher process, all the guest relative data should be cloned in the Lguest kernel module and assigned to the new launcher process.

When the "child" guest is done, the "parent" guest resumes execution from the hypercall location (as both its memory image and kernel information were not changed during the "child" execution), but with a different return value allowing it to follow a different execution path. This is analogous to the POSIX command fork(2).

Forking guests for different purposes was previously discussed in [LWS+09] for different purposes and not for kernel based hypervisors.

3.3.3 Kernel Multitasking and Profiling

In section 3.2.2, we discussed multi-processing and claimed that if the profiling is done by the kernel on itself we will require locking that will affect the traced process. Since Lguest only runs on single launcher process then hypercalls are always generated in a serial fashion and no such locking is done or required by LgDb.

In order to support kernel profiling in a multitasking environment, LgDb needs to support multiple threads entering the same code segment or charging the same wallet across multiple kernel threads which it does.

However, there are two known issues with LgDb behavior in a multitasking environment. Both issues occur because the hypervisor is not aware which kernel thread is currently executing on the guest or even which kernel threads exist in the guest.

When profiling kernel code, time is charged to the process when we charge a certain wallet regardless of the process state. If the process is "ready", "waiting for I/O" or otherwise not executing then it is arguable that we should not charge its wallet. Fixing these issues requires generating messages to the hypervisor from the scheduler and from the elevator. We did not implement the support for this feature yet.

Another known issue is when a kernel module is being forked. In this case, we need to be aware of the two processes and charge accordingly. The current version of LgDb does not handle this case.

3.4 Tools

3.4.1 Code Coverage

While running the following code (taken from [SBP])

```
int init_module(void)
{
    Major = register_chrdev
        (0, DEVICE_NAME, &fops);
    if (Major < 0) {
        printk(KERN_ALERT
        "Registering char device

        failed with %d\n", Major);
        return Major;
    }
    // more code follows
}
```

We expect register_chrdev to succeed most of the time. However, test procedure dictates that the "if" clause must also be executed during testing and its result is monitored. How can we simulate the case when register_chrdev fails?

In order to solve such a case, we created a set of hypercalls to support a full code coverage testing in the kernel module. When the hypercall is received in the hypervisor, the hypervisor forks the running VM. The hypervisor first runs the VM code with the hypercall failing to perform all code that should handle the failed case.

The guest keeps running while generating logs that can be inspected by the developer until a second hypercall is called to indicate that all the logs, which are relevant to the case of the failed call, have been generated. The hypervisor now terminates the guest VM (where the call failed) and resumes running the other VM where the code runs normally.

Upon receiving the 3rd hypercall, which indicates that all code coverage tests are complete or at any time during the running of the program, the user can inspect the hypervisor log and find a detailed output of all the successful and all the failed runs of the module.

Algorithm 1 Code Coverage Logic

```
Run Lguest hypervisor inside host Linux kernel
Run Guest in supervisor mode
while Coverage test not completed do
  Guest generates a hypercall when reaching critical function
  Hypervisor forks the running process and runs the failing process first
  Guest checks if it should enter the successful or the failing branch code
  if Guest fails in the branching point then
    Run all relevant code to failing
    Inform the hypervisor that child has to be terminated
  else
    run everything normally
  end if
end while
```

The involved hypercalls are defined in the <lguestcov.h> header and includes

```
#define testCovStart \\
    (function, failval, X...) \\
    (lguest_cov_fork() ? \\
    failval : function(X))
```

```
#define testCovEnd() \\
     lguest_cov_end()

#define testCovFinish() \\
     lguest_cov_fin()
```

By using the coverage functions in <lguestcov.h>, we can change the original code example to the following and ensures that register_chrdev is checked in successful and failing cases.

```
int init_module(void)
{
    Major = testCovStart(register_chrdev,-1,
        0 , DEVICE_NAME, &fops);
    if (Major < 0) {
        printk(KERN_ALERT
        "Registering char device
        failed with %d\n", Major);
        testCovEnd();
        return Major;
    }

    // more code follows
}

int clean_module(void)
{
// standard termination routine.
    testCovFinish();
}
```

3.4.2 Profiling

LgDb approach to profiling is based on "wallets" similar to the TAU system described in [SM06] but works in kernel space.

When profiling, we are interested in code sections where the majority of the processing time is spent on. We use a profiler in order to detect congestion, or find certain code segments to optimize.

The profiler is implemented using a hypercall that issue a report whenever we enter a point of interest. This can be either a function or loop or any other part of the program. In the hypervisor, we trace the time spent in each of those parts using a "wallet". When we call our hypercall, we start charging time to a certain wallet. When hypercall is called again, we stop charging time for this wallet. Last, we can provide the user with trace of hypercalls that are typically running when charging specific account.

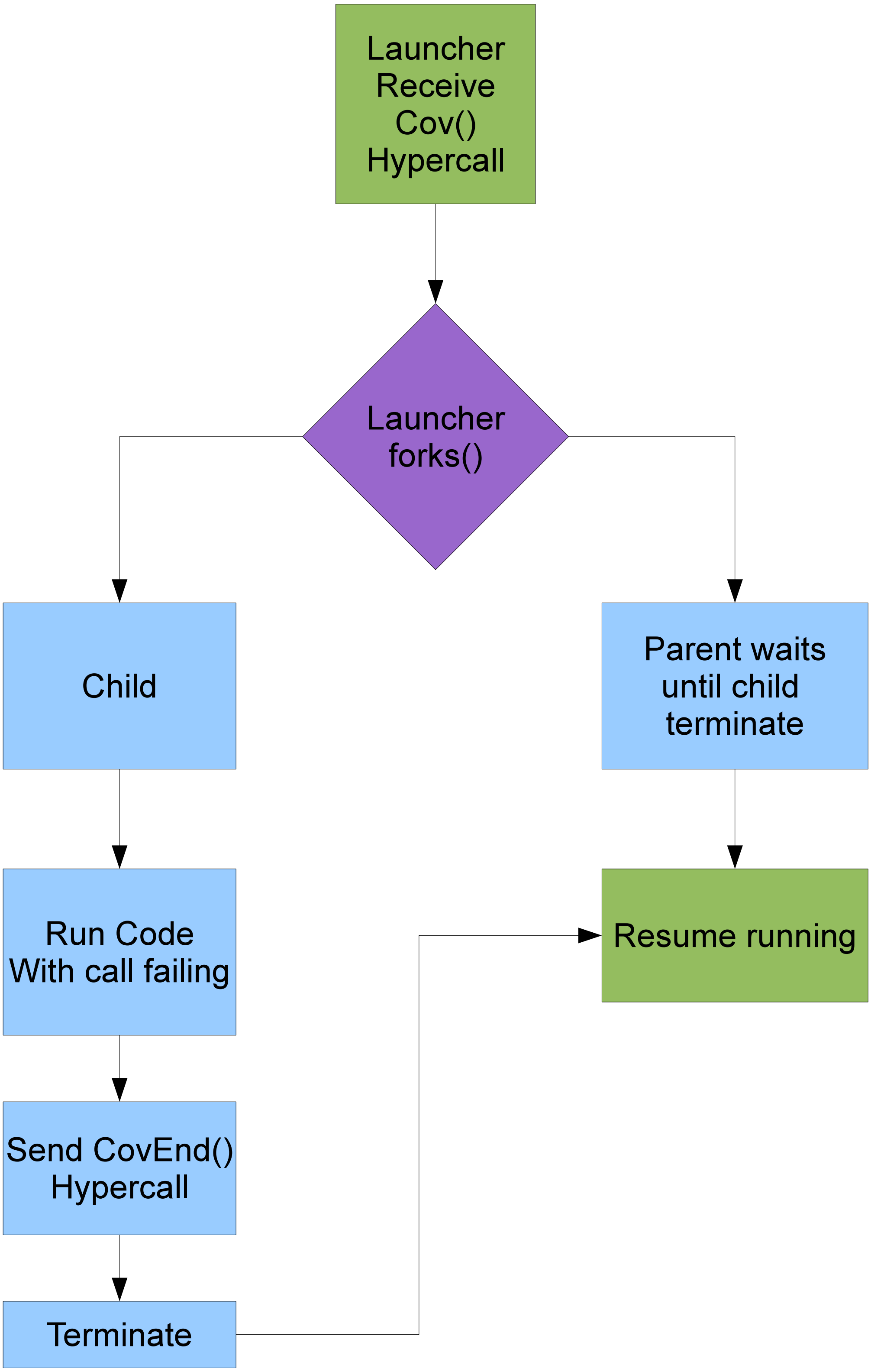

Launcher Receive Cov() Hypercall
Launcher forks()
Child
Parent waits until child terminate
Run Code With call failing
Resume running
Send CovEnd() Hypercall
Terminate

FIGURE 16 Flow Chart of Code Coverage Operation

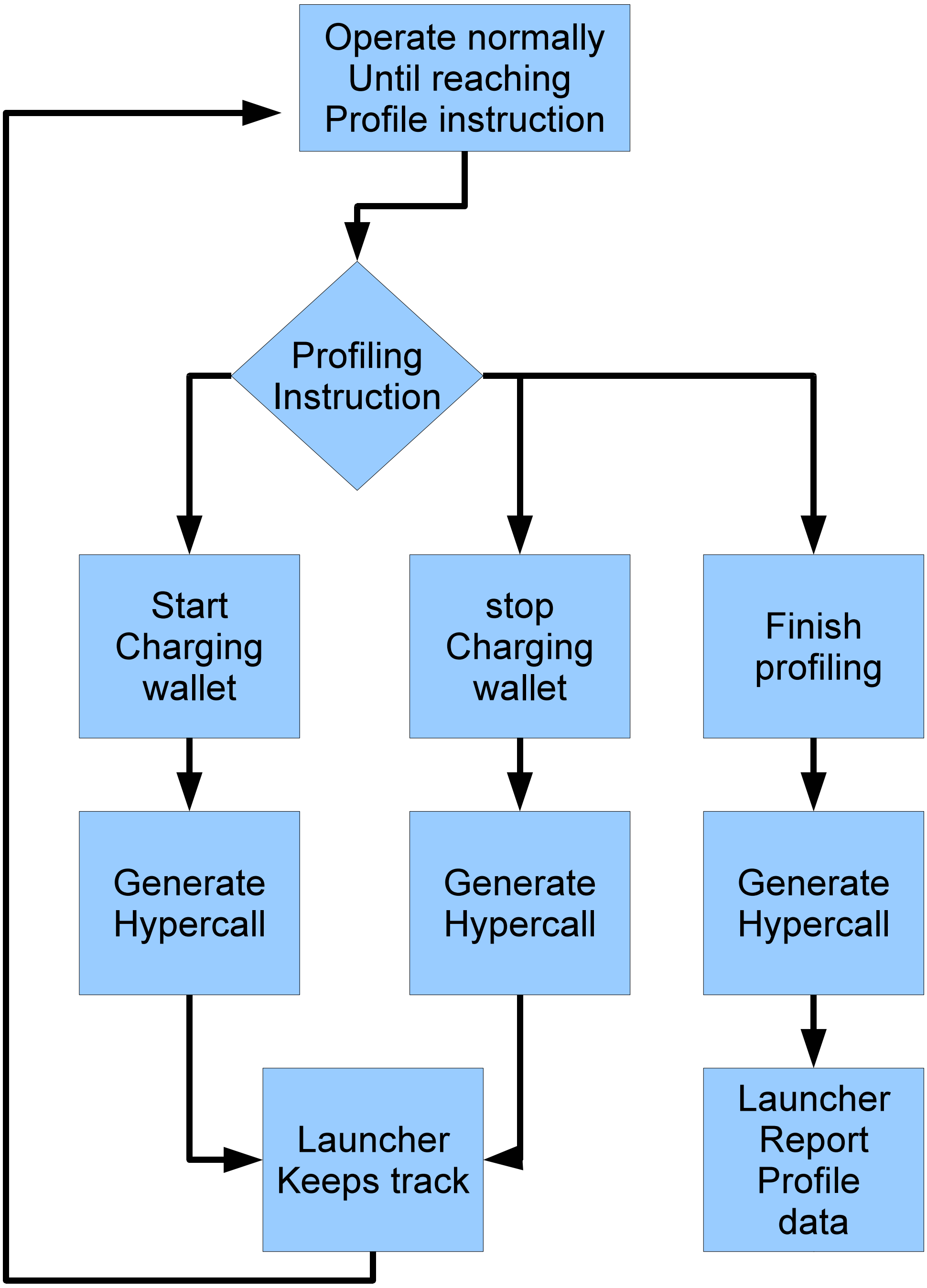


FIGURE 17 Flow Chart of Profile Operation

Algorithm 2 Profiler Algorithm

```
  hypervisor keeps "wallets" to charge processing time to
  When reaching interesting function, guest uses hypercall to report
  Hypervisor charges time while guest is running
  Guest sends hypercall when leaving a function
```

In order to use the LgDb profile routines, we provide the following API

```
// This function starts charging
void lguest_charge_wallet(int wallet);

// This function stop charging time
void lguest_stop__wallet(int wallet);
```

For example:

```
#define NETLINK 1
#define SCOPE_A 2
#define SCOPE_B 3

int func_a()
{
    lguest_charge_wallet(SCOPE_A);
// code...
    lguest_stop_wallet(SCOPE_A);
}

int funct_b()
{
    lguest_charge_wallet(SCOPE_B);
// code
    lguest_charge_wallet(NETLINK);
// code
    lguest_stop_wallet(NETLINK);
// code
    lguest_stop__wallet(SCOPE_B);
}

static int msg_cb
    (struct sk_buff *skb,
    struct genl_info *info)
{
    lguest_charge_wallet(NETLINK);
    // code
    lguest_stop_wallet(NETLINK);
}
```

3.5 Related work

There are many user space programs that offer profiling and code coverage for user space code. Some were mentioned in the introduction. In this section, we will describe some of the virtualized testing tools for kernel developers and compare LgDb with Oprofile and gcov/lcov that offer similar services in the Linux kernel.

Oprofile and lcov are now officially a part of the Linux kernel though they are considered "experimental". None of the virtualization environments for kernel debugging have been accepted as an official part of the kernel.

3.5.1 Debugging in virtual environment

LgDb is unique in the sense that it offers a complete framework for testing and debugging kernel applications. However, the notion of using an hypervisor to assist in the development is not new.

Using Xen[BDF$^+$03] for Linux kernel development where an unmodified kernel runs as guest on the hypervisor is suggested by [KNM06]. While [KNM06] work shares the same software architecture (inspected guest running on hypervisor) it does not offer profiling and code coverage tools as offered by LgDb.

Using Oprofile on a KVM [Kiv07] hypervisor with the QEMU [Bel05] launcher is discussed in [DSZ10]. Our system differs in the sense that it actually uses the hypervisor as a part in the profiling environment allowing only inspected kernel module measurements. XenOprof [MST$^+$] is an open source similar tool that runs Oprofile but uses Xen instead of KVM. Its internal components are described in [MST$^+$05].

VTSS++ [BBM09] is a similar tool for profiling code under VM. It differs from all the above tools in the sense that it does not work with Oprofile but requires a similar tool to run on the guest.

VMWare vmkperf[inc] is another hypervisor based performance monitoring tool for VMWare ESX Server. Aside from the obvious difference between VMWare ESX server (Type I hypervisor) and Lguest (Type 2 hypervisor), it is not programmable and cannot measure specific code segment system usage and therefore it is not related to our work.

3.5.2 LgDb vs. Gcov

LgDb enjoys a few of advantages over gcov. Firstly, gcov does not simulate failure, therefore, critical paths may not be covered. In this matter, LgDb could be used to validate the system's stability at all possible scenarios. Second, gcov is a compiler specific tool when LgDb is (by concept) platform independent. Third, LgDb allows for this ability to follow a specific path and therefore does not overflow the user with unnecessary information.

LgDb will probably not be useful for testing a driver code as the inspected kernel uses virtual I/O devices and special drivers for them.

3.5.3 LgDb vs. Oprofile

LgDb enjoys a few architecture advantages over Oprofile. First, LgDb consumes significantly less resources than Oprofile. In LgDb, the only source for profiling overhead is associated with calling an hypercall.

Calling hyper call takes approximately 120 machine instructions for the context switch between guest and host. Each profiling request contains 2 hypercalls (one hypercall to start profiling and one hypercall to stop profiling) therefore each profiling request should contain around 240 instructions. On the other hand, Oprofile has overhead for each sample taken for handling the none maskable interrupt. It also has overhead for the work of the Oprofile's daemon. I.e. the user space to analysis of the profiling data.

Second, LgDb is non-intrusive for kernel components that are not being profiled.

Oprofile profiles the entire system and much of the NMI samples will probably take place in the context of processes and modules which are not being inspect. Thirdly, LgDb can merge calls from different functions into one wallet such as network operations or encryption algorithms. Fourthly, it allows to profile code segments that are smaller than a function.

Lastly, but most important, Oprofile is statistical based and thus it is less reliable than LgDb.

Oprofile's advantage is that any part of the kernel can be profiled. As stated on the Gcov comparison, the profiling device driver code becomes impossible with LgDb. Furthermore, Oprofile requires no changes to the inspected code itself.

3.5.4 LgDb vs. LTTNG 2.0 and FTrace

The LTTng [Des] project aims at providing highly efficient tracing tools for Linux. Its tracers help tracking down performance issues and debugging problems involving multiple concurrent processes and threads. Tracing across multiple systems is also possible.

FTrace [Ros] is an internal tracer designed to help developers and designers of systems find what is going on inside the kernel. FTrace can be used for debugging or analyzing latencies and performance issues that take place outside user-space.

LTTNG works by patching the kernel and by using kernel features such as FTrace. LTTNG does not use virtualization. LTTNG requires a LTTNG patched kernel and only works on a certain version.

By comparing LgDb to LTTNG we can profile parts of the kernel that were prepared for LTTNG. LTTNG can also debug parts of the driver code that LgDb cannot. From FTrace we can automatically get a lot more information than we can automatically get from LgDb. It will require a lot of effort in FTrace. Furthermore, FTrace will always require code injections. Future version of LgDb, which will use a kernel debugger, will not require code injection and can be completely automated.

3.5.5 LgDb vs. Performance Counters

Performance Counters [GM] are special registers that can be used to measure internal performance in the kernel without using VM. The system can offer similar benefits to our system with respect to profiling (but not code coverage) when the hardware provides the registers.

One benefit of the hypervisor approach that is employed by LgDb is that one can use the launcher to produce additional information such as call trees, tracing the current number of running threads and providing information that performance counters do not have the registers to use for keeping track of this information. However, the current version of LgDb does not implement call trees.

3.6 Conclusion

We presented LgDb, which is a proof of concept tool for profiling and code coverage based on lguest hypervisor.

LgDb demonstrate that kernel development using a virtual machine and has many advantages and can ultimately lead to a more reliable and efficient kernel code. We hope that LgDb will lead the way to "virtualizing" our kernel developments.

3.6.1 Performance Results

Profiling and checking code coverage using this extension has very little overhead where almost all of which is due to lguest. Compared with normal profilers or kernel profilers in which the inspected process runs significantly slower than a normal non profiled process, our profiler consume very little system resources.

It is hard to find generic benchmark for kernel only profiling without measuring the effects of kernel profiling on running processes or the combination of kernel and user processes.

We developed a synthetic benchmark in which matrices were multiplied in the OS kernel. No effect on user process or any user work was measured.

The benchmark results in seconds where low measures are better:

native (without lguest), without profiling 53.008323746
lguest, without profiling 53.048017075
lguest, with profiling 53.160033226
native, with Oprofile 58.148187103

These results, which are expected. The low overhead generated by the system was produced by the hypercall procedure. The overhead is low because only the switcher code, which performs the guest to host switch was added. This switch sums to 120 machine instructions for two hypercalls (profiling_start and profiling_end) each goes both ways (guest-to-host, host-to-guest). [1]

This benchmark is ideal for the system because it involves mathematical operations only and no virtual I/O operations which would require communication between the guest and the host.

In addition, this is unlike Oprofile that sends non maskable interrupts to the inspected kernel. Unrelated kernel modules are unaffected by LgDb. When running Oprofile, the performance of the entire system is degraded.

3.6.2 Innovation

The architecture, which is described in this thesis, runs a kernel code coverage and profiling by using an hypervisor that is part of the profiling process, is a new concept. Using hypervisor for code coverage is also a new concept.

Although tools for profiling and coverage kernel code already exist, LgDb allows more features that were unavailable before (as stated in sections 3.5.3 and 3.5.2).

Furthermore, the concept of forking a running guest is new to lguest and can be used to demonstrate features such as replication, recovery and applications.

3.7 Looking forward and LgDb 2.0

We continue our work on LgDb in order to achieve more reliable and efficient kernel coding. Our main objectives are to detect causes in memory access violations, memory leaks and kernel "oops"es and assist in other situations where the guest kernel crashed.

In LgDb 2.0 we connected LgDb with kernel debugger (KDB) via virtual serial driver to the guest. By doing so we eliminated the need for code injections by replacing the calls to hypercalls with breakpoints mechanism in the debugger. We obtain all process information we need by reading the kernel module DWARF file. In LgDb 2.0 we kept the hypercalls mechanism to allow for custom wallets but they are no longer required. This feature alone makes LgDb much more usable. LgDb 2.0 is not ready to be pushed upstream yet and it will be pushed shortly after this dissertation is published.

For kernel developers that use kernel debuggers, we would like to enable the ability for the host launcher to access debug information on terminated guest thus enabling LgDb to act as kernel debugger. We expect this feature to be released with LgDb 2.1 or other future version.

We also work on improved multitasking support by integrating it with the

[1] 30 machine instruction per direction per hypercall were needed

guest scheduler. This will allow us to distinguish between busy and idle time on the guest and to profile multi-thread kernel modules.

3.8 Availability

LgDb is available from

```
http://www.scipio.org/kerneldebugger
```

4 ASYNCHRONOUS REPLICATION OF BLOCK DEVICES AND VM

This chapter describes the AMirror system. AMirror is a system designed for asynchronous replication of block devices and virtual machines. AMirror was developed with Tomer Margalit for his M.Sc thesis. Replication of Lguest was developed with Eviatar Khen for his M.Sc thesis. The system was published in [AMZK11].

This chapter relies on technologies that are described in appendixes 3, 6 and 5.

4.1 Introduction

4.1.1 Disaster Recovery problem, replication and VM replication

Today, it is becoming more and more common for small offices and even home users/offices to contain vital business data on organization servers. Since vital data can not be lost, server administrators and business owners in such environments face the problem of protecting their data and their vital systems.

An administrator can attempt to protect data by using local mirroring (RAID) of the vital data on to a secondary disk on the same system enabling survival of single disk failures. However, even systems with redundant disks are still vulnerable to fires, theft and other local disasters. By local we mean disasters that affect the entire site and not just a single computing component.

Usually, by Enterprise systems we mean multiple vendors that provide disaster protection by replicating the disk into a remote site. When disaster strikes on the primary site, the user will be able to recover his system from the disaster recovery site. Enterprise systems provide not only data recovery but also full system recovery for systems running on system virtual machines (hypervisors).

Recovering virtual machines in their running state allows for minimal recovery time and almost immediate operation continuity as the system is already recovered for operational status. The advantages of replicating the whole virtual machines system are two fold. Firstly, the user does not have to wait for the system to boot reducing down time required before returning to operation continuity. Secondly, system restoration and reconfiguration can be skipped since the system is already restored in a running state. Since the system is already operational when recovered no additional down time for booting the servers is expected.

The main difference between enterprise and small-office/home-office (SOHO) systems from a disaster recovery point of view is that in an enterprise system we can afford to replicate every byte that is written on the disk and in certain cases even every byte that is written to the memory. SOHO users do not have the required resources and therefore have to settle for solutions that use bandwidth more efficiently.

AMirror borrows from the enterprise world to the home user world by bringing the enterprise level replication to the reach and bandwidth capacity of SOHO users.

4.1.2 Terminology

Below is a short list of storage and virtualization terms used in this chapter.

Replication: Refers to the replication of raw data or higher level objects from one site to another. Replication is demonstrated in figure 18.

Synchronous/Asynchronous replication: There are two ways to replicate data.

Synchronous Replication Ensures that every write is transferred to a remote location before reporting write completion. The flow chart for synchronous replication is shown in figure 19.

Asynchronous Replication buffers the writes and reports success immediately after a write has been buffered and written locally (but not to the disaster recovery site). An asynchronous replication system can later flush the buffer to the disaster recovery site and write it at its leisure. The flow chart for asynchronous replication is shown in figure 20.

Synchronous replication has the benefit of having *zero data loss* in case of disasters. Asynchronous replication has the benefit of reduced cost as less bandwidth capacity is required. A secondary benefit of asynchronous replication is the possibility of having a greater distance between the primary and disaster recovery sites. Since the speed of light is finite and most installations limit the extra latency that is acceptable due to mirroring, it is rare to find a Synchronous Replication solution where the distance between the primary and disaster recovery sites is greater than 50km. In contrast, in asynchronous mirroring scenarios the distance between the primary and the disaster recovery sites can be practically unlimited.

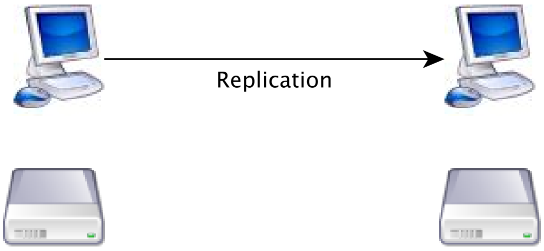


FIGURE 18 Generic replication System

Because of the nature of the target systems, we chose to use asynchronous replication. In the reminder of this chapter, we use the terms replication and asynchronous replication interchangeably.

Recovery Point Objective (RPO) is the maximal data size we consider as an acceptable loss in case of a disaster. We will measure RPO in terms of data generated in MB even though it may be more common to address RPO in terms of time.

Recovery Time Objective (RTO) is the length of time it takes to fully recover the system after a disaster. Usually this includes the time required to obtain the backups and the time required for recovery from backups and bringing the system to an operational state.

Deduplication is a technique used to eliminate redundant data. In our case. we use deduplication to avoid sending redundant data by detecting data that already exists on the other end.

The Linux block device layer is the block I/O layer in Linux. This layer is located below the file system layer and above the disk drivers layer. This layer is common to all disks types and to all file systems.

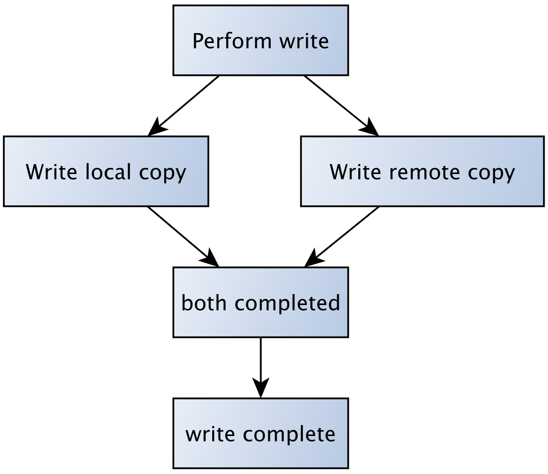


FIGURE 19 Synchronous replication System

VM migration refers to the situation where a running VM is "moved", uninterrupted[1] from the host it runs on to a different host.

VM replication refers to the repeated migration of a VM system every preset time slice.

Lguest [Rus] is a lightweight Linux-on-Linux system hypervisor developed by Rusty Russell[Rus]. Lguest is described in detail in Appendix 6.

QEMU-KVM KVM [Kiv07] is a virtualization framework that is developed inside the Linux kernel. It is sponsored and used by Red Hat. QEMU [Bel05] is an open source system hypervisor that can use KVM. QEMU and KVM are described in detail in Appendix 3.

Continuous Data Protection (CDP) refers to a backup environment that continuously logs write operations. By marking snapshots on the log we can rollback to any snapshot. By rolling the log we can get to any snapshot.

[1] There are likely to be environmental interruptions if the machine is moved to a different network while being connected to an external host such as TCP/IP connection timeouts, but we see these as being beyond the scope of the dissertation.

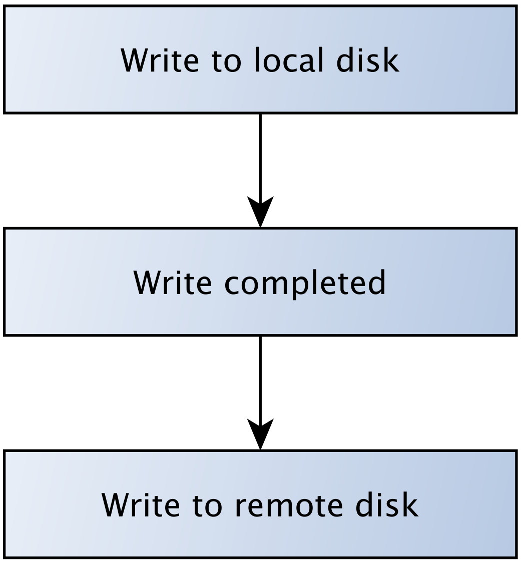


FIGURE 20 Asynchronous replication System

4.2 Deduplication

AMirror uses a design method called deduplication. Deduplication is the art of not doing things twice. For example, in AMirror we are replicating blocks from the primary site into the disaster recovery site. If a block is already transferred then we do not want to transfer it again. We detect this by keeping hashes of all the blocks already present in the disaster recovery site. If a hash is already found in the disaster recovery site then we do not need to transfer the block again only its ID.

There are plenty of Deduplication solutions in the storage and networking world and AMirror is just another implementation of the deduplication concept.

4.3 Related work

4.3.1 Replication

The Linux kernel includes an asynchronous mirroring system from kernel 2.6.33 called DRBD [DRB10]. AMirror block level replication offers a similar design to the design that DRBD offers in the sense that both provide a virtual block device based on a physical block device that being replicated. Furthermore, AMirror offers similar performance to DRBD when AMirror is running with data compression and deduplication switched off. However, a key difference in the design of AMirror and DRBD is that AMirror performs most logical decisions in the user space allowing us to add several key features to AMirror that do not exist in DRBD which is all kernel solution. Putting the logic in user space allows us to add features such as CDP, volume group replication, data compression and deduplication of blocks. These features are a vital part of our approach to low bitrate replication.

There are plenty of other open and closed source replication solutions with varying similarities to ours. Storage vendors provide their own proprietary replication solutions. For example, IBM's PPRC [CHLS08] and EMC's SRDF [Coo09] are both replication protocols between high end storage servers while EMC Recoverpoint provides replication and CDP via SCSI write command replication. While these solutions offer the same set of tools as we do (and more) they are limited to hardware which is far beyond the reach of our target audience.

Replication can also occur in the file system level. For instance, Mirrorfs [Jou07] and Antiquity [WECK07] are another file based logging approach for replication.

There are also further distributed solutions for reliability such as Google file system [GGL03] and HADOOP [SKRC10]. Google file system implements the replication of data to recover from local node crashes in a cluster. Such clustering approaches are very useful in the enterprise level, especially, if the cluster is deployed on several physical sites but are beyond the reach of our target users.

Mirroring in the OS level was implemented as well for example in Mosix [BS11]. DRM [WVL10] is an object based approach for replication. We feel that replication in the block level is superior to OS or filesystem replication. Block level replications allows our "SOHO and home user" to use his favorite Linux distribution and applications while using any file system that is suitable to his needs and with minimal configuration requirements.

4.3.2 Replication and Migration of Virtual Machine

There is no live migration or replication support for Lguest in Linux kernel 2.6.38 though stubs for mirroring do exist.

The Kemari [TSKM08] project did some work on replicating QEMU-KVM. The live migration in QEMU-KVM and Kemari is done on the same LAN when the storage is shared. We alleviate this requirement by using our block device replication.

There are several replication solutions in other hypervisors.

The Remus project [CLM+08] provides solutions for replicating Xen based virtual machines. Similarly, the VMWare product line offers multiple (closed source) products in the VMotion product line for VM replication. Other commercial solutions include Novell's PlateSpin and Oracle's solution for VirtualBox.

Understandably, most other implementations have developed software for the enterprise world. Because the target audience for these solutions is the enterprise market, the requirements for good system performance and minimal RPO are emphasized. As a result, these systems replicate volumes of data that are unmanageable by a normal home user or small office. On the other hand, we feel that for most home users in most day-to-day usage the underlying hardware is often idle. Therefore, we do not see the need to put the system performance as a prime requirement. We also argue that for our target audience, RPO of the current time minus approximately 5 minutes or so is sufficient for our target audience.

Lastly we see bandwidth as an expensive resource. We believe home users will not be willing or able to pay for the replication uplink that is common in enterprise environments. With these constraints in mind we designed our system.

4.3.3 Storage Concepts of Deduplication and CDP

The concept of CDP is well researched. Two recent implementations include [LXY08, SWHJ09]. Industrial CDP products like EMC RecoverPoint serve many purposes such as creating multiple restore points, elimination of the necessity of backup windows, etc.

We require CDP as part of our solution in order to ensure that VMs migrate consistent memory and storage.

Deduplication is also widely used in the industry, especially in virtual tape libraries such as EMC Data Domain or IBM ProtectTIER and other such products. The common task of deduplication is to ensure that only one copy of identical data is kept. Deduplication [ZZFfZ06] was researched in object storage and [ZMC11] demonstrates the PACK system for the deduplication of network traffic. Our algorithm is much simpler than PACK due to the block based nature of our storage in comparison to variable block size in PACK.

4.4 Design goals

We focused our design on features that will be useful and affordable to our target audience. We did not care for features such as clusters or cloud support, etc. Instead, we put a major emphasis on maintaining decent performance with low bandwidth requirements.

We pushed the implementation of any feature that made sense to the block device layer. Working on the block device layer allows our project to be used with multiple hypervisors and any file system with minimal changes. We kept the changes on the hypervisor level to a minimum by requiring only a way to serialize the VM's current state or accumulate the changes from the VM last recorded state.

4.4.1 Block Device Replication

The following requirements were made for our block device replication product:

File system support: Supports any file system without modification to the file system.

Application support: Supports any application using the block device without modifications to the application.

Network support: Supports any network that provides TCP/IP connections between source and target.

Minimal recovery time: Recovery of the latest known position on the disaster recovery site should take minimal time. Recovery to an older snapshot may take longer.

CDP: In order to provide consistent recovery of a VM memory and storage, we require the ability to take snapshots of the virtual machine memory and storage at an exact point in time. Therefore, continuous data protection is required.

Low bandwidth: Minimal amount of bandwidth should be used while replicating. It can be assumed that at least parts of the data can be deduplicated. For example, loading a program to memory that already exists on the disk.

Support direction switch: The replication direction can be switched. This feature allows the user to work from home while replicating the data to his office and vice versa as required.

4.4.2 Mirroring and Replication of VMs

From each of the supported hypervisors we need the following features to support migration and replication.

Serialize the state of a running VM: We must be able to save and load the state of a running VM.

Freeze the VM state: We must be able to freeze the VM to take a consistent snapshot.

Detecting "dirty pages" between migrations: This feature is a "nice to have" and is implemented by QEMU but not in Lguest. It saves us the need to scan the entire memory for changes but as we implement Deduplication of pages, pages that exist in the other end will not be transferred. This feature set exists in QEMU (QEMU migration) and had to be implemented for Lguest.

4.4.3 Failing to meet RPO objective

Most industrial asynchronous mirror products define a maximal RPO that the user is willing to lose in case of a disaster. The RPO is measured either in time units or bytes. If we get more data then our upload rate can manage. This can occur if the user is running something with heavy disk usage such as software installation. Then AMirror can provide the user with the choice of permitting the write to occur or causing the write to fail. If we allow the write to occur, we cannot commit on the RPO guarantee if a disaster actually occurs.

Enterprise level replication solutions keep the RPO guarantee and causes writes to fail. We feel that in home or SOHO environments, blocking the user from writing is doing an ill service. We also feel that most massive disk I/O operations in our target environments come from software installation or other reproducible content. Therefore, we leave the requirement to cause write operations to fail or block as a configurable option with our default to allow the writes.

In such cases where we cannot meet the RPO, we notify the user, increase the size of our logs and we hope to be able to deliver the backlog of the data as soon as possible.

4.5 System Architecture

We consider two parts in our application: The asynchronous mirror for storage product and the VM migration code.

We will discuss each part of the architecture separately.

Asynchronous mirror: The application consists of kernel space components and user space components.

The kernel space components are in charge of generating logical, replicating, block devices to operate on. The kernel module intercepts block writes to these block devices (by write(2), mmap(2) or any other means) and delivers the write operations to the user mode components. Reads are passed to the underlying hardware without intervention.

The user mode components are responsible for acquiring write operations from the kernel module and delivering them to the disaster recovery site and for saving it at the disaster recovery site.

Lastly, a programmable console allows for the creation of block devices, the creation of volume groups, the management of volume groups and the generation of consistent backups using previous backups and the CDP log.

User space process: The user space part of the asynchronous mirror product reads the shared buffer from the kernel and is responsible for delivery to the disaster recovery site. It is also in charge of doing deduplication and compression.

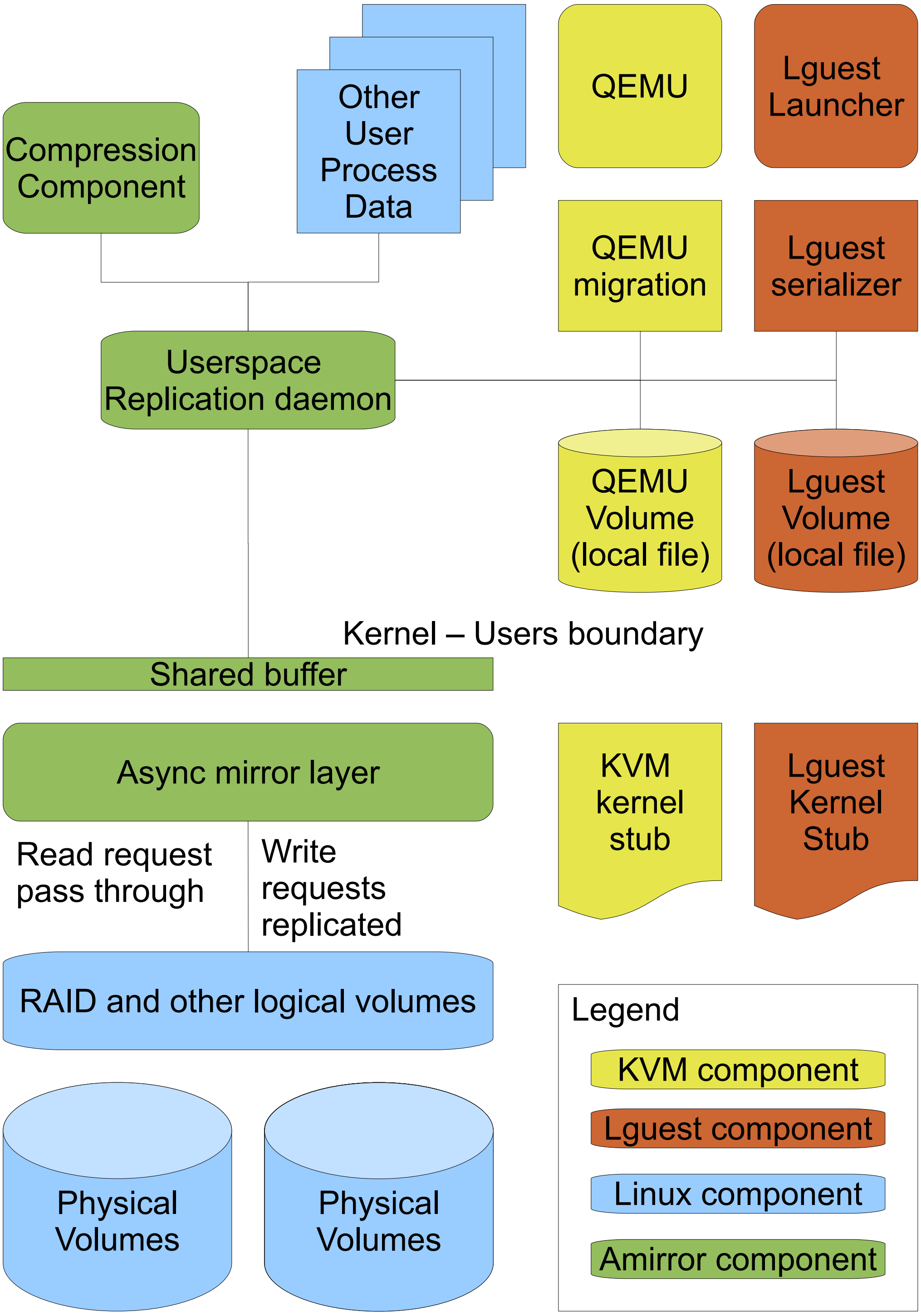


FIGURE 21 Schematic Architecture of the AMirror Tool

A second user process is responsible for saving the changes on a local block device at the disaster recovery site.

Kernel process: We use the AMirror kernel module. The Kernel module is responsible for exporting the kernel buffer and for the management of the block devices. The AMirror kernel module is also responsible for queueing the write operations into the kernel buffer.

Deduplication: We implement the deduplication layer in our solution because we feel that VM memory tends to have identical contents throughout the VM runtime.

We calculate the MD5 hash for each 8k page that we get and keep a set of existing hashes. If identical hashes exist we do not transmit the data. Instead, we only transmit the address where the page is located and the page number.

CDP: Continuous data protection (CDP) is important to our solution because we serialize the CPU and memory of the VM in a specific time to match the VM storage. All the CDP implementations use some log form. When a snapshot is created, a mark is placed on the log indicating a restore point. When requested to restore to a certain point in time, we need to roll the log until we reach the mark.

Volume group management: Our replication solution uses a single queue and a single buffer for all the replicated block devices including VM disks and memory that is placed in a single volume group. The concept of a shared queue for all disks in the group ensures that if one disk lags no writes will take place on the other disks in the same group. Thus, we ensure that the system will always be recovered in a consistent state. Volume group management is illustrated in figure 22.

4.6 VM Replication

The abstract logic for replicating a VM is as follows:

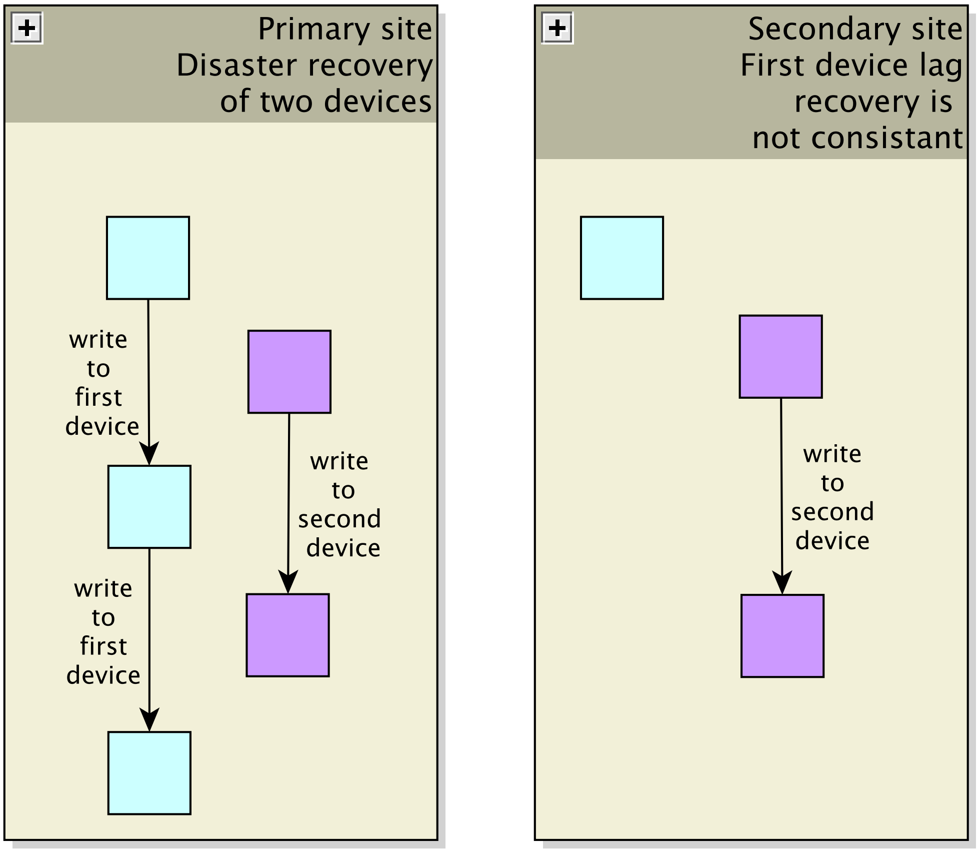


FIGURE 22 Why Volume Groups are Required

Algorithm 3 VM Replication Logic

```
while VM is running do
    Stop VM
    Mark disk storage restore point (point in time snapshot)
    Fork VM user process
    if parent then
        Start marking dirty pages for next replication.
        The child replicates the kernel VM structures.
        Resume VM, sleep until next replication time.
    else
        Read the VM Kernel memory and move to the other side.
        Replicate VM main memory and other data (Virt queues etc.)
    end if
end while
```

Individual VMs have minor differences and therefore differ slightly from this model.

4.7 Low Bitrate Considerations

All prior and related solutions that we surveyed replicate every page modified (dirty pages), and thus the bit rate of replicating a complete system may be considerable. In theory, we may require the memory bus to be identical to the network bus.

To save on bandwidth the following steps were taken:

We increased the RPO buffer size: In this way, we have more chances to eliminate redundant writes.

We use references to blocks that already exists in the disaster recovery site: In our implementation of block/page level deduplication, we keep a hash of all the pages contents. Before a write is sent, we check if the contents of the written page exist in another page. If a copy of the data already exists in the disaster recovery site a reference to that page is sent instead of the page content. This way we avoid the transfer of pages moved in memory.

We compress the transmitted data: If the user requires it, we use compression to further reduce the bandwidth usage.

4.8 Benchmarks

The Asynchronous mirror solution utilizes almost 100% of the device speed when doing read tests. We reach close to 100% of either disk or LAN speed when doing

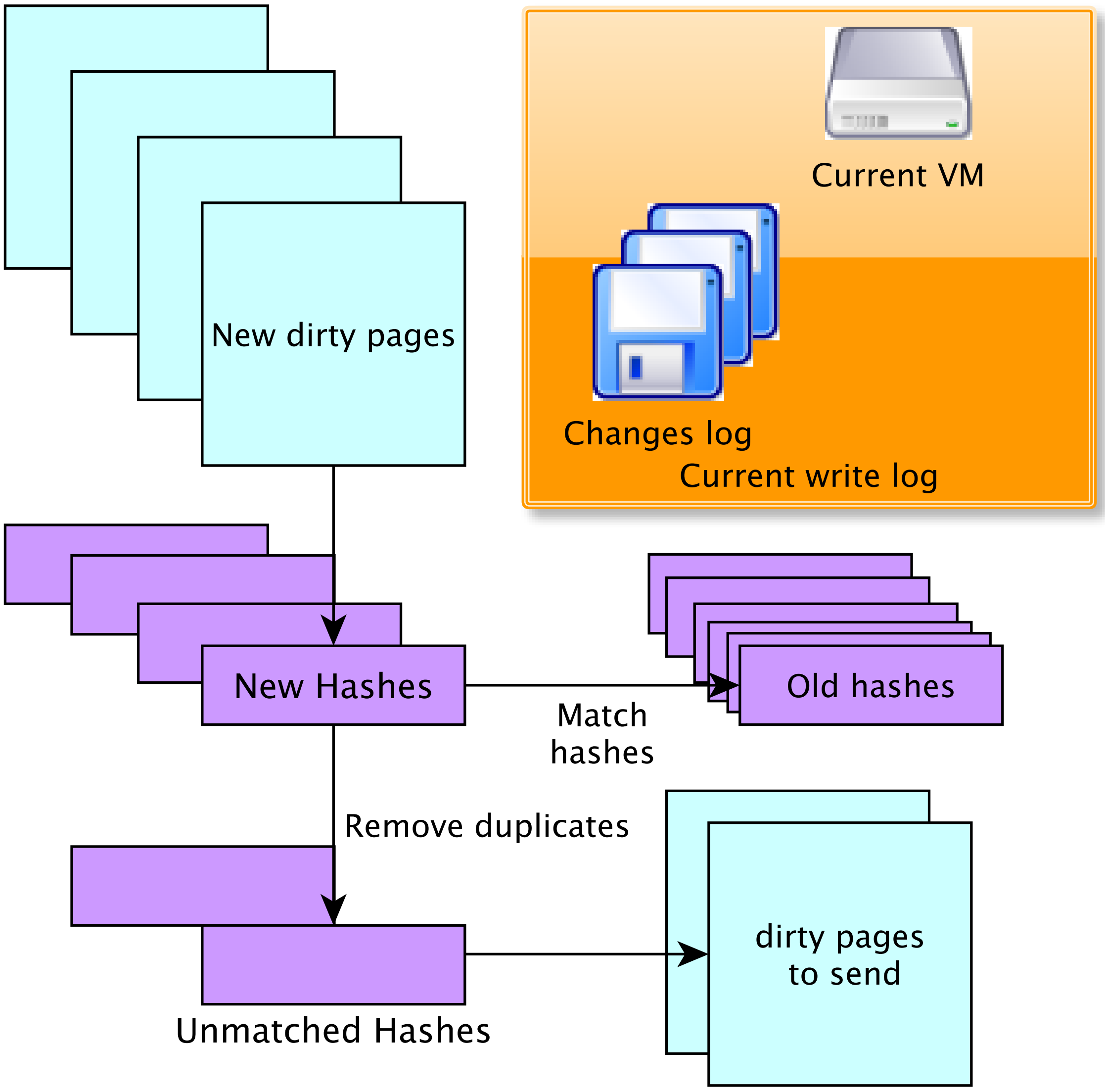


FIGURE 23 Deduplication on the Network Traffic

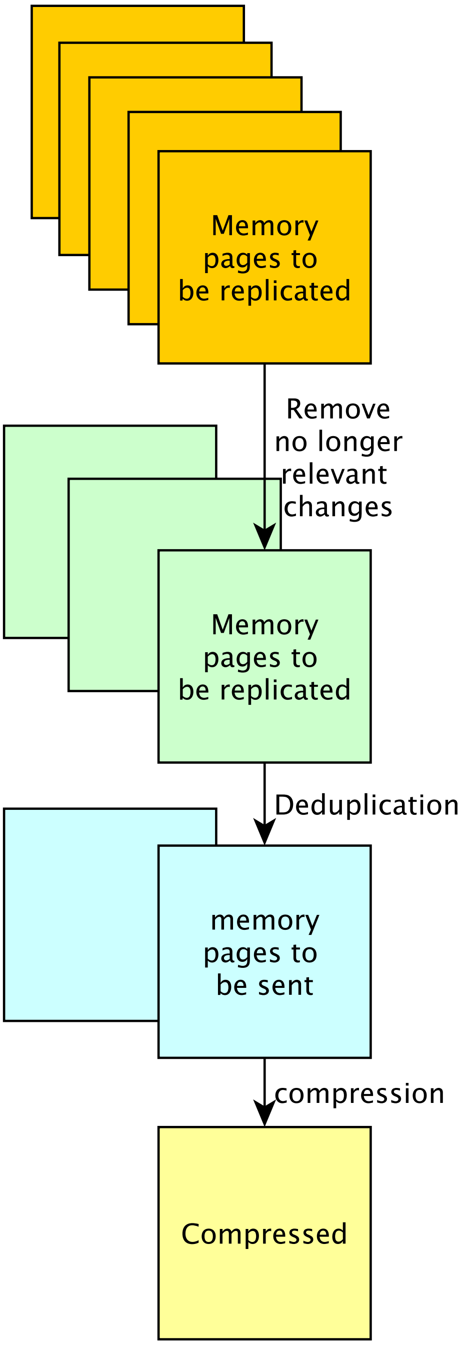

Memory pages to be replicated
Remove no longer relevant changes
Memory pages to be replicated
Deduplication
memory pages to be sent
compression
Compressed

FIGURE 24 Saving of Bandwidth

write tests on LAN. This is in comparison with the standard remote copy or ftp.

Activating compression and/or deduplication dramatically affects the performance. Performance drops to about 10–20% of the original rate. We argue that this rate should be sufficient for home users that do not have the uplink bandwidth for a faster migration.

4.9 Future development

The AMirror solution lends itself to future extensions in the field of Cloud computing. We plan to develop a windows based version of AMirror and use AMirror as a Platform-as-a-Service environment for DRP.

4.10 Availability

The code for this project is available under dual GNU/BSD license. The Sourceforge project URL is:

http://amirror.git.sourceforge.net/git/gitweb-index.cgi

5 TRULY-PROTECT

5.1 Introduction

A rising trend in the field of virtualization is the use of VM based digital rights and copy protection. The two goals of introducing VM to digital rights protection are to encrypt and to obfuscate the program. Forcing the hackers to migrate from a familiar x86 environment to an unfamiliar and obfuscated virtual environment is intended to pose a greater challenge in breaking the software copy protection.

A generic and semi-automatic method for breaking VM based protection is proposed by Rolles [Rol09]. It assumes that the VM is, broadly speaking, an infinite loop with a large switch statement called the op-code dispatcher. Each case in this switch statement is a handler of a particular op-code.

The first step a reverse-engineer should take according to Rolles method is to examine the VM and construct a translator. The translator is a software tool that maps the program instructions from the VM language to some other language chosen by the engineer, for example x86 assembly. The VM may be stack based or register based. The reverse-engineer work is similar in both cases.

Rolles calls language that the translators translates the code it reads into, an intermediate representation (IR). The first step is done only once for a particular VM based protection, regardless of how many software systems are protected using the same VM. In the second step, the method extracts the VM op-code dispatcher and the obfuscated instructions from the executable code. The op-codes of these instructions, however, do not have to correspond to those of the translator: the op-codes for every program instance can be permuted differently. In the third step, the method examines the dispatcher of the VM and reveals the meaning of each op-code from the code executed by its handler. Finally, the obfuscated instructions are translated to IR. At this point, the program is not protected anymore since it can be executed without the VM. Rolles further applies a series of optimizations to achieve a program which is close to the original one. Even by using Rolles' assumptions, we argue that a VM, which is unbreakable by the Rolles' method, can be constructed. In this chapter, we will describe how to develop such a VM.

In this chapter, we do not try to obfuscate the VM. Its source code is publicly available and its detailed description appears herein. We protect the VM by holding secretly the op-code dispatcher. By secretly we mean inside the CPU internal memory. Holding the op-code dispatcher in secret makes it impossible to perform the second step described by Rolles.

Moreover, we claim that the security of the system can be guaranteed under the following assumptions:

- The inner state of the CPU cannot be read by the user.
- The CPU has a sufficient amount of internal memory.

The former assumption simply states that the potential attacker can not access the internal hardware of the CPU. The second assumption however is more vague, so the properties of such an internal memory are discussed in section 5.6.

This chapter has the following structure: Section 5.2 provides an overview of related work. Section 5.3 outlines a step-by-step evolution of our system. Final details of the evolution are provided in Section 5.4. Section 5.5 describes the security of the proposed system. Section 5.6 describes how to use different facilities of modern CPUs to implement the required internal memory. The performance is analyzed in section 5.7. Section 5.8 provides an example of a C program and its corresponding encrypted bytecode.

5.2 Related work

5.2.1 Virtual machines for copy protection

The two goals of introducing VM to trusted computing are to encrypt and to obfuscate the program. Forcing the hackers to migrate from a familiar x86 environment to an unfamiliar and obfuscated virtual environment is intended to pose a greater challenge in breaking the software copy protection.

However, not much has been published on the construction of virtual machines for digital rights protection as it would be counter productive for the obfuscation efforts that were the main reason for using VMs. Hackers, on the other hand, have opposite goals and tend to publish their results more often. Therefore, we often learn about VM protection from their breakers instead of their makers. For example, [sch] is the most complete documentation of Code Virtualizer internals available outside Oreans.

Examples of using virtualization for copy protection include the hypervisor in Sony Play Station 3 [SON] and XBox 360. The Cell OS Level 1 is a system virtual machine which is not exposed to the user. The XBox 360 hypervisor also ensures that only signed code will be executed[DS07]. For PC software, Code Virtualizer [Ore] or VMProtect [VMP] are both software protection packages based on process virtual machines.

In all cases, very little was published by the software provider. However, the means for attacking virtual machine protection has been published by PS3 hackers [bms10] and by Rolles [Rol] with respect to VMProtect.

5.2.2 Hackers usage of Virtualization

By running the protected software in a virtual environment it became possible to disguise it as a different system. For example, running OS X on a standard hardware using a VM disguised as an Apple machine [Gra08]. Protection methods against virtualization were researched by [KJ03]. Inspecting and freezing CPU and memory of a running system to obtain, for example, a copy of a copyrighted media, is another threat faced by media protectors such as NDS PC Show [NDS]. It is also possible to construct malware that will obfuscate itself using a VM. How malware obfuscated by VM can be detected is described in [SLGL09].

5.2.3 Execution Verification

Protecting and breaking software represent a long struggle between vendors and crackers that began even before VM protection gained popularity. Users demand a mechanism to ensure that the software they acquire is authentic. At the same time, software vendors require users to authenticate and ensure the validity of the software license for business reasons.

Execution verification and signatures is now a part of Apple Mach-O [App] object format and Microsoft Authenticode [Mic]. It also exists as an extension to Linux ELF [mil].

The trusted components have been heavily researched in industry [Tru] and academia [SBPR08] among others with mixed results. Both software rights protectors and hackers were able to report on a partial success.

From the hackers' camp, [Ste05] is a fundamental paper dissecting all Microsoft's security mistakes in the first XBox generation.

5.3 System Evolution

In this section, we describe the evolution of the proposed system in several phases which are fictional interim versions of the system. For each version, we describe the system and discuss its advantages, disadvantages and fitness to today's world of hardware and software components.

We explore the means to improve the analyzed version and consider the implementation details worth mentioning as well as any related work.

The first version describes the system broken by Rolles. The second version cannot be broken by Rolles but has much stronger assumptions. The rest of the evolution process consists of our engineering innovation. The sixth version is as secure as the second one but requires only the assumptions of the first version.

In the seventh version, we propose a completely different approach: a just-in-time decryption mechanism which incurs only minor performance penalty.

The last section presents a parallelization scheme for the sixth version, which can theoretically improve its performance by utilizing an additional core present at a potential CPU. This idea was not implemented and thus described at the end of this section.

5.3.1 Dramatis Personae

The following are actors that participate in our system use cases:

Hardware Vendor manufactures. The Hardware Vendor can identify components he manufactured. The Hardware Vendor is trust worthy. A possible real world example is Sony as the Hardware Vendor of Play Station 3.

Software Distributor distributes copy protected software. It is interested in providing conditional access to the software. In our case, the Software Distributor is interested in running the software on one End User CPU per license. A possible real world example is VMProtect.

“Game": The software we wish to protect. It may be a computer game, a copyrighted video or other piece of software.

End User purchases at least one legal copy of the “Game" from a Software Distributor. The End User may be interested in providing other users with illegal copies of the “Game". The End User is not trustworthy.

The goal of the system described herein is to prevent any of the End Users from obtaining even a single illegal copy of the “Game".

VM: A software component developed and distributed by a Software Distributor.

VM Interpreter: A sub–component of VM that interprets the instructions given by the “Game".

VM Compiler: A software component used by the software distributor to convert a “Game" code developed in high level programming language to instructions interpretable by the VM Interpreter.

Malicious End User: The Malicious End User would like to obtain illegitimate copy of the game. The VM Compiler, VM Interpreter and VM are tools manufactured by the software distributor and hardware vendor that prevent the Malicious End User from achieving her goal : a single unlicensed copy. The Malicious End User may enlist one or more End User with legal copies of the game to achieve her goal.

5.3.2 Evolution

The building of the system will be described in steps.

System Version 1 The VM Interpreter represents virtual, unknown instruction set architecture (ISA) or permutation of a known instruction set such as MIPS in our case. The VM Interpreter runs a loop:

Algorithm 4 System 1 — VM Interpreter Run Loop

```
while VM is running do
    fetch next instruction
    choose the instruction handling routine
    execute the routine
end while
```

The VM Compiler reads a program in a high level programming language and produces the output in the chosen ISA.

System Version 1: Discussion Cryptographically speaking, this is an implementation of a replacement cipher on the instruction set. This method was described by Rolles [Rol09] and used by VMProtect. Of course, the VM may include several other obfuscating subcomponents that may even provide greater challenge to a malicious user but this is beyond our scope. The protection is provided by the VM complexity and by the user's lack of ability to understand it and, as stated previously, in additional obfuscations.

Rolles describes a semi-automatic way to translate a program from the unknown ISA to intermediate representation and later to the local machine ISA. Understanding how the VM works is based on understanding the interpreter. This problem is unavoidable. Even if the interpreter is implementing a secure cipher such as AES, it will be unable to provide a tangible difference as the key to the cipher will also be stored in the interpreter in an unprotected form.

Therefore, it is vital to use a hardware component that the End User cannot reach to provide an unbreakable security.

System Version 2 The Hardware Vendor cooperates with the Software Distributor. He provides a CPU that holds a secret key known to the software distributor. Furthermore, the CPU implements an encryption and decryption algorithms.

The compiler needs to encrypt the program with the CPU's secret key. This version does not require a VM since the decryption takes place inside the CPU and its operation is similar to that of a standard computer.

System Version 2: Discussion This version can implement any cipher including AES, which is considered strong. This form of encryption was described by

Best [Bes80]. Some information about the program such as memory access patterns can still be obtained.

This method requires manufacturing processors with cryptographic functionality and secret keys for decrypting every fetched instruction. Such processors are not widely available today.

System Version 3 This system is based on system version 2 but the decryption algorithm is implemented in software. We alleviate the hardware requirements of system version 2. The CPU stores a secret key which is also known to the Software Distributor. The VM Compiler reads the "Game" in high level programming language and provides the "Game" in an encrypted form where every instruction is encrypted using the secret key. The VM knows how to decrypt the value stored in one register with a key stored in another register.

The VM Interpreter runs the following loop:

Algorithm 5 System 3 — VM Interpreter Run Loop

```
while VM is running do
    fetch next instruction
    decrypt the instruction
    choose the instruction handling routine
    execute the routine
end while
```

System Version 3: Discussion This version, is as secure as system version 2 assuming the VM internal state is stored at all times inside the CPU internal memory.

If only the VM runs on the CPU then we can make sure that the state of the VM such as its registers never leave the CPU. The VM just has to access all the memory blocks incorporating its state once in a while. The exact frequency depends on the cache properties.

This method dramatically slows down the software. For example, decrypting one instruction using AES takes up to 112 CPU cycles on a CPU core without AES acceleration. However, Intel's newest processors can reduce the decryption time to 12 cycles per instruction with AES specific instructions.

System Version 4 System version 3 took a dramatic performance hit which we now try to improve.

By combining versions 1 and 3, we implement a substitution cipher as in version 1. The cipher is polyalphabetic and special instructions embedded in the code define the permutation that will be used for the following instructions.

Similar to system version 3, we use the hardware for holding a secret key that is known also to the Software Distributor.

The VM Interpreter runs the following code

Algorithm 6 System 4 — VM Interpreter Run Loop

```
while VM is running do
    fetch next instruction
    decrypt the instruction
    if current instruction is not special then
        choose the instruction handling routine
        execute the routine
    else
        decrypt the instruction arguments using the secret key
        build a new instruction permutation
    end if
end while
```

System Version 4: Discussion Section 5.7 defines a structure of the special instructions and a means to efficiently encode and reconstruct the permutations.

Dependent instructions should have the same arguments as justified by the following example which is extracted from the Pi Calculator described in section 5.8):

```
01: $bb0_1:
02: lw $2, 24($sp)
03: SWITCH (X)
04: lw  $3, 28( $sp)
05: subu $2, $2, $3
06: beq $2, $zero, $bb0_4
07: ...
08: $bb0_3:
09: ...
10: lw $3, 20($sp)
11: SWITCH (Y)
12: div $2, $3
13: mfhi $2
14: SWITCH (Z)
15: sw $2, 36($sp)
16: $bb0_4:
17: sw $zero, 32($sp)
18: lw $2, 28($sp)
```

This is a regular MIPS code augmented with three special instructions on lines 3, 11 and 14. The extraction consists of three basic blocks labeled bb0_1, bb0_3 and bb0_4. Note that we can arrive at the first line of bb0_4 (line 17) either from the conditional branch on line 6 or by falling through from bb0_3. In the first case, line 17 is encoded by X and in the second case, it is encoded by Z. The interpreter should be able to decode the instruction regardless of the control flow, thus X should be equal to Z. In order to characterize precisely the dependencies between SWITCH instructions we define the term "flow" and prove some facts about it.

Although, a flow can be defined on any directed graph, one might want to imagine a control flow graph derived from some function. Then, every basic block of the graph corresponds to a vertex and an edge connecting x and y suggests that a jump from x to y might occur.

A flow comprises two partitions of all basic blocks. We call the partitions *left* and *right*. Every set of basic blocks from the left partition has a corresponding set of basic blocks from the right partition and vice versa. In other words, we can think of these partitions as of a set of pairs. Every pair (A, B) has three characteristics: the control flow can jump from a basic block in A only to a basic block in B; the control flow can jump to a basic block in B only from a basic block in A; the sets A and B are minimal, in the sense that no basic blocks can be omitted from A and B.

The importance of these sets emerges from the following observation. In order to guarantee that the control flow arrives at a basic block in B with the same permutation, it is enough to make the last SWITCH instructions of basic blocks in A share the same argument. This is so, because we arrive at a basic block in B from some basic block in A. The formal proof follows.

Definition 5.3.1. *Given a directed graph $G = (V, E)$ and a vertex $v \in V$. A* flow *is a pair (A_v, B_v) defined iteratively:*

- $v \in A_v$;
- *If $x \in A_v$ then for every $(x, y) \in E$, $y \in B_v$;*
- *If $y \in B_v$ then for every $(x, y) \in E$, $x \in A_v$.*

No other vertices appear in A or B.

A flow can be characterized in another way, which is less suitable for computation but simplifies the proofs. One can easily see that the two definitions are equivalent.

Definition 5.3.2. *Given a directed graph $G = (V, E)$ and a vertex $v \in V$. A* flow *is a pair (A_v, B_v) defined as follows: $v \in A_v$ if there is a sequence $u = x_0, x_1, \ldots, x_k = v$ s.t. for every $1 \le i \le k$, there is $y_i \in V$ for which $(x_{i-1}, y_i), (x_i, y_i) \in E$. We call a sequence with this property a* chain.

B_v is defined similarly.

We use the above definition to prove several lemmas on flows. We use them later to justify the characterization of dependent SWITCHes.

Since the definition is symmetric with respect to the chain direction, the following corollary holds.

Corollary 1. *For every flow, $v \in A_u$ implies $u \in A_v$.*

Lemma 1. *If $v \in A_u$ then $A_v \subseteq A_u$.*

Proof A chain according to the definition is $u = x_0, x_1, \ldots, x_k = v$. Let $w \in A_v$ and let $v = x'_0, x'_1, \ldots, x'_{k'}$ be the chain that corresponds to w. The concatenation of these chains proves that $w \in A_u$. Therefore, $A_v \subseteq A_u$. □

Lemma 2. *If A_u and A_v are not disjoint then $A_u = A_v$.*

Proof A chain according to the definition is $u = x_0, x_1, \ldots, x_k = v$. Let $w \in A_u \cap A_v$. From the corollary, $u \in A_w$. The previous Lemma implies that $A_u \subseteq A_w$ and $A_w \subseteq A_v$, thus $A_u \subseteq A_v$. The other direction can be proved in a similar manner. □

Lemma 3. *If A_u and A_v are not disjoint or if B_u and B_v are not disjoint, then $A_u = A_v$ and $B_u = B_v$.*

Proof We omit the proof since it is similar to the proof of the previous lemma. □

Claim 1. *Let $G = (V, E)$ be a control flow graph s.t. V is the set of basic blocks and $(x, y) \in E$ if the control flow jumps from x to y. Two* SWITCH *instructions should share the same argument if and only if they are the last* SWITCH *instructions in the basic blocks u and v s.t. $A_u = A_v$. We assume that every basic block contains a* SWITCH *instruction.*

Proof Consider the instruction γ. We need to prove that the interpreter arrives at γ with the same permutation regardless of the execution path being taken.

If there is a SWITCH instruction α preceding γ in the same basic block then every execution path passes through α in its way to γ, so the interpreter arrives at γ with the permutation set at α.

If there is no SWITCH instruction preceding γ in its basic block w, then consider two execution paths P and Q and let u and v be the basic blocks preceding w in P and Q, respectively. Denote by α the last SWITCH of u and by β the last SWITCH of v.

Clearly $w \in B_u$ and $w \in B_v$, and thus by the last lemma, $A_u = A_v$. Therefore, α and β share the same argument and on both paths the interpreter arrives at γ with the same permutation. □

The proposed system allows calling or jumping only to destinations known at compile-time, otherwise the dependency graph can not be constructed reliably. Nevertheless, polymorphic behavior still can be realized. Consider a type hierarchy in which a function F is overridden. The destination address of a call to F can not be determined at compile-time. Note however that such a call can be replaced by a switch statement, that dispatches to the correct function according to the source object type.

System Version 5 We rely on the previous version but give up on the assumption that the CPU is keeping a secret key that is known to the software distributor. Instead we run a key exchange algorithm [Sch96b].

Algorithm 7 Key Exchange in System 5

```
1: The Software Distributor publishes his public key
2: The VM chooses a random number. The random number acts as the secret key.The random number is stored inside one of the CPU registers.
3: The VM encrypts it using a sequence of actions using the software distributor public key.
4: The VM sends the encrypted secret key to the Software Distributor.
5: The Software Distributor decrypts the value and gets the secret key.
```

System Version 5: Discussion The method is secure if and only if we can guarantee that the secret key was randomized in a real (non-virtual) environment where it is impossible to read CPU registers. Otherwise it would be possible to run the program in a virtual environment where the CPU registers, and therefore, the secret key, are accessible to the user. Another advantage of random keys is that different “Game"s have different keys. Thus, breaking the protection of one “Game" does not compromise the security of others.

System Version 6 This version is built on top of the system described in the previous section. Initially, we run the verification methods described by Kennell and Jamieson [KJ03].

Kennel and Jamieson propose a method of hardware and software verification that terminates with a shared “secret key" stored inside the CPU of the remote machine. The method is described in algorithm 8.

Using the algorithm of Kennel and Jamieson we can guarantee the genuinity of the remote computer, i.e. the hardware is real and the software is not malicious.

A simpler way to perform such a verification is described below. In order to ensure that the hardware is real we can require any CPU to keep an identifier which is a member of a random sequence. This will act as a shared secret in the identification algorithm. The algorithm is performed by the

Algorithm 8 System 6 — Genuine Hardware and Software Verification

1: The OS on the remote machine sends a packet to the distributor containing information about its processor.
2: The distributor generates a test and sends a memory mapping for the test.
3: The remote machine initializes the virtual memory mapping and acknowledges the distributor.
4: The distributor sends the test (a code to be run) and public key for response encryption.
5: The remote machine loads the code and the key into memory and transfers control to the test code. When the code completes computing the checksum, it jumps to a (now verified) function in the OS that encrypts the checksum and a random quantity and sends them to the distributor.
6: The distributor verifies that the checksum is correct and the result was received within an allowable time, and if so, acknowledges the remote host of success.
7: The remote host generates a new session key. The session key acts as our shared secret key, concatenates it with the previous random value, encrypts them with the public key and then sends the encrypted key to the distributor.

VM without knowing the identifier itself. Identification algorithms are described in greater details in [Sch96c].

In order to ensure the authenticity of the software, we can initiate the chain of trust in the CPU itself as in the XBox 360. The CPU will initiate its boot sequence from an internal and irreplaceable ROM. The memory verified by algorithm 8 should reside in the internal memory of the CPU. This issue is discussed in greater detail in section 5.6.

System Version 6: Discussion System 6 alleviates the risk of running inside a VM that is found in system version 5.

System Version 7 Modern virtual machines like JVM [LY99] and CLR [Box02], employ just-in-time compilers to increase the performance of the program being executed. It is natural to extend this idea to the just-in-time decryption of encrypted programs. Instead of decrypting only a single instruction each time, we can decrypt an entire function. Clearly, decrypting such a large portion of the code is safe only if the CPU instruction cache is sufficiently large to hold it. When the execution leaves a decrypted function either by returning from it or by calling another function, the decrypted function is erased and the new function is decrypted. The execution continues. The benefit of this approach is obvious: every loop that appears in the function is decrypted only once, as opposed to being decrypted on every iteration by the interpreter. The relatively low cost of decryption, allows us to use a stronger and thus less efficient cryptographic functions, making this approach more resistant to crypt-analysis.

This approach uses the key-exchange protocol described in system version 6. We assume that there is a shared secret key between the Software Distributor and the End User. The Software Distributor encrypts the binary program using the shared key and sends the encrypted program to the End User. The virtual machine loads the program to the memory in the same fashion that the operating system loads regular programs to main memory. After the program is loaded and just before its execution begins, the virtual machine performs the following steps:

1. Make a copy of the program's code in another location.
2. Overwrite the original program's code with some value for which the CPU throws an illegal op-code exception, e.g. 0xFF on x86.
3. Set a signal handler to catch the illegal op-code exception.

We call the memory location containing the illegal op-codes as the "text segment" or "T". The copy, which was made on the first step, is called the "copy segment" or "C". After performing these steps, the program execution begins and then immediately throws an illegal op-code exception. This, in turn, invokes the handler set on step 3.

This mechanism is similar to just-in-time compilation. The handler is responsible for:

1. Realizing which function is absent.
2. Constructing it.

The first step can be done by investigating the program stack. We begin by finding the first frame whose instruction pointer is inside T. The list of instruction pointers can be obtained through the "backtrace" library call. Next, we have to identify the function that contains this address. This can be done either by naively traversing the entire symbol table giving us linear time complexity, or by noting that this problem can be solved by the "interval tree" data structure [CLRS01]. The "interval tree" provides a logarithmic complexity: each function is a memory interval that contains instructions. The instruction pointer is a point and we want to find an interval that intersects with this point.

After finding the function F to be constructed in T, we can compute its location in C, copy F from C to T and finally decrypt it in C.

Note that in contrast to just-in-time compilers, we need to destroy the code of the previously decrypted function before handling the new function. The easiest way to do this is to write 0xFF over the entire text segment.

System version 7: Discussion Nowadays, when CPUs are equipped with megabytes of cache, the risk of instruction eviction is low even if the entire functions of moderate size are decrypted at once. Moreover, we propose to hold in the cache several frequently used functions in a decrypted form. This way, as can be seen in Fig. 26, we improve the performance drastically. We did not

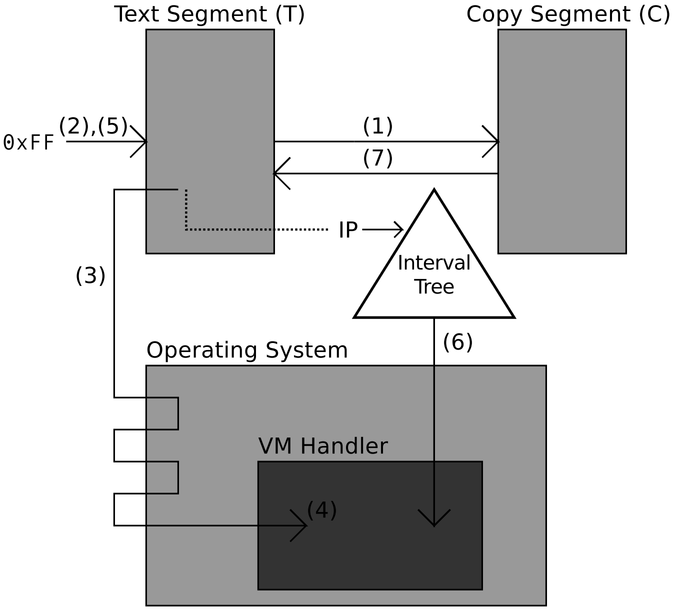


FIGURE 25 Just-In-Time Decrypting

Algorithm 9 Just-In-Time Decryption

```
while Execution continues do
    The program is copied from T to C.
    T is filled with illegal instructions.
    Illegal op-code exception is thrown and the operating system starts handling this exception.
    The execution is transferred to the VM handler.
    T is filled with illegal instructions.
    Intersection is found between the instruction pointer and an interval in the interval tree.
    The corresponding function is copied from C to T and decrypted.
end while
```

explore in-depth the function erasure heuristic, i.e. which functions should be erased upon exit and which should remain. However, we believe that the naive approach described below will suffice, meaning it is sufficient to hold the most frequently used functions such that the total size is limited by some fraction of the cache size. This can be implemented easily by allocating a counter for each function and counting the number of times the function was invoked.

Parallel System: Future Work Modern CPUs consist of multiple cores and a cache is shared between these cores. This system is based on system version 6 that tries to increase its performance by utilizing the additional cores available on the CPU.

The key observation is that the decryption of the next instruction and the execution of the current instruction can be done in parallel on different cores. In this sense, we refer to the next instruction as the one that will be executed after the execution of the current instruction. Usually, the next instruction is the instruction that immediately follows the current instruction.

This rule, however, has several exceptions. If the current instruction is SWITCH, then the next instruction, decrypted by another thread, is decrypted with the wrong key. If the current instruction is a branch instruction, then the next instruction, decrypted by another thread, will not be used by the main thread. We call the instructions of these two types “special instructions". In all the other cases, the next instruction is being decrypted while the current instruction is executed.

These observations give us a distributed algorithm for the interpreter:

Algorithm 10 Core I Thread

```
while VM is running do
    read instruction at $PC + 1$
    decrypt the instruction
    wait for Core II to execute the instruction at $PC$
    erase (write zeros over) the decrypted instruction
end while
```

Even better performance can be achieved in systems where the CPU chip contain CPU and GPU. (system on Chip) In such cases GPGPU-version of Truly protect can be achieved where the GPU decipher the instruction that are executed by the CPU. (This is the reason why we require system on chip. We cannot allow deciphered instructions to travel on a bus) Using block AES ciphers great speed up can be achieved.

System Version 7: Discussion We did not implement the proposed system and it is a work in progress. Clearly, it can substantially increase the system performance. Note that we benefit here from a polyalphabetic cipher, since it is practically impossible to use a block cipher like AES in this context.

Algorithm 11 Core II Thread

```
while VM is running do
    if previous instruction was special then
        decrypt instruction at $PC$
    end if
    fetch next instruction at $PC$
    choose instruction handler routine
    execute instruction using handler routine
    if previous instruction was special then
        erase (write zeros over) the decrypted instruction
    end if
    wait for Core I to decrypt the instruction at $PC + 1$
end while
```

Block ciphers operate on large portions of plaintext or ciphertext, so they may require the decryption of many instructions at once. After branching to a new location, we will have to find the portion of a program that was encrypted with the current instruction and decrypt all of them. Obviously, this is far from being optimal.

5.4 Final Details

5.4.1 Scenario

In this section, we provide a scenario that involves all the dramatis personae. We have the following participants: Victor — a Hardware Vendor, Dan — a Software Distributor, Patrick — a programmer developing PatGame and Uma — an End User.

Uma purchased a computer system supplied by Victor with Dan's VM preinstalled as part of the operating system. Patrick, who wants to distribute his "Game", sends it to Dan. Dan updates his online store to include PatGame as a new "Game".

Uma, who wants to play PatGame, sends a request for PatGame to Dan via his online store. Dan authenticates Uma's computer system, possibly in cooperation with Victor, as described in system version 6. After the authentication is completed successfully, Uma's VM generates a random secret key R, encrypts it with Dan's public key D and sends it to Dan. Dan decrypts the message obtaining R. This process was described in version 5. As described in version 4, Dan compiles the PatGame with the key R and sends it to Uma. Uma's VM executes the PatGame decrypting the arguments of special instructions with R.

A problem arises when Uma's computer is rebooted since the key R is stored in a volatile memory. Storing it outside the CPU will compromise its secrecy and thus the security of the whole system. We propose to store the key R on Dan's side.

Suppose Uma wants to play an instance of PatGame already residing on her computer. Uma's VM generates a random secret key Q, encrypts it with Dan's public key D and send it to Dan. Dan authenticates Uma's computer. After the authentication completes successfully, Dan decrypts the message obtaining Q. Dan encrypts the stored key R with the key Q, using AES for example, and sends it back to Uma. Uma decrypts the received message obtaining R, which is the program's decryption key. Thus the encrypted program doesn't have to be sent after every reboot of Uma's computer.

5.4.2 Compilation

Since the innovation of this chapter is mainly the 4th version of the system, we provide here a more detailed explanation of the compilation process. See section 5.8 for an example program passing through all the compilation phases.

The compiler reads a program written in a high level programming language. It compiles it as usual up to the phase of machine code emission. The compiler then inserts new special instructions, which we call SWITCH, at random with probability p before any of the initial instructions. The argument of the SWITCH instruction determines the permutation applied on the following code up to the next SWITCH instruction. Afterwards, the compiler calculates the dependencies between the inserted SWITCH instructions. The arguments of the SWITCH instructions are set randomly but with respect to the dependencies.

The compiler permutes the instructions following SWITCH according to its argument. In the final pass we encrypt the arguments of all SWITCHes by AES with the key R.

5.4.3 Permutation

In order to explain how the instructions are permuted, we should describe first the structure of the MIPS ISA we use. Every instruction starts with a 6-bit op-code that includes up to three 5-bit registers and, possibly, a 16-bit immediate value. The argument of the SWITCH instruction defines some permutation σ over 2^6 numbers and another permutation τ over 2^5 numbers. σ is used to permute the op-code and τ is used to permute the registers. Immediate values can be encoded either by computing them as described by Rolles [Rol09] or by encrypting them using AES.

5.5 Security

The method described by Rolles requires a complete knowledge of the VM's interpreter dispatch mechanism. This knowledge is essential for implementing a mapping from bytecode to intermediate representation (IR). In the described system, a secret key, which is part of the dispatch mechanism, is hidden from an adversary. Without the secret key, the permutations are constructed secretly. Without the permutations the mapping from bytecode to IR can not be reproduced.

The described compiler realizes an auto-key substitution cipher. This class of ciphers is on one hand more secure than the substitution cipher used by VMProtect, and, on the other hand, does not suffer from the performance penalties typical to the more secure AES algorithm.

As discussed by Goldreich [Gol86], some information can be gathered from memory accessed during program execution. The author proposes a way to hide the access patterns, thus not allowing an adversary to learn anything from the execution.

In an effort to continue improving the system performance we have considered using an efficient Low Level Virtual Machine (LLVM) [LA04]. Unfortunately modern virtual machines with efficient just-in-time compilers are unsuitable for software protection. Once the virtual machine loads the program, it allocates data structures representing the program which are stored unencrypted in memory. Since this memory can be evicted from cache at any time, these data structures become a security threat in a software protection system.

5.6 Assumptions in Modern CPUs

We posed two assumptions on the CPU that guarantee the security of the entire system. This section discusses the application of the system to the real world CPUs. In other words, we show how to use the facilities of modern CPUs to imply the assumptions.

Let us first recall the assumptions:

- The inner state of the CPU can not be read by the user.
- The CPU has a sufficient amount of internal memory.

As to the later assumption, we should first clarify the purpose of the internal memory. In essence this memory stores three kinds of data. The first one is the shared secret key. The second is the state of the virtual machine, specifically the current permutation and the decrypted instruction. The third kind of data is some parts of the kernel code and the VM code. The reason behind the last kind is less obvious, so consider the following attack.

An attacker lets the algorithm of Kennel and Jamieson to complete successfully on a standard machine equipped with a special memory. This memory can be modified externally and not by the CPU. Note that no assumption prohibits the attacker to do so. Just after the completion of the verification algorithm the attacker changes the memory containing the code of the VM to print every decrypted instruction. Clearly this breaks the security of the proposed system. Observe that the problem is essentially the volatility of critical parts of the kernel code and the VM code. To overcome this problem we have to disallow modification of the verified memory. Since the memory residing inside the CPU can not be modified by the attacker, we can assume it to remain unmodified.

Note that the first and the second kinds which hold the shared secret key and the VM's state should be readable and writeable, while the third kind which holds the critical code should be only readable.

On Intel CPUs, we propose to use registers as a storage for the shared secret key and the VM's state and to use the internal cache as a storage for the critical code.

To protect the key and the state we must store them in registers that can be accessed only in kernel mode. On Intel CPUs only the special purpose segment registers can not be accessed in user mode. Since these registers are special we can not use them for our purposes. However on 64-bit CPUs running 32-bit operating system, only half of the bits in these registers are used to provide the special behaviour. The other half can be used to store our data.

The caching policy in Intel CPUs can be turned on and off. The interesting thing is that after turning it off the data is not erased from the cache. Subsequent reads of this data return what is stored in the cache even if the memory has been modified. We use this property of the cache to extend the algorithm of Kennel and Jamieson not only to validate the code of the kernel and the VM before the key generation, but also to guarantee that the validated code will never change. Two steps should be performed just after installing the virtual memory mapping received from the distributor in verification algorithm: loading the critical part of the kernel and the VM into the cache and turning the cache off. This behaviour of Intel CPUs is documented in section 11.5.3 of [Int12].

The first assumption disallows the user to read the above-mentioned internal state of the CPU physically, i.e. by opening its case and plugging wires into the CPU's internal components. Other means of accessing the internal state are controlled by the kernel, and so are guaranteed to be blocked.

5.7 Performance

In this section, we analyze in details the performance of the proposed cipher. Throughout this section we compare our cipher to AES. This is due to recent advances in CPUs that make AES to be the most appropriate cipher for program encryption [Int10]. We provide both theoretical and empirical observations proving our algorithm's supremacy.

We denote the number of cycles needed to decrypt one word of length w using AES by α.

The last subsection, compares system version 7 to a regular, unprotected execution.

5.7.1 Version 3 Performance

Consider version 3 of the proposed system and assume it uses AES as a cipher. Every instruction occupies exactly one word such that n instructions can be decrypted in $n\alpha$ cycles.

5.7.2 Switch Instructions

As described above, the switch instruction is responsible for choosing the current permutation σ. This permutation is then used to decrypt the op-codes of the following instructions.

Some details were omitted previously, since they affect only the system's performance but do not affect its security or overall design:

- How does the argument of a SWITCH instruction encode the permutation?
- Where is the permutation stored inside the processor?

Before answering these questions we introduce two definitions:

Definition 5.7.1. *Given an encrypted program, we denote the set of all instructions encrypted with σ by I_σ and call it color-block σ.*

Definition 5.7.2. *Given a set I of instructions, we denote by $D(I)$ the set of different instructions (those having different op-codes) in I.*

The key observation is that it is enough to define how σ acts on the op-codes of $D(I_\sigma)$, which are instructions that occur in the color-block σ. Likewise, we noticed that some instructions are common to many color-blocks, while others are rare.

Denote by $F = \{f_1, f_2, \dots, f_\ell\}$ the set of the ℓ most frequent instructions. Let $R(I)$ be the set of rare instructions in I, i.e. $R(I) = D(I) - F$. The argument of SWITCH preceding a color-block σ has the following structure (and is encrypted by AES as described in version 5):

$$\begin{gathered}\sigma(f_1), \sigma(f_2), \dots, \sigma(f_\ell), \\ r_1, \sigma(r_1), r_2, \sigma(r_2), \dots, r_k, \sigma(r_k)\end{gathered}$$

where $R(I_\sigma) = \{r_1, r_2, \dots, r_k\}$. If σ acts on m-bits long op-codes, then the length of σ's encoding is $\phi = (\ell + 2k)m$ bits. Thus, it's decryption takes $\frac{(\ell+2k)m}{w}\alpha$ cycles.

5.7.3 Version 4 Performance

Consider a sequence of instructions between two SWITCHes in the program's control flow. Suppose these instructions belong to I_σ and the sequence is of length n. The VM Interpreter starts the execution of this sequence by constructing the permutation σ. Next, the VM Interpreter goes over all the n instructions, decrypts them according to σ and executes them, as described in version 5.

The VM Interpreter stores $\sigma(f_1), \sigma(f_2), \ldots, \sigma(f_\ell)$ in the CPU registers and the rest of σ in the internal memory. This allows decryption of frequent instructions in one cycle. Decryption of rare instructions takes $\beta + 1$ cycles where β is the internal memory latency in cycles. Denote by q the number of rare instructions in the sequence.

We are now ready to compute the number of cycles needed to decrypt the sequence:

$$\begin{aligned} \sigma \text{ construction:} \quad & \frac{(\ell + 2k)m}{w}\alpha + \\ \text{frequent instructions:} \quad & (n - q) + \\ \text{rare instructions:} \quad & q(\beta + 1) \end{aligned}$$

5.7.4 Comparison

On MIPS op-codes are $m = 6$ bits long and $w = 32$. The best available results for Intel newest CPUs argue that $\alpha = 14$ [Wik]. Typical CPUs' Level-1 cache latency is $\beta = 3$ cycles. The correlation between ℓ, ϕ and q is depicted in table 1.

We have noticed that most of the cycles are spent constructing permutations, and this is done every time SWITCH is encountered. The amount of SWITCHes, and thus the time spent constructing permutations, varies with the probability p of SWITCH insertion. The security, however, varies as well. Figure 27 compares program decryption time (in cycles) using AES and our solution with different values of p.

Note that on CPUs not equipped with AES/GCM [Wik], like Pentium 4, $\alpha \geqslant 112$. In this case our solution is even more beneficial. Figure 28 makes the comparison.

5.7.5 Version 7 Performance

The performance is affected mainly by the amount of function calls, since each call to a new function requires the function decryption. This performance penalty is reduced by allowing several functions to be stored in a decrypted form simultaneously.

Figure 26 compares the running times of the same program in 3 configurations and with inputs of different sizes. The program inverses the given matrix using Cramer's rule. The program consists of three functions computing determinant, minor, and finally inversion of a square matrix. The determinant is computed recursively, reducing the matrix order on each step by extracting its minor. We run this program on matrices of different sizes.

The three configurations of the program include:

1. Non-encrypted configuration — just a regular execution.
2. Encrypted configuration allowing two functions to reside in the cache simultaneously.

3. Encrypted configuration allowing a single function to reside in the cache simultaneously.

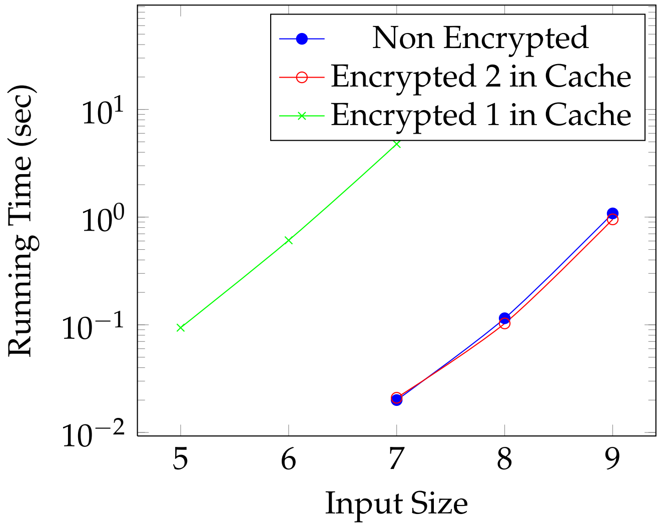


FIGURE 26 Just-In-Time Decryption Performance. Program Running Time in Seconds as a Function of Input Size. Note the Logarithmic Scale.

TABLE 1 Correlation Between ℓ, ϕ and q. Here $p = 0.2$

ℓ	0	1	2	3	4	5	6
ϕ	58	39	42	41	58	46	48
$\frac{q}{n}$	100%	70%	50%	34%	34%	26%	21%

5.8 Example

We provide a detailed example of a program passing through all the compilation phases. The original program is written in C. It computes the sum of the first 800 digits of π.

```
int main(){
  int a=10000,b,c=2800,d,e,f[2801],g,s;

  for(;b-c;) f[b++]=a/5;

  for(;d=0,g=c*2;c-=14,e=d%a){
    for(b=c;
      d+=f[b]*a,f[b]=d%--g,d/=g--,--b;
          d*=b);
        s += e+d/a;
```

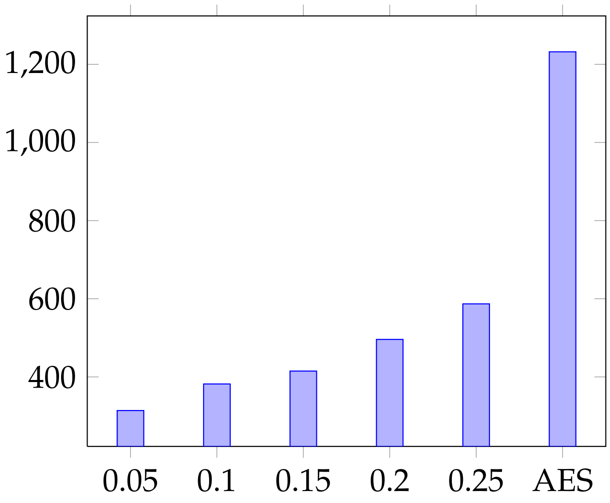


FIGURE 27 Program Decryption Time (in cycles) Using AES and our Cipher with Different Values of p. $\ell = 4$. $\alpha = 14$.

```
  }

  return s;
}
```

The corresponding output of the compiler is listed below. It is a combination of MIPS assembly and MIPS machine instructions. The leftmost column contains the instruction number. The second column contains the machine instruction. The rightmost column contains the assembly instruction. Block labels are emphasized.

Instructions of the form

```
 addiu $zero, $zero, ...
```

are an implementation of the SWITCH instruction. Colors correspond to the permutation that should be applied to the instructions.

For example, consider the instruction at 000c. It sets the current permutation to 5 (corresponding to gray color). As a result, all following instructions (up to the next SWITCH) are colored gray. Note that the instruction at 000c is not colored gray, since it should be encoded by the previous permutation (pink, corresponding to number 4 and set at 0005).

After the instructions are permuted according to their colors, the final phase takes place: the compiler encrypts special instructions' arguments using AES with the secret key.

$main (1)

```
0001 24000002 addiu $zero, $zero, 2


0002 24000003 addiu $zero, $zero, 3
```

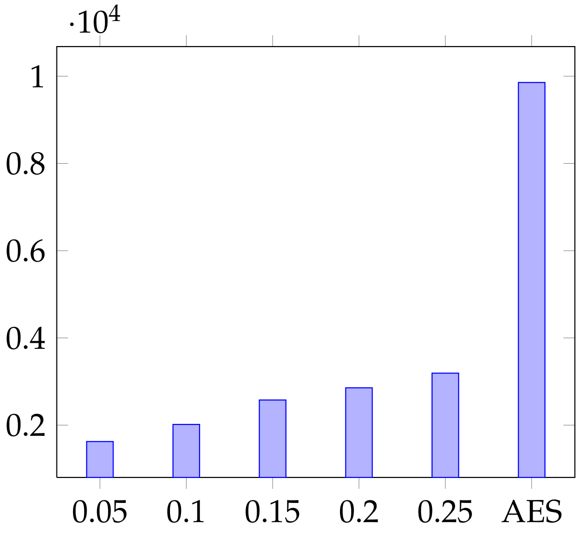


FIGURE 28 Program Decryption Time (in cycles) Using AES and Our Cipher with Different Values of p. $\ell = 4$. $\alpha = 112$.

```
0003 27bdd400 addiu $sp, $sp, -11264
0004 24022710 addiu $2, $zero, 10000
0005 24000004 addiu $zero, $zero, 4
```

```
0006 afa00010 sw $zero, 16($sp)
0007 24030af0 addiu $3, $zero, 2800
0008 afa20014 sw $2, 20($sp)
0009 afa3001c sw $3, 28($sp)
```

$bb0_1 (4)

```
000b 8fa20018 lw $2, 24($sp)
000c 24000005 addiu $zero, $zero, 5
```

```
000d 8fa3001c lw $3, 28($sp)
000e 00431023 subu $2, $2, $3
000f 10020000 beq $2, $zero, $bb0_4
```

```
0011 3c026666 lui $2, 26214
0012 34426667 ori $2, $2, 26215
0013 8fa30014 lw $3, 20($sp)
0014 8fa40018 lw $4, 24($sp)
0015 00620018 mult $3, $2
0016 00001010 mfhi $2
0017 00400803 sra $3, $2, 1
0018 24000006 addiu $zero, $zero, 6
```

```
0019 0040f801 srl $2, $2, 31
001a 27a50030 addiu $5, $sp, 48
001b 00801000 sll $6, $4, 2
001c 24840001 addiu $4, $4, 1
001d 24000004 addiu $zero, $zero, 4
```

```
001e 00621021 addu $2, $3, $2
001f 00a61821 addu $3, $5, $6
0020 afa40018 sw $4, 24($sp)
0021 ac620000 sw $2, 0($3)
0022 0800000a j $bb0_1
```

$bb0_3 (7)

```
0024 8fa20020 lw $2, 32($sp)
0025 24000008 addiu $zero, $zero, 8
```

```
0026 8fa30014 lw $3, 20($sp)
0027 0043001a div $2, $3
0028 00001012 mflo $2
0029 8fa30024 lw $3, 36($sp)
002a 8fa42bf8 lw $4, 11256($sp)
002b 00621021 addu $2, $3, $2
002c 00821021 addu $2, $4, $2
002d afa22bf8 sw $2, 11256($sp)
002e 8fa2001c lw $2, 28($sp)
002f 2442fff2 addiu $2, $2, -14
0030 afa2001c sw $2, 28($sp)
0031 8fa20020 lw $2, 32($sp)
0032 8fa30014 lw $3, 20($sp)
0033 24000009 addiu $zero, $zero, 9
```

```
0034 0043001a div $2, $3
0035 00001010 mfhi $2
0036 24000005 addiu $zero, $zero, 5
```

```
0037 afa20024 sw $2, 36($sp)
```

$bb0_4 (5)

```
0039 afa00020 sw $zero, 32($sp)
003a 8fa2001c lw $2, 28($sp)
003b 00400800 sll $2, $2, 1
003c afa22bf4 sw $2, 11252($sp)
003d 10020000 beq $2, $zero, $bb0_8
```

```
003f 8fa2001c lw $2, 28($sp)
0040 afa20018 sw $2, 24($sp)
```

$bb0_6 (5)

```
0042 8fa20018 lw $2, 24($sp)
0043 27a30030 addiu $3, $sp, 48
0044 00401000 sll $2, $2, 2
0045 00621021 addu $2, $3, $2
0046 2400000a addiu $zero, $zero, 10
```

```
0047 8c420000 lw $2, 0($2)
0048 8fa40014 lw $4, 20($sp)
0049 8fa50020 lw $5, 32($sp)
004a 00440018 mult $2, $4
004b 2400000b addiu $zero, $zero, 11
```

```
004c 00001012 mflo $2
004d 00a21021 addu $2, $5, $2
004e afa20020 sw $2, 32($sp)
004f 8fa42bf4 lw $4, 11252($sp)
0050 2484ffff addiu $4, $4, -1
0051 afa42bf4 sw $4, 11252($sp)
0052 8fa50018 lw $5, 24($sp)
0053 00a01000 sll $5, $5, 2
0054 0044001a div $2, $4
0055 00001010 mfhi $2
0056 00651821 addu $3, $3, $5
0057 ac620000 sw $2, 0($3)
0058 8fa22bf4 lw $2, 11252($sp)
0059 2400000c addiu $zero, $zero, 12
```

```
005a 2443ffff addiu $3, $2, -1
005b afa32bf4 sw $3, 11252($sp)
005c 8fa30020 lw $3, 32($sp)
005d 0062001a div $3, $2
005e 00001012 mflo $2
005f afa20020 sw $2, 32($sp)
0060 24000007 addiu $zero, $zero, 7
```

```
0061 8fa20018 lw $2, 24($sp)
0062 2442ffff addiu $2, $2, -1
0063 afa20018 sw $2, 24($sp)
0064 10020000 beq $2, $zero, $bb0_3
```

```
0066 8fa20018 lw $2, 24($sp)
0067 2400000d addiu $zero, $zero, 13

0068 8fa30020 lw $3, 32($sp)
0069 00620018 mult $3, $2
006a 00001012 mflo $2
006b 24000005 addiu $zero, $zero, 5

006c afa20020 sw $2, 32($sp)
006d 08000041 j $bb0_6
```

$bb0_8 (5)

```
006f 8fa22bf8 lw $2, 11256($sp)
0070 27bd2c00 addiu $sp, $sp, 11264
0071 03e00008 jr $ra
```

We discussed several steps toward software protection. Our system did not include obfuscation procedures beyond using a VM. Therefore, very little can be said about our system protection and performance compared to industrial products that use obfuscation procedures as a means of primary protection. Obfuscation is an integral component and may provide the bread and butter of many real life products. Since such measures may increase the challenges faced by software hackers, we make no claim regarding the vulnerability of such products.

Our system is designed only to mask the code of the software from a malicious user. The system can be used to prevent the user from either reverse engineering the "Game" or to create an effective key validation mechanism. It does not prevent access to the "Game" data which may be stored unprotected in memory. Even if the entire "Game" is encoded with this system, a player may still hack the "Game" data to affect his high score, credits or "lives". A different encryption mechanism, which can be protected by Truly-Protect, can be added to prevent it. This was not covered in the chapter. For similar reasons, the system cannot be used to prevent copying of copyrighted content, such as movies and audio, unless the content is also encrypted. It can be used to mask decoding and decryption properties.

An interesting side effect of the Truly-Protect system is in using similar technology in virus, spyware and other malicious software. If we run malware via encrypting virtual machine it may be very hard to identify the malware and protect the user against it.

While the system can indeed mask the virus software, there are three defenses that are still available to the user.

1. The software distributor of the VM has to be reliable.
2. The malware can still be detected by intrusion detection system and spyware removal tools that analyze its behavior.
3. Software protection relies on [KJ03] to prevent running on VMs. If the user is running on a VM the malware cannot run defeating its purpose by default.

6 LLVM PREFETCHING AND PRE-EXECUTION

We describe a system to speed up I/O bound scientific computation and number crunching. This system can be used to create a number of prefetch threads running in parallel to the main computation.

The system is designed to run on a specific environment of I/O bound computation with just one process running on designated number crunching machine. This system is not likely to be beneficial and may be harmful if the system is not I/O bound or if other processes are running on the same machine as the prefetching threads we run consume CPU power and may actually generate extra load on the machine.

6.1 Introduction

Today applications find ample amount of CPU power but suffer greatly from I/O latency. We propose to employ pre-execution to trade some of this CPU power for I/O latency improvement: the I/O relative instructions are extracted from the original program to separate threads. They are called prefetching threads. They are faster than the original since they contain less instructions, which allows them to run ahead of the main computation. We implemented LLVM-PREFETCH — a virtual machine equipped with a just-in-time compiler that is able to:

(a) Find the I/O instructions that are benefited most from pre-execution.

(b) Construct multiple pre-execution threads for parallel prefetching.

(c) Synchronize between the pre-executed thread and the main computation.

(d) Terminate threads that do not improve the overall performance;

(e) Optimize the pre-executed threads aggressively based on the runtime data. Our virtual machine derives from the Low Level Virtual Machine (LLVM) project.

6.2 Introduction

6.2.1 CPU–I/O Performance Gap

Recent advances in CPU technologies provide scientific applications with great power to achieve exciting results. However, I/O and disk technologies did not improve at the same pace. This was repeatedly shown in [Lud04, May01, SR97, Ree03, WG97, SCW05] among other sources proving that scientific computing performance suffers greatly from I/O latency and poor response time.

The root cause for the performance degradation comes from the improvement nature. While CPU rate and multi-core technology are developed at almost exponential rate (according to or close by Moore's law), I/O speed experiences linear improvement. Today's disks may be 2-3 times faster than disks manufactured 10 years ago while CPUs today offer 4 times more cores.

The notably slower improvement rate of disk performance when compared to CPU performance has created what is now known as the I/O wall or the performance gap.

There have been massive efforts in reducing the I/O latency and improving the I/O performance in multiple fields. In the physical level, multiple disks are now used in the RAID environment allowing linear improvement in read and write speed by the use of more spindles. Advances in network and in interconnect technologies with the introduction of high speed interconnects such as Infiniband [LMVP04], FCoE [DJ08], etc. have led to the use of caching and distributed systems. Distributed file systems, such as Google file system [GGL03], Linux's Ceph [Wei07], IBM's Global-Parallel file system (GPFS) [SH02], and other systems ([SKRC10, Clu02, CLRT00]). These systems use multiple disk storage from remote interconnected locations. This distributed technique can usually increase read and write throughput in linear proportion to the size of the cluster, however, it cannot improve access latency since the data is still stored on high latency disk drives.

Two physical approaches to reduce access latency are caching file pages in memory and new solid state disk drives. Caching refers to storing data in RAM instead of disk drive, typically where the data is located near the server cache. A common implementation of this method is memcached [Fit04], which is found in LiveJournal, Facebook and Wikipedia.

Using a caching server is a by product of a recent observation that for the first time since computer networks were developed, they provide faster response time than local disk. Caching allows for high performance and low latency access to random data but comes with a very high price tag. However, when a dataset is very large, a system to decide which files to load to the cache is still needed.

Using solid state disks is an interim solution for reducing access times that still bears a very high price tag. Solid state disks today are found only on high end servers and laptops and are yet to appear on mundane number crunchers. Furthermore, even solid state disk drives are inefficient when compared to the

CPU power of even a modest number cruncher.

I/O access patterns of a distributed memory caching system include a large number of small irregular accesses as shown in [KN94, Ree03]. Furthermore, the problem of I/O modeling, condensing multiple I/O requests to a single request and data sieving are handled in [TGL99, Ree03]. It is clear that in many cases, it is impossible to combine small I/O requests due to the nature of the application.

In this chapter, we focus on I/O prefetching — another commonly used technique to hide I/O latency by bringing I/O pages to memory long before they are needed. Several previous studies on prefetching [DJC$^+$07, EK90, May01, Pat97, Ree03, PS05] demonstrated performance gains and successes. However, as processor technology continues to improve and the performance gap widens, more aggressive prefetching methods are required.

Today, the CPU computation rate is often a million times faster than disk access time. CPUs today usually have several cores whose performance is measured in nanoseconds, while disk access time is still measured in miliseconds. A huge gap creates the necessity for more complex systems to predict and prefetch the pages needed for the running of the application. Furthermore, more complex prediction systems now have the opportunity to be successful in prefetching the required pages. This is due to the availability of 64bit systems, lower memory prices, OS support for buffer caches and the aforementioned technologies such as Infiniband and FCoE that provide much greater and cheaper I/O bandwidth.

6.2.2 Virtualization

Process virtual machines (VM) or application virtual machines run as normal applications inside the OS and support a single running process as opposed to a system virtual machines such as KVM [Kiv07] or Xen [BDF$^+$03] that run a complete system.

A VM is created when the process is created and then it is destroyed when the process is terminated. VM provides a platform independent programming environment and means to access storage, memory or threading resources.

VM provides high level abstraction for programming languages such as Java or C# via JVM [LY99] or CLR [Box02], respectively, or low level abstraction for a programming language such as C that uses LLVM.

Furthermore, in some cases the VM acts as an agent that provides additional services besides execution and platform independency. Such services may include distributed execution such as PVM [GBD$^+$94], byte code verification [LY99] and memory leaks and thread debugging [NS03].

6.2.3 Main Results

By considering all these new technology trends and observations, we propose a pre-execution prefetching approach that improves the I/O access performance. It avoids the limitation of traditional prediction-based prefetching approaches that have to rely on perceivable patterns among I/O accesses. It is applicable to many application types including those with unknown access patterns and random accesses.

Several researchers have proposed to implement the prefetching technique inside the pre-processor. However, this solution is either time-consuming, since an huge amount of code should be emitted and compiled, or requires a profile-driven compilation in order to locate the most critical spots where this technique is then applied.

Our implementation is part of VM: the program compilation time remains unchanged and no profiling phases are required. During the program's execution, VM locates the critical spots — program instructions that will benefit most from prefetching — and applies the prefetching technique to them.

The proposed approach runs several prefetching threads in parallel to utilize the CPU power more aggressively. This was enabled by a dependency analysis technique, which we call layer decomposition.

The prefetching threads are then optimized by eliminating rarely used instructions. This is achieved through program instrumentation carried by the VM during runtime.

The rest of the chapter is organized as follows: section 6.3 explains what "pre-execution" is and how it can help I/O-bound applications. Section 6.4 presents the means used to isolate prefetching threads from the main computation and the reason behind the need for such an isolation. Section 6.5 introduces LLVM, which is a framework we used for VM construction. The actual construction of the prefetching threads is described in section 6.6. A library, which was developed to support VM, is described in section 6.7. Sections 6.9 and 6.10 present the experimental results measured during the evaluation of our system. While the former section gives a broad overview of the results, the later provides a detailed analysis of a specific program execution. We review the related work in section 6.11. Conclusions are discussed in section 6.12.

6.3 Pre-execution

The runtime of a computer program can be seen as an interleave between CPU-bound and I/O-bound segments. In the CPU-bound segments, the software waits for some computation to be completed, and likewise, in I/O-bound segments, the system waits for the completion of the I/O operations [SGG08].

A well known idea is that if I/O-bound and CPU-bound segments can be executed in parallel then the computation time can be significantly improved. That is the key feature in the proposed approach: we would like the VM to automatically to overlap the CPU-bound and the I/O-bound segments via prefetching without the need for manual injections of functions like madvise(2).

Since we already discussed the performance gap between CPU and I/O, we can assume that more often than not the system runs I/O bound segments. We assume that during process runtime there is the available CPU power that can be used for prefetching. Furthermore, we can also assume that the I/O has significantly high latency while CPU has almost no latency so prefetching should also contribute to eliminate system idle time.

The pre-execution of code is done by additional threads that are added to the program by VM. This is called prefetching threads. The prefetching thread is composed of I/O only related operations of the original thread. The original source code is transformed by the Just-In-Time (JIT) compiler to obtain prefetching threads. The prefetching threads execute faster than the main thread because they contain only the essential instructions for data address calculation. Therefore, prefetching threads are able to produce effective prefetches for the main original computation thread. The prefetching threads are supported by the underlying prefetching library that provides normal I/O function calls for the prefetch counterparts. It collects speculated future references, generates prefetch requests and schedules prefetches. The prefetching library can also track function-call identifiers to synchronize the prefetching threads and the computation thread I/O calls and force the prefetching threads to run properly.

The proposed prefetching approach has many technical challenges that include generating accurate future I/O references, guaranteeing expected program behavior, constructing the pre-execution threads efficiently, synchronizing the pre-execution threads with the original thread as necessary and performing prefetching with a library support. Finally, everything should be done while the program runs. We address these challenges in the next sections.

6.4 Isolation of Pre-execution

This section explores various aspects of the prefetching threads construction main problem: How to prefetch precisely the same pages as the main computation without interfering with it? The actual method of thread construction is discussed in section 6.6.

The prefetching threads run in the same process and at the same time as the run of the main thread. Usually it is ahead of the main thread to trigger I/O operations earlier to reduce the access latency of the original thread.

This approach essentially tries to overlap the expensive I/O access with the computation in the original thread as much as possible. The main design considerations include two criteria: correctness and effectiveness. Correctness means that the prefetching must not compromise the correct behavior of the original computation. Since the prefetching threads share certain resources with the main thread such as memory address space and opened file handler, an inconsiderate design of the prefetching might result in unexpected results. We discuss in detail our design to guarantee that the prefetching does not disturb the main thread with regards to memory and I/O behaviors. This design provides a systematic way to perform prefetching effectively and to generate accurate future I/O references.

6.4.1 Memory Isolation

We do not guarantee the correctness of the prefetch threads in the sense that they can generate results that differ from the results of the main computation. Therefore, we have to prevent these threads to alter the state of the main thread through a shared memory.

Our method that deals with a shared memory is similar to [KY04, Luk04, CBS+08, ZS01]. We remove all store instructions from the prefetching threads to guarantee that the main thread's memory will not be altered by any of the prefetching threads, thus preserving from the main computation its correctness.

While this method alleviates the need for additional memory allocation and managing, it decreases the accuracy of the prefetching threads. This inaccurate behavior will not affect the correctness of the program though. It merely decreases the accuracy of the prefetching, and thus affects its effectiveness. We did not research other methods of memory management.

However, as we shall will in section 6.7, the library used by the prefetching threads can detect such as inaccurate behavior and terminates the misbehaving threads. In this sense, the proposed optimization technique does not never cause any harm.

6.4.2 I/O Isolation

To simplify the discussion and to focus on the methodology itself, we only deal with memory-mapped files. We made this decision based on the fact that memory regions are shared between all threads. Thus, by reading from files, prefetching threads can alter the state of the main computation.

The underlying prefetching library provides functionality to support the proposed approach. Specifically, it provides the PREFETCH function, which not only prefetches data but also synchronizes between the prefetching threads and the main computation.

We decided not to make strong assumptions regarding the operating system in order to increase the portability of LLVM-PREFETCH.

6.5 Introduction to LLVM

The proposed solution is heavily based on the Low Level Virtual Machine (LLVM). This section provides a short introduction to the most important elements in LLVM.

LLVM is a compiler infrastructure that provides the middle layers of a complete compiler system by taking intermediate representation (IR) code from the compiler and emitting an optimized IR. This new IR can then be converted and linked to the machine-dependent assembly code for a target platform. LLVM can also generate binary machine code at runtime.

LLVM code representation is designed to be used in three different forms: as an in-memory compiler IR, as an on-disk bytecode representation (suitable for fast loading by a Just-In-Time compiler) and as a human readable assembly language representation. This allows LLVM to provide a powerful IR for efficient compiler transformations and analysis while providing natural means to debug and visualize the transformations. The three different forms of LLVM are all equivalent.

LLVM supports a language-independent instruction set and type system. Each instruction is in a static single assignment form (SSA). This means that each variable called a typed register is assigned once and then frozen. This helps in simplifying the dependency analysis among variables. LLVM allows code to be left for late-compiling from the IR to machine code in a just-in-time compiler (JIT) in a fashion similar to Java.

LLVM programs are composed of "Module"s where each is a translation unit of the input programs. Each module consists of functions, global variables and symbol table entries. Modules may be combined together with the LLVM linker, which merges function (and global variable) definitions, resolves forward declarations and merges symbol table entries.

A function definition contains a list of basic blocks forming the Control Flow Graph (CFG) for the function. Each basic block may as an option start with a label giving the basic block a symbol table entry that contains a list of instructions and ends with a terminator instruction such as a branch or function return.

Every instruction contains a possibly empty list of arguments. While in the human readable assembly language, every argument is represented by a previously defined variable, the in-memory IR holds a pointer to the instruction defining this variable. Thus, the in-memory IR forms a data dependency graph, which is required by the slicing mechanism.

6.6 Prefetching Threads Construction

Prefetching threads as well as prefetching instructions can be inserted manually. Linux and most other operating systems support this feature while providing madvise(2) or an equivalent API. However, the process of inserting such instruction is difficult, time consuming and bug prone.

In this section, we present a prototype design of a VM that is equipped with a just-in-time compiler to address the challenges of constructing the prefetching threads automatically and efficiently.

We augment the LLVM program execution with the program slicing technique that was discussed in [CBS$^+$08, Wei84] to automatically construct prefetching threads. The program slicing technique was originally proposed for studying program behavior since it knows which results depend on other results and it can greatly assist in debugging and detecting a bugs' root cause. Nowadays, program slicing is a rich set of techniques for program decomposition, which allows to extract instructions relevant to specific computation within a program. Program slicing techniques rely on the Program Dependence Graph (PDG) [FOW87] analysis. This is a data and control dependencies analysis which can be carried out easily with LLVM.

The construction of the prefetching threads and choosing the code that runs in the prefetching threads is, in its essence, a program slicing problem because the prefetching threads are running a sub-set ("slice") of the original program where I/O instructions are of interest.

6.6.1 Hot Loads

Since we deal with memory-mapped files, I/O operations are disguised as memory accesses specifically "load" instructions. If we can find the memory accesses that cause the operating system to perform I/O operations, we will be able to reveal the instructions that require prefetching. Then, if we slice the original program with these load instructions and their arguments as slice criteria, we obtain all I/O related operations. These I/O operations and the critical computations might affect those I/O operations.

We call the load instructions that cause the operating system to perform most I/O operations "hot loads". In order to find the hot loads, we use a profiling technique. We instrument the original program with a counter for every load instruction. Just before the load instruction, we insert a code that asks the operating system whether the page accessed by the load instruction resides in main memory. If it does not, i.e. the operating system has to perform I/O operations to fetch it, the corresponding counter is incremented by one.

After some time, we can pause the programs execution and traverse the counters in order to find the load instructions that caused 80% of the I/O operations which are the hot loads. Other load instructions either are not fetched from a memory-mapped file or are rarely accessed. In any event, the benefit of prefetching the corresponding memory addresses is insignificant, thus making it unworthy to apply the prefetching technique to these instructions.

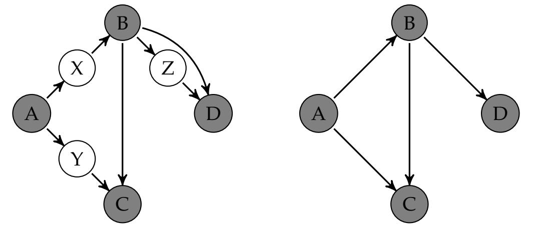
6.6.2 Slicing with LLVM

We begin this section with the definition of a slice.

Definition 6.6.1. *Consider a set I of instructions. A* slice *with respect to I is a set $S(I) \supseteq I$ such that if $i \in S(I)$ belongs to a basic block B then*

1. *All the i arguments are in $S(I)$;*
2. *The termination instruction of B is in $S(I)$;*
3. *If i branches, conditionally or not, to a basic block B' then the termination instruction for B' is in $S(I)$.*

It may seem that the first requirement is sufficient. Recall, however, that every basic block should end with a termination instruction. This is why we need the second requirement. Finally, if a basic block is empty it is automatically



(a) The graph G (b) An L-minor of G

FIGURE 29 (a) The original graph G. It has seven vertices. The subset L of G vertices contains the vertices A, B, C and D colored gray. (b) The L-minor of G.

removed, causing an error if some instruction branches to it. This is why we need to retain at least one instruction in these blocks, giving us the need to have the third requirement.

6.6.3 Threads Construction

We construct the prefetching threads by slicing with respect to a family of disjoint sets $L_1, L_2, \ldots, L_n$ of frequently used memory access instructions which are the hot loads. Note that if the result of some load instruction is never used in the slice then this load instruction can be replaced by a prefetch instruction which is a request from the prefetch library to fetch a page that contains the given address from the disk. The prefetch instruction does not fetch the page by itself but rather places an appropriate request on the prefetch queue. Then, the library later dequeues the request and fetches the page hopefully before it is needed by the main thread.

We introduce now several definitions:

Definition 6.6.2. *A graph $G = (V, E)$ is a data dependency graph of some function if V is the set of the function's instructions and $(x, y) \in E$ if y computes a value used (directly) by x.*

Specifically, we are interested in a subset of G vertices that correspond to the hot loads and in the relations between them.

Definition 6.6.3. *An L-minor of a data dependency graph G is the graph $G_L = (L, E)$ for $L \subseteq V(G)$ s.t. $(x, y) \in E$ if G contains a path from x to y not passing through the vertices in L besides x,y.*

Figure 29 presents an example of L-minor.

If we are fortunate to have a cycle-free minor, then we can decompose its vertices into disjoint sets such that the decomposition has a desirable property.

Definition 6.6.4. *Given a tree $T = (V, E)$. Its layer decomposition is a family of disjoint sets covering the vertex set $V = L_1 \uplus L_2 \uplus \cdots \uplus L_n$ constructed as follows: L_1 is the set of T leaves, which are vertices with indgree 0, L_2 is the set of $T - L_1$ leaves, L_3 is the set of $T - L_1 - L_2$ leaves and so on.*

Proposition 6.6.5. *Let $V = L_1 \uplus L_2 \uplus \cdots \uplus L_n$ be the layer decomposition of the tree $T = (V, E)$. For every edge $(x, y) \in E$ with $x \in L_i$ and $y \in L_j$ we have $i \leq j$.*

Proof Assume in contradiction that there is an edge $(x, y) \in E$ with $x \in L_i, y \in L_j$ and $i > j$. From definition the layer L_j was built before the layer L_i. Just before the construction of L_j, both x and y were present in the tree. Thus, the indegree of y was not 0. From the layer decomposition definition, $y \notin L_j$ which contradicts the assumption. □

Proposition 6.6.5 enables us to construct sets that guarantee that after some point in time, neither the main thread nor the prefetch threads will encounter a cache miss due to a hot load. Let G be the data dependency graph, L be the set of all hot loads and G_L is the G L-minor. Assume, meanwhile, that G_L is a tree and denote by $L_1, L_2, \ldots, L_n$ its layer decomposition.

Theorem 6.6.6. *There is a point in time after which no hot load encounters a cache miss.*

Proof All slices (their hot loads, to be precise) access the same page sequence $p_1, p_2, \ldots, p_i, \ldots$. Slices, which access a different sequence, are terminated by the prefetch library as described in section 6.7. We denote by $t_{i,j}$ the time at which page p_i was accessed by slice S_j. Note that S_{j+1} contains all the instructions of S_j and additional ones that compute the loads of the next layer. Moreover, the prefetch instructions of S_j may appear in S_{j+1} as load instructions. Thus, the slice S_j is faster than S_{j+1} by some factor, i.e. $t_{i,j-1} < t_{i,j}/c_j$ for some $c_j > 1$.

We assume that the computation is sufficiently long in the sense that for every slice S_j and a time t almost all pages are accessed by this slice after t. More formally, there exists an index r such that for all $i \geq r$, $t_{i,j} > t$.

Denote by T the maximal time needed to prefetch a page such that the following holds: if the page p was prefetched at time t_0 then no access to p will encounter a cache miss if the access occurs at time $t > t_0 + T$. This occurs under the assumption that no pages are evicted from the memory. This is guaranteed by the prefetching library as described in section 6.7.

We claim that every slice S_j that starts from some point in time does not encounter any cache miss. There exists r, which depends on j, such that for all $i \geq r$, $t_{i,j-1} > \frac{T}{c_j - 1}$ implying:

$$t_{i,j} > t_{i,j-1}c_j = t_{i,j-1} + t_{i,j-1}(c_j - 1) > t_{i,j-1} + T$$

. In other words, page p_i is accessed by slice S_j at least T seconds after it has been accessed by slice S_{j-1}. From the definition of T, page p_j resides in memory cache when accessed by S_j. By choosing the maximum between all such r, we observe that no thread encounters a cache miss on any page past and including p_r. □

If G_L is not a tree, then we compute the graph of its strongly connected components H. The layer decomposition is applied to H rather than to G_L, where for a strongly connected component C the meaning of $C \in L_i$ is that all its vertices are in L_i.

6.6.4 Optimizing the Termination Instructions

As can be seen experimentally, a termination instruction may introduce undesired instructions to the slice such as instructions that compute a result that is not used in the near future and thus may be omitted. However, omitting these instructions poses a dependency problem, since the result is an argument in the termination instruction. Before describing the solution, we explain first how to determine whether some result is likely to be used in the near future.

Recall that a basic block contains no termination instructions till its very end, thus if some instruction of a basic block was executed then all of them should be executed. By adding a counter to every basic block, we can determine for every instruction how frequently it is executed. If some termination instruction was not executed at all then we can (optimistically) assume that it will not be executed in the near future, thus we can omit all the instructions that compute its arguments.

After removing the unnecessary instructions, we can take care of the termination instruction arguments. Clearly these arguments should be removed where possible or replaced by some default value. The exact solution depends on the termination instruction type.

Note that this optimization can not be performed at compile time which makes our runtime approach beneficial. This method is similar to dynamic slicing [XQZ+05].

6.6.5 Effectiveness of Prefetching Threads

The prefetching threads are able to run ahead of the original thread and are effective in fetching data in advance to overlap the computation and the I/O accesses for the following reasons. As the previous discussion illustrates, the irrelevant code to I/O operations is sliced away, which makes the prefetching thread to contain only the essential I/O operations. Therefore, the prefetching I/O thread is not involved in enormous computations and runs much faster than the main thread. Secondly, the prefetch version of I/O calls are used within the prefetching threads to replace normal I/O calls. These prefetch calls are implemented with non-blocking accesses to accelerate the prefetching threads.

6.7 I/O Prefetching Library

This section discusses the design of the underlying library support for I/O prefetching. The goal of the library is to provide I/O functionality that is missing in the operating system. Although some operating systems do provide a partial implementation of the library's functionality, we chose to re-implement it in order to achieve a high portability of our system. It enabled us to run the system on Linux, Free BSD and Mac OS X with equal success.

The library provides only two functions: FETCH and PREFETCH. Both func-

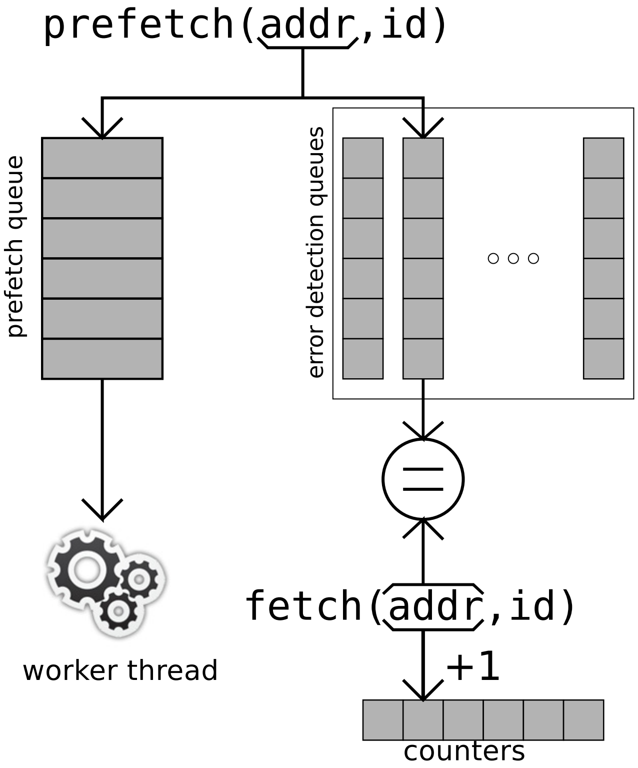


FIGURE 30 Prefetching Library. PREFETCH enqueues the address in the PREFETCH-QUEUE and in the error detecting queue. A worker thread checks periodically whether the queue is not empty. If so it dequeues the oldest address and accesses the corresponding memory. If a page is not cached, FETCH increments the counter that corresponds to the ID. Likewise, FETCH dequeues an address from an error detecting queue and compares it against its argument to reveal a prefetch thread that is misbehaved.

tions have two arguments: ID, whose meaning is explained later, and an address that should be either fetched (accessed synchronously) or prefetched (accessed asynchronously). Figure 30 demonstrates the internal organization of the library.

The PREFETCH function enqueues the address in the PREFETCH-QUEUE. A worker thread checks periodically whether the queue is non-empty. If so it dequeues the oldest address and accesses the corresponding memory.

Recall that the optimization is performed in two phases:

1. Finding the hot loads;
2. Running slices to prefetch the hot loads.

The FETCH function has a different goal during each of the two phases.

During the first phase, the FETCH function counts the number of cache misses encountered by each load instruction as follows. The main thread, which is the only one during this phase, invokes the FETCH function just before each of its load instructions, passing to it the memory address being accessed by the load instruction. The ID parameter we have mentioned previously is the unique identifier of the load instruction that invoked FETCH. The FETCH function asks the operating system whether the corresponding page resides in memory using POSIX's MINCORE system call. If the system call replies negatively, i.e. the page is not cached, FETCH increments the counter that corresponds to the ID. When this phase is completed, the counters are traversed to find the hot loads and the slices are constructed accordingly.

During the second phase, the function is looking for prefetch threads that are misbehaved. Note that for every hot load instruction, there is exactly one call to FETCH with some ID h and exactly one call to PREFETCH with the same ID h. The sequence of addresses passed to these functions should be exactly the same. A divergence of these sequences indicates that the corresponding prefetch thread is misbehaving. The library compares the two sequences as follows. It allocates a queue for each ID, which is a circular buffer that holds the addresses already passed to PREFETCH but not yet passed to FETCH. On every invocation, PREFETCH enqueues the address in the corresponding queue of the ID. FETCH dequeues the address from the corresponding queue and compares it against its argument. If the addresses are not equal then a misbehaved prefetch thread, which is the thread that enqueued the address, is found.

The last responsibility of the library is to synchronize between the main thread and the prefetching threads. Without synchronization, the prefetching threads will populate the memory too fast causing eviction of previously prefetched pages which were not yet accessed by the main thread.

The synchronization mechanism is simple: when the PREFETCH function is called to check whether the corresponding buffer queue has at least ℓ elements then if it is true it suspends the thread by waiting until the queue's size decreases. This mechanism guarantees that the hot loads and the corresponding prefetch instructions do not cause an eviction of pages that were prefetched but not yet used. Other load instructions can affect the page cache. However, the effect of these instructions is minor since they are invoked rarely. Thus, to overcome this problem,

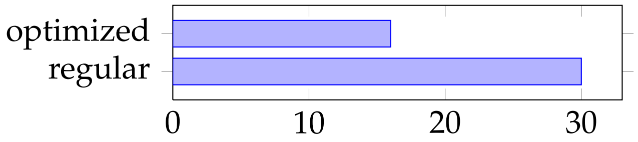


FIGURE 31 Running time (in seconds) of the optimized version of bucket sort algorithm versus the regular unoptimized algorithm.

it is enough to decrease a little the value of ℓ. Note, that page cache is system-wide and other I/O intensive applications might evict the prefetched pages. Thus, one should ensure that no other I/O intensive applications are running at the same time.

6.8 POSIX Implementation and Porting to other Platforms

POSIX and later versions of POSIX known as Single UNIX Specification are the standard to which UNIX systems adhere. FreeBSD, Linux and Mac OS X are all UNIX systems that implement the POSIX API.

The POSIX API includes the system call MINCORE which we rely on. The MINCORE system call provides information about whether pages are core resident, i.e. stored in memory and access to those pages does not require I/O operations.

Provided the existence of MINCORE, it is relatively simple to port our software to multiple platforms.

6.9 Experimental Results

We performed experiments to verify the benefits of the proposed prefetching technique. The conducted experiments prove the necessity of having different components of the proposed system. Some of the experiments were based on synthetic programs that nevertheless represent a large class of real-life applications.

The simplest program BUK in our benchmark implements an integer bucket sort algorithm, which is part of the NAS parallel benchmark suit [BBLS91]. This program contains a single memory access instruction executed in a loop. A single pre-execution thread is constructed to prefetch the data ahead of the main computation. The obtained results demonstrate as shown in Fig. 31 that our implementation of prefetching is effective in this case.

Next we want to show that the optimization should be applied only to those memory accesses that cause many cache misses specifically called the "hot loads" in our terminology. Other memory accesses have insignificant impact on the overall program performance. Even a total removal of these instructions will

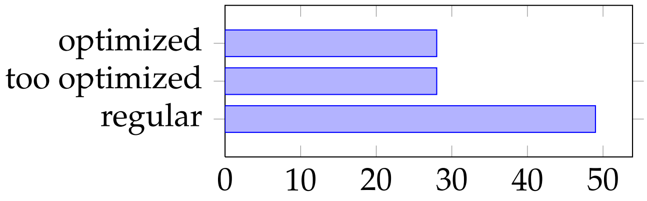


FIGURE 32 Running time (in seconds) of the matrix multiplication algorithm.

not reduce the running time notably. This is demonstrated by the matrix "multiplication" program. Formally, the program computes

$$\sum_{i=1}^{N}\sum_{j=1}^{N}\sum_{k=1}^{N} f(a_{ik}, b_{kj})$$

where a_{ij}, b_{ij} are elements of the matrices A and B, respectively, stored on a disk row-by-row. Clearly, most cache misses are encountered due to accesses to B, thus the optimizer constructs a pre-execution thread, which prefetches only the elements of B. We compare this behaviour to a regular execution with no prefetch threads and to an execution in which the elements of A are prefetched as well. The obtained results demonstrated in Fig. 32 exhibit that our strategy is optimal and it is not worse than the fully optimized execution. The figure demonstrate The running time of both programs executed regularly (without optimizations), with an unnecessary optimizations on both memory accesses and with a smart optimization of a problematic memory access (our approach).

The main innovation in this chapter is the construction of parallel pre-execution threads for prefetching. The third program in our benchmark shows the benefit of using the parallel prefetching approach. This program compute

$$\sum_{i=1}^{N} a_{b_{i \cdot P}} c_i$$

where a, b and c are arrays of N elements, P is the page size of the machine and all the indices are computed modulo N. The program computes the products sums of $a_j \cdot c_i$ where $j = b_{i \cdot P}$, i.e. the elements of b contain the indices of the elements in a that should be multiplied with the elements of c. We have three memory-access instructions α, β and γ that access the arrays a, b and c, respectively. Clearly, only every P invocation of γ encounters a cache miss in comparison to every invocation of β, and, probably, to every invocation of α. This is true with high probability for random b and large N. The analysis suggests that α and β should be pre-executed and γ should be ignored. Since the *addresses* of α depend of the *values* of β, the proposed system constructs two pre-execution threads: the first issues prefetch requests with the addresses of β while the second uses the values of β (which should be prefetched by the first thread) to compute the addresses of α and to issue the corresponding prefetch requests.

We examine differences among the performances of four systems:

1. Regular execution with no prefetch threads;

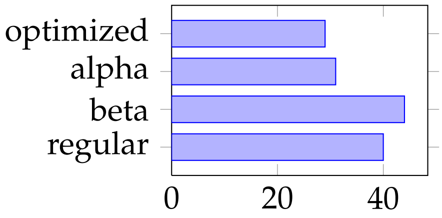


FIGURE 33 Running time (in seconds) of the indirect array access program (on average for single iteration).

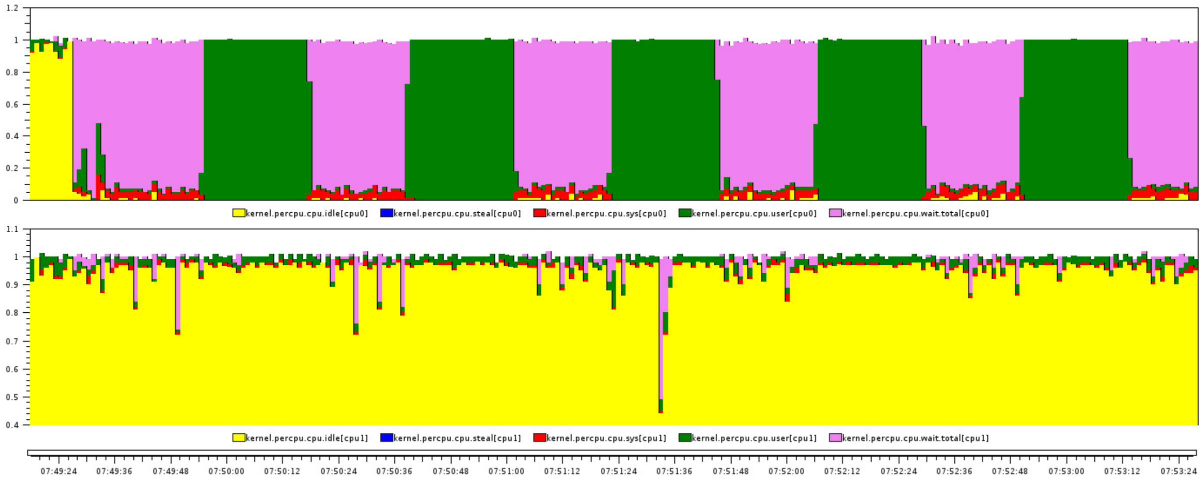


FIGURE 34 Regular non-optimized execution of the matrix multiplication program. Only one core is utilized. Yellow means that the core is idle, green means that the program calculations are performed, cyan means that the I/O operations are performed, red means that the OS calculations are performed. The green sections are the CPU-bound segments and the cyan sections are the I/O bound segments.

2. A single prefetch thread that prefetches β;
3. A signal prefetch thread that executes β and prefetches α;
4. The proposed optimization (having two prefetch threads).

Figure 33 compares between the four executions under consideration.

1. A non-optimized (regular) execution;
2. An execution which does not prefetch β but prefetches α;
3. An execution which does not prefetch α but prefetches β;
4. Finally, the proposed execution which prefetches both α and β.

As can be seen in Fig. 33, parallelizing the pre-execution improves the performance when sufficient computational power is available.

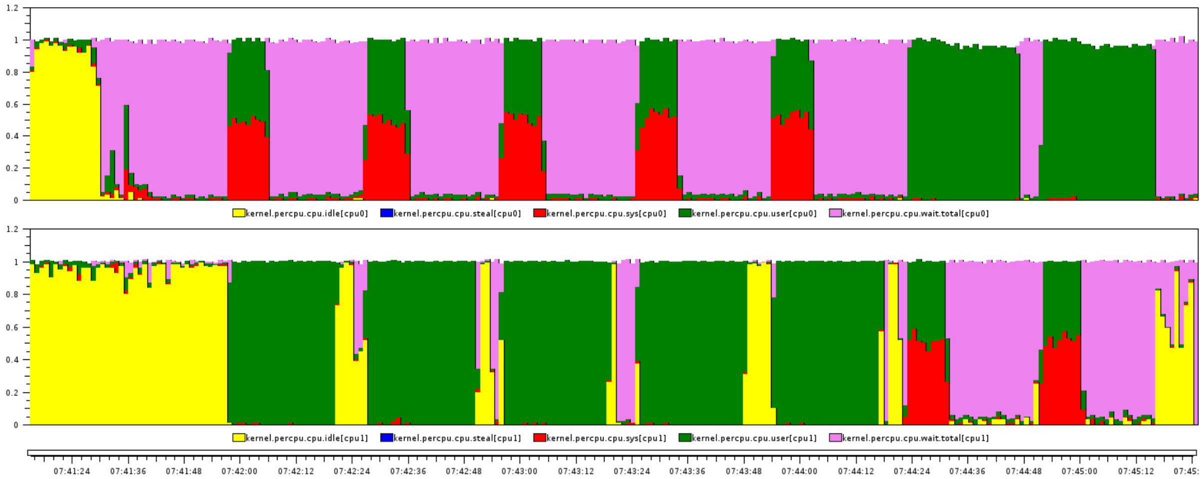


FIGURE 35 Optimized execution of the matrix multiplication program. The execution begins with the main thread occupying the first core. The prefetching threads start after 5 seconds and replaces the main thread. The main thread runs on the second core for about two minutes. Then, it is swapped with the main computation. Yellow means that the core is idle, green means that program calculations are performed, cyan means that I/O operations are performed, red means that OS calculations are performed. The green sections are the CPU-bound segments and the cyan sections are the I/O bound segments.

6.10 Case Study: Matrix Multiplication

We have run massive matrix multiplication benchmarks on an intel Quad Core i7 3.06 Ghz with 24 GB of RAM. The disks that were used in the benchmark were Intel SSDs.

The matrix multiplication program demonstrates clearly the problem we are trying to solve and the achieved performance. Essentially, our approach tries to overlap the I/O-intensive program sections with CPU-intensive sections, by utilizing an additional CPU core.

A typical program, in our case, the matrix multiplication program, is described in section 6.9, is in general neither I/O-intensive nor CPU-intensive. Its intensiveness type varies over time. As can be seen in Fig. 34, the program is broadly half of the time I/O intensive and half of the time CPU-intensive. Since the program is single-threaded, the second core is mostly idle. It executed some background programs for example PCP Charts used to create graph and other OS services.

Pre-execution uses the idle core to pre-execute the main computation and prefetches the required data. Figure 35 shows CPU utilization during the optimized execution of the matrix multiplication program. The execution begins where only the main computation is executing on the first core for 5 seconds. During these 5 seconds, the optimizer collects profiling data for the optimization. After these 5 seconds, the pre-execution thread is constructed and starts its execution on the first core. The main computation moves to the second core. The two threads swap cores after about two minutes. The time axis can be divided into 30 seconds segments (intervals) of CPU-intensive sections of the main computation. Note that these segments start with a CPU-intensive section in the prefetching thread followed by an I/O-intensive section. In other words, the CPU-intensive section of the main computation overlap (almost entirely) with the I/O-intensive section of the prefetching thread.

6.11 Related Work

Initial work on prefetching in LLVM is described in [SH]. This work includes a prefetch instruction similar to MADVISE in Linux but does not include any prediction or slicing.

Outside the virtualization realm, there has been a great effort invested in prefetching which can be classified into two main approaches: speculative execution and heuristic based prediction as discussed in [May01]. Our approach includes aspects of both.

The heuristic approach for prefetching uses observed patterns among past history of I/O requests to predict future requests. Naturally, heuristic prefetching can only work if the application follows regular and perceivable patterns in its I/O requests. When the application I/O requests follow random or unknown patterns then obviously heuristic prefetching cannot improve the application performance.

The speculative execution approach for prefetching is a more general approach. Theoretically, speculative execution can work for every application and if done correctly it has a good chance to circumvent future I/O originated lags.

Our approach, which uses slicing and prefetching threads, is a speculative execution approach while some decision making (in the case of termination instructions' argument removal) is of heuristic origin.

Other speculative execution approaches for prefetching include the SpecHint [Cha01] system and the prefetching TIP system [PGG$^+$95].

Both SpecHint and TIP approaches demonstrate the feasibility of accurate speculation of future I/O requests ahead of time in order to provide the information for the underlying system so that I/O will be prefetched in advance. Both of these methods are relatively conservative in terms of the number of CPU cycles dedicated to prefetching. In comparison, our approach is significantly more aggressive and provides better prefetching results.

The aggressive pre-execution approach has also been studied extensively in [CBS07, KY04, ZS01, Luk04] to reduce memory access latency to attack the "memory wall" problem. Similarly to our system, these approaches contain source code transformation and prefetching instruction injection. However, in our case, the prefetching instructions are all inserted in separate threads.

6.12 Conclusion

The performance gap between CPU and I/O is already very significant and there are no indications that it will be either eliminated or improved in the near future.

As long as disk performance is so far behind CPU power, I/O performance will have significant effect on computation run time. As a result, more aggressive and complex measures for I/O prefetching will be required.

Furthermore, programmers wages continue to be high. Thus, manual insertion of prefetching instruction into the code is expensive.

LLVM-prefetch addresses the performance gap issue by overlapping computation with future disk access pre-executed in parallel.

The main contributions of the system described in this chapter are:

1. We propose an innovative pre-execution approach for trading computing power for more effective I/O use. This approach allows the VM to pre-execute operations and automatically prefetches the needed pages without programmer intervention;
2. We present system implementation benchmarks. It was tested and its performance is compared to naive running of the program and to specific running of the same program after prefetching instruction insertion by the programmer;
3. The system was implemented for LLVM;
4. A careful design consideration for constructing the pre-execution thread and a VM with JIT compiler for automatic program slicing is presented. This system can later be re-implemented for other environments such as JVM, CLR, etc;
5. The proposed environment was tested on several popular UNIX systems.

The above described approach shows a great promise, especially with the recent trend in using process virtual machines. Decreasing I/O latency provides great improvements in computation times and allows the VM to do so automatically and it decreases the cost of man power. The experimental results confirm that the proposed approach is beneficial and has real potential to eliminate I/O access delay, expedites the execution time, and improves system performance.

7 APPLICATIONS IN PEER-2-PEER STREAMING

We describe here a system that builds peer-2-peer multicast trees. The system proposes a unique algorithm that incorporates a real time, priority based scheduler into graph theory. The system includes robust implementation that supports multiple platforms. The system proposes a fine example of the power of cloud computing and to network virtualization.

Special considerations were given to conditional access and trust computing concerns.

The algorithm described in section 7.6.2 is a joint effort with Amir Averbuch and Yehuda Roditi.

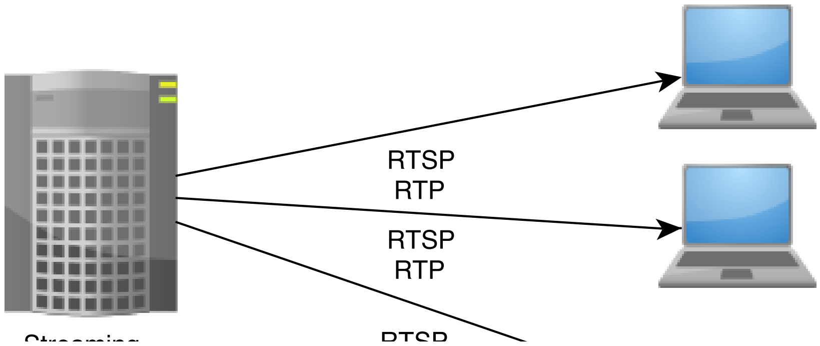
Implementations details are given in Appendix 7

7.1 Introduction

The bandwidth cost of live streaming prevents cost-effective broadcasting of rich multimedia content to Internet users.

For Internet streaming, the old-fashioned client-server model puts a huge cost burden on the broadcaster. In the client-server model, a client sends a request to a server and the server sends a reply back to the client. This type of communication is at the heart of IP[Pos81a] and TCP[Pos81b] protocol, and most of UDP[Pos80] traffic as well. In fact almost all upper layers of communication such as HTTP[FGM+99], FTP[Bhu71], SMTP[Pos82] etc and the client-server model contains this type of communication well. The client-server communication model is known as unicast where a one-to-one connection exists between the client and the server. If ten clients ask for the same data at the same time, ten exact replicas of the same replies will come from the server to each of the clients as demonstrated in figure 36. This model remains the same regardless of the number of concurrent requests from the same number of unique clients, placing additional stress on the server with each additional user.

Furthermore, The problem exists to a much greater extent in live stream-



FIGU

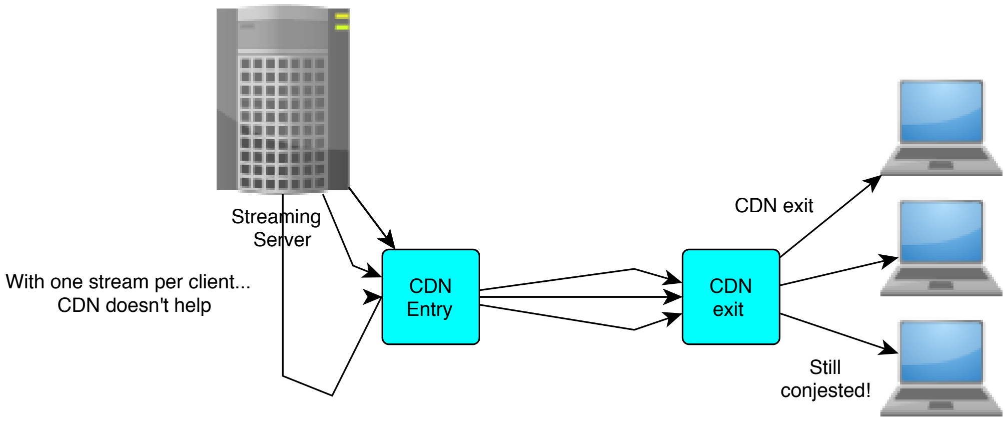


FIGURE 37 CDN approach does not provide the solution for last mile congestion.

ing scenarios with large crowds of listeners such as sport events etc. as caching techniques such as proxies do not work with live streaming.

These problems also arise even when Content Delivery Networks (CDNs) are used for replicating static content to other servers at the edge of the Internet. Even when CDNs are used every client is still served by one stream from a server , resulting in the consumption of a great amount of bandwidth (see figure 37). These infrastructure and cost challenges place a significant hurdle in front of existing and potential Web casters. While the media industry is seeking to bring streaming content with TV-like quality to the Internet, the bandwidth challenges often restrict a feasible, profitable business model.

In order to reduce the dependence on costly bandwidth, a new method of Internet communication was invented called "multicast". Rather than use the one-to-one model of unicast, multicast is a "one-to-selected-many" communica-

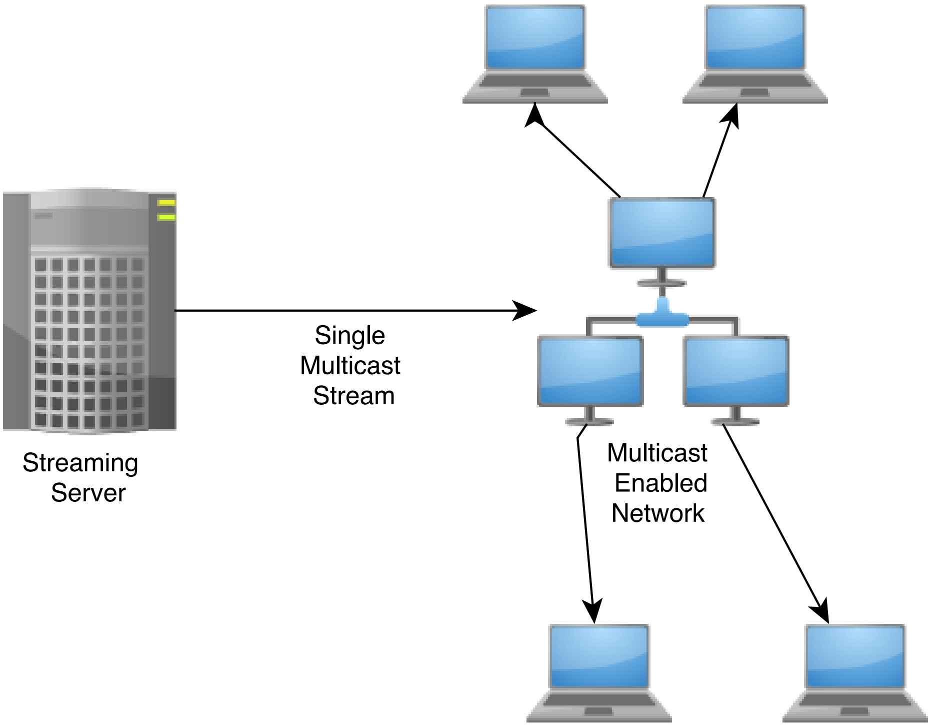


FIGURE 38 Multicast streaming could provide a solution.

tion method. However, multicast is not available on the current Internet infrastructure IPv4 and may never be available outside private networks. An example of how multicast streaming looks like is demonstrated in figure 38.

A solution commonly proposed is to operate Internet users as "broadcasters" using peer–2–peer[Zai01, FL08, Liu07] connections as an ad-hoc CDN.

In this chapter, we describe our system that implements peer-2-peer streaming and our algorithm that handles network events.

7.2 System design

The software industry has already anticipated the need for cost-effective, high-quality streaming and has developed applications that support multicast. Our peer-2-peer streaming system, called PPPC (Peer-2-Peer Packet Cascading) bypasses the lack of multicast in IPv4 internet by providing multicast-like capabilities via peer-2-peer, and allows use of the already available multicast software.

The concept of peer-2-peer streaming is a distributed architecture concept designed to use the resource of a client's (a home user with desktop computer or laptop) upstream in order to alleviate congestion in the broadcaster streaming server. Using the client upstream doesn't affect the client ability to surf or download files. The upstream resource is usually idle for most clients not involved in peer-2-peer activity, such as bittorrent [Coh03] or e-donkey.

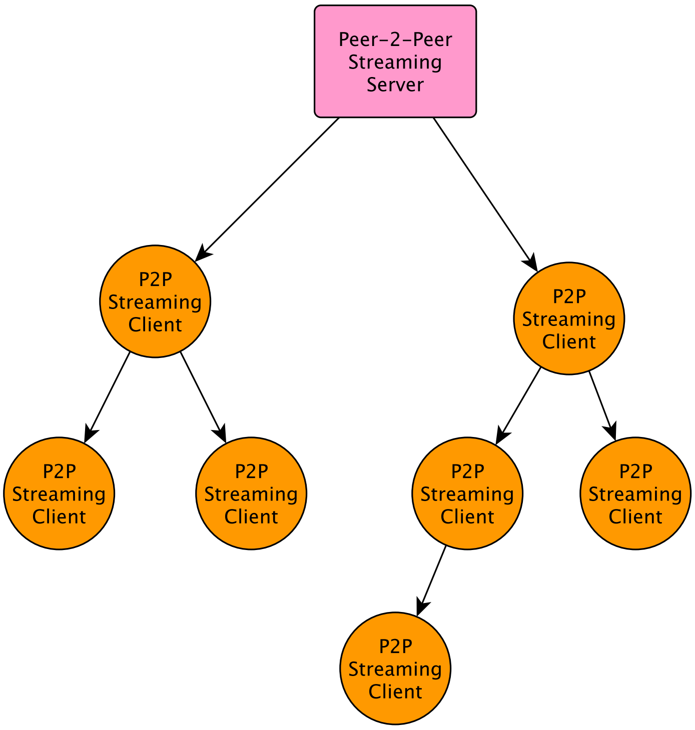


FIGURE 39 Only peer-2-peer streaming solves streaming problem on the Internet.

In a peer-2-peer streaming system, the server only serves a fraction of selected simultaneous clients requesting the same stream and turns them into relay stations. Hereafter, the other clients who are requesting the same data will be served from one of the clients who received the stream first.

The clients shall only maintain a controlled connection to the server for receiving control input and reporting information. Also, we shall use every client as a sensor, to detect stream rate drops, reporting the problem, and complementing the missing packet from either the PPPC router or another client. It is vital to detect any streaming issues in advance before the Media Player has started buffering or the viewer has noticed anything. Therefore, by following Peer-2-peer streaming concept and serving a fraction of the users, the server can serve a lot more users with the same bandwidth available. This is shown in figure 39.

Peer-2-peer packet cascading, or PPPC, is an implementation of the peer-2-peer concept to the streaming industry. PPPC provides reliable multicasting protocol working on and above the standard IP layer. A PPPC system consists of the PPPC router. and the PPPC protocol driver. The PPPC router stands between a generic media server, such as MS Media Server, a Real Media server or a QuickTime server, and the Viewers. (see figure 40).

The PPPC driver is in charge of the distribution of data and the coordination of the clients.

In an PPPC live stream, the PPPC router will receive a single stream from the media server and will route it directly to several "root-clients". These clients will then forward the information to other clients and so on and so forth. Users

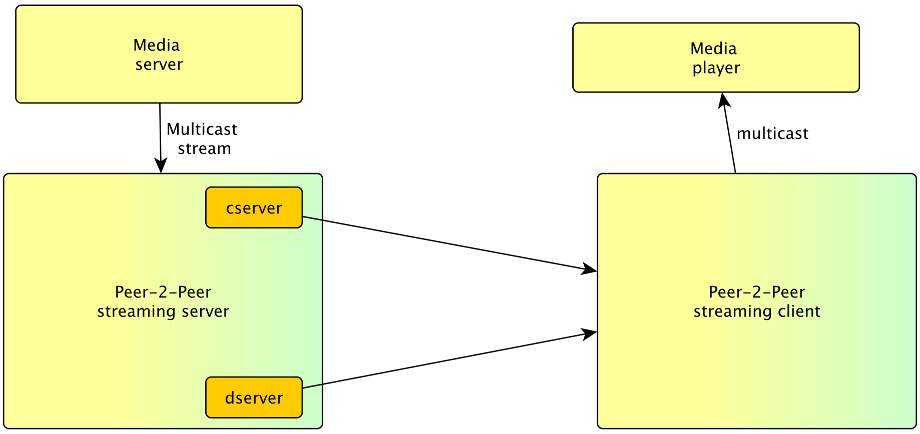


FIGURE 40 PPPC data flow.

connecting to each other will be relatively close network-wise. In this method, the data is cascaded down the tree to all users while the PPPC router only serves directly (and pays bandwidth costs) for the root clients. As users join and leave, the trees are dynamically updated. Moreover, the more users join the event, the PPPC router can build better trees saving even more, eliminating the financially terrible linear connection between cost of streaming and number of users.

7.3 PPPC System Overview

Peer-2-Peer Packet Cascading is a system designed to provide audio and video streaming clients with the capability to receive data from other clients and relay other clients to them. PPPC system consists of the PPPC router and PPPC Driver. The PPPC router contains two logical components the Coordinating Server (CServer) and Distributing Server (DServer).

PPPC driver installed on a client workstation (any standard PC) consists of thin client software that handles the reception and relay of the packets, and also "feeds" them to any Media Player. The client does not interact with a media player, it only delivers packets to the media player. **Coordinating Server** (CServer), is a command and control system in charge on all PPPC drivers listening to a single stream. CServer is responsible for all the decisions in the system. For example, for a given client, from which client it should receive data, and to which client should it transfer data, how the tree should be rebuilt after a new client arrived, what to do if a client in the middle of the tree was disconnected, and what happens when any given client reports he has problems with receiving stream from his parent. **Distributing Server** (DServer) is a data replication and relay system. DServer receives a multicast (data-only) stream and encapsulates the data in PPPC format (recognized by PPPC driver). DServer delivers the encapsulated packets to the roots of PPPC clients' trees (root clients). The CServer decides who are the root clients.

7.3.1 Data flow in the PPPC system

In a PPPC system, PPPC router must receive a data-only stream (i.e. no meta-data) from a generic media server and is responsible for delivering the stream to clients. In some ways, a PPPC router acts very much like a multicast capable router. (Data-only stream is required because a stream with meta-data will require the decoding of the stream and the right meta-data to be sent to each of the clients thus missing the system generality goal.)

Most standard media servers can provide data only stream, either directly or via a "multicast" option. DServer in our system will receive the multicast stream or other data only stream and pass it forward to the root clients. The PPPC drivers running on root clients workstations pass the stream to other drivers on other clients. Therefore, each client acts as a small server, reusing DServer code for this purpose.

When a PPPC driver, regardless of whether the PPPC driver also forwards the stream to other clients, receives the stream, it will forward it to the media player pretending to be the original media server using multicast or fake IP if necessary. This is not real network traffic, but local traffic on local host blocked in the kernel. Then, the media player will receive the stream and will act as if it received the stream directly from the media server. The PPPC driver will send a stream just like a media server.

Thus, media server sends a standard (usually multicast) data stream, and media player receives a standard stream. This enables the system to work with any media server, any media player and any codec etc, without the need to have any specific integration.

7.3.2 Detailed description of the DServer, CServer and theirs components

One instance of the server handles one media stream. Multiple instances of the server are possible in order to handle more than one stream. Parts (entities) within the Server communicate with each other by TCP enabling them to run on several computers.

7.3.2.1 Distributing server (DServer)

The Distributing Server transfers the stream contents to root clients and serves as a backup source for clients (DServer is also a backup server). It contains two physical components:

- A single **Receiver**, which gets the raw data from media server via Multicast or UDP. The DServer Receiver then encapsulates the data arriving from the Media Server in PPPC packets (recognized by PPPC clients.)
- One or more **distributors** which receive the information directly from **receiver** and serve the root clients.

The distributors share packet relay and connection code with the drivers, but they are the only entities that are allowed to receive the stream directly from the receiver.

The separation of DServer components to **receiver** and multiple **distributor**s is suggested in order to provide optimal scalability and allows the deployment of the distributors across several CDN sites.

7.3.2.2 Coordinating Server (CServer)

The Coordinating Server maintains the control connection with every client. It decides which clients connect between them, i.e., it constructs the PPPC tree. Our algorithm is implemented within the CServer. CServer updates dynamically the PPPC tree on such events as connection/departure of clients, unexpected disconnections, etc.

CServer, similar to the DServer, also contains two components:

- A single centralized **main** module, where all the users (of a single stream) data is saved. **Main** module provides all the logic and decisions to the system.
- One or more **Proxy** who receives client connections and pass requests and responses to/from CServer's **main** module.

In a large scale PPPC system, where several proxies can exist each maintains a connection to a large group of clients. **Main** module is the only place where complete information and decision making regarding all clients is kept for decision making regarding the clients tree. Reports on clients connections and disconnections are handled by the **main** module.

The PPPC driver is a very light client which consumes very little system resources apart from the relatively free upstream.

7.3.3 Viewing a stream with PPPC - life cycle

This life cycle assumes that the clients select the stream using WWW.

1. The user accesses a page on WWW which provides him with stream information.
2. The file is silently downloaded to the user's computer.
3. PPPC driver parses the file which includes a standard media-player activation file. PPPC driver reads CServer and DServer IP address as well as other parameters and invokes the standard media player to handle the stream.
4. The client connects simultaneously to the CServer and DServer.
5. DServer sends data to the client which is immediately displayed to the user.
6. In a certain event[1] a CServer decides to rebuild the PPPC client trees.
7. CServer sends to the client messages with information about its new stream source (another client or the DServer) and possibly address of other clients that should be served the stream.

[1] For example, after some other new clients arrived, or old clients disconnected.

8. A client connects to specified clients and starts to receive information from the parent and relays it to its children. The arrival of data is viewed through the Media Player to the user.
9. CServer may decide during the operation to rebuild the tree and sends again the corresponding messages to the client, which disconnects its older connections and creates newer ones.
10. When the user decides to stop viewing the stream, the PPPC client recognizes it and sends the message 'I'm going to disconnect" to the CServer and quits. Meanwhile, the CServer updates the clients' tree if necessary.

7.3.4 Maintaining a Certain Level of QoS

In order to maintain and guarantee a certain level of QoS we will add a stream rate detection unit to every client. The stream rate is published by the CServer when clients join the stream. If a client detects that the stream rate has dropped below a certain level, he will be connected to DServer to complement the missing packets or as an alternative stream source. Numerous reasons cause the packet rate to be dropped: parent disconnection (packet rate drops to zero), sudden drop in packet rate when the parent used his upstream to send an email, or a high CPU load on the parent machine. He might also report that his previous parent was a "bad parent", then the CServer will not assign new children to a "bad parent".

Switch between parents and going to DServer should be done very fast (within the streaming buffer time found in the client). If all packets arrived before the buffer expire, the user will never notice the switch between the parents.

We will describe the exact method in which "bad parents" are detected in section 7.5.

7.4 Avoiding Flat Trees, Distress and Guaranteeing Certain Level of QoS

In this section we describe all the system engineering issues which are connected to the appearance of what we call "flat trees". Flat trees are trees that have a very large number of root clients compared to the total number of clients and very small number of peer-2-peer connections. We will also describe how these issues are solved.

We encountered several reasons for having extremely flat trees, most of them were related to our goal to achieve a high level of QoS. This section describes our solution, which provides high streaming quality to clients who can receive the stream. This is done while we maintain peer-2-peer connection with a high bandwidth saving ratio. We realized that QoS and the flat trees problem are closely connected. Several problems have been identified:

- Parents that can not serve clients constantly received new clients which caused decrease in QoS.

- Parents which declared bad parents, never received new clients and caused to have flat trees.
- Clients that were unable to view the stream pass from parent to parent declared them all to be bad parents (hereby bad clients). Such a client can easily mark all clients in a tree as bad parents which will surely result in a flat tree.
- In case of a transmission problem in the upper layers of the tree, many clients in lower layers of the tree did not receive the stream and report their parents to be bad parents. This causes to multiplicities of bad parents.
- Clients that detected problems are usually not the direct children of the clients that caused the problem. Thus, many of the clients were declared to be bad for no apparent reason. (Same as above!).
- Due to the above conditions lower layers of the tree received poor quality stream.

As we can see from above, most of the problems occurred due to faulty behavior when served by an unsatisfying packet rate. We shall hereby call this situation *distress*.

7.5 Bad Parents & Punishments

When a client reports to the CServer that his parent does not provide him with sufficient packet rate, the CServer will mark the parent as a bad parent. In this case the bad parent's maximum children number is set to its current children number.

The client, that reported the bad parent, will also connect to the DServer either to compliment the missing packets or to replace its current bad parent with DServer. Therefore, bad parent cannot have any more children. We will not disconnect any of the other children he already had. We will allow new children to replace one of the old ones if they were disconnected. If previous client did not have any children then he will have no children anymore.

We "punish" bad parents so harshly to prevent any connection of new clients to them. Thus, we avoided a situation where a client connected to several "bad parents" before receiving the stream. Thus, the QoS was degraded.

The isolation of bad parents plays a very important role in guaranteeing high QoS. We realized that a stream is never disrupted in real world scenarios by the sudden disconnection of parents or fluctuations in their upstream. However, bad parents were often one of the main reasons for having flat trees. Clients could not find themselves a suitable parent, because all possible parents were marked as bad parents and could not accept any new children. This frequently happened when a client had faulty uplink and reported several parents as bad or when we streamed radio stream (audio only) and the users used their computers for other purposes while listening. These other usages often caused high cpu load or temporary use of the uplink and thus temporary faulty stream.' Therefore, we gave a chance for a bad parent to recover. We set a punishment time stamp where the punishment time was assigned to each of the bad parents. To recover from this situation we introduce bad parents rehabilitation process (see section 7.5.1). There are many temporary situations such as sending and e-mail which hogs the upstream, starting Microsoft Office, which causes a CPU surge for couple of seconds, and many more. A "bad parent" can recover from this "temporary" situations. This should not prevent him from future productive service to clients.

7.5.1 Bad Parent Rehabilitation

There are many reasons for punishing a client which marks it as bad parent in the system. Nevertheless, we realized that the punishment mechanism on the PPPC network should be temporary. Network turns to free over time as activities that consumed the client uplink are completed. We shall associate a time stamp with the punishment time when a client is punished. After a period of time we will rehabilitate the parent and allow it to receive new connections.

The rehabilitation thread is in charge of bad parents rehabilitation. The suggested time period for rehabilitation in our experience is between 5 and 15 minutes.

7.5.2 Distress Logic: Marking of Bad Parents

A distress state is the state in which a client does not receive enough information within a PACKET_RATE_CALCULATION_PERIOD. There are two variables that dictate distress state

1. Parent distress is a boolean variable that indicates whether the parent sent any indication of entering into a distress state.
2. Current distress is a variable that may be equal to either no-distress, light distress, or major distress.

These variables introduce six different distress states:

No distress - The standard state. Packet rate is fine and the father has not informed otherwise.

Light distress - The state that occurs when the client receives less packets then DISTRESS_MIN_PACKET_RATE and there is no notification from the parent that he reached similar state.

Parent distress - The parent indicates that he is in a light distress state but the information still flows fine.

Parent and light distress - Indicates that both the current client and its father experienced light distress.

Major distress - Indicates that current packet rate is below MIN_PACKET_RATE.

Major and parent distress - Indicates that current packet rate is below MIN_PACKET_RATE and the parent is also experiencing difficulties.

7.5.2.1 Entering into a distress state

A packet rate threshold, DISTRESS_MIN_PACKET_RATE, is used to determine the upper bound of entering into a "light distress" state. A client in "light distress" does not complain about bad parent, but opens a connection to the DServer to complement missing packets from there. The client only sends a "Bad Parent" message when packet rate reaches MIN_PACKET_RATE, then it connects to the DServer (hereby major distress).

When a client enters into a distress state it will inform its children about its state. When a client enters into a major distress it will not report his parent as a bad parent if his parent is also in a distress state.

7.5.3 Bad Client

Some clients, for whatever reasons may be, are simply unable to receive the stream. Reasons may vary from insufficient downstream, congestion at the ISP or backbone, busy CPU, poor network devices or others.

Those clients will reach major distress state regardless of the parent they were connected to. In this can an "innocent" parent will be marked as "bad" parent. In order to prevent this from happening we add new logic to the PPPC driver.

The client should stop complaining about bad parents when the problem is was its own ability to receive the stream.

If a client enters major distress when receiving a stream from DServer or if 3 clients were not able to serve given client within a certain time frame (we used 1 minute) then the client will stop complaining about bad parents.

7.6 The algorithm

7.6.1 The structure of the Internet - from a Peer–2–Peer Streamer Perspective

The Internet nodes that view the stream, each come from an location with Internet connection, often such organization is the user home. The system we developed is capable of detecting multiple nodes in the same location (such as two users in the same LAN or home) via multicast messages. The system insure that on any location only one users will stream in or out of the location. This way we eliminate congestion and as a by product guarantee only one user in each location is visible to the algorithm.

Connections between Internet nodes tends to be loss packets (It is typical to have about 1% packet loss) and add latency. Furthermore, not all connections are equal. When connecting to a "nearby" user we can expect significantly less latency and packet loss then when connecting to a "far" user. Latency, specifically, is increased and can differ from a few milliseconds to hundreds of milliseconds depending on the network distance.

We will now define *near* and *far* in Internet terms. Each Internet connection belongs to an "autonomous system". An autonomous system is usually an ISP[2] and sometimes a large company (such as HP or IBM) that is connected to at least two other autonomous systems. Two users from the same autonomous systems will be considered nearby.

We have created two additional levels of hierarchy.

Below the autonomous system we have created a "subnet" level. Each IP address belongs to a subnet that define a consistent range of IP addresses. Autonomous systems gets their IP range as a disjoint union of subnets. Often each subnet belongs to a different location that can be very far from each other (Such as the east and west coast of the US). Thus, when possible, we prefer to connect to a user from the same Subnet.

Autonomous systems are interconnected. Some autonomous systems can be considered "hubs" connected to many other autonomous systems. We have created "Autonomous System Families" centered on the hubs. (containing all autonomous systems that connect to the hub) When a user from the same autonomous system can't be found we will prefer a user from the same Autonomous System Family (hereby ASF).

An autonomous system usually belongs to more then one autonomous system family.

Thus, we chose clients to connect to each other, we prefer clients that share a subnet. If none is found we prefer clients that belong to the same autonomous system, If none is found we prefer clients that belong to the same autonomous system family. Clients that have no shared container will be considered far and will not connect to each other.

7.6.2 Minimum Spanning Trees of Clients

The algorithm uses containers that hold all clients in a certain level. Since we can consider all clients that share a container and doesn't share any lower level container to be of identical distance from each other We can store all clients in a container in "heap-min" and only consider the best client in each heap to connect to. Best client will be considered using distance from source and best available up link.(best up link considering other peers served by client)

The algorithm 12 strives to maintain the graph as close to the MST as possible while responding to each new request (vertex joining, vertex removal) in nano-seconds. Indeed our solution often involves finding an immediate solution such as connecting directly to the source and improves the solution over time until it reaches the optimal state. The proposed algorithm can handle very large broadcast trees (millions of listeners) in near optimal state. As server load increases (with more users connecting to the stream) we may be further away from the optimal solution but we will not be too far and the stream quality for all users will be well tested.

[2] Internet service provider

Algorithm 12 Real time graph analysis

```
Read the subnet to autonomous systems and the autonomous systems to autonomous systems family files. Store information in a map.
Create global data structure spawn interface thread and parents rehabilitation thread and interface thread.
while Main thread is alive do
    if There are new requests then
        handle new request, touch at most 100 containers.
        Inform interface thread when you are done.
    else
        if there are dirty containers then
            clean at most 100 containers
            inform interface thread when you are done
        else
            wait for new request
        end if
    end if
end while
```

Clean and *dirty* in the algorithm sense are containers that are optimal and containers that are sub optimal for a variety of reasons.

For example assuming a client has disconnected. We will try to handle the disconnection by touching no more then 100 containers, by connecting all the "child" nodes directly to the source. We will mark all the container containing the nodes as dirty. At some point we will clean the container and fix any none optimal state.

7.7 Related systems

We have been involved with Peer-2-Peer streaming company vTrails ltd that has operated in peer-2-peer streaming scene in 1999-2002. vTrails no longer operates. Many of the concepts and system design may have originated in the authors period with vTrails though the system have been written from scratch.

In recent years several peer-2-peer streaming system have been proposed, some with similar design. Our main contribution in this work is the peer-2-peer algorithm designed to calculate graph algorithms based on real time approaches. Some system approach such as the handling of distress state is also innovative.

ChunkySpeed[VFC06] is a related system that also implements peer-2-peer multicast, in a robust way. Unlike PPPC ChunkySpeed doesn't take internet distances into account.

ChunkCast[CWWK06] is another multicast over peer-2-peer system. ChunkCast deals with download time which is a different problems all together. (In streaming, guaranteed constant bitrate is required. This requirement doesn't exist in content download which is not vulnerable to a fluctuation in download speed and only the overall download time really matters.

Climber[PPK08] is a peer-2-peer stream based on an initiative for the users to share. It is in our experience that user sharing is not the main problem but rather the main problem is that broadcasters are not willing to multicast their content on peer-2-peer networks. Thus Climber does not solve the real problem.

Microsoft[PWC04] researched peer-2-peer streaming in multicast environment and network congestion but had a completely different approach which involved multiple sub stream for each stream based on the client abilities.

Liu et al [LBLLN09] recently researched P2P streaming servers handling of bursts flash crowds which is easily handled by the algorithm easily thanks to it's real time capabilities.

7.8 Experimental results

The PPPC system strives to provide QoS over bandwidth savings. We will connect clients to the DServer directly whenever the stream they receive from another client is not stable enough. Therefore, bandwidth saving is only best effort and we do not offer any guaranteed results.

One can construct a network where no user has the available Internet upload connection to serve any other users. However, this case has never appeared in our testing. Usually a saving ratio ($1 - \frac{Rootclients}{TotalClients}$) of at least 50% and up to 99% can be expected.

Following are several results that we got in real world experimental testing and in simulated networks. They should demonstrate what a user can expect in average scenario.

7.8.1 Internet Radio Streaming

Internet radio streaming is characterized by a relatively low bitrate streaming. In this benchmark, we limited each user to serve to no more than 3 other users the stream. The goal is avoid what the users may consider as too much abuse of their uplink. However, even with this limitation we proved that 70% bandwidth saving is feasible on 100-200 radio users spread around the world and up to 99% saving when the users are all local users. This difference can be attributed in part to our algorithm that does not handle users as "connectable" outside of an Autonomous System Family. Therefore, when we had many users we could always found somebody to connect the user to. Something we weren't able to do with relatively small number of world wide users.

Figure 41 demonstrate the system life time and bandwidth saving with high spread of users. These results were obtained daily on a classical music radio station with world wide spread of listeners Figure 42 demonstrate the system life time and bandwidth saving with low spread of users. These results were obtained during the daytime on a local radio station with listeners are mostly local users from Israel. We found it frequent that more then one stream was used when we had remote listeners. (People from abroad etc.)

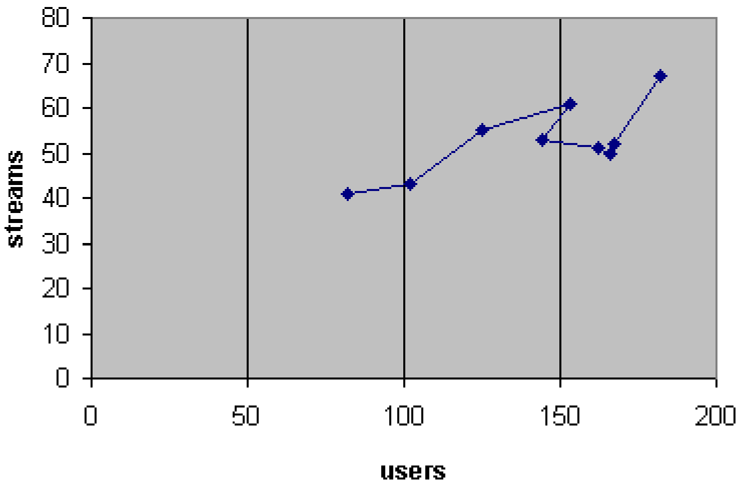


FIGURE 41 Results on high spread radio users

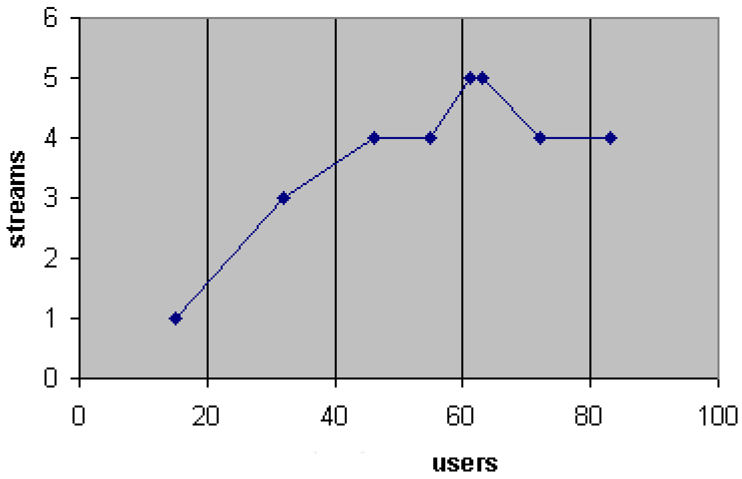


FIGURE 42 Results on low spread radio users

7.8.2 Video streaming

Video streaming is more demanding between peer-2-peer links. Streaming high quality video to multiple users on internet links may often be beyond the ability of some users.

When video streaming benchmarks in Enterprises and Universities were running, we were able to reach 92% saving on enterprise streaming.

Furthermore, we always had no more then 1 connection to each enterprise location at any given time.

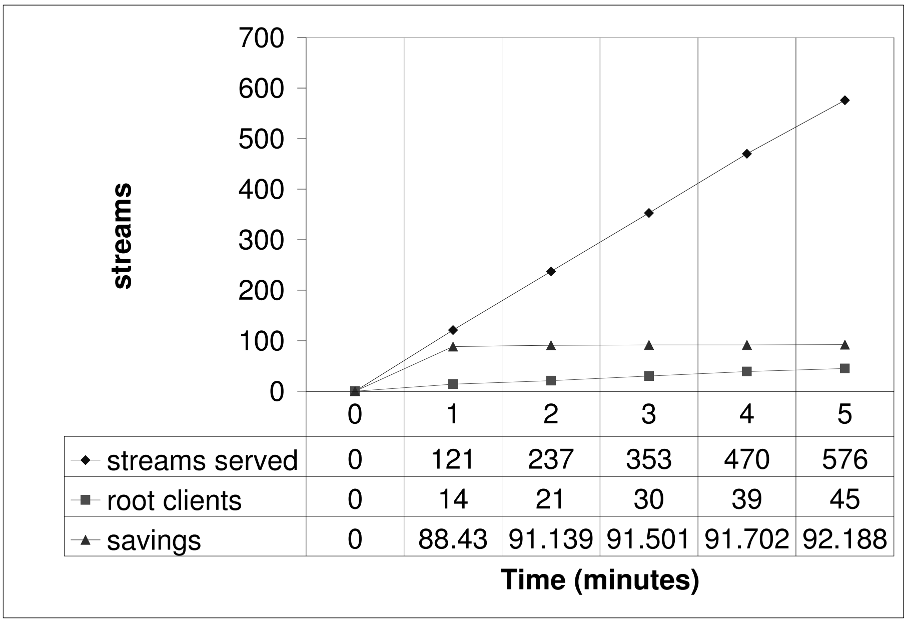
The following benchmark represent an online class given by the IUCC (Inter University Computation Center) in Israel. No more then 1 connection was made to each separate academic institute.

7.8.3 Simulated results

We produced the following results on a complete system with simulated clients. In this test, we produced clients from 20,000 different subnets that were picked randomly from the entire Internet. These subnets may or may not belong to any common ASF.

At this test, we produced 2000 clients where 2 clients join every second and the average lifetime of a client is 1,000 seconds. 1% of the client were bad parents. A client could support either 0, 1, 2, or 3 clients with equal chances. Packet loss between normal clients was not simulated in this benchmark, but bad parents were unable to serve clients. In this simulated benchmark, we did video streaming.

Figure 43 demonstrate the system behaviour over time as it starts accumulating clients. We are starting a buildup of random clients with spread IPs. (The network was simulated off course but the algorithm received random IP address belonging to random IPs worldwide)

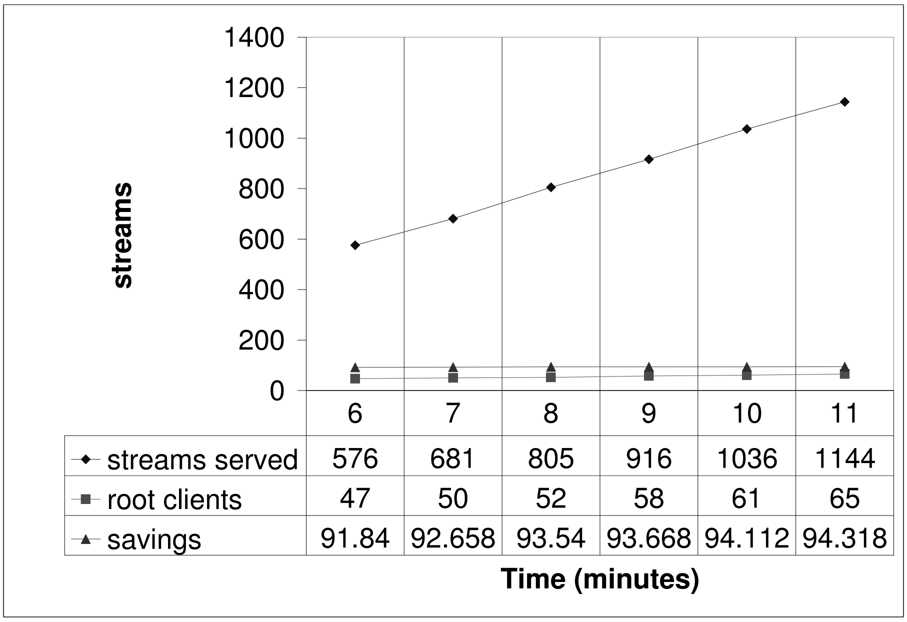


	0	1	2	3	4	5
◆ streams served	0	121	237	353	470	576
■ root clients	0	14	21	30	39	45
▲ savings	0	88.43	91.139	91.501	91.702	92.188

FIGURE 43 The beginning of the simulated test.

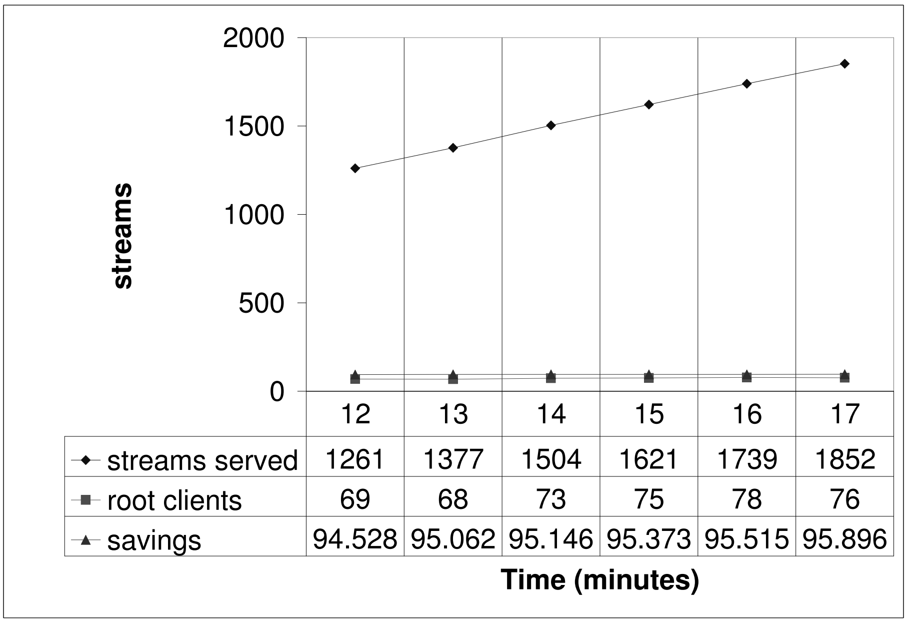
Figure 44 demonstrates the system's behavior as the number of clients grows over time. As more clients were added, we can see that the number of the root clients for which the Internet broadcaster consumes bandwidth, grows as the number of clients grows. However, the saving ratio ($\frac{rootclients}{totalclients}$) actually improves as the number of clients increases. That is because we can now find more local clients for each new IP that we receive.

Figure 45 demonstrates the next 6 minutes as the number of clients in the graph grows. We see more of the same, i.e. the saving ratio improves as the number of clients grows.



	6	7	8	9	10	11
◆ streams served	576	681	805	916	1036	1144
■ root clients	47	50	52	58	61	65
▲ savings	91.84	92.658	93.54	93.668	94.112	94.318

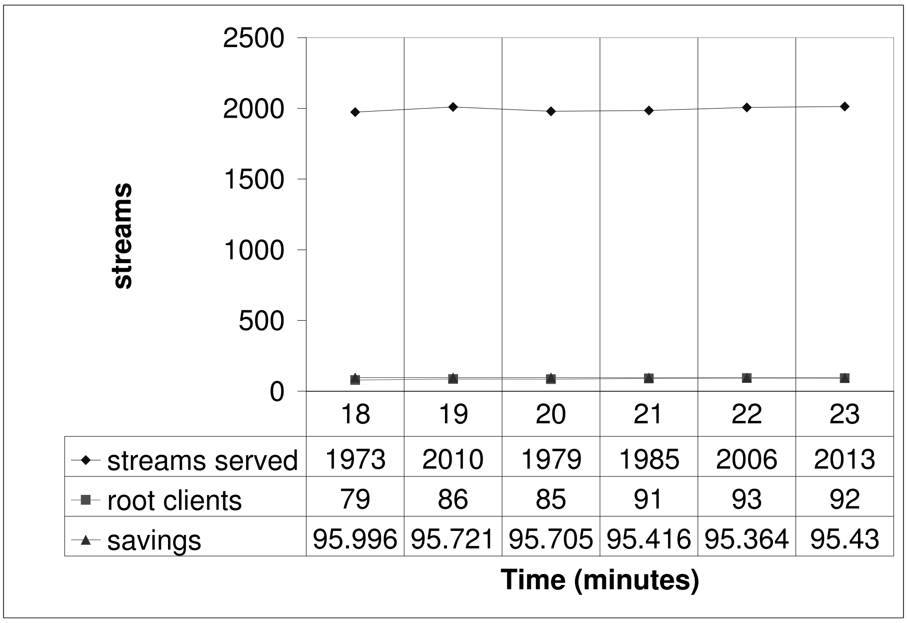
FIGURE 44 This image represents the system status upon reaching 1,000 clients



	12	13	14	15	16	17
◆ streams served	1261	1377	1504	1621	1739	1852
■ root clients	69	68	73	75	78	76
▲ savings	94.528	95.062	95.146	95.373	95.515	95.896

FIGURE 45 Minutes 12-17, 2000 clients.

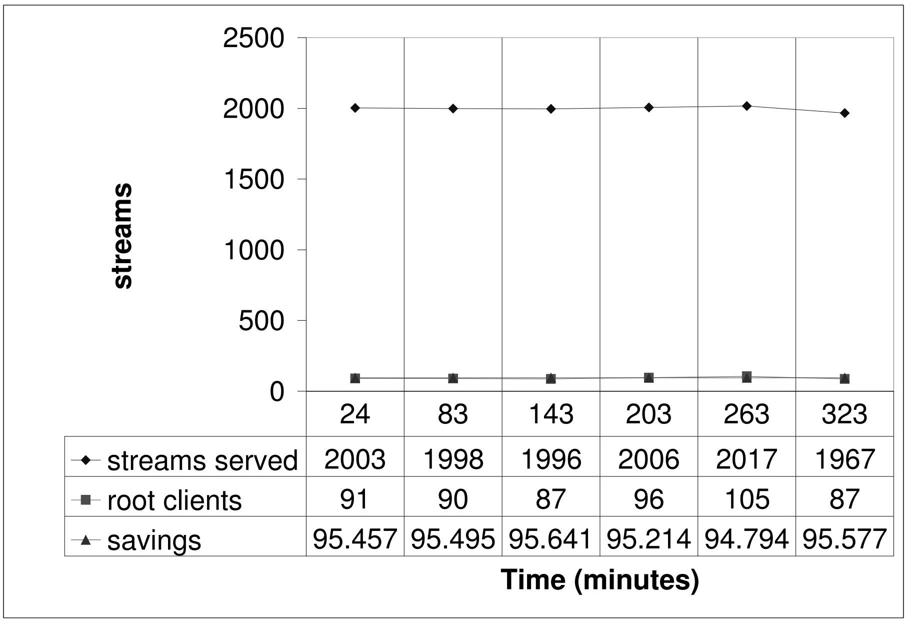
When the predicted maximum clients number is for this experiment (2000) is reached, the system is still stable in bandwidth saving and the number of root clients is as demonstrated in figure 46. The stability in saving ratio and the system remains over time.



	18	19	20	21	22	23
◆ streams served	1973	2010	1979	1985	2006	2013
■ root clients	79	86	85	91	93	92
▲ savings	95.996	95.721	95.705	95.416	95.364	95.43

FIGURE 46 This image shows the test results upon reaching maximum clients limit

We left the system running for a few hours. We checked the results every hour and saw no real change in either the stream quality or in the bandwidth consumption. These results are demonstrated in figure 47. The saving ratio of the system remains stable.



Time (minutes)	24	83	143	203	263	323
◆ streams served	2003	1998	1996	2006	2017	1967
■ root clients	91	90	87	96	105	87
▲ savings	95.457	95.495	95.641	95.214	94.794	95.577

FIGURE 47 We sampled the system running every hour.

While the system was running, we kept a client connected to a stream. The stream quality was flawless at all times.

Other experiments with more clients and different attribute produce similar results.

7.9 Conclusion and Future work

We have demonstrated how significant amount of the broadcaster bandwidth can be saved by the peer-2-peer streaming, while the broadcaster maintain a complete list of all stream participants (for billing and DRM purposes).

We demonstrated real life scenario where the broadcaster could save around 70% of the streaming bandwidth.

We feel more research may be in place to provide Trusted Computing peer-2-peer streaming environment where the stream is never passed in a way that an outsider can intercept the none encrypted stream.

8 FUTURE WORK

We continue to develop the systems mentioned in this work. Usually future extensions of the existing systems were mentioned in each chapter. Additionally, we will continue our work on other virtualization systems. Some of the systems that we currently work on are described below.

8.1 Job control System for EGI

The European Grid Initiative (EGI) is a multination project based in CERN to provide massive computation environment for research institutes all over Europe.

The GRID system has no online scheduler or job management system forcing users to trace each of the job submitted manually. Furthermore, no limitation is currently applied on job submitted even by new and inexperienced users that may flood EGI with random, uselss and often duplicate jobs.

Together with Tamir Erez and Zeev Vaxman Fisher (who heads Israel Grid consurtium) we have developed a job management and submission that is already deployed and widely used by several researchers in Israel. The work is now completed with the cooperation of Redhat.

The final version of the system will be published shortly and offered to other national grid organizations. (NGO)

8.2 Creating an Efficient Infrastructure as a Service Environment

Our work on Lguest and Asynchronous mirror replication led us to observing the amount of shared memory pages across multiple running VMs that share similar OS. In such cases most of the storage and memory will be indentical (with the sole exception being the pages that are actually developed particularly for this VM.)

This project has recently began.

8.3 Bridging Grid Platform as a service and Infrastructure as a Service

Current EGI grid infrastructure is great platform-as-a-service service providing massive amount of processing power for those who need.

But some users simply cannot use the grid as platform-as-a-service is not suitable for them. Some users needs specific environment requiring infrastructure-as-a-service.

Our work involves running a system virtual machine on EGI infrastructure with memory and storage deduplication allowing users to actually submit virtual machines snapshots as jobs for the grid.

This project has recently began.

8.4 Optimizing memcached

Memcached [Fit04] is a clustered key-value storage caching environement that was developed after the observation that nowdays the network is often significantly faster then the storage (disks) and a distributed cached (which may involve a network request to a nearby host) outperforms the caching environment were a local cache miss causes immediete disk access.

Memcached does improve performance greatly but uses only a naive LRU caching algorithm. Our work involves increasing memcached performance by replacing the memcached caching algorithm with more efficient algorithms such as ARC[MM03]. The clustered nature of memcached makes this problem more interesting.

This project is currently in it's implementation phase. Initial results demonstrate about 20% improvement compared to memcached caching as demonstrated by figures 48 and 49.

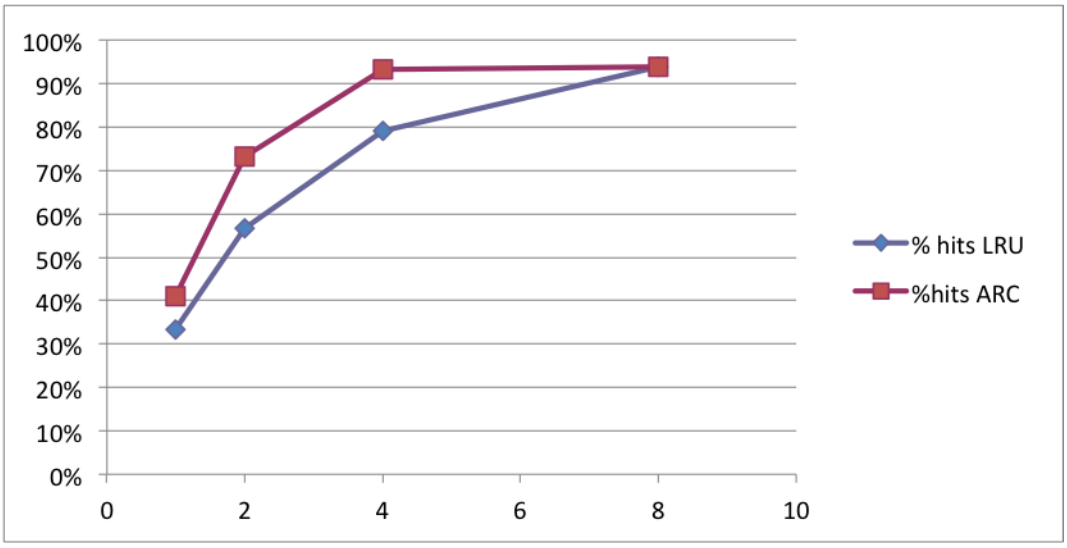


FIGURE 48 LRU vs. ARC on 1,2,4,8 CPUs. 15462400 Operations, 95% get 5% set, 745K window size.

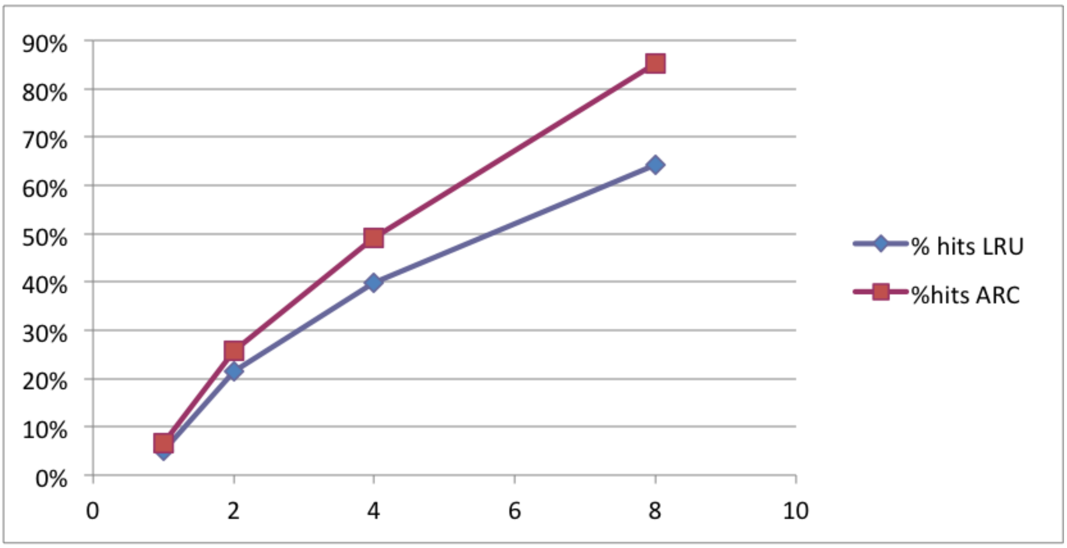


FIGURE 49 LRU vs. ARC on 1,2,4,8 CPUs. 16650240 Operations, 85% get 15% set, 2439K window size.

9 CONCLUSION

We presented five systems for solving specific problems where each system used a different virtualization form as a critical design feature.

LgDb: In the *LgDb* tool, we used a system virtual machine that provides a controlled debug environment for Linux kernel modules. In LgDb, the virtualization abstraction allows the software developer to disengage from the hardware layer to provide an execution environment for the development tools. During the work on LgDb, we realized that while virtualization can be beneficial to kernel modules development, The virtualization layer fails to provide a suitable environment for device drivers debugging.

AMirror: In the *AMirror* system, we used the abstraction provided by our logical volume manager (storage virtualization) to create self replicating block devices. Furthermore, by using Lguest and QEMU/KVM system virtualization environments, we were able to insert an abstraction layer for the whole system migration. In this system, virtualization did not introduce any limitations. However, using virtual machines (instead of physical machines) typically introduces a slight performance penalty.

Truly-Protect: In the *Truly-Protect* environment, we used the process virtual machine to create an encrypted execution environment. The encrypted execution environment affected the system performance but also provided us with a trusted computing environment without the need to have specialized hardware.

LLVM-Prefetch: In the *LLVM-Prefetch* system, we used a process virtual machine to pre-execute program instructions. This process usually boosts application performance as it allows computation and I/O to be performed simultaneously. However, if the application is already coded to allow parallel execution of computation and I/O, then the use of LLVM-Prefetch may result in performance degradation due to the extra threads created by the system.

PPPC: In the *PPPC* system, we used network virtualization and cloud computing that provides a platform as a service environment. In PPPC, the network virtualization layer provides a multicast network where none existed in IPv4. It also cuts the link between the broadcaster and the audience, thus, it introduces problems from trusted computing perspective that the system is meant to take care of.

Each system, that was presented in this dissertation, introduced a programming problem followed by a system that provides the solution. Each solution used virtualization that serves as "one extra level of indirection". This reinforces David Wheeler's fundamental theorem of Computer Science. i.e.: "All problems in computer science can be solved by another level of indirection" [Spi07].

We believe that the methods, which were described herein, will be beneficial to system designers.

On the other hand, adding the virtualization layer component introduced new problems such as the inability to get native debug device drivers in LgDb and performance degradation in Truly-protect.

Thus, in addition, we validated Kelvin Henney's Corollary "All problems in computer science can be solved by another level of indirection ... except for the problem of too many layers of indirection".

YHTEENVETO (FINNISH SUMMARY)

Esittelemme tässä väitöskirjassa useita järjestelmiä, jotka hyödyntävät virtualisointiteknologian eri piirteitä. Järjestelmäkäyttö vaihtelee tekijänoikeussuojauksesta, täystuhosta toipumisesta ja Linux-ytimen kehitystyökaluista joukkoviestimien suoratoistoon ja tieteellisen laskennan optimointiin. Järjestelmät käyttävät useita virtualisoinnin piirteitä, kuten sovellusvirtuaalikoneita, järjestelmävirtuaalikoneita, tallennusvirtualisointia, pilvilaskentaa ja niin edelleen.

Jokaisessa järjestelmässä on jokin ainutlaatuinen virtualisoinnin piirre, mutta ne myös havainnollistavat tärkeitä menetelmiä järjestelmäsuunnittelijoille. Kaikissa järjestelmissä virtualisointia käytetään "uuden epäsuoruuden tason" lisäämiseen. Lisäämällä tämän "uuden epäsuoruuden tason" voimme käyttää sellaisia työkaluja, joita emme voisi muuten hyödyntää. Virtualisointitekniikat ovat osoittaneet olevansa elintärkeitä työkaluja kaikissa kehitetyissä järjestelmissä.

Monissa tapauksissa järjestelmän ajo tai kehittäminen on mahdotonta tai se olisi ainakin ollut huomattavasti vaikeampaa, jos virtualisointia ei olisi lainkaan käytetty. Monissa näissä järjestelmissä virtualisointi johtaa myös heikentyneeseen toimintaan ja suoritustehoon. Käsittelemme täten myös virtualisoinnin seurauksia ja kustannuksia.

Väitämme, että virtualisointi on tärkeä työkalu, jota suunnittelijat eivät voi jättää huomiotta suunnitellessaan tulevaisuuden järjestelmiä. Kaikki tämän väitöskirjatyön puitteissa kehitetyt järjestelmät ovat vapaasti käytettävissä GNU GPL – lisenssin alla.

REFERENCES

[AB] Christian Rechberger Andrey Bogdanov, Dmitry Khovratovich. Biclique cryptanalysis of the full aes. urlhttp://research.microsoft.com/en-us/projects/cryptanalysis/aesbc.pdf.

[AKZ11a] Amir Averbuch, Eviatar Khen, and Nezer Jacob Zaidenberg. LgDB - virtualize your kernel code development. In *Proceedings of SPECTS 2011*, June 2011.

[AKZ11b] Amir Averbuch, Michael Kiperberg, and Nezer Jacob Zaidenberg. An Efficient VM-Based Software Protection. In *NSS 2011*, 2011.

[AKZ12] Amir Averbuch, Michael Kiperberg, and Nezer Jacob Zaidenberg. An Efficient VM-Based Software Protection. In *submitted*, 2012.

[AMZK11] Amir Averbuch, Tomer Margalit, Nezer Jacob Zaidenberg, and Eviatar Khen. A low bitrate approach to replication of block devices and virtual machines. In *Networking, Architecture and Storage (NAS), 2011 IEEE Sixth International Conference on*, 2011.

[And01] R. Anderson. *Security Engineeringm – A guide to building dependable distributed systems*. John Wiley and Sons, 2001.

[And02] Andrew “bunnie" Huang. Keeping secrets in hardware: the microsoft xboxtm case study. Technical report, Massachusetts Institute of Technology, 2002. http://web.mit.edu/bunnie/www/proj/anatak/AIM-2002-008.pdfa.

[App] Apple Ltd. Mac OS X ABI Mach-O File Format Reference. http://developer.apple.com/library/mac/#documentation/DeveloperTools/Conceptual/MachORuntime/Reference/reference.html.

[ARZ11] Amir Averbuch, Yehuda Roditi, and Nezer Zaidenberg. An algorithm for creating a spanning graph for peer-2-peer internet broadcast. In *Proccedings of CAO2011: CCOMAS Thematic Conference on Computational Analysis and Optimization*, 2011.

[ARZ12] Amir Averbuch, Yehuda Roditi, and Nezer Zaidenberg. An algorithm for creating a spanning graph for peer-2-peer internet broadcast. In *Computational Analysis and Optimization: Dedicated to Professor P. Neittaanmaki on his 60þBirthday*, 2012.

[BBLS91] D.H. Bailey, J. Barton, T. Lasinski, and H. Simon. The nas parallel benchmarks. *Technical Report RNR-91-002, NASA Ames Research Center*, 1991.

[BBM09] Stanislav Bratanov, Roman Belenov, and Nikita Manovich. Virtual machines: a whole new world for performance analysis. *Operating Systems Review*, 43:46–55, 2009.

[BDF$^+$03] Paul Barham, Boris Dragovic, Keir Fraser, Steven H, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proc. of SOSP 2003*, pages 164–177, 2003.

[Bel05] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *FREENIX Track: 2005 USENIX Annual Technical Conference*, 2005.

[Bes80] Robert M. Best. Preventing software piracy with crypto-microprocessors. In *Proceedings of IEEE Spring COMPCON 80*, pages 466–469, February 1980.

[Bhu71] A.K. Bhushan. File Transfer Protocol. RFC 114, April 1971. Updated by RFCs 133, 141, 171, 172.

[bms10] bushing, marcan, and sven. Console hacking 2010 ps3 epic fail. In *CCC 2010: We come in peace*, 2010.

[Box02] D. Box. *"Essential .NET, Volume 1: The Common Langugage Runtime"*. Addison-Wesley, 2002.

[BS11] A. Barak and A. Shiloh. Mosix. http://www.mosix.org, 2011.

[CBS07] Yong Chen, Surendra Byna, and Xian-He Sun. Data access history cache and associated data prefetching mechanisms. In *SC '07 Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007.

[CBS$^+$08] Yong Chen, Surendra Byna, Xian-He Sun, Rajeev Thakur, and William Gropp. Hiding i/o latency with pre-execution prefetching for parallel applications. In *SC '08 Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

[Cha01] F. Chang. *Using Speculative Execution to Automatically Hide I/O Latency*. Carnegie Mellon, 2001. PhD dissertation CMU-CS-01-172.

[CHLS08] Donald Chesarek, John Hulsey, Mary Lovelace, and John Sing. Ibm system storage flashcopy manager and pprc manager overview. http://www.redbooks.ibm.com/redpapers/pdfs/redp4065.pdf, 2008.

[CLM$^+$08] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High availability via asynchronous virtual machine replication. In *5th USENIX Symposium on Network Systems Design and Implementation*, 2008.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Interval tree. In *Introduction to Algorithms (2nd ed.)*, chapter 14.3, pages 311–317. MIT Press and McGraw-Hill, 2001.

[CLRT00] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. Pvfs: A parallel file system for linux clusters. In *IN PROCEEDINGS OF THE 4TH ANNUAL LINUX SHOWCASE AND CONFERENCE*, pages 391–430. MIT Press, 2000.

[Clu02] Cluster File System Inc. Lustre: A scaleable, high performance file system, 2002. whitepaper.

[Coh03] Bram Cohen. Incentives build robustness in bittorrent. Technical report, Bittorrent inc, May 2003.

[Coo09] EMC Cooperation. Implementing emc srdf/star protected composite groups on open systems. http://japan.emc.com/collateral/software/white-papers/h6369-implementing-srdf-star-composite-groups-wp.pdf, 2009.

[CWWK06] Byung-Gon Chun, Peter Wu, Hakim Weatherspoon, and John Kubiatowicz. Chunkcast: An anycast service for large content distribution. In *Proccedings of IPTPS 06'*, 2006.

[Des] Mathieu Desnoyers. Linux trace toolkit next generation manual. http://git.lttng.org/?p=lttv.git;a=blob_plain;f=LTTngManual.html.

[DH76] W. Diffie and M.E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22:644–654, November 1976.

[DJ08] Claudio DeSanti and Joy Jiang. Fcoe in perspective. In *ICAIT '08 Proceedings of the 2008 International Conference on Advanced Infocomm Technology*, 2008.

[DJC+07] Xiaoning Ding, Song Jiang, Feng Chen, Kei Davis, and Xiaodong Zhang. Diskseen: Exploiting disk layout and access history to enhance I/O prefetch. In *2007 USENIX Annual technical conference*, 2007.

[DM08] Stanislav Shwartsman Darek Mihocka. Virtualization without direct execution or jitting: Designing a portable virtual machine infrastructure. Technical report, http://bochs.sourceforge.net/, 2008.

[DRB10] DRBD. http://www.drbd.org, 2010.

[DS07] Felix Domka and Michael Steil. Why Silicon-Based Security is still that hard: Deconstructing Xbox 360 Security. In *CCC 2007*, 2007.

[DSZ10] Jiaqing Du, Nipun Sehrawat, and Willy Zwaenepoel. Performance profiling in a virtualized environment. In *Proceedings of USENIX HotCloud 2010*, 2010.

[Eag07] Michael J. Eager. Introduction to the dwarf debugging format. Technical report, PLSIG UNIX International, February 2007.

[Ehr10] David Ehringer. The dalvik virtual machine architecture. Technical report, Google, March 2010.

[EK90] Carla Schlatter Ellis and David Kotz. Prefetching in file systems for mimd multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1, 1990.

[FGM$^+$99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785, 6266.

[Fit04] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004, August 2004.

[FL08] Chen Feng and Baochun Li. On large-scale peer-to-peer streaming systems with network coding. *ACM Multimedia*, 2008.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9:319–349, 1987.

[GBD$^+$94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidyalingam S. Sunderam. *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Network Parallel Computing*. MIT Press, 1994.

[gcc] gcc developers. gcov. http://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of 19th ACM Symposium on Operating Systems Principles*, 2003.

[GKM] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof – a call graph execution profiler. http://docs.freebsd.org/44doc/psd/18.gprof/paper.pdf.

[Gle11] Kyle Gleed. vmotion - what's going on under the covers? http://blogs.vmware.com/uptime/2011/02/vmotion-whats-going-on-under-the-covers.html, 2011.

[GM] Thomas Gleixner and Ingo Molnar. Performance counters. http://lwn.net/Articles/310176/.

[Gol86] Oded Goldreich. Toward a theory of software protection. In *Proceedings of advances in cryptology – CRYPTO86*, 1986.

[Gra08] Alexander Graf. Mac os x in kvm. In *KVM Forum 2008*, 2008.

[IBM] IBM. Quantify (rational purifyplus). http://www-01.ibm.com/software/awdtools/purifyplus/enterprise/.

[inc] VMWARE inc. Vmkperf.

[Int10] Intel Cooperation. Breakthrough aes performance with intel aes new instructions, 2010.

[Int12] Intel Cooperation. *IntelÂ® 64 and IA-32 Architectures Developer's Manual: Vol. 3A*. 2012.

[Jou07] Nikolai Joukov. mirrorfs. http://tcos.org/project-mirrorfs.html, 2007.

[Kiv07] Avi Kivity. Kvm: The kernel-based virtual machine. In *Ottawa Linux Symposium*, 2007.

[KJ03] Rick Kennell and Leah H. Jamieson. Establishing the genuinity of remote computer systems. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[KN94] David Kotz and Nils Nieuwejaar. Dynamic file-access characteristics of a production parallel scientific workload. In *Supercomputing '94 Proceedings of the 1994 conference on Supercomputing*, 1994.

[KNM06] Nitin A. Kamble, Jun Nakajima, and Asit K. Mallick. Evolution in kernel debugging using hardware virtualization with xen. In *Ottawa Linux Symposium*, 2006.

[KY04] Dongkeun Kim and Donald Yeung. A study of source-level compiler algorithms for automatic construction of pre-execution code. *ACM TRANS. COMPUT. SYST*, 22, 2004.

[LA03] Chris Lattnar and Vikram Adve. Architecture for a next-generation gcc. In *First Annual GCC Developer Summit*, May 2003.

[LA04] Chris Lattner and Vikram Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *GCO '04 international symposium on Code generation and optimization: feedback-directed and runtime optimization*, 2004.

[Lat07] Chris Lattnar. Llvm 2.0 and beyond! In *Google tech talk*, July 2007.

[Lat08] Chris Lattnar. LLVM and Clang: next generation compiler technology. In *Proc. of BSDCan 2008: The BSD Conference*, May 2008.

[Lat11] Chris Lattnar. LLVM and Clang: advancing compiler technology. In *Proc. of FOSDEM 2011: Free and Open Source Developers' European Meeting*, February 2011.

[LBLLN09] Fangming Liu, Lili Zhong Bo Li+, Baochun Li, and Di Niu*. How p2p streaming systems scale over time under a flash crowd? In *Proccedings of IPSTS 09'*, 2009.

[Lin] Linus Turvalds. About kernel debuggers. ttp://lwn.net/2000/0914/a/lt-debugger.php3.

[Liu07] Y. Liu. On the minimum delay peer-to-peer video streaming: How realtime can it be? In *Proc. of ACM Multimedia*, 2007.

[LMVP04] Jiuxing Liu, Amith Mamidala, Abhinav Vishnu, and Dhabaleswar K Panda. Performance evaluation of infiniband with pci express. *IEEE Micro*, 25:2005, 2004.

[Lov10] Robert Love. *Linux Kernel Development*. Addison-Welsey, June 2010.

[LTP] LTP developers. Linux testing project. http://http://ltp.sourceforge.net/.

[Lud04] Thomas Ludwig. Research trends in high performance parallel input/output for cluster environments. In *4th International Scientific and Practical Conference on Programming*, 2004.

[Luk04] Chi-Keung Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *ACM SIGARCH Computer Architecture News - Special Issue: Proceedings of the 28th annual international symposium on Computer architecture (ISCA '01*, volume 29, 2004.

[LWS$^+$09] H. A. Lagar-Cavilla, Joseph A. Whitney, Adin Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and M. Satyanarayanan. Snowflock: Rapid virtual machine cloning for cloud computing. In *3rd European Conference on Computer Systems (Eurosys)*, 2009.

[LXY08] Xu Li, Changsheng Xie, and Qing Yang. Optimal implementation of continuous data protection (cdp) in linux kernel. In *Networking, Architecture and Storage (NAS), 2008 IEEE International Conference on*, pages 28–35, 2008.

[LY99] T. Lindoholm and F. Yellin. *"The Java Virtual Machine Specification, 2nd ed."*. Addison–Wesley, 1999.

[May01] John M May. *Parallel I/O for High Performance Computing*. Morgan Kaufmann, 2001.

[Mic] Microsoft Cooperation. Windows authenticode portable executable signature form. http://msdn.microsoft.com/en-us/windows/hardware/gg463180.aspx.

[mil] millerm. elfsign. http://freshmeat.net/projects/elfsign/.

[Mit05] Chris Mitchell. *Trusted computing*. The Institution of Engineering and Technology, 2005.

[MM63] J. C. Miller and C. J. Maloney. Systematic mistake analysis of digital computer programs. *Communications of the ACM*, 3:58–63, 1963.

[MM03] Nimrod Megiddo and Dharmendra S. Modha. Arc: A self-tuning, low overhead replacement cache. In *Proccedings of USENIX FAST 03*, 2003.

[MR08] Alex Ionescu Mark Russinovich, David A. Solomon. *Windows Internals: Including Windows Server 2008 and Windows Vista, Fifth Edition (Pro Developer)*. Microsoft Press, 2008.

[MRW06] M. A. Hennell M. R. Woodward. On the relationship between two control-flow coverage criteria: all jj-paths and mcdc. *Information and Software Technology*, 48:433–440, 2006.

[MST$^+$] Aravind Meno, Jose Renato Santos, Yoshio Turner, G. John Janakiraman, and Willy Zwaenepoel. Xenoprof. http://xenoprof.sourceforge.net/.

[MST$^+$05] Aravind Meno, Jose Renato Santos, Yoshio Turner, G. (John) Janakiraman, and Willy Zwaenepoe. Diagnosing performance overheads in the xen virtual machine environment. In *VEE*, volume 5, pages 13–23, 2005.

[NDS] NDS. PC Show. http://www.nds.com/solutions/pc_show.php.

[NS03] Nicholas Nethercote and Julian Sewardb. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89:44–66, October 2003.

[Opr] Oprofile's developers. Oprofile: a system profiler for linux. http://oprofile.sourceforge.net/news/.

[Opr11] Oprofile developers. Interpreting call graph. In *Oprofile user guide*, chapter 5.3. Oprofile, 2011.

[Ore] Oreans Technologies. Code Virtualizer. http://www.oreans.com/products.php.

[Para] Parasoft. Insure++. http://www.parasoft.com/jsp/products/insure.jsp?itemId=63.

[Parb] Parasoft. Jtest. http://www.parasoft.com/jsp/products/jtest.jsp/.

[Pat97] R H Patterson. *Informed Prefetching and Caching*. Carnegie Mellon, 1997. PhD dissertation CMU-CS-97-2004.

[Pea02] Siani Pearson. Trusted computing platforms, the next security solution. Technical report, Hewlett-Packard Laboratories, 2002.

[PG74] G. J. Popek and R. P. Goldberg. Formal requirements for virtualization third-generation architecutres. *Communications of the ACM*, pages 412–421, July 1974.

[PGG$^+$95] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *In Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 79–95. ACM Press, 1995.

[Pos80] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980.

[Pos81a] J. Postel. Internet Protocol. RFC 791 (Standard), September 1981. Updated by RFC 1349.

[Pos81b] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFCs 1122, 3168, 6093.

[Pos82] J. Postel. Simple Mail Transfer Protocol. RFC 821 (Standard), August 1982. Obsoleted by RFC 2821.

[PPK08] Kunwoo Park, Sangheon Pack, and Taekyoung Kwon. Climber: An incentive-based resilient peer-to-peer system for live streaming services. In *Proccedings of IPTPS 08'*, 2008.

[PS05] Athanasios E. Papathanasiou and Michael L Scott. Aggressive prefetching: an idea whose time has come. In *Proceedings of 10th Workshop on Hot Topics in Operating Systems*, 2005.

[PWC04] Venkata N. Padmanabhan, Helen J. Wang, and Philip A. Chou. Supporting heterogeneity and congestion control in peer-to-peer multicast streaming. In *Proccedings of IPTPS 04'*, 2004.

[Ree03] Daniel A Reed. *Scalable Input/Output: Achieving System Balance*. The MIT Press, October 2003.

[Roe11] Kevin Roebuck. *Encryption*. Tebbo, 2011.

[Rol] R. E. Rolles. Unpacking VMProtect. http:/http://www.openrce.org/blog/view/1238/.

[Rol09] Rolf Rolles. Unpacking virtualization obfuscators. In *Proc. of 4th USENIX Workshop on Offensive Technologies (WOOT '09)*, 2009.

[Ros] Steven Rostedt. ftrace - function tracer. http://lxr.linux.no/linux+v3.1/Documentation/trace/ftrace.txt.

[Rus] Rusty Russell. Rusty's remarkably unreliable guide to Lguest. http://lguest.ozlabs.org/lguest.txt.

[Rus08] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *Electronic Notes in Theoretical Computer Science*, 42, July 2008.

[RWS05] Stephan A. Rago Richard W. Stevens. *Advanced Programming for the UNIX Environment 2nd Edition*. Addison–Wesley Professional, 2005.

[San] Santa Cruz Operations Inc. *SYSTEM V APPLICATION BINARY INTERFACE*.

[SBP] Peter Jay Salzman, Michael Burian, and Ori Pomerantz. Linux kernel module programming guide. http://tldp.org/LDP/lkmpg/2.6/html/.

[SBPR08] Mudhakar Srivatsa, Shane Balfe, Kenneth G. Paterson, and Pankaj Rohatgi. Trust management for secure information flows. In *Proceedings of 15th ACM Conference on Computer and Communications Security*, October 2008.

[sch] scherzo. Inside Code Virtualizer. http://rapidshare.com/files/16968098/Inside_Code_Virtualizer.rar.

[Sch96a] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, 1996.

[Sch96b] Bruce Schneier. Key–exchange algorithms. In *Applied Cryptography 2nd ed.*, chapter 22. Wiley, 1996.

[Sch96c] Bruce Schneier. Key–exchange algorithms. In *Applied Cryptography 2nd ed.*, chapter 21. Wiley, 1996.

[SCW05] Xian-He Sun, Yong Chen, and Ming Wu. Scalability of heterogeneous computing. In *ICPP '05 Proceedings of the 2005 International Conference on Parallel Processing*, 2005.

[SE94] Amitabh Srivastava and Alan Eustace. Atom: A system for building customized program analysis tools. In *PLDI '94 Proceedings of the ACM SIGPLAN*, pages 196–205. ACM, 1994.

[SGG08] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating Systems Concepts*. Wiley, July 2008.

[SH] Nathan Snyder and Q Hong. Prefetching in llvm final report. http://www.cs.cmu.edu/~qhong/course/compiler/final_report.htm.

[SH02] Frank Schmuck and Roger Haskin. Gpfs: A shared-disk file system for large computing clusters. In *Proceedings of the Conference on File and Storage Technologies(FAST'02*, pages 231–244, 2002.

[SKRC10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of 26th IEEE (MSST2010) Symposium on Massive Storage Systems and Technologies*, 2010.

[SLGL09] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Automatic reverse engineering of malware emulators. In *Proc. of the 30th IEEE Symposium on Security and Privacy*, 2009.

[SM06] S. Shende and A. D. Malony. The tau parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–331, Summer 2006.

[SON] SONY Consumer Electronics. Playstayion 3. http://us.playstation.com/ps3/.

[Spi07] Diomidis Spinellis. Another level of indirection. In Andy Oram and Greg Wilson, editors, *Beautiful Code: Leading Programmers Explain How They Think*, chapter 17, pages 279–291. O'Reilly and Associates, Sebastopol, CA, 2007.

[SR97] Evgenia Smirni and Daniel A. Reed. Workload characterization of input/output intensive parallel applications. In *Proceedings of the 9th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 169–180. Springer-Verlag, 1997.

[Ste05] Michael Steil. 17 Mistakes Microsoft Made in the XBox Security System. In *22nd Chaos Communication Congress*, 2005.

[SW] Ben Smith and Laurie Williams. A survey on code coverage as a stopping criterion for unit testing. ftp://ftp.ncsu.edu/pub/tech/2008/TR-2008-22.pdf.

[SWHJ09] Yonghong Sheng, Dongsheng Wang, Jin-Yang He, and Da-Peng Ju. Th-cdp: An efficient block level continuous data protection system. In *Networking, Architecture and Storage (NAS), 2009 IEEE International Conference on*, pages 395–404, 2009.

[Tea] FAA Certification Authorities Software Team. What is a “decision" in application of modified condition/decision coverage (mc/dc) and decision coverage (dc)? http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-10.pdf.

[TGL99] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective i/o in romio. In *FRONTIERS '99 Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, 1999.

[Tim] Tim M Jones. Virtio: An i/o virtualization framework for linux paravirtualized i/o with kvm and lguest.

[Tru] Trusted Computing Group. TPM Main Specification. http://www.trustedcomputinggroup.org/resources/tpm_main_specification.

[TSKM08] Yoshiaki Tamura, Koji Sato, Seiji Kihara, and Satoshi Moriai. Kemari: Virtual machine synchronization for fault tolerance. http://www.osrg.net/kemari/download/kemari_usenix08_poster.pdf, 2008.

[VFC06] Vidhyashankar Venkataraman, Paul Francis, and John Calandrino. Chunkyspread: Multi-tree unstructured peer-to-peer multicast. In *Proccedings of IPTPS 06'*, 2006.

[VMP] VMPSoft. VMProtect. http://www.vmprotect.ru.

[VMW] VMWare. Vmware. http://www.vmware.com.

[WECK07] Hakim Weatherspoon, Patrick Eaton, Byung-Gon Chun, and John Kubiatowicz. Antiquity: Exploiting a secure log for wide-area distributed storage. In *Proceedings of the 2nd ACM European Conference on Computer Systems (Eurosys '07)*, 2007.

[Wei84] M. Weiser. Program slicing. *IEEE Trans. on Software Engineering SE-10*, 4, 1984.

[Wei07] Sage A Weil. *Ceph: Reliable, Scalable, and High-Performance Distributed Storage*. University of California, Santa Cruz, December 2007. Ph.d Thesis.

[WG97] David E. Womble and David S. Greenberg. Parallel I/O: An introduction. *Parallel Computing*, 23, 1997.

[Wik] Wikipedia, the free encyclopedia. AES Instruction Set.

[WVL10] Qingsong Wei, B. Veeravalli, and Zhixiang Li. Dynamic replication management for object-based storage system. In *Networking, Architecture and Storage (NAS), 2010 IEEE Fifth International Conference on*, pages 412 – 419, 2010.

[XQZ$^+$05] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhonggiang Wu, and Lin Chen. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, 30, 2005.

[Zai01] Nezer J. Zaidenberg. *sFDPC - a P2P approach for streaming applications*. Tel Aviv University (M.Sc thesis), September 2001.

[ZMC11] Eyal Zohar, Osnat Mokryn, and Israel Cidon. The power of prediction: Cloud bandwidth and cost reduction. In *Proc. of SIGCOMM 2011*, August 2011.

[ZS01] Craig Zilles and Gurindar Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001.

[ZZ03] Wensong Zhang and Wenzhuo Zhang. Linux virtual server clusters: Build highly-scalable and highly-available network services at low cost. *Linux Magazine*, November 2003.

[ZZFfZ06] Ke Zhou, Qiang Zou, Dan Feng, and Ling fang Zeng. Self-similarity in data traffic for object-based storage system. In *Networking, Architecture, and Storages, 2006. IWNAS '06. International Workshop on*, 2006.

APPENDIX 1 TRUST COMPUTING

Trusted computing is a branch of the computer security field. Historically, computer security has dealt with software security between users in multiple user computer systems.

This has led to the development of concepts such as permission systems, access control lists and multi-level security policies developed and incorporated into operating systems.

With the reduced price of computers and their ubiquity, the majority of we people own computer systems at home. This has led to new kind of security research. Systems were designed to protect their owner against viruses, attacks, and, recently, spam and malware.

As computer became more common, demand for rich content and home user software has increased. Willing to supply this demand, content owners started to seek means to deliver content (games, media content etc.) while enforcing digital rights.

Purchasing software and media services over the Internet requires infrastructure capable of securing the transactions and ensuring protected intellectual property will be used according to license.

In a sense, trusted computing is different from other security models. In previous security cases we tried to secure the user from other malicious users or from malicious software that may attack the user data. In trusted computation, we try to secure content belonging to content owner and provide the user with conditional access mode, or, as some point out, we protect the system against its owner.

Thus, in a trusted computing environment we assume the hardware owner himself may be malicious (wishing to create illegal copies of the software) and try to protect the content against such users.

This chapter does not claim to be a complete reference on trusted computing. We refer the reader to [Mit05] for a more complete work on trusted computing. We wish to provide some basic foundation about the environment in which our system described in 5 operates.

APPENDIX 1.1 Foundations

Anderson defines trusted system in [And01]:

> A trusted system or component is defined as one whose failure can break the security policy.
>
> A trustworthy system is a system who will not fail.

The TCG defines trusted system as follows:

> A trusted system or component is one that behaves in the expected way for particular purpose.

From the above definitions it follows that we look for systems having a trusted component that cannot be tempered by the system owner. A software vendor should be able to use the trusted component to write trusted software that ensure proper digital rights.

APPENDIX 1.2 Digital Rights Management and Copyright Protection

Copy protection, also known as content protection, copy obstruction, copy prevention or copy restriction, refers to techniques used for preventing reproduction of software, films, music, and other media, usually for copyright reasons. Digital rights management (DRM) refers to technologies whose purpose is to limit the usage of some digital content or device. DRM software components are employed by hardware manufacturers, publishers, copyright holders and individuals. The term can also refer to restrictions associated with specific instances of digital works or devices.

Companies such as Sony, Amazon.com, Apple, Microsoft, AOL and BBC use digital rights management. There are multiple forms of copy protection and DRM methods such as serial numbers or key files, encrypted content and others.

In 1998 the Digital Millennium Copyright Act (DMCA) was passed in the United States to impose criminal penalties on those who circumvent encryption.

While DRM and copyright protection software are essential part of the digital world and even though most countries have criminal laws against breaking such systems, copyright systems and DRM systems are attacked by hackers constantly leading to a development of new copy protection systems.

APPENDIX 1.3 Trusted computing – When trusted hardware is available

Pearson defines a trusted computing platform in [Pea02] as

> a computing platform that has a trusted component, which it uses to create a foundation of trust for software processes.

Indeed when the user hardware has a trusted component software vendors and digital rights owner can use the trusted components to enforce trust.

Several attempts have been made to develop trusted computing platforms for home users.

APPENDIX 1.3.1 CPUID in Intel processors

The first attempt to create a trusted computing environment in the user home was carried over by Intel.

Intel tried to augment its CPUID instruction to include a unique processor identifier.

CPUID is an x86 instruction for CPU identification. The instruction's main use is to determine the capabilities of the CPU to execute special instructions such as MMX, SSE and other instructions which are not common to all x86 processors.

In their Pentium III processors, Intel introduced a new behaviour to the CPUID instruction, which returned the processor's serial number. This number could serve as a trusted hardware component. Unfortunately due to privacy concern the above behaviour of CPUID was disabled in later processors (including processors that are widespread today). Furthermore the CPUID instruction has never provided the serial number on AMD processors and many other non-Intel x86 compatible processors.

APPENDIX 1.3.2 TCG and TPM

The most significant attempt at trusted computing thus far has been by the Trusted Computing Group (TCG). The TCG is an industrial group that defines and publishes standards in the field of trusted computing.

TCG has published the standard for the Trusted Platform Module (TPM). TPM is a hardware module that needs to be installed on the customer's machine to enable a TCG supported trusted computing environment. Software components can query the TPM module in order to identify and authenticate the owner and ensure the owner identity and ensure rightful use of protected content.

Unfortunately, the TPM penetration is very low — it practically does not exist in home-users' machines today.

APPENDIX 1.3.3 Market Penetration of Trusted Computing Environment

Overall, as we have seen, the market is not ready for trusted computing hardware. There are significant objections regarding privacy concern as well as customers not willing to pay for trusted computing platform.

In today's world, if trusted computing means using trusted hardware then the world is not ready.

In the next section we will demonstrate attempts made by companies to achieve a trusted computing environment using standard components.

APPENDIX 1.4 Trusted Computing – When The Trusted Component is Unavailable

When the trusted component is unavailable (as is the case with most user computer systems) we have to begin our trusted computation environment based on standard components.

We no longer rely on the components to be unbreakable but rather very difficult to break. Instead we have to rely on the complexity of some component so that it will be too expensive or too complex for the hackers to attack.

Several companies use obscurity as a means to defend their code. Doing meaningless instructions on some registers or threads, running program counters in non standard CPU registers and in a non linear way (jumps between instructions, moving the program counter backwards etc.) as a means of masking the true work they are doing.

Such methods have received mixed success. Some companies have been able to rely on obfuscation for quite a while without being broken (for example NDS PC-Show). Others have already been broken by hackers several times.

Naturally it is very hard to prove anything on systems that rely on obfuscations.

Other systems choose to rely on other components such as the OS, external plug or USB dongle for copy protection. Practically all of these were broken by hackers and such systems are not widely used any more.

Yet more systems make assumptions about the complexity of the hardware itself. Typical assumptions include assumptions that the CPU or GPU state is too complex for the user to break. Therefore as long as the CPU or GPU registers are not compromised, we can begin our chain of trust in that component. This assumption is found reliable in several products including Sony PlayStation 3 and Microsoft XBOX 360.

Other assumptions of the sort include relying on a special ROM chip on the motherboard (broken in Sony PlayStation 1 and Sony PlayStation 2) or relying on the system bus to be "super-fast" so that the user cannot tap into it (broken in Microsoft XBOX[And02])..

From all the above assumptions, the only assumption that could be acceptable is that the CPU or GPU registers can be considered trustworthy if we prevent the system from being virtualized.

Luckily, Kennell[KJ03] has provided us with the means to prove that a system is running on genuine (none virtualized) hardware.

APPENDIX 1.5 Trusted Computing Concepts in our Systems

Truly-protect, described in chapter 5 is a trusted computing environment. In Truly-protect the chain of starts includes the user CPU. Truly-protect allows the running of encrypted programs via a virtual machine without allowing the user to know what the instructions are that actually take place.

Our peer-2-peer streaming platform discussed in chapter 7 required taking trusted computing consideration into the system design.

APPENDIX 2 CRYPTOGRAPHIC FOUNDATION

This chapter discusses the cryptographic foundations used in the Truly-protect system described in chapter 5.

The presentation here is not complete, and we refer the reader to [Sch96a, Roe11] for a more details.

APPENDIX 2.1 Encryption

Encryption is the process of transforming information (referred to as plaintext) using an algorithm (called a cipher) to make it unreadable to anyone except those possessing special knowledge, usually referred to as a key.

The result of the process is encrypted information (in cryptography, referred to as ciphertext). The reverse process, i.e., to make the encrypted information readable again, is referred to as decryption (i.e., to make it unencrypted).

APPENDIX 2.2 American Encryption Standard (AES)

AES, an acronym for the American Encryption Standard, is a commonly used encryption method. AES was selected as a standard by National Institute for Standards and Technology (NIST) to supersede the previous cryptography standard, DES.

AES is based on a design principle known as a *substitution permutation network*, which can be implemented efficiently in both software and hardware [6].

AES has a fixed block size of 128 bits and a key size of 128, 192, or 256 bits. AES operates on a 4x4 column-major order matrix of bytes, termed the state. Most AES calculations are done over a finite field. The AES cipher comprises several transformation rounds that convert the plaintext into a ciphertext. Each round consists of several processing steps, including one that depends on the encryption key. A set of reverse rounds can be applied to transform the ciphertext back into the plaintext using the same encryption key.

Since [AB] had shown that it is enough to check "only" one fourth of all possible keys to break AES, cryptographers have declared this encryption method as broken. However, it is still safe for every practical use, since not only regular computers but even clusters are not able to break AES encryption thus the vulnerability is not practical.

Algorithm 13 High level description of AES algorithm

```
KeyExpansion - Round keys are derived from the cipher key using Rijndael's key schedule
Initial Round - AddRoundKey - each byte of the state is combined with the round key using bitwise xor
for all Rounds but first and last do
    SubBytes - a non-linear substitution step where each byte is replaced with another according to a lookup table.
    ShiftRows - a transposition step where each row of the state is shifted cyclically a certain number of steps.
    MixColumns -a mixing operation which operates on the columns of the state, combining the four bytes in each column.
    AddRoundKey
end for
Final Round (no MixColumns)
SubBytes
ShiftRows
AddRoundKey
```

APPENDIX 2.3 Identification Schemes

Identification scheme refers to a method in which communicating parties recognize each other. Identification schemes are used to prevent masquerading attacks — in which a malicious user pretends to be somebody else.

Identification schemes are extremely important in DRM and conditional access environments.

We describe here the Feige-Fiat-Shamir identification scheme that can be used in Truly-protect and refer the reader to [Sch96a] for other identification schemes.

The Feige-Fiat-Shamir protocol is:

Algorithm 14 Feige-Fiat-Shamir Identification Scheme

```
Peggy picks a random $r$, when $r$ is less than $n$.
Peggy compute $x = r2modn$
Peggy sends x to Victor.
Victor sends Peggy a random binary string $k-$bits long: $b_1, b_2, ..., b_k$.
Peggy computes $y = r * (s_1^{b_1} * s_2^{b_2} * ... * s_k^{b_k})modn$.
Peggy sends y to Victor.
Victor verifies that $x = y2 * (v_1^{b_1} * v_2^{b_2} * ... * v_k^{b_k})modn$.
Peggy and Victor repeat this protocol $t$ times, until Victor is convinced that Peggy knows $s_1, s_2, ..., s_k$.
```

The chance that Peggy can fool Victor is 1 in 2^{kt}.

APPENDIX 2.4 Key Exchange Techniques

Truly protect uses a key-exchange scheme to exchange keys with the software distributor. We present here a simple example of key exchange techniques that can be used in Truly-protect and refer the reader to [Sch96a] for other key-exchange techniques.

Diffie-Hellman[DH76] is the first public-key algorithm ever invented. Diffie-Hellman relies on the fact that it is much easier to calculate (discrete) exponents than logarithms. The algorithm is simple: first, Alice and Bob agree on a large prime, n and g, such that g is primitive mod n. These two integers do not have to be secret; Alice and Bob can agree to them over some insecure channel. They can even be common among a group of users. It does not matter. Then, the algorithm 2.4 goes as follows:

Algorithm 15 Diffie-Hellman algorithm

```
Alice chooses a random large integer $x$
Alice sends Bob $X = g * xmodn$
Bob chooses a random large integer $y$
Bob sends Alice $Y = g * ymodn$
Alice computes $k = Y * xmodn$
Bob computes $k' = X * ymodn$
Both $k$ and $k'$ are equal to $gxymodn$.
```

The following illustration demonstrates the Diffie-Hellman scheme.

No one eavesdropping on the channel can computes $g * x * ymodn$. At worst they can know $n, g, X, andY$. Unless an eavesdropper compute the discrete logarithm and recovers x or y they do not solve the problem. So, k is the secret key that both Alice and Bob computed independently.

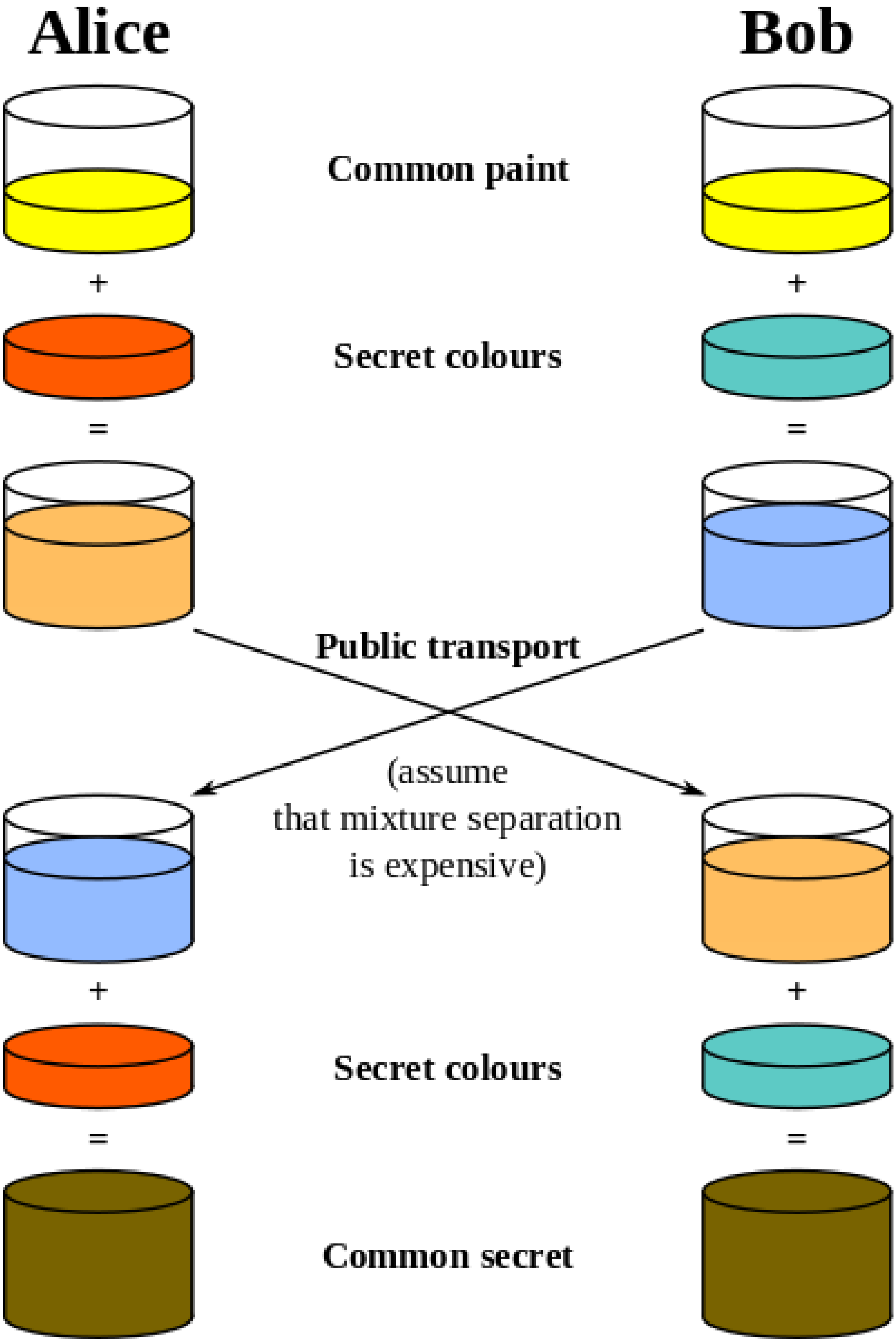


FIGURE 50 The Diffie-Hellman key exchange illustrated

APPENDIX 3 QEMU AND KVM

QEMU[Bel05] is a machine emulator. QEMU relies on dynamic binary translation and allows for reasonable execution speed for emulated binaries. QEMU provides CPU emulation and a set of device drives. Thus QEMU can be considered an hypervisor. QEMU can be used in conjunction with a virtualization platform such as Xen or KVM.

The KVM (Kernel Virtual Machine)[Kiv07] is an industrial grade virtualization environment implemented in the Linux kernel. KVM operates as a Linux kernel module that provides a user space process access to the hardware virtualization features of various processors. Coupling KVM with QEMU allows QEMU to offer viable para-virtualization for x86, PowerPC, and S/390 guests. When the target architecture is the same as the host architecture, QEMU can make use of KVM particular features, such as acceleration.

QEMU was developed by Fabrice Bellard. KVM was released by Qumranet. Qumranet was bought by RedHat in 2008 and Red Hat now control the KVM code base. KVM is maintained by Avi Kivity and Marcelo Tosatti. Both products are released under Open source licenses.

Both KVM and QEMU are living, evolving projects. This chapter describes the system status as of version 0.15 of QEMU and KVM that ships with Linux kernel 3.0.0

APPENDIX 3.1 QEMU – internal design

QEMU has two operating modes:

`User mode emulation` In this mode QEMU runs single Linux or Darwin/-Mac OS X programs that were compiled for a different CPU. System calls are thunked for endianness and for 32/64 bit mismatches. Fast cross-compilation and cross-debugging are the main targets for user-mode emulation.

`Computer emulation` In this mode QEMU emulates a full computer system, including peripherals. It can be used to provide virtual hosting for several virtual computers on a single computer. QEMU can boot many different guest operating systems, including Linux, Solaris, Microsoft Windows, DOS, and BSD. QEMU supports emulation of several hardware platforms, including x86, x86-64, ARM, Alpha, ETRAX CRIS, MIPS, MicroBlaze, PowerPC and SPARC.

APPENDIX 3.1.1 QEMU and storage devices

QEMU stores virtual hard disk images in a proprietary format (qcow or qcow2). These only take up disk space that the guest OS actually uses. This way, an emulated 120 GiB disk can still take up only several hundred megabytes on the host. The QCOW2 format also allows the creation of overlay images that record the difference to another base image file which is not modified. This can be useful to have the possibility of reverting the disk's contents to an earlier state. For example, a base image could hold a fresh install of an operating system that is known to work, and the overlay images are worked with. Should the guest system be unusable for example as a result of a virus attack, the overlay can be deleted and recreated, restoring the system to a stable state.

APPENDIX 3.1.2 QEMU – Tiny Code Generator (TCG)

The Tiny Code Generator (TCG) aims to remove the shortcomings of relying on a particular version of GCC or any compiler, instead of relying on particular compiler versions. QEMU has incorporated the code generator part of the compiler into other tasks performed by QEMU in run-time. The architecture translation task thus consists of two parts: blocks of target code (TBs) being rewritten in TCG ops - a kind of machine-independent intermediate notation, and subsequently this notation being compiled for the host's architecture by TCG. Optional optimization passes are performed between them.

TCG requires dedicated code to support each supported architecture. Additionally TCG requires that the target instruction translation be rewritten to take advantage of TCG ops.

Starting with QEMU Version 0.10.0, TCG ships with the QEMU stable release.

APPENDIX 3.1.3 Serializing guests

QEMU allows guest operating systems to be serialized for guest migration support as well as saving and loading guests.. However, QEMU compresses the guest memory blocks which renders deduplication of guests impossible.

APPENDIX 3.2 KVM

The Kernel-based Virtual Machine (KVM)[Kiv07] is a virtualization infrastructure for the Linux kernel. KVM supports native virtualization on processors with hardware virtualization extensions. KVM originally supported x86 and x86_64 processors and has been ported to other platforms such as S/390, PowerPC and IA-64.

A wide variety of guest operating systems work with KVM, including many flavors of Linux, BSD, Solaris, Windows and OS X.

Limited para-virtualization support is available for Linux and Windows guests using the Virt I/O framework. This supports a para-virtual Ethernet card, a para-virtual disk I/O controller, a balloon device for adjusting guest memory-usage, and a VGA graphics interface using SPICE or VMware drivers. Linux 2.6.20 (released February 2007) was the first to include KVM

Unlike Lguest, KVM is an industrial grade hypervisor and has a significantly larger code base.

APPENDIX 3.3 Using KVM and QEMU technologies in our systems

We are using QEMU as a system virtual machine platform for VM migration in AMirror discussed in chapter 4.

APPENDIX 3.3.1 Guest serializations for VM migration

Both our Asynchronous mirror product and our current in-development infrastructure as a service product relies on KVM and QEMU migration and guest serialization.

APPENDIX 4 LLVM - LOW LEVEL VIRTUAL MACHINE

The Low Level Virtual Machine (LLVM) is an optimized compiler and execution infrastructure[LA04].

LLVM, originally designed for C, C++ and Objective-C, supports now many more programming languages including Fortran, Ada, Haskell, Java, Python and more.

The LLVM project was founded by Vikram Adve and Chris Lattner at the University of Illinois at 2000, and released under BSD-like open source license. In 2005 Apple Inc. hired LLVM project founder Chris Lattner and started supporting the project, which was later integrated into Apple's Mac OS X operation system. Today LLVM is an integral part of apple development tools for both Mac OS X an iOS.

LLVM is a live project that changes constantly. The majority of this chapter describes LLVM 2.9 and is based among others on [Lat11].

APPENDIX 4.1 Motivation

LLVM developers felt existing compilers, mainly GCC, were based on old and complicated code base. Old compilers were not designed to be modular and therefore, changing a specific component or a behaviour was difficult. Furthermore, due to the aging code base and complicated structure, compilation times were very long.

APPENDIX 4.2 System Design

LLVM's primary goal was to build a set of modular compiler components for each of the compiler tasks, including code generation for different architectures (x86, ARM, PowerPC, etc.), generating ELF[San] and other executable formats, generating DWARF[Eag07] debug symbols, linking, just-in-time compiling and optimizing, I/O handling and front ends for various programming languages.

These compiler components were easy to construct and were shared among several compilers. Furthermore, the right tools for one implementation may not be suitable for another, so each implementation can now share the right tools for the job.

Dividing a large system as compiler to a smaller components has several benefits. Firstly, development of smaller components is much easier, and allows the code to be maintained easily and for its documentation to remain readable. Secondly, many components can be reused by compiler for different languages: once the language processor has generated intermediate representation (LLVM assembly) then the same optimization steps can run regardless of the original programming language. Similarly the LLVM intermediate representation can be translated to many different architectures with all architectures benefiting from LLVM's just-in-time compiler and other optimizations simply by replacing the machine code emitter component.

The LLVM community is a strong, industrial supported community that provides help to developers wishing to create new compiler tools.

APPENDIX 4.3 Intermediate Representation

The intermediate representation is designed to provide high-level information about programs that is needed to support sophisticated analyses and transformations, while being low-level enough to represent arbitrary programs and to permit extensive optimization in static compilers.

LLVM provides an infinite set of typed virtual registers in Static Single Assignment (SSA) form, which can hold values of primitive types. SSA form provides the data flow graph, which can be utilized by various optimizations, including the one incorporated in LLVM-prefetch. LLVM also makes the control flow graph of every function explicit in the representation. A function is a set of basic blocks, and each basic block is a sequence of LLVM instructions, ending in exactly one terminator instruction. Each terminator explicitly specifies its successor basic blocks.

LLVM defines three equivalent forms of the intermediate representation: textual, binary, and in-memory (i.e., the compiler's internal) representations. Being able to convert LLVM code between these representations without information loss makes debugging transformations much simpler, allows test cases to be written easily, and decreases the amount of time required to understand the in-memory representation.

APPENDIX 4.4 Compiler

The LLVM compiler supports two front ends. The traditional GNU GCC front end and it's own Clang front end.

APPENDIX 4.4.1 GCC front-end

LLVM was originally written to be a replacement for the code generator in GCC stack[LA03]. Using the standard GCC front-end allows LLVM to support many programming languages including Ada, Fortran, etc.

Using the GCC front-end we relay on GCC to compile the programming language to intermediate representation and we use LLVM infrastructure to compile the intermediate representation, link and execute the byte code.

This front-end is now deprecated and the Clang front-end is preferred.

APPENDIX 4.4.2 Clang front-end

Clang is the new interface for LLVM to compile C-derived languages (C, C++, Objective-C). With the second version of LLVM, its community started to replace the old llvm-gcc 4.2 interface in favor of the newer Clang front end[Lat07]. Clang is a completely new environment not based on GCC sources.

The motivation behind Clang development includes easier integration with modern integrated development environment, wider support for multi-threading, and clearer error messages.

APPENDIX 4.4.3 Just-in-time Compiler optimization

LLVM performs compiler and runtime optimizations. for example take this code designed to switch between two BGRA444R to BGRA8888 video formats.

```
for each pixel {
  switch (infmt) {
    case RGBA5551:
       R = (*in >> 11) & C
       G = (*in >> 6) & C
       B = (*in >> 1) & C
  ...
  }
  switch (outfmt) {
    case RGB888:
      *outptr = R << 16 |
                G << 8 ...
  }
}
```

This code runs on every pixel and performs both switch statements on each pixel. LLVM can detect that the switch is unnecessary since it always results in the same branch. Removing the switch statements collapses the code to

```
for each pixel {
 R = (*in >> 11) & C;
  G = (*in >> 6) & C;
  B = (*in >> 1) & C;
  *outptr = R << 16 |
ï¿Œï¿Œï¿Œï¿Œï¿ŒG << 8 ...
ï¿Œ}
```

This compiler optimization may result in up to 19.3x speed-up and 5.4x average speed up according to [Lat08].

APPENDIX 4.5 Code Generation

Before execution, a code generator is used to translate from the intermediate representation to native code for the target platform, in one of two ways. In the first option, the code generator is run statically, to generate high performance native code for the application, possibly using expensive code generation techniques.

Alternatively, a just-in-time Execution Engine can be used which invokes the appropriate code generator at runtime, translating one function at a time for execution.

LLVM-prefetch uses the Execution Engine to generate native code for the newly constructed program slice.

APPENDIX 4.6 Virtual Machine Environment

As its name suggests LLVM is a "low-level" virtual machine. In contrast with other popular virtual machines such as JVM[LY99] or CLR[Box02], LLVM doesn't provide memory management and abstraction. LLVM relies on the user to handle these issues.

LLVM includes an interpreter, just-in-time compiler and ahead-of-time compiler. LLVM allows the user to use native shared libraries, even those not compiled for LLVM.

APPENDIX 4.7 Other tools

The LLVM project includes a rich tool-chain that includes LLDB — a debugger, and various other tools for linkage, library, and binary manipulation. These tools were not critical part in the efforts and are only briefly described for completeness.

APPENDIX 4.8 How LLVM is being used in our systems

We are using LLVM in LLVM-prefetch system. We have also considered using LLVM in Truly-Protect but found it unsuitable for this purpose.

APPENDIX 4.8.1 LLVM-Prefetch

We use LLVM run-time optimization environment to construct and run our pre-execution threads. LLVM's intermediate representation allowed to construct the dynamic slices of the program easily.

APPENDIX 4.8.2 Truly-Protect

We considered using LLVM in Truly-protect. Unfortunately, LLVM holds the intermediate representation of the program in the memory in an unprotected form. The intermediate representation is not just a buffer stored in the memory but rather an object graph incorporating covering a significant portion of LLVM's code. Thus, wrapping these objects with a cryptographic layer is not cost-effective.

APPENDIX 5 LINUX BLOCK I/O LAYER

Linux block I/O layer is an inside layer that resides between the layers that request I/O operations such as file systems (usually) and the disk drives.

From the file system perspective it provides identical interfaces to all storage devices regardless of storage medium. From the disk perspective it is the only interface that generate requests to the disks.

This chapter describes state of the kernel 2.6.35 which is the version we used for AMirror development. (chapter 4)

Due to the complete lack of documentation on the block layer internals and due to the fundamental nature of the block layer in our system, this chapter provides a complete documentation of the block layer.

The block layer shadows the layer that makes the request (the file system) from the layer that commits the I/O (the disk). All I/O requests directed to block devices (which includes all disk and tape drives) go through the block I/O layer.

The block layer is a "queue of requests" where the queue is represented by a huge structure called "struct request_queue" which is essentially:

1. Double linked list of "requests" (using kernel list_head link structure) these request were submitted by the upper layers and need to be executed by the block devices. (Hard drives for example)
2. A set of pointers to specific methods defined by the device driver to execute operations on a queue.
3. A pointer to an I/O scheduler module, that is a set of methods which (in addition to the basic functions defined in the block layer) manage the queue (Order the requests in the queue, the minimal time when some request can be fetched from the queue etc.), the I/O scheduler may allocate and manage its own data structures per queue, and to store requests there to put them back into the queue at some specific points only.

APPENDIX 5.1 The Structure of the Block I/O Queue

The I/O queue consists of a list of requests. Each request is a structure that is managed inside the drivers and block layer. The structure that is known to the file systems and page cache is called struct bio (for block I/O) and struct bio consists of an array of structures called io_vecs. Each member of the I/O vec contains a page, offset from the beginning of the page and length of the data in a page.

TABLE 2 struct bio (block I/O) members

Member	Usage
request_queue	Doubly linked list of request.
request bio	List of (adjacent) bios.
Bio	Vector of i/o (io_vec)
buffer_head	The buffer in memory usually resides in kernel space but may be in user space for the case of direct I/O
io_vec	page , offset , length
Page	Pointer to address_space (which is a page cache), Pointer to buffer_head (which may be linked in a circled list (by means of *b_this_page field). The page usually represents a block of 4KB whose beginning address divides evenly on 4096 (or 8 sectors) , that is why the direct I/O blocks should comply with this limitations also.

All the addresses and lengths defined in the block layer are in granularity of complete sectors. Sectors are the minimal unit a disk / DMA controller may process, usually 512 bytes.

APPENDIX 5.2 Typical Block Layer Use Scenario

Struct bio request is submitted to the block layer, by calling to submit_bio() (detailed in 5.8). When bio is submitted the block layer verifies the bio integrity. The block later checks that the bio is sent to a block device that is actually a partition. If it is sent to a partition it remaps the sectors in the bio to the actual addresses (adds the partition offset). The block layer then calls __make_request() (see section 5.8) to build a request struct from a bio struct and push it into a request queue. At first make_request() may do some memory re mappings (bouncing highmem/lomem) then asks the I/O scheduler if this bio is adjacent to some existing request, if so it merges this bio into the existing request (Under the assumption that the request is made to a magnetic hard disk were seeking has high cost) and accounts for merged requests. Otherwise the block layer calls to get_request_wait() (see section 5.9) to allocate a new request structure . If the queue is full the Block layer process will sleep until the queue will free up and only then will get a desired structure. When a new structure is allocated it is pushed into queue by means of __elv_add_request() (see section 5.9).

The I/O device may then be plugged or unplugged (see section 5.11). When the request_fn() callback function is called (by generic_unplug_device() for example) the driver starts to fetch requests from the queue by calling to blk_fetch_request() (see fetch request section). Then the block layer transforms the request into a scatter gather list by a call to blk_rq_map_sg() ,passes this list to a DMA controller and then goes to sleep. When the DMA controller raises an interrupt reporting successful completion of a request a device driver then completes the request by calling to blk_end_request() (see completion section). The driver then fetches and starts to process next request from a queue. The function blk_end_request() calls the completion methods of all the bios in the request, accounts for completed request, response time of a request and number of completed bytes. Afterward we free and deallocate the request by calling to blk_put_request()(see section 5.14).

APPENDIX 5.3 Services the Block Layer Provides

The block layer provides the following services to the upper layer dependent on it. (such as file system)

partition remapping Correct I/O request addresses to physical address to account for partition location on the disk.

highmem/lomem remapping] If the DMA controller does not support highmem (above 4GB) the memory is remapped.

Bidirectional requests(bidi) support Some devices may both read and write operations in one DMA, thus a request in the opposite direction may be appended to another request by the block layer.

Timeout handling A time limit is appended to each request and when it times out a special method is called to handle the error case.

Integrity checking An integrity payload may be appended to each request. The integrity payload can be used to check its validity (upon reading).

Barrier requests Special requests can be submitted to force all requests in the queue be flushed, and to define a strict order on a request, such that no request that arrives after the barrier will be serviced before the flushing process finishes.

Tag management for tagged (TCQ) requests The block layer can add tags to define the order on the requests and to ensure the requests will be serviced by a disk device (for queue management / flushing purposes). TCQ applies to SCSI devices. SATA devices define a similar method for their internal queues management called NCQ.

Elevator The block layer includes the I/O scheduler (also called elevator. The block layer is highly flexible and has the ability to change its behavior almost entirely by defining new I/O schedulers.

Cgroups The block layer manages priorities between processes (actually groups of processes) , for block I/O and to make a per process group block for I/O accounting (implemented inside CFQ I/O scheduler only).

Per partition I/O accounting (separately for reads and writes) The block layer tracks total I/O operations completed, total bytes completed, cumulative response time, total merges and queue sizes.

Handle partial requests completion some devices can not complete a request in one DMA command. Thus it is necessary to have the option to complete only a partial number of bytes of the request and track multiple fragments for request completion.

queueing The block layer is responsible for requeuing the existing requests in separate drivers queue.

APPENDIX 5.4 The Block Layer Interface to Disk Drives

The following methods are defined by each I/O driver (such as disk controller driver). These methods are the interfaces that the block layer has with the underlying disk drive layer.

TABLE 3 I/O driver interfaces

Method	Usage
request_fn_proc *request_fn	The entry point to the driver itself. When called, the driver should start executing I/O request. The I/O requests are fetched from the queue one by one. (see section 5.11)
make_request_fn *make_request_fn	Pushes a bio into a queue (see section 5.8) (usually __make_request()).
prep_rq_fn *prep_rq_fn	May process the request after it is fetched from the queue. (see section 5.12) For most devices this function is NULL.
unplug_fn *unplug_fn	Unplug event handler (see section 5.11) For most devices this is generic_unplug_device()).
merge_bvec_fn *merge_bvec_fn	This function is not called by the block layer (but it is called by I/O drivers) The code comments - *"Usually queues have static limitations on the max sectors or segments that we can put in a request. Stacking drivers may have some settings that are dynamic, and thus we have to query the queue whether it is OK to add a new bio_vec to a bio at a given offset or not. If the block device has such limitations, it needs to register a merge_bvec_fn() to control the size of bio's sent to it"*.
prepare_flush_fn *prepare_flush_fn	Called before a block layer issues a flush on a queue. (see section 5.13).
softirq_done_fn *softirq_done_fn	Called on a completion of a software irq on a block I/O.
rq_timed_out_fn *rq_timed_out_fn	Called when the requests timed outs for error handling, (see start_request method in section 5.12 , and finish_request method in section 5.15).
dma_drain_needed_fn *dma_drain_needed	From the comment to this function. *"Some devices have excess DMA problems and can't simply discard (or zero fill) the unwanted piece of the transfer. They have to have a real area of memory to transfer it into..."* This function should return true if this is a case, if so the queue, "silently" appends a buffer dma_drain_buffer of size dma_drain_size into each scatter_gather list.
lld_busy_fn *lld_busy_fn	*"...Check if underlying low-level drivers of a device are busy this function is used only by request stacking drivers to stop dispatching requests to underlying devices when underlying devices are busy..."* Returns true if the device is busy.

APPENDIX 5.5 The I/O Scheduler (Elevator)

The is a set of methods, which order and make decisions about each step of the I/O request life cycle. i.e. allocation, insertion into queue, merging with other requests, fetching from queue, completion and deallocation.

The I/O scheduler acts as an "elevator" in which the request "goes up and down" queues. Therefore the name elevator is used for the I/O scheduler in the kernels and both terms will be used interchangeably in this work.

The Linux kernel developers intention was to make I/O scheduler store the requests inside its own data structures and then put the requests back into a main queue during the call to make_request, however as time and version went by schedulers (especially CFQ) became so complicated that many of the block layers logic , was re-implemented inside them. For example CFQ (The default I/O scheduler) stores many queues and calls directly to devices driver request_fn() (by run_queue() for ex.) while giving the driver one of its internal queues as a parameter (which overrides the plug/unplug mechanism). So currently the I/O scheduler is more closely tied to the block layer and therefore we cover it. The kernel currently contains 3 implementations of I/O schedulers:

noop The Noop scheduler takes no decision (does nothing) it manages a FIFO queue and responses are handled in first come first served policy.

CFQ (complete fair queuing) hashes between the requests and the ids of the processes that issued them (I/O contexts), stores queue for each hashed entry and service them in a round robin manner. This I/O scheduler is the elevator analog of the CFS in task scheduling in Linux.

deadline The older I/O scheduler. This I/O scheduler processes I/O requests with emphasis on the completion deadline.

TABLE 4 The methods that the I/O scheduler implements

Function	Usage
elevator_merge_fn	Called on query requests for merge with a bio
elevator_merge_req_fn	Called when two requests are merged. The request that gets merged into the other will be never seen by the I/O scheduler again. I/O-wise, after being merged, the request is gone.
elevator_merged_fn	Called when a request in the scheduler has been involved in a merge. It is used in the deadline scheduler to re-position the request if its sorting order has changed.
elevator_allow_merge_fn	Called when the block layer tries to merge bios. The I/O scheduler may still want to stop a merge at this point if the merge will results in some sort of internal conflict. This hook allows the I/O scheduler to do that.
elevator_dispatch_fn	Fills the dispatch queue with ready requests. I/O schedulers are free to postpone requests by not filling the dispatch queue unless the "force" argument is non-zero. Once dispatched, I/O schedulers are not allowed to manipulate the requests as they belong to generic dispatch queue.
elevator_add_req_fn	Called to add a new request into the scheduler
elevator_queue_empty_fn	Returns true if the merge queue is empty.
elevator_former_req_fn	Returns the request before the one specified in disk sort order.
elevator_latter_req_fn	Returns the request after the one specified in disk sort order.
elevator_completed_req_fn	Called when a request is completed.
elevator_may_queue_fn	Returns true if the scheduler wants to allow the current context to queue a new request even if it is over the queue limit.
elevator_set_req_fn	Allocate elevator specific storage for a request.
elevator_put_req_fn	Deallocate elevator specific storage for a request.
elevator_activate_req_fn	Called when device driver first sees a request. I/O schedulers can use this callback to determine when actual execution of a request starts.
elevator_deactivate_req_fn	Called when device driver decides to delay a request by requeuing it
elevator_init_fn	Allocates elevator specific storage for a queue.
elevator_exit_fn	Deallocates elevator specific storage for a queue.

APPENDIX 5.6 File List

The block layer code is located under ./block/ directory in the kernel source tree. Headers files, like blkdev.h are located under ./include/linux/

TABLE 5 The methods that the I/O scheduler implements

File	Contents
blk-barrier.c	Barrier request management , queue flushing
blk-cgroup.c, blk-cgroup.h	Cgroup block I/O management
blk-core.c	Main functions of the block layer
blk-exec.c	Helper functions and wrappers
blk.h	Definitions and static inline methods
blk-integrity.c	Integrity functionality
blk-ioc.c	I/O context, links between a process and a specific I/O request, used for request allocation (see section 5.9) and by CFQ I/O scheduler
blk-iopoll.c	I/O multiplexing
blk-lib.c	Helper functions and wrappers
blk-map.c	Memory mapping functions between bios and requests
blk-merge.c	Functions to merge adjacent requests and blk_rq_map_sg()
blk-settings.c	Initialization of the data structures
blk-softirq.c	Software IRQ
blk-sysfs.c	/sys filesystem entries that refer to the block layer
blk-tag.c	Tagged requests management
blk-timeout.c	Timeout management for error handling
bsg.c	Block layer SCSI generic driver
cfq.h, cfq-iosched.c	CFQ I/O scheduler
compat_ioctl.c	Ioctl compatibility
deadline-iosched.c	Deadline I/O scheduler
elevator.c	Common I/O schedulers functions and wrappers
genhd.c	Gendisk data structure. Gendisk is the abstract base class for all disk drives.
ioctl.c	IOCTL implementation
noop-iosched.c	No-op I/O scheduler
scsi_ioctl.c	IOCTL for scsi devices.

APPENDIX 5.7 Data structure

The following figure describe the block layer data structure relations

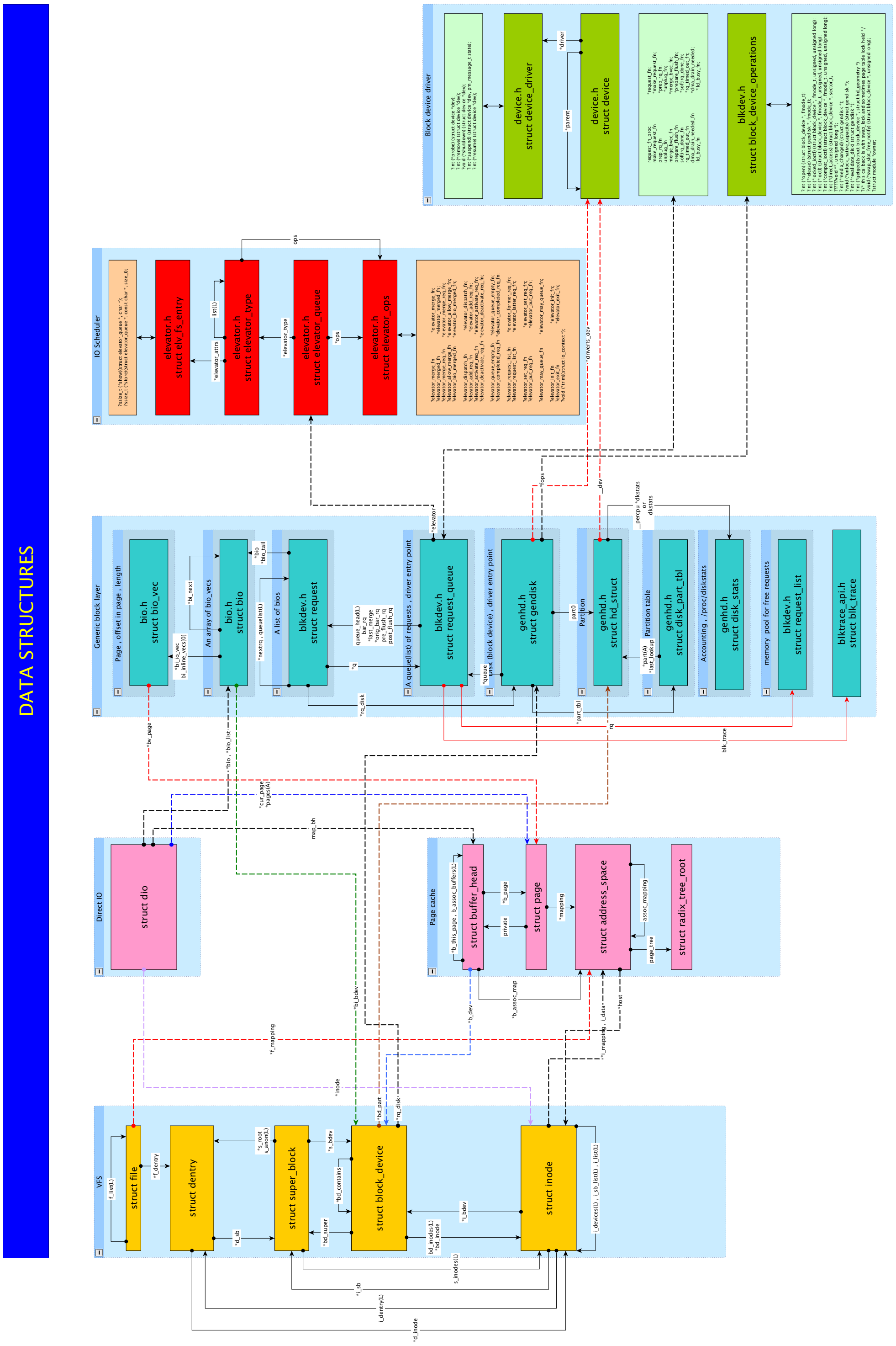


FIGURE 51 Block layer data structures

APPENDIX 5.8 Block I/O Submission

Block I/O submission is the process of turning a bio into a request and inserting it into the queue (see section 5.10). The functions that handle bio submission must prepare the bio and pass the request in a form appropriate to a specific lower-level driver. These functions are responsible for the following procedures:

1. Memory reallocation (high-memory to low-memory) of a request, if the driver cannot hold a request in high memory (above 4GB) then the block layer creates a copy of the bio with identical data (destination buffers/page) in low-memory area, and defines its completion routine to restore the original bio and copy (in the case of read request) the data back to the original buffer for the application to use.
2. Perform bio verification, like limits on its size , checking whether it does not extends the physical size of a device , or some threshold in queue. (the bio will fail if it does not pass the verification.)
3. Partition remapping, the device driver doesn't know about partitions. However, the addresses in the io_vec array of the bio are relative to the start of the partition and should be corrected relative to the beginning of a physical device. The block layer perform this correction.

submit_bio() is the block layer interface toward higher layers. (Usually the file system and page cache which calls submit_bio through the submit_bh() function call. These are the main interface the block layer provide the upper layers.

blk_make_request() Allocates a request and insert the whole bio chain. Direct pointers, instead of a common list structure, chain the bios. A chain of bios forms a request, this function assumes that the bio points to a chain of bios. This chain is may be a part of some reassembled request by a stacking drivers, such as software RAID. This function does not insert the request into the queue and returns it to the caller.

blk_requeue_request() Requeue the bio request. Drivers often keep queuing requests until the hardware cannot accept any more, when that condition happens we need to put the request back in the queue. (See section 5.10). This function clears the request timeout by stopping its timer (see section 5.9), clearing its completion flag state, and ending its tagged status(if it was tagged), then this function requeues the request. (i.e. putting it back into queue (see section 5.10)).

blk_insert_request() Practically unused. Inserts a soft barrier request into queue (see section 5.10), from the comment to the function: *"... Many block devices need to execute commands asynchronously, so they don't block the whole kernel from preemption during request execution. This is accomplished normally by inserting artificial requests tagged as REQ_TYPE_SPECIAL in to the corresponding request queue, and letting them be scheduled for actual execution by the request queue. .."*

generic_make_request() This function delivers a buffer to the right device driver for I/O by receiving a new bio, getting the queue associated with it's block device, verifying the bio against the queue, performing the partition remapping , and verifying it again and then trying to insert it into queue by calling to q->make_request_fn().

The make request function is usually __make_request() but may be some other function specified by the driver (for example some stacking drivers for software RAIDS or network devices want to override the I/O scheduler) , thus specifying their own make_request(). The comments to the function say

"... generic_make_request() is used to make I/O requests of block devices. It is passed a &struct bio, which describes the I/O that needs to be done. generic_make_request() does not return any status. The success/failure status of the request, along with notification of completion, is delivered asynchronously through the bio->bi_end_io function (bio_endio() , req_bio_endio()). (see section 5.15)

The caller of generic_make_request must make sure that bi_io_vec are set to describe the memory buffer, and that bi_dev and bi_sector are set to describe the device address, and the bi_end_io and optionally bi_private are set to describe how completion notification should be signaled.

generic_make_request and the drivers it calls may use bi_next if this bio happens to be merged with someone else, and may change bi_dev and bi_sector for remaps as it sees fit. So the values of these fields should NOT be depended on after the call to generic_make_request. ..."

Generic make request is described in detail in the figure 52.

submit_bio() This function is the most common entry point to the block layer and its logic is described in figure 53. submit_bio() makes some verification and accounting and calls to generic_make_request(), this is the most common method to submit bio into a generic block layer.

__make_request() This is the most commonly used function for making request. It is described in detail in figure 54

__make_request() first performs the memory remapping on the request (from high to low memory). Then, __make_request() calls the I/O scheduler in an attempt to find a request for the bio to be merged with. If __make_request() fails to find a bio request to merge with then __make_request() will allocates a new request and inserts the new request into the queue (see section 5.10. __make_request() also tries to merge sequential requests. This functionality is defined in blk-merge.c.

blk_execute_rq_nowait(), blk_execute_rq() These functions insert a fully prepared request at the back of the I/O scheduler queue for execution and do not wait/wait for completion. When waiting for completion the function

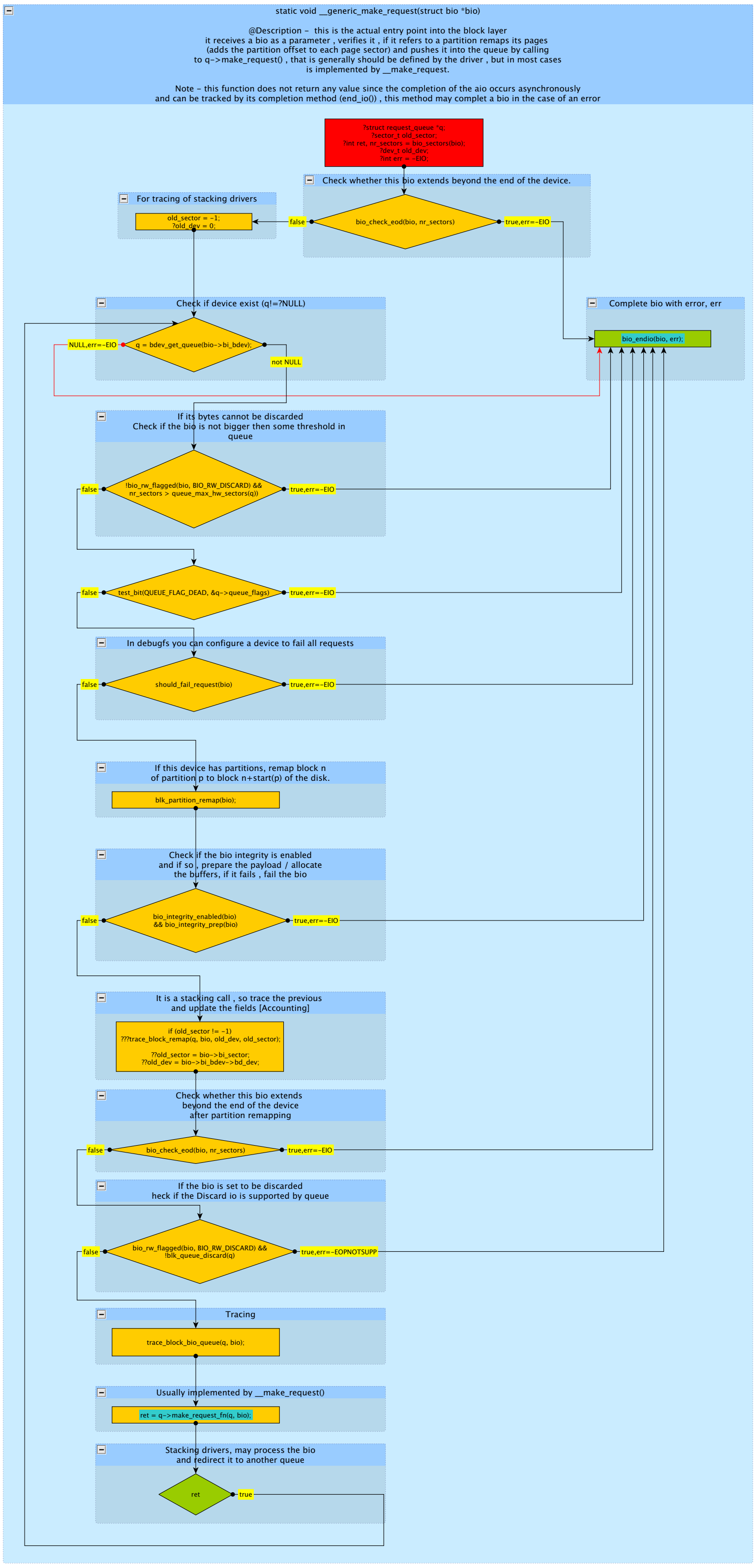

static void __generic_make_request(struct bio *bio)
@Description – this is the actual entry point into the block layer
it receives a bio as a parameter , verifies it , if it refers to a partition remaps its pages
(adds the partition offset to each page sector) and pushes it into the queue by calling
to q->make_request() , that is generally should be defined by the driver , but in most cases
is implemented by __make_request.
Note – this function does not return any value since the completion of the aio occurs asynchronously
and can be tracked by its completion method (end_io()) , this method may complet a bio in the case of an error
?struct request_queue *q;
?sector_t old_sector;
?int ret, nr_sectors = bio_sectors(bio);
?dev_t old_dev;
?int err = –EIO;
Check whether this bio extends beyond the end of the device.
bio_check_eod(bio, nr_sectors)
false
true,err=–EIO
For tracing of stacking drivers
old_sector = –1;
?old_dev = 0;
Check if device exist (q!=?NULL)
q = bdev_get_queue(bio->bi_bdev);
NULL,err=–EIO
not NULL
Complete bio with error, err
bio_endio(bio, err);
If its bytes cannot be discarded
Check if the bio is not bigger then some threshold in
queue
!bio_rw_flagged(bio, BIO_RW_DISCARD) &&
nr_sectors > queue_max_hw_sectors(q))
false
true,err=–EIO
test_bit(QUEUE_FLAG_DEAD, &q->queue_flags)
false
true,err=–EIO
In debugfs you can configure a device to fail all requests
should_fail_request(bio)
false
true,err=–EIO
If this device has partitions, remap block n
of partition p to block n+start(p) of the disk.
blk_partition_remap(bio);
Check if the bio integrity is enabled
and if so , prepare the payload / allocate
the buffers, if it fails , fail the bio
bio_integrity_enabled(bio)
&& bio_integrity_prep(bio)
false
true,err=–EIO
It is a stacking call , so trace the previous
and update the fields [Accounting]
if (old_sector != –1)
???trace_block_remap(q, bio, old_dev, old_sector);
??old_sector = bio->bi_sector;
??old_dev = bio->bi_bdev->bd_dev;
Check whether this bio extends
beyond the end of the device
after partition remapping
bio_check_eod(bio, nr_sectors)
false
true,err=–EIO
If the bio is set to be discarded
heck if the Discard io is supported by queue
bio_rw_flagged(bio, BIO_RW_DISCARD) &&
!blk_queue_discard(q)
false
true,err=–EOPNOTSUPP
Tracing
trace_block_bio_queue(q, bio);
Usually implemented by __make_request()
ret = q->make_request_fn(q, bio);
Stacking drivers, may process the bio
and redirect it to another queue
ret
true

FIGURE 52 Generic make request flow

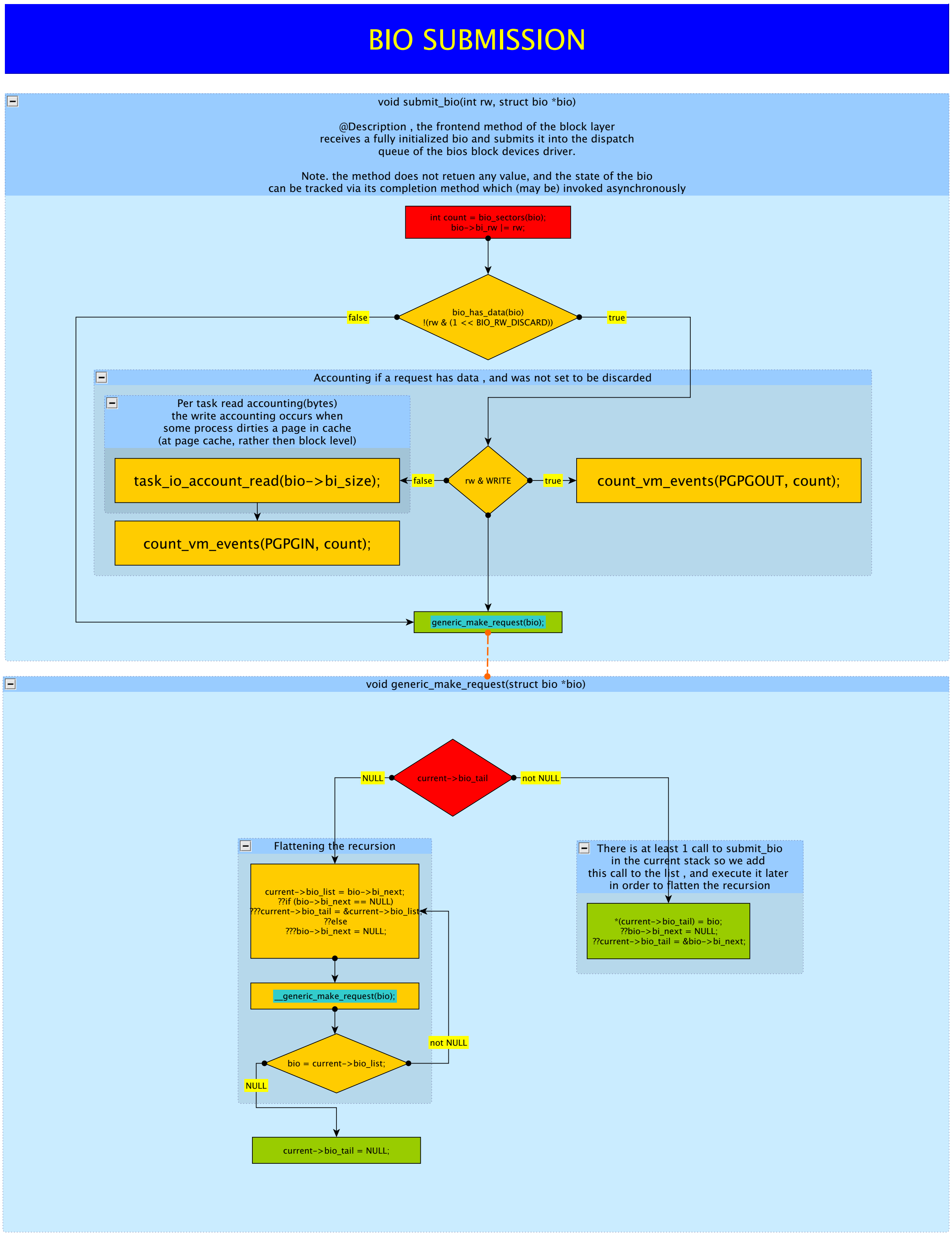

BIO SUBMISSION
void submit_bio(int rw, struct bio *bio)
@Description , the frontend method of the block layer
receives a fully initialized bio and submits it into the dispatch
queue of the bios block devices driver.
Note. the method does not retuen any value, and the state of the bio
can be tracked via its completion method which (may be) invoked asynchronously
int count = bio_sectors(bio);
bio->bi_rw |= rw;
bio_has_data(bio)
!(rw & (1 << BIO_RW_DISCARD))
false
true
Accounting if a request has data , and was not set to be discarded
Per task read accounting(bytes)
the write accounting occurs when
some process dirties a page in cache
(at page cache, rather then block level)
task_io_account_read(bio->bi_size);
false
rw & WRITE
true
count_vm_events(PGPGOUT, count);
count_vm_events(PGPGIN, count);
generic_make_request(bio);
void generic_make_request(struct bio *bio)
NULL
current->bio_tail
not NULL
Flattening the recursion
current->bio_list = bio->bi_next;
??if (bio->bi_next == NULL)
???current->bio_tail = &current->bio_list;
??else
???bio->bi_next = NULL;
__generic_make_request(bio);
not NULL
bio = current->bio_list;
NULL
current->bio_tail = NULL;
There is at least 1 call to submit_bio
in the current stack so we add
this call to the list , and execute it later
in order to flatten the recursion
*(current->bio_tail) = bio;
??bio->bi_next = NULL;
??current->bio_tail = &bio->bi_next;

FIGURE 53 submit bio flow

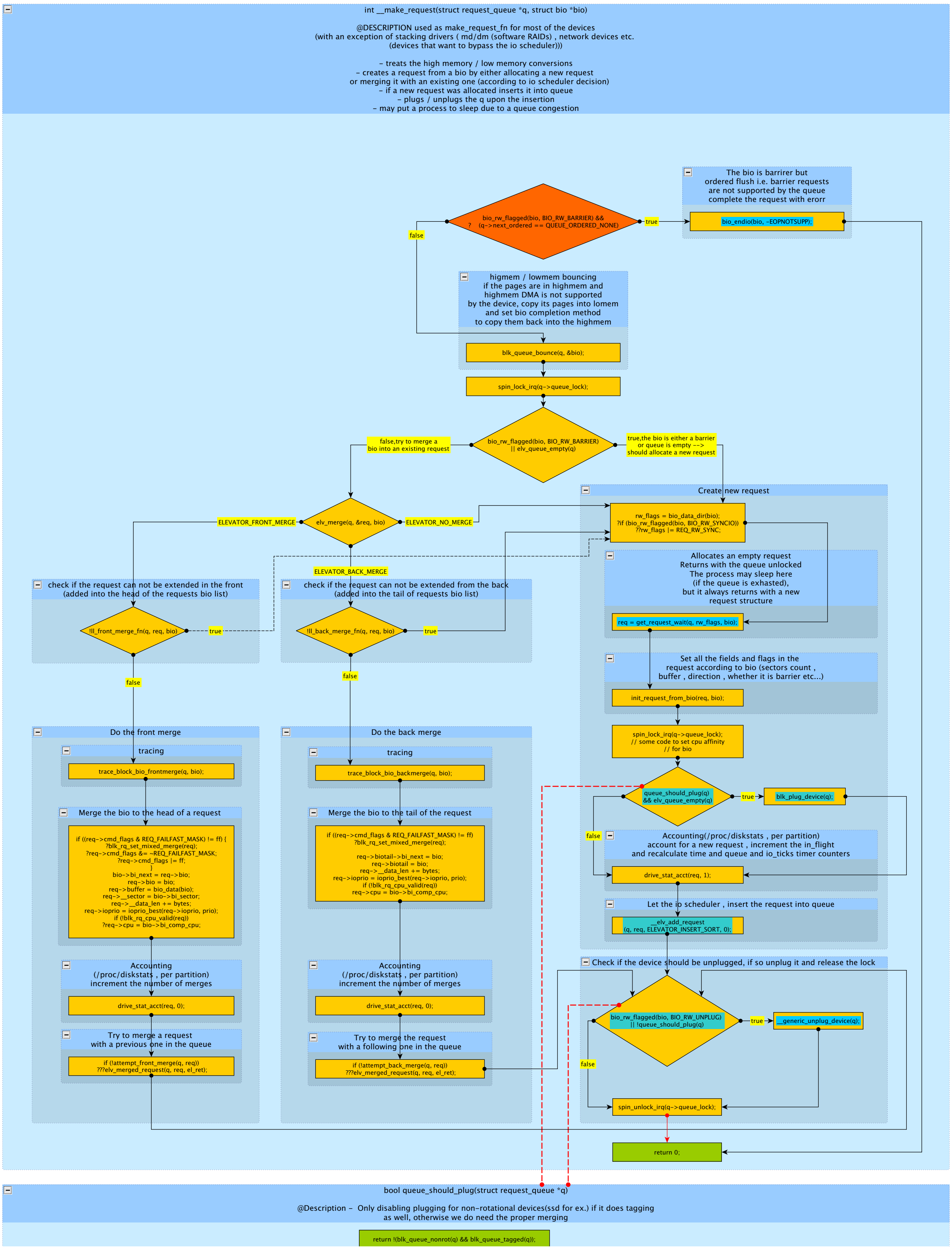

int __make_request(struct request_queue *q, struct bio *bio)
@DESCRIPTION used as make_request_fn for most of the devices
(with an exception of stacking drivers (md/dm (software RAIDs) , network devices etc.
(devices that want to bypass the io scheduler)))
- treats the high memory / low memory conversions
- creates a request from a bio by either allocating a new request
or merging it with an existing one (according to io scheduler decision)
- if a new request was allocated inserts it into queue
- plugs / unplugs the q upon the insertion
- may put a process to sleep due to a queue congestion
bio_rw_flagged(bio, BIO_RW_BARRIER) &&
? (q->next_ordered == QUEUE_ORDERED_NONE)
false
true
The bio is barrirer but
ordered flush i.e. barrier requests
are not supported by the queue
complete the request with erorr
bio_endio(bio, -EOPNOTSUPP);
higmem / lowmem bouncing
if the pages are in highmem and
highmem DMA is not supported
by the device, copy its pages into lomem
and set bio completion method
to copy them back into the highmem
blk_queue_bounce(q, &bio);
spin_lock_irq(q->queue_lock);
false,try to merge a
bio into an existing request
bio_rw_flagged(bio, BIO_RW_BARRIER)
|| elv_queue_empty(q)
true,the bio is either a barrier
or queue is empty -->
should allocate a new request
ELEVATOR_FRONT_MERGE
elv_merge(q, &req, bio)
ELEVATOR_NO_MERGE
ELEVATOR_BACK_MERGE
check if the request can not be extended in the front
(added into the head of the requests bio list)
!ll_front_merge_fn(q, req, bio)
true
false
check if the request can not be extended from the back
(added into the tail of requests bio list)
!ll_back_merge_fn(q, req, bio)
true
false
Create new request
rw_flags = bio_data_dir(bio);
?if (bio_rw_flagged(bio, BIO_RW_SYNCIO))
??rw_flags |= REQ_RW_SYNC;
Allocates an empty request
Returns with the queue unlocked
The process may sleep here
(if the queue is exhasted),
but it always returns with a new
request structure
req = get_request_wait(q, rw_flags, bio);
Set all the fields and flags in the
request according to bio (sectors count ,
buffer , direction , whether it is barrier etc...)
init_request_from_bio(req, bio);
spin_lock_irq(q->queue_lock);
// some code to set cpu affinity
// for bio
queue_should_plug(q)
&& elv_queue_empty(q)
true
blk_plug_device(q);
false
Accounting(/proc/diskstats , per partition)
account for a new request , increment the in_flight
and recalculate time and queue and io_ticks timer counters
drive_stat_acct(req, 1);
Let the io scheduler , insert the request into queue
__elv_add_request
(q, req, ELEVATOR_INSERT_SORT, 0);
Do the front merge
tracing
trace_block_bio_frontmerge(q, bio);
Merge the bio to the head of a request
if ((req->cmd_flags & REQ_FAILFAST_MASK) != ff) {
?blk_rq_set_mixed_merge(req);
?req->cmd_flags &= ~REQ_FAILFAST_MASK;
?req->cmd_flags |= ff;
}
bio->bi_next = req->bio;
req->bio = bio;
req->buffer = bio_data(bio);
req->__sector = bio->bi_sector;
req->__data_len += bytes;
req->ioprio = ioprio_best(req->ioprio, prio);
if (!blk_rq_cpu_valid(req))
?req->cpu = bio->bi_comp_cpu;
Accounting
(/proc/diskstats , per partition)
increment the number of merges
drive_stat_acct(req, 0);
Try to merge a request
with a previous one in the queue
if (!attempt_front_merge(q, req))
???elv_merged_request(q, req, el_ret);
Do the back merge
tracing
trace_block_bio_backmerge(q, bio);
Merge the bio to the tail of the request
if ((req->cmd_flags & REQ_FAILFAST_MASK) != ff)
?blk_rq_set_mixed_merge(req);
req->biotail->bi_next = bio;
req->biotail = bio;
req->__data_len += bytes;
req->ioprio = ioprio_best(req->ioprio, prio);
if (!blk_rq_cpu_valid(req))
req->cpu = bio->bi_comp_cpu;
Accounting
(/proc/diskstats , per partition)
increment the number of merges
drive_stat_acct(req, 0);
Try to merge the request
with a following one in the queue
if (!attempt_back_merge(q, req))
???elv_merged_request(q, req, el_ret);
Check if the device should be unplugged, if so unplug it and release the lock
bio_rw_flagged(bio, BIO_RW_UNPLUG)
|| !queue_should_plug(q)
true
__generic_unplug_device(q);
false
spin_unlock_irq(q->queue_lock);
return 0;
bool queue_should_plug(struct request_queue *q)
@Description - Only disabling plugging for non-rotational devices(ssd for ex.) if it does tagging
as well, otherwise we do need the proper merging
return !(blk_queue_nonrot(q) && blk_queue_tagged(q));

FIGURE 54 Make request flow

unplugs the device or calls to request_fn directly after the insertion of a request into queue. These functions invokes __elv_add_request(). (see section 5.10)

APPENDIX 5.9 Get Request

The first step in submitting I/O to the block layer is to allocate a request. These are the main goals of the mechanism:

1. Fast and efficient memory allocation.
2. Initializing request fields to default values.
3. Notifying the I/O scheduler about the new I/O.
4. Avoiding queue congestion.
5. Avoiding starvation of requests.

The 1stis achieved by using memory pool, the 3rdby calling to elv_set_request() upon request allocation and the 4th and the 5th by using a special mechanism called batching process or batching context.

A special data structure called request_list is managed in every queue for request allocation: (include/linux/blkdev.h)

```
struct request_list {
/*
 * count[], starved[], and wait[] are indexed by
 * BLK_RW_SYNC/BLK_RW_ASYNC
 */
    int count[2];
/*
 * the number of requests currently allocated
 * in each direction
 */

    int starved[2];

/*
 * whether a queue was unable to allocate a request in this
 * direction * due to memory allocation failure or some
 * I/O scheduler decision
 */

    int elvpriv;
/*whether I/O scheduler should be notified ?*/

    mempool_t *rq_pool;
/* memory pool for requests */

    wait_queue_head_t wait[2];
/*
 * wait queue for a tasks sleeping while
 * waiting for some more requests to become
 * free (failed to allocate a request at first time)
 * The direction of each request is actually is whether
 * it's write = 1 or read, synchronous write=2
 * i.e. the one that avoids write page-cache
 * the counters in each direction are managed separately
 * for more flexibility in congestion \ batching decisions.
 */
};
```

Allocation for the data structure is performed by "block_alloc_request()" and calls to "blk_rq_init()" to initialize the fields. The I/O scheduler is also notified by calling to "elv_set_request()" to allocate its private fields. Each queue status and its triggered behavior is treated separately for both sync and async requests.

APPENDIX 5.9.1 Queue Congestion Avoidance and Batching Context

The main mechanisms of "get_request()" / "put_request() combination at "get_request()" side is processed by a "get_request_wait()" and "get_request()" functions and it works as following:

The queue is considered congested if in one of the directions it has allocated more requests then the value of nr_congestion_on. The simplest way was to send a process to sleep until a queue becomes decongested (i.e has less requests in the specific direction) then nr_congetstion_off. However this could easily lead to a starvation of some processes at high I/O rates. Thus a more flexible mechanism is introduced:

APPENDIX 5.9.2 Congestion and starvation avoidance

Under the new approach for block device congestion the queue has 2 states: congested and full. The queue is considered full when it has at least q->nr_requests = BLKDEV_MAX_RQ = 128 requests, and the queue is considered congested when it has at least q->nr_requests*7/8 = 112 requests. The congested queue is no longer congested when it has at most q->nr_requests*13/16 = 104.

1. If a process notices that the queue will be congested after the allocation, it is marked as congested, if in addition it will become full the process is marked as batching.
2. If a process tries to allocate a request but the queue is already full the request is then added to a kernel wait_queue in the appropriate direction of the request_list of a queue. The process then goes to sleep then until the queue is

unplugged to process its requests and free them by put_request(see section 5.14). By calling put_request we awaken the processes in a wait_queue if the queue becomes non-congested. When the process is awoken it becomes a batching process.

3. A batching process can allocate at least 1 and at most 32 requests (despite the queue been congested) within a predefined time.
4. Even with a lot of batching processes that are awakened one by one the queue may end up with too many requests. Thus if a queue contains too many requests (specifically 3/2*q- >nr_requests) the process is sent to sleep despite being batching.

APPENDIX 5.9.3 Exception to Congestion Avoidance

The data regarding the state of a process is stored in an ioc field of type io_context in task_struct. The I/O scheduler may override the congestion mechanism by returning ELV_MQUEUE_MUST by elv_may_queue()

APPENDIX 5.10 Add Request

This is the main functionality of the I/O scheduler, given a request and a request queue, the I/O schedulers put the request in the right position. The method that does adds the request to queue is part of the general elevator API is elv_add_request(). The add_request() function receives in addition to the request and the queue where it should be inserted 2 additional parameters :

Plug whether the queue should be plugged before the insertion.

Where may take 4 possible values:

1. ELEVATOR_INSERT_FRONT - inserts the request to the front of a main dispatch queue (no interference from the specific scheduler implementation).
2. ELEVATOR_INSERT_BACK - forces the specific scheduler implementation to move all of the requests from its internal data structures to the main dispatch queue, then it insert the request at the tail of the main dispatch queue, and finally run the queue i.e unplug it and let the driver process all the requests. (see section 5.11).
3. ELEVATOR_INSERT_SORT - let the specific implementation of a scheduler insert the request (into its own data structures).
4. ELEVATOR_INSERT_REQUEUE - does front insertion if the queue is not orderly flushed at that very moment. Otherwise insert the request into a position according to an ordered sequence (into the main dispatch queue). In addition it prevents the queue from being unplugged immediately after.

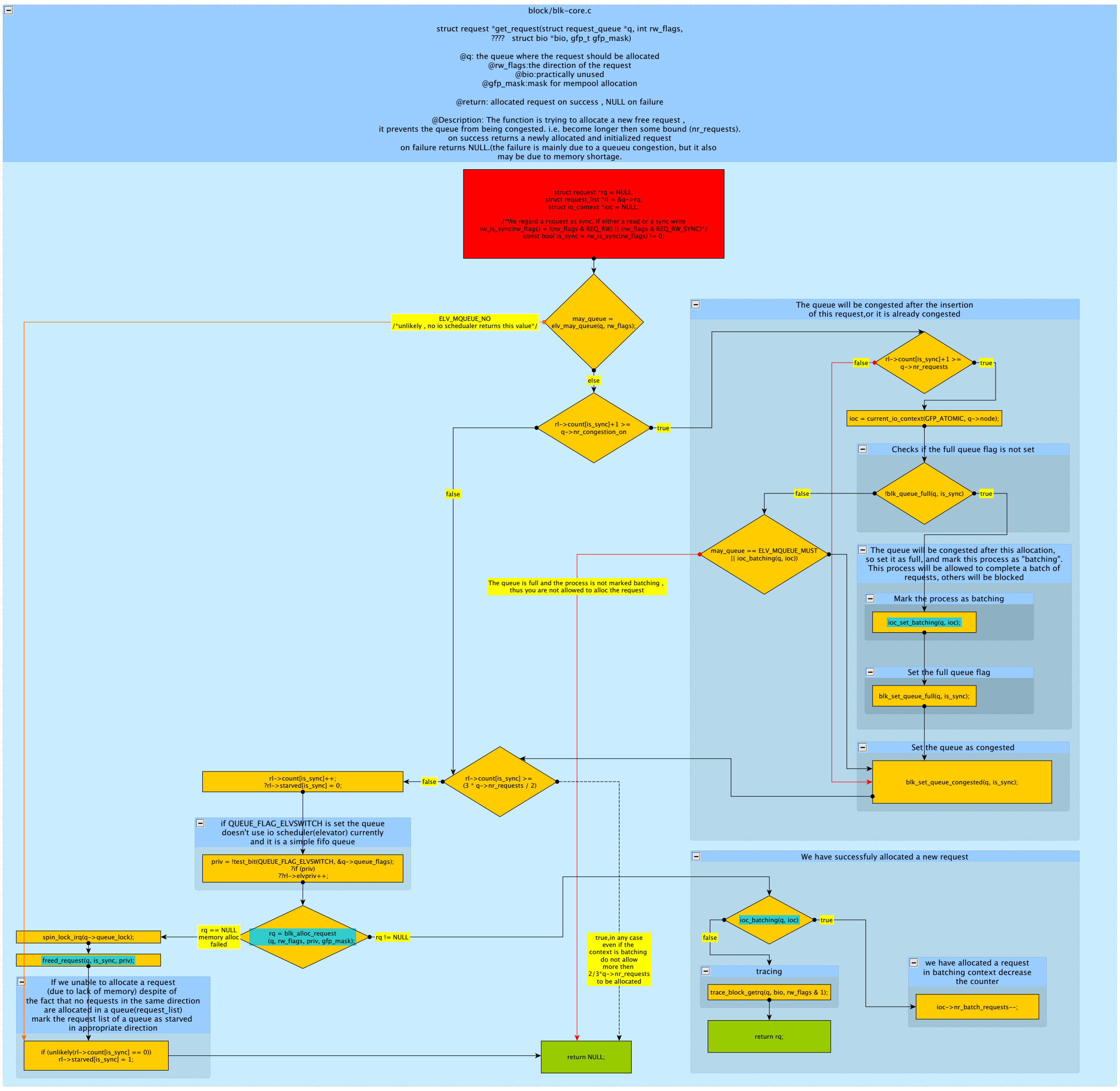

block/blk-core.c
struct request *get_request(struct request_queue *q, int rw_flags,
???? struct bio *bio, gfp_t gfp_mask)
@q: the queue where the request should be allocated
@rw_flags:the direction of the request
@bio:practically unused
@gfp_mask:mask for mempool allocation
@return: allocated request on success , NULL on failure
@Description: The function is trying to allocate a new free request ,
it prevents the queue from being congested. i.e. become longer then some bound (nr_requests).
on success returns a newly allocated and initialized request
on failure returns NULL.(the failure is mainly due to a queueu congestion, but it also
may be due to memory shortage.
struct request *rq = NULL;
struct request_list *rl = &q->rq;
struct io_context *ioc = NULL;
/*We regard a request as sync, if either a read or a sync write
rw_is_sync(rw_flags) = !(rw_flags & REQ_RW) || (rw_flags & REQ_RW_SYNC)*/
const bool is_sync = rw_is_sync(rw_flags) != 0;
may_queue =
elv_may_queue(q, rw_flags);
ELV_MQUEUE_NO
/*unlikely , no io schedualer returns this value*/
else
rl->count[is_sync]+1 >=
q->nr_congestion_on
true
false
The queue will be congested after the insertion
of this request,or it is already congested
rl->count[is_sync]+1 >=
q->nr_requests
false
true
ioc = current_io_context(GFP_ATOMIC, q->node);
Checks if the full queue flag is not set
!blk_queue_full(q, is_sync)
false
true
may_queue == ELV_MQUEUE_MUST
|| ioc_batching(q, ioc))
The queue is full and the process is not marked batching ,
thus you are not allowed to alloc the request
The queue will be congested after this allocation,
so set it as full, and mark this process as "batching".
This process will be allowed to complete a batch of
requests, others will be blocked
Mark the process as batching
ioc_set_batching(q, ioc);
Set the full queue flag
blk_set_queue_full(q, is_sync);
Set the queue as congested
blk_set_queue_congested(q, is_sync);
rl->count[is_sync] >=
(3 * q->nr_requests / 2)
false
rl->count[is_sync]++;
?rl->starved[is_sync] = 0;
if QUEUE_FLAG_ELVSWITCH is set the queue
doesn't use io scheduler(elevator) currently
and it is a simple fifo queue
priv = !test_bit(QUEUE_FLAG_ELVSWITCH, &q->queue_flags);
?if (priv)
??rl->elvpriv++;
rq = blk_alloc_request
(q, rw_flags, priv, gfp_mask);
rq == NULL
memory alloc
failed
rq != NULL
spin_lock_irq(q->queue_lock);
freed_request(q, is_sync, priv);
If we unable to allocate a request
(due to lack of memory) despite of
the fact that no requests in the same direction
are allocated in a queue(request_list)
mark the request list of a queue as starved
in appropriate direction
if (unlikely(rl->count[is_sync] == 0))
rl->starved[is_sync] = 1;
true,in any case
even if the
context is batching
do not allow
more then
2/3*q->nr_requests
to be allocated
return NULL;
We have successfuly allocated a new request
ioc_batching(q, ioc)
true
false
tracing
trace_block_getrq(q, bio, rw_flags & 1);
return rq;
we have allocated a request
in batching context decrease
the counter
ioc->nr_batch_requests--;

FIGURE 55 Get request flow

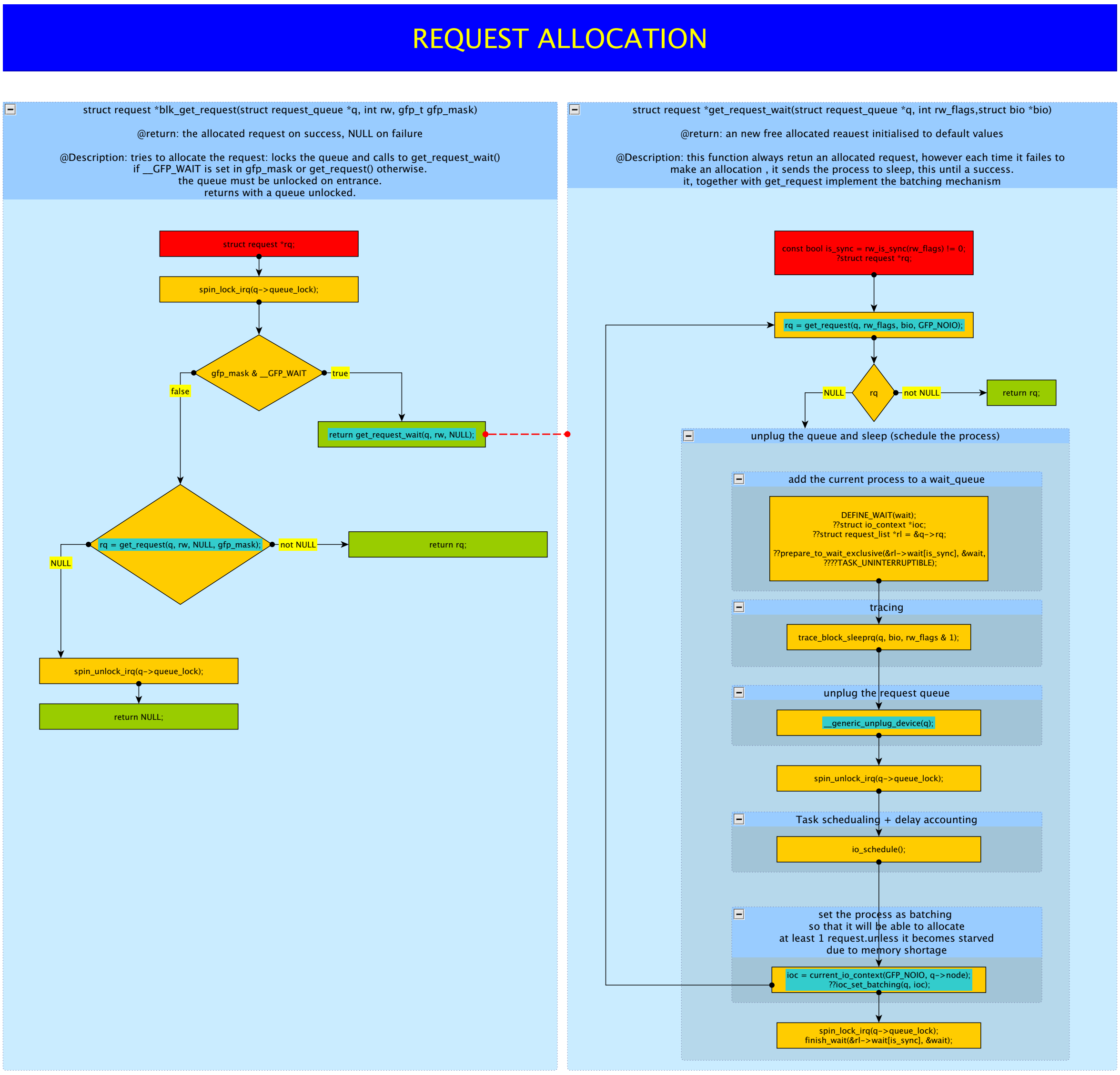
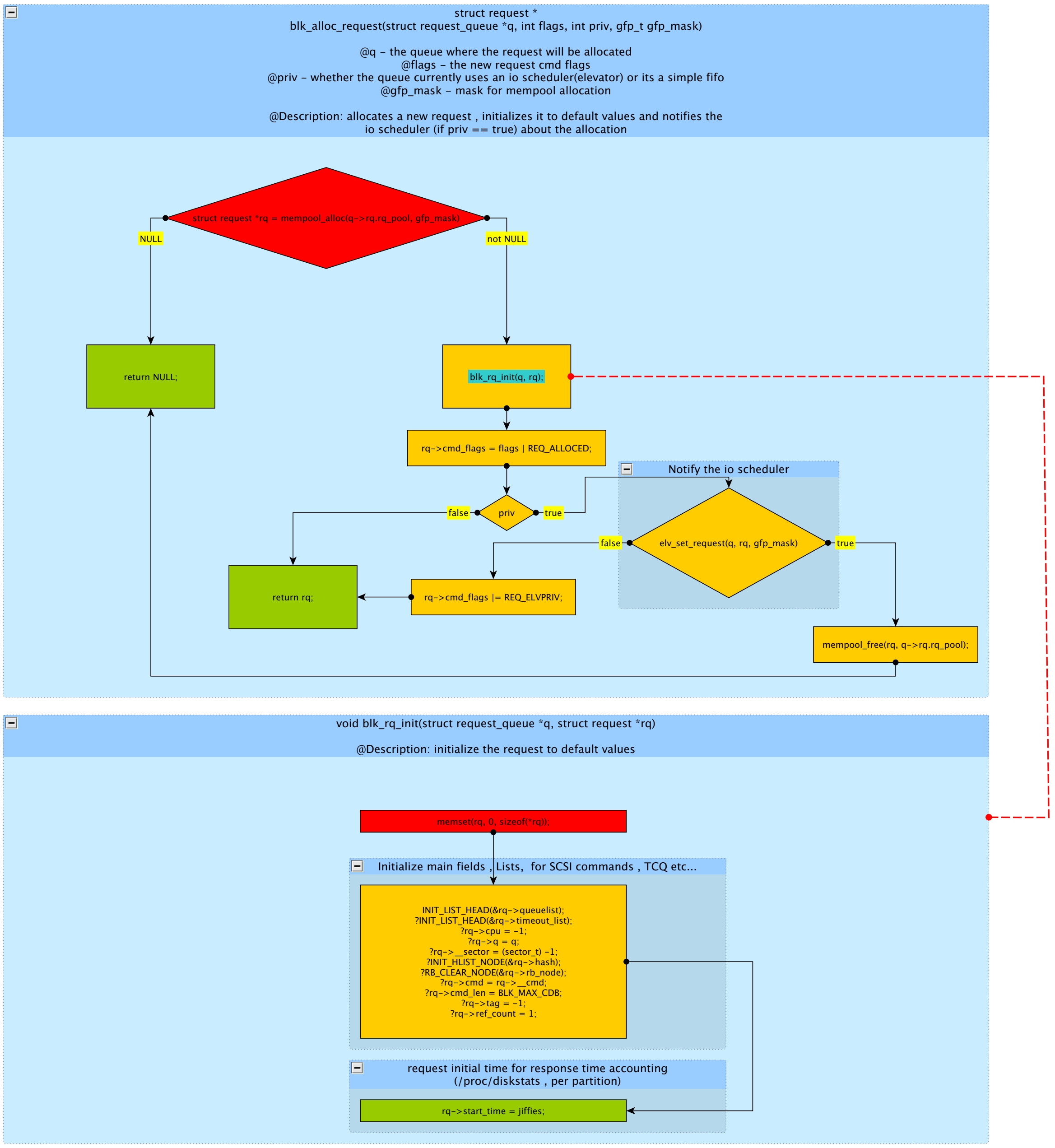

REQUEST ALLOCATION
struct request *blk_get_request(struct request_queue *q, int rw, gfp_t gfp_mask)
@return: the allocated request on success, NULL on failure
@Description: tries to allocate the request: locks the queue and calls to get_request_wait()
if __GFP_WAIT is set in gfp_mask or get_request() otherwise.
the queue must be unlocked on entrance.
returns with a queue unlocked.
struct request *rq;
spin_lock_irq(q->queue_lock);
gfp_mask & __GFP_WAIT
true
false
return get_request_wait(q, rw, NULL);
rq = get_request(q, rw, NULL, gfp_mask);
not NULL
return rq;
NULL
spin_unlock_irq(q->queue_lock);
return NULL;
struct request *get_request_wait(struct request_queue *q, int rw_flags,struct bio *bio)
@return: an new free allocated reauest initialised to default values
@Description: this function always retun an allocated request, however each time it failes to
make an allocation , it sends the process to sleep, this until a success.
it, together with get_request implement the batching mechanism
const bool is_sync = rw_is_sync(rw_flags) != 0;
?struct request *rq;
rq = get_request(q, rw_flags, bio, GFP_NOIO);
NULL
rq
not NULL
return rq;
unplug the queue and sleep (schedule the process)
add the current process to a wait_queue
DEFINE_WAIT(wait);
??struct io_context *ioc;
??struct request_list *rl = &q->rq;
??prepare_to_wait_exclusive(&rl->wait[is_sync], &wait,
????TASK_UNINTERRUPTIBLE);
tracing
trace_block_sleeprq(q, bio, rw_flags & 1);
unplug the request queue
__generic_unplug_device(q);
spin_unlock_irq(q->queue_lock);
Task schedualing + delay accounting
io_schedule();
set the process as batching
so that it will be able to allocate
at least 1 request.unless it becomes starved
due to memory shortage
ioc = current_io_context(GFP_NOIO, q->node);
??ioc_set_batching(q, ioc);
spin_lock_irq(q->queue_lock);
finish_wait(&rl->wait[is_sync], &wait);

FIGURE 56 Get request wait flow



struct request *
blk_alloc_request(struct request_queue *q, int flags, int priv, gfp_t gfp_mask)

@q – the queue where the request will be allocated
@flags – the new request cmd flags
@priv – whether the queue currently uses an io scheduler(elevator) or its a simple fifo
@gfp_mask – mask for mempool allocation

@Description: allocates a new request , initializes it to default values and notifies the
io scheduler (if priv == true) about the allocation

struct request *rq = mempool_alloc(q–>rq.rq_pool, gfp_mask)
NULL
not NULL
return NULL;
blk_rq_init(q, rq);
rq–>cmd_flags = flags | REQ_ALLOCED;
Notify the io scheduler
false
priv
true
false
elv_set_request(q, rq, gfp_mask)
true
return rq;
rq–>cmd_flags |= REQ_ELVPRIV;
mempool_free(rq, q–>rq.rq_pool);

void blk_rq_init(struct request_queue *q, struct request *rq)

@Description: initialize the request to default values

memset(rq, 0, sizeof(*rq));
Initialize main fields , Lists, for SCSI commands , TCQ etc...
INIT_LIST_HEAD(&rq–>queuelist);
?INIT_LIST_HEAD(&rq–>timeout_list);
?rq–>cpu = –1;
?rq–>q = q;
?rq–>__sector = (sector_t) –1;
?INIT_HLIST_NODE(&rq–>hash);
?RB_CLEAR_NODE(&rq–>rb_node);
?rq–>cmd = rq–>__cmd;
?rq–>cmd_len = BLK_MAX_CDB;
?rq–>tag = –1;
?rq–>ref_count = 1;
request initial time for response time accounting
(/proc/diskstats , per partition)
rq–>start_time = jiffies;

FIGURE 57 Allocate request flow

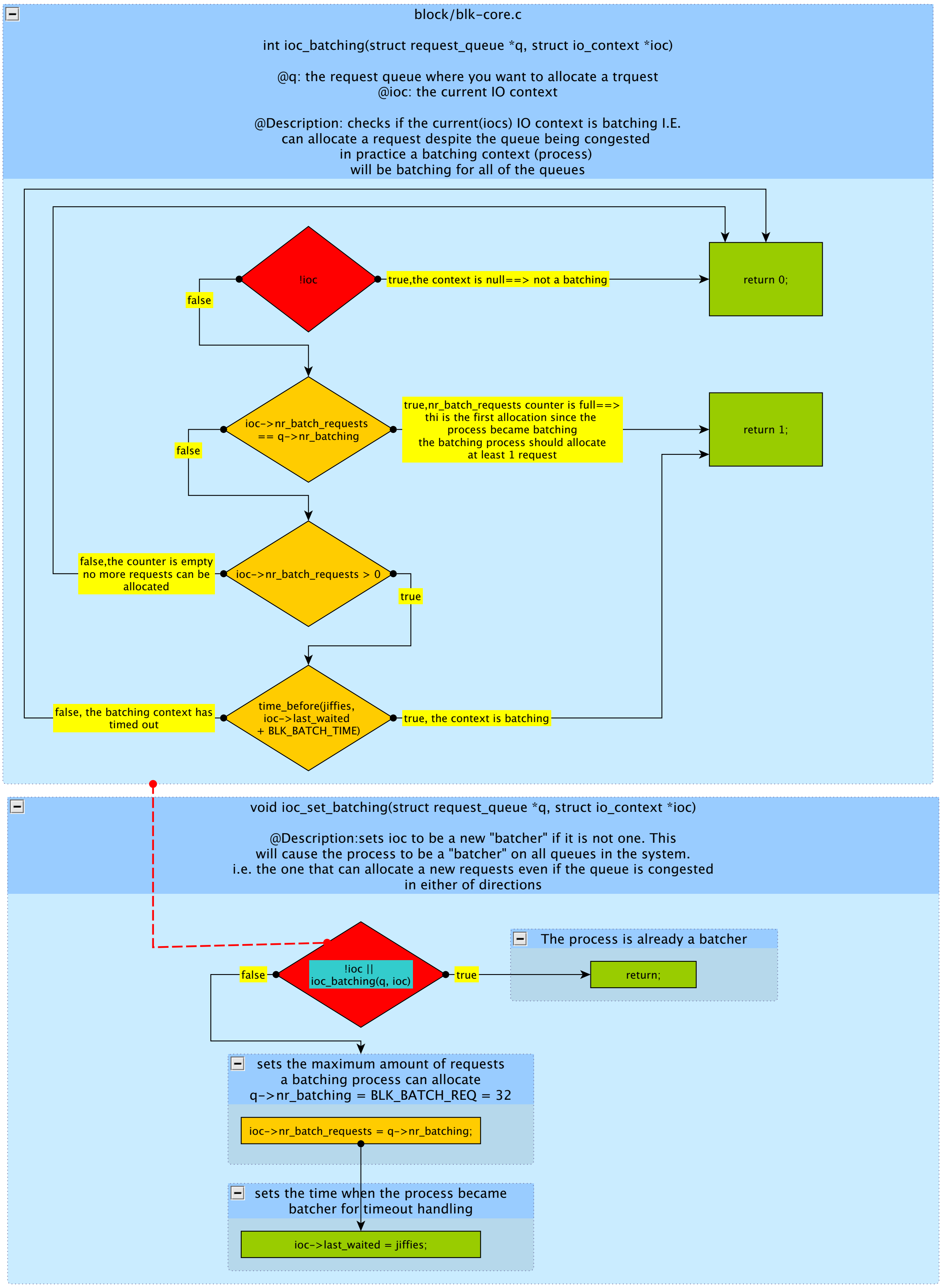

block/blk-core.c
int ioc_batching(struct request_queue *q, struct io_context *ioc)
@q: the request queue where you want to allocate a trquest
@ioc: the current IO context
@Description: checks if the current(iocs) IO context is batching I.E.
can allocate a request despite the queue being congested
in practice a batching context (process)
will be batching for all of the queues
!ioc
true,the context is null==> not a batching
return 0;
false
ioc->nr_batch_requests
== q->nr_batching
true,nr_batch_requests counter is full==>
thi is the first allocation since the
process became batching
the batching process should allocate
at least 1 request
return 1;
false
false,the counter is empty
no more requests can be
allocated
ioc->nr_batch_requests > 0
true
false, the batching context has
timed out
time_before(jiffies,
ioc->last_waited
+ BLK_BATCH_TIME)
true, the context is batching
void ioc_set_batching(struct request_queue *q, struct io_context *ioc)
@Description:sets ioc to be a new "batcher" if it is not one. This
will cause the process to be a "batcher" on all queues in the system.
i.e. the one that can allocate a new requests even if the queue is congested
in either of directions
The process is already a batcher
false
!ioc ||
ioc_batching(q, ioc)
true
return;
sets the maximum amount of requests
a batching process can allocate
q->nr_batching = BLK_BATCH_REQ = 32
ioc->nr_batch_requests = q->nr_batching;
sets the time when the process became
batcher for timeout handling
ioc->last_waited = jiffies;

FIGURE 58 Batching I/O flow

In addition the method unplugs the queue, if the number of the requests in the request queue exceeds the unplug threshold. elv_add_request() calls elv_insert() which does the actual job. elv_add_requests() is called by elv_requeue_request() which makes a little more logic and essentially masks the ELEVATOR_INSERT_REQUEUE flag. In the block layer it is called by __make_request and by similar functions like blk_insert_request or blk_insert_cloned_request etc. These function do essentially the same thing but with less functionality (like bypassing the scheduler, inserting an already existing / prepared request etc.).

APPENDIX 5.11 Plug/Unplug Mechanism

The idea behind the device plugging mechanism is simple: due to natural characteristics of rotational block devices, it is much more efficient to issue the requests in bursts i.e. when the queue length > 1. Bursts are efficient often due to the length of time required for the seek operation of the device. Sometimes by waiting for more requests, we can accumulate bursts and seek more efficiently.

Thus we want to prevent the requests from being served immediately after it was submitted (if the queue length is small) and wait until (possibly) more requests are added to the queue.

To do the this block layer simply raises a flag to notify the driver that it should stop processing requests from the queue - this is referred to as device plugging. The plugging occurs at certain conditions (see figures 61, 62) during bio submission in "__make_request()". "elv_add_request()" . It will plug the device also if the parameter is not set to 0.

However we do not want our bios to be starved and the unplugging mechanism is introduced:

1. When the device is plugged it starts a timer that when and timeout, pushes an unplug method (usually generic_unplug_device()) , into work_queue which schedules it to run.
2. In some cases the device is unplugged by direct calling to the unplug function (when the queue is exhausted for example) The generic_unplug_device() method , when the call deletes the timer , clears the flag , and calls to request_fn() to process the requests in a queue by a block device driver one by one.

Some times, I/O schedulers need to force the request_fn() to start producing the requests. Thus the run_queue() method is introduced, it removes plug, and either calls directly to request_fn() or if it was already called in the recursion stack, schedules it within the same work queue.

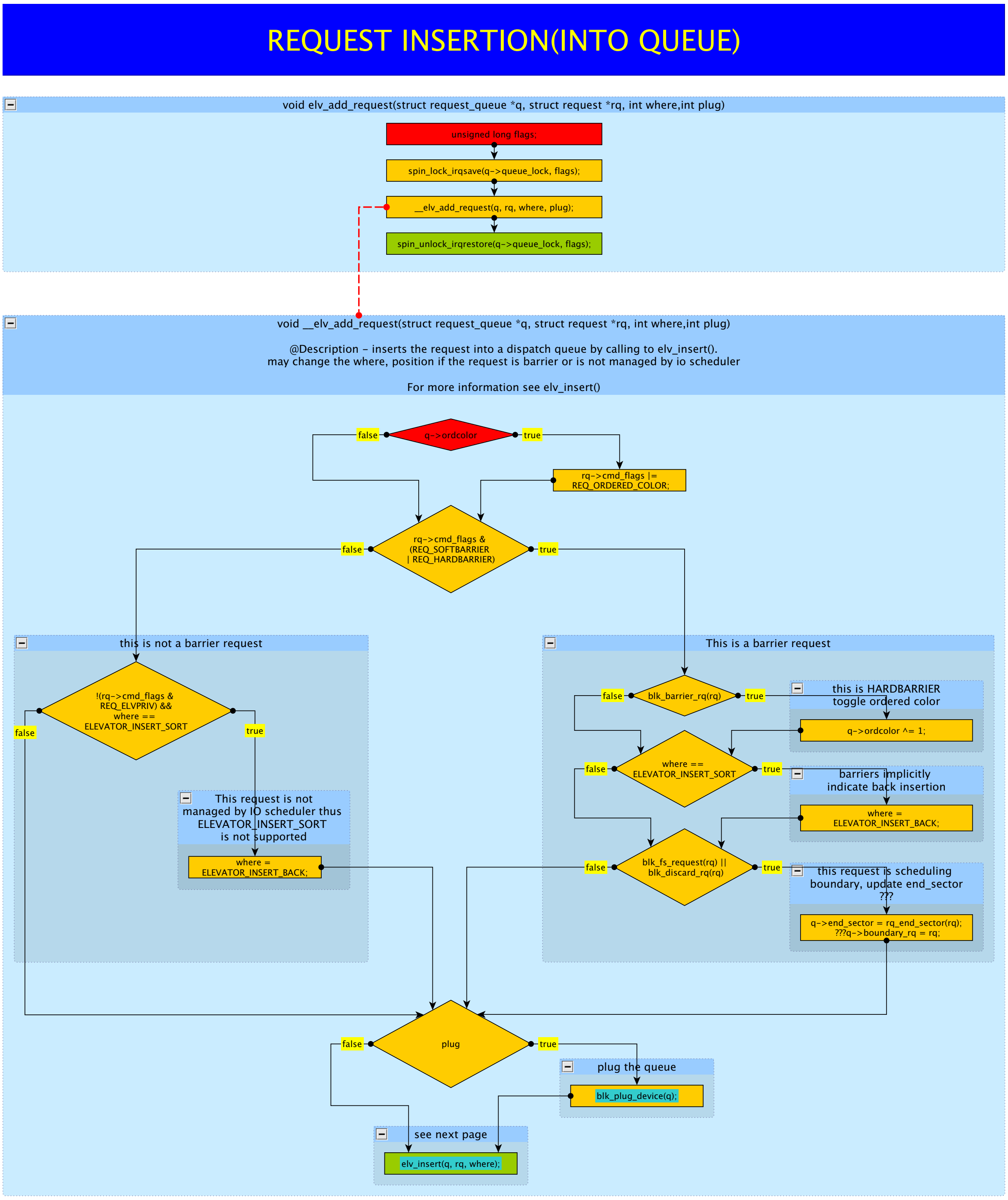

REQUEST INSERTION(INTO QUEUE)
void elv_add_request(struct request_queue *q, struct request *rq, int where,int plug)
unsigned long flags;
spin_lock_irqsave(q->queue_lock, flags);
__elv_add_request(q, rq, where, plug);
spin_unlock_irqrestore(q->queue_lock, flags);
void __elv_add_request(struct request_queue *q, struct request *rq, int where,int plug)
@Description - inserts the request into a dispatch queue by calling to elv_insert().
may change the where, position if the request is barrier or is not managed by io scheduler
For more information see elv_insert()
false
q->ordcolor
true
rq->cmd_flags |= REQ_ORDERED_COLOR;
false
rq->cmd_flags & (REQ_SOFTBARRIER | REQ_HARDBARRIER)
true
this is not a barrier request
!(rq->cmd_flags & REQ_ELVPRIV) && where == ELEVATOR_INSERT_SORT
false
true
This request is not managed by IO scheduler thus ELEVATOR_INSERT_SORT is not supported
where = ELEVATOR_INSERT_BACK;
This is a barrier request
false
blk_barrier_rq(rq)
true
this is HARDBARRIER toggle ordered color
q->ordcolor ^= 1;
false
where == ELEVATOR_INSERT_SORT
true
barriers implicitly indicate back insertion
where = ELEVATOR_INSERT_BACK;
false
blk_fs_request(rq) || blk_discard_rq(rq)
true
this request is scheduling boundary, update end_sector ???
q->end_sector = rq_end_sector(rq); ???q->boundary_rq = rq;
false
plug
true
plug the queue
blk_plug_device(q);
see next page
elv_insert(q, rq, where);

FIGURE 59 Add request flow

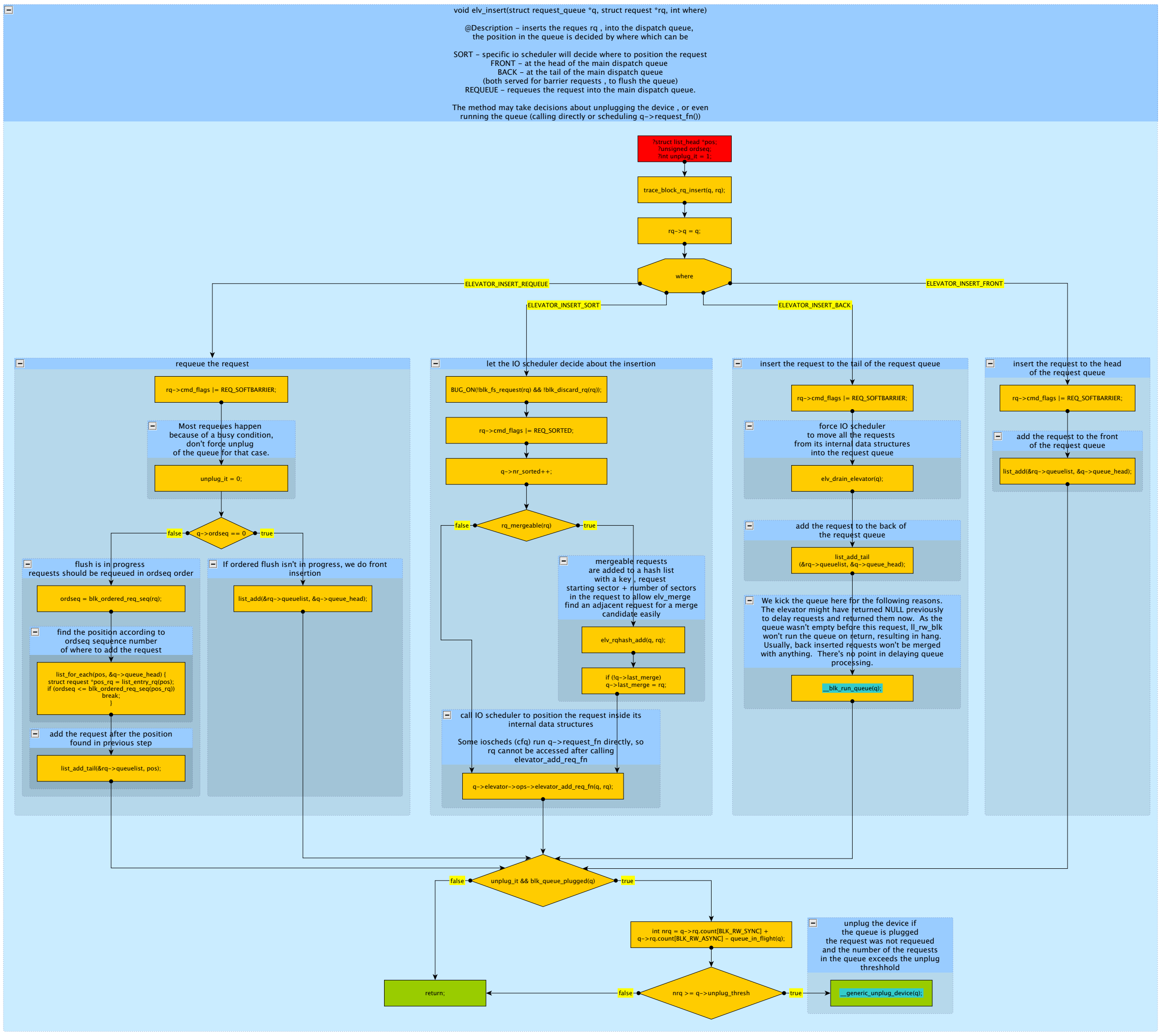

void elv_insert(struct request_queue *q, struct request *rq, int where)
@Description – inserts the reques rq , into the dispatch queue,
the position in the queue is decided by where which can be
SORT – specific io scheduler will decide where to position the request
FRONT – at the head of the main dispatch queue
BACK – at the tail of the main dispatch queue
(both served for barrier requests , to flush the queue)
REQUEUE – requeues the request into the main dispatch queue.
The method may take decisions about unplugging the device , or even
running the queue (calling directly or scheduling q->request_fn())
?struct list_head *pos;
?unsigned ordseq;
?int unplug_it = 1;
trace_block_rq_insert(q, rq);
rq->q = q;
where
ELEVATOR_INSERT_REQUEUE
ELEVATOR_INSERT_SORT
ELEVATOR_INSERT_BACK
ELEVATOR_INSERT_FRONT
requeue the request
rq->cmd_flags |= REQ_SOFTBARRIER;
Most requeues happen
because of a busy condition,
don't force unplug
of the queue for that case.
unplug_it = 0;
false
q->ordseq == 0
true
flush is in progress
requests should be requeued in ordseq order
ordseq = blk_ordered_req_seq(rq);
find the position according to
ordseq sequence number
of where to add the request
list_for_each(pos, &q->queue_head) {
struct request *pos_rq = list_entry_rq(pos);
if (ordseq <= blk_ordered_req_seq(pos_rq))
break;
}
add the request after the position
found in previous step
list_add_tail(&rq->queuelist, pos);
If ordered flush isn't in progress, we do front
insertion
list_add(&rq->queuelist, &q->queue_head);
let the IO scheduler decide about the insertion
BUG_ON(!blk_fs_request(rq) && !blk_discard_rq(rq));
rq->cmd_flags |= REQ_SORTED;
q->nr_sorted++;
false
rq_mergeable(rq)
true
mergeable requests
are added to a hash list
with a key , request
starting sector + number of sectors
in the request to allow elv_merge
find an adjacent request for a merge
candidate easily
elv_rqhash_add(q, rq);
if (!q->last_merge)
q->last_merge = rq;
call IO scheduler to position the request inside its
internal data structures
Some ioscheds (cfq) run q->request_fn directly, so
rq cannot be accessed after calling
elevator_add_req_fn
q->elevator->ops->elevator_add_req_fn(q, rq);
insert the request to the tail of the request queue
rq->cmd_flags |= REQ_SOFTBARRIER;
force IO scheduler
to move all the requests
from its internal data structures
into the request queue
elv_drain_elevator(q);
add the request to the back of
the request queue
list_add_tail
(&rq->queuelist, &q->queue_head);
We kick the queue here for the following reasons.
The elevator might have returned NULL previously
to delay requests and returned them now. As the
queue wasn't empty before this request, ll_rw_blk
won't run the queue on return, resulting in hang.
Usually, back inserted requests won't be merged
with anything. There's no point in delaying queue
processing.
__blk_run_queue(q);
insert the request to the head
of the request queue
rq->cmd_flags |= REQ_SOFTBARRIER;
add the request to the front
of the request queue
list_add(&rq->queuelist, &q->queue_head);
false
unplug_it && blk_queue_plugged(q)
true
int nrq = q->rq.count[BLK_RW_SYNC] +
q->rq.count[BLK_RW_ASYNC] – queue_in_flight(q);
unplug the device if
the queue is plugged
the request was not requeued
and the number of the requests
in the queue exceeds the unplug
threshhold
return;
false
nrq >= q->unplug_thresh
true
__generic_unplug_device(q);

FIGURE 60 Elevator insert flow

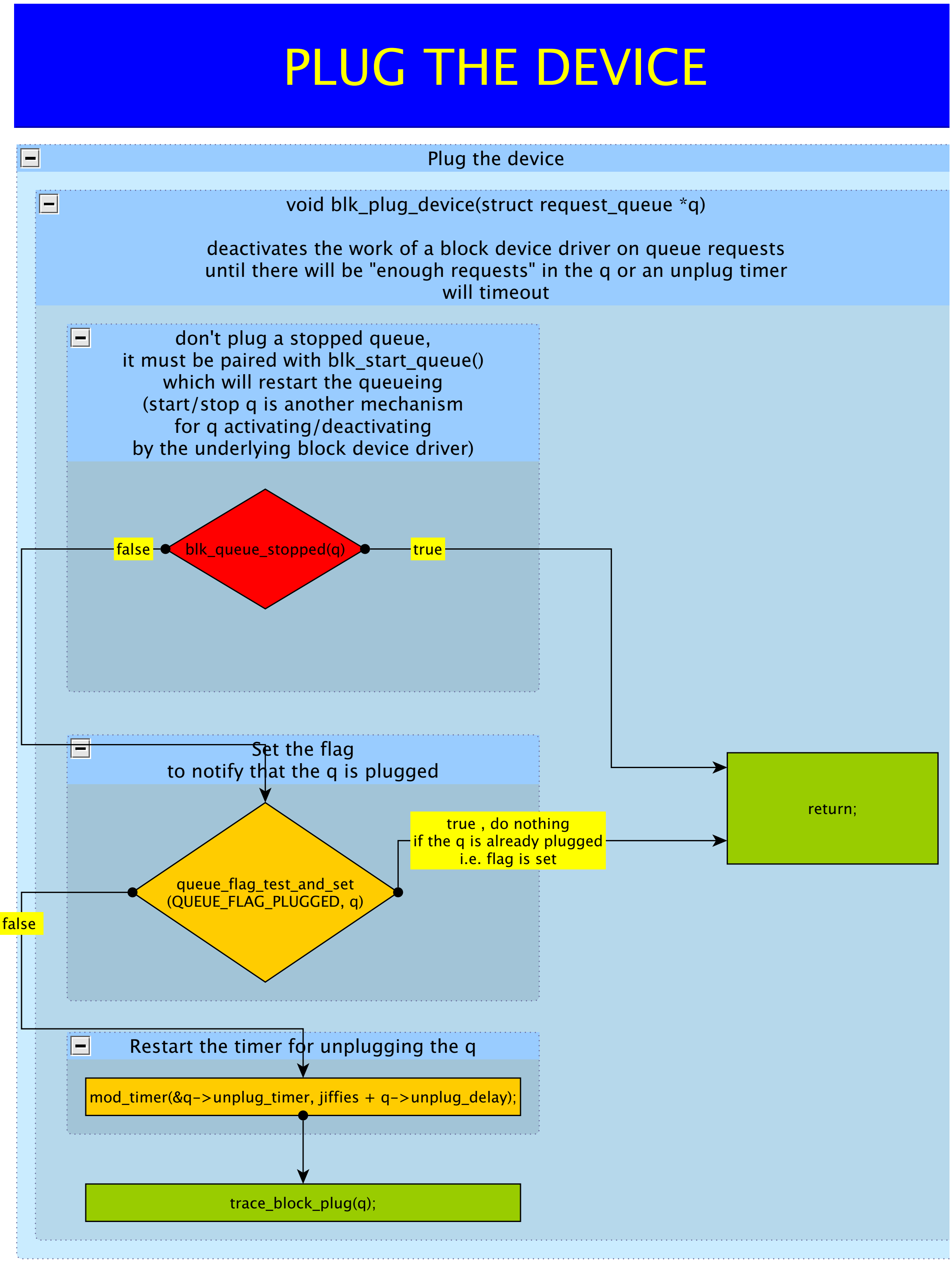
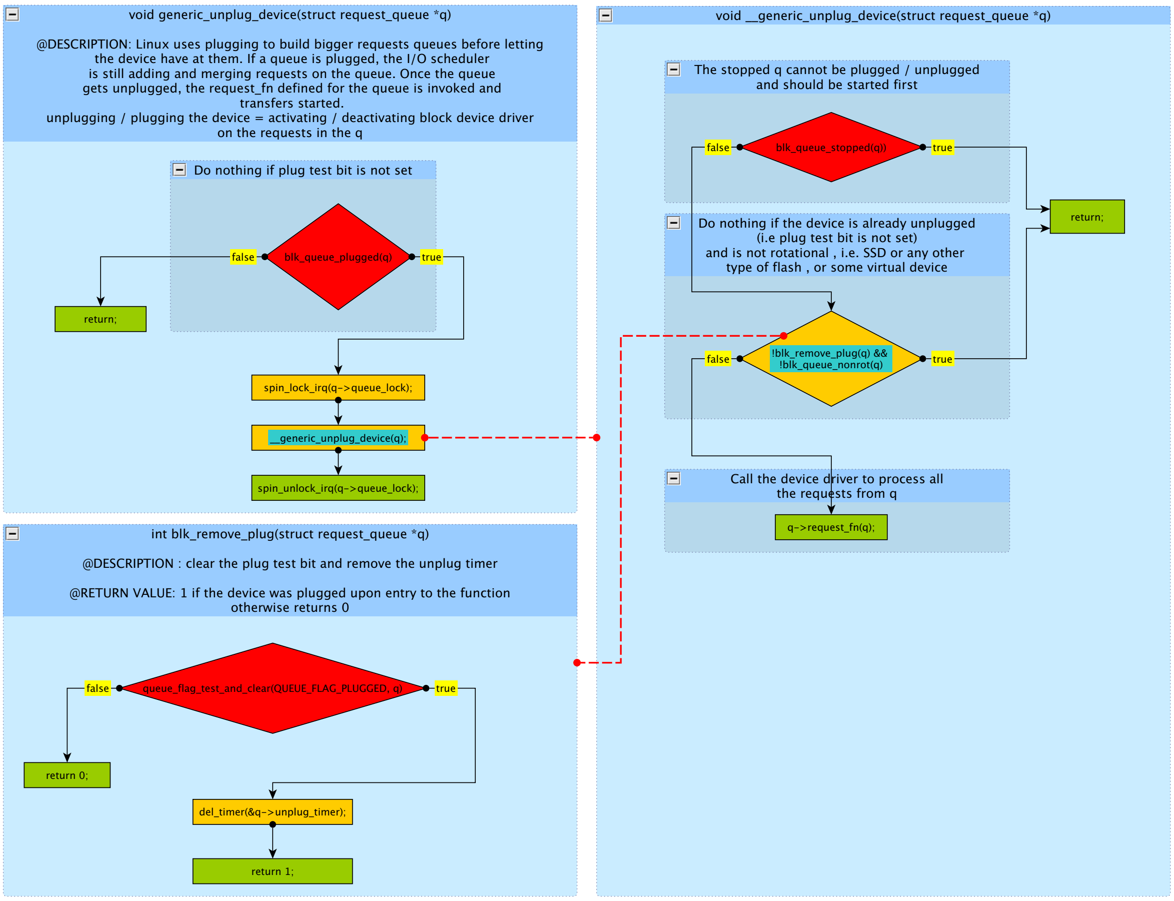

PLUG THE DEVICE
Plug the device
void blk_plug_device(struct request_queue *q)
deactivates the work of a block device driver on queue requests
until there will be "enough requests" in the q or an unplug timer
will timeout
don't plug a stopped queue,
it must be paired with blk_start_queue()
which will restart the queueing
(start/stop q is another mechanism
for q activating/deactivating
by the underlying block device driver)
false
blk_queue_stopped(q)
true
Set the flag
to notify that the q is plugged
return;
true , do nothing
if the q is already plugged
i.e. flag is set
queue_flag_test_and_set
(QUEUE_FLAG_PLUGGED, q)
false
Restart the timer for unplugging the q
mod_timer(&q->unplug_timer, jiffies + q->unplug_delay);
trace_block_plug(q);

FIGURE 61 Plug Device flow

UNPLUG THE DEVICE



void generic_unplug_device(struct request_queue *q)

@DESCRIPTION: Linux uses plugging to build bigger requests queues before letting the device have at them. If a queue is plugged, the I/O scheduler is still adding and merging requests on the queue. Once the queue gets unplugged, the request_fn defined for the queue is invoked and transfers started.
unplugging / plugging the device = activating / deactivating block device driver on the requests in the q

Do nothing if plug test bit is not set

int blk_remove_plug(struct request_queue *q)

@DESCRIPTION : clear the plug test bit and remove the unplug timer

@RETURN VALUE: 1 if the device was plugged upon entry to the function otherwise returns 0

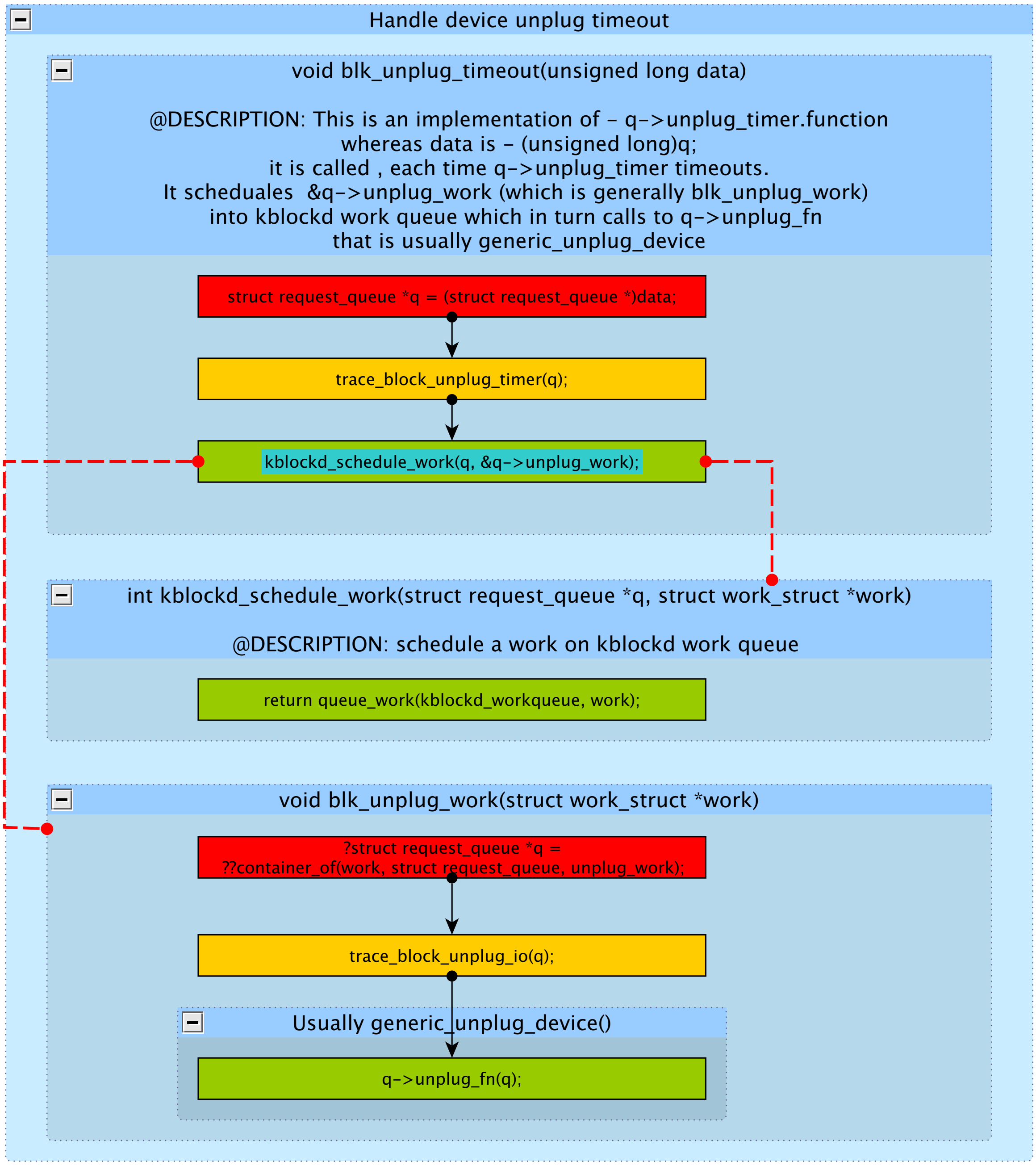
void __generic_unplug_device(struct request_queue *q)

The stopped q cannot be plugged / unplugged and should be started first

Do nothing if the device is already unplugged (i.e plug test bit is not set) and is not rotational , i.e. SSD or any other type of flash , or some virtual device

Call the device driver to process all the requests from q

FIGURE 62   Unplug Device flow



Handle device unplug timeout

void blk_unplug_timeout(unsigned long data)

@DESCRIPTION: This is an implementation of – q->unplug_timer.function
whereas data is – (unsigned long)q;
it is called , each time q->unplug_timer timeouts.
It scheduales &q->unplug_work (which is generally blk_unplug_work)
into kblockd work queue which in turn calls to q->unplug_fn
that is usually generic_unplug_device

```
struct request_queue *q = (struct request_queue *)data;
trace_block_unplug_timer(q);
kblockd_schedule_work(q, &q->unplug_work);
```

int kblockd_schedule_work(struct request_queue *q, struct work_struct *work)

@DESCRIPTION: schedule a work on kblockd work queue

```
return queue_work(kblockd_workqueue, work);
```

void blk_unplug_work(struct work_struct *work)

```
?struct request_queue *q =
??container_of(work, struct request_queue, unplug_work);
trace_block_unplug_io(q);
```

Usually generic_unplug_device()

```
q->unplug_fn(q);
```

FIGURE 63 Unplug Device - Timeout flow

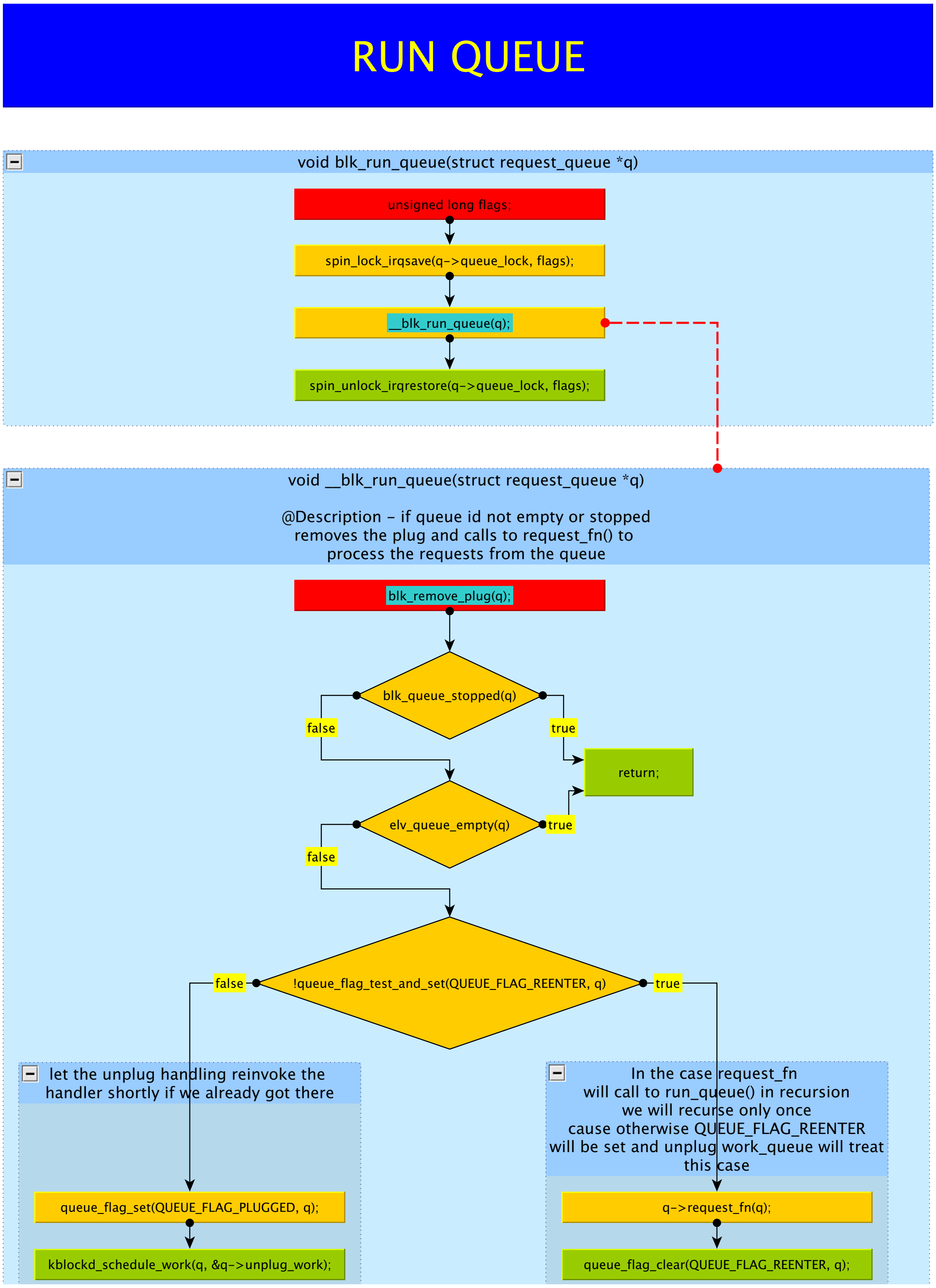

RUN QUEUE
void blk_run_queue(struct request_queue *q)
unsigned long flags;
spin_lock_irqsave(q->queue_lock, flags);
__blk_run_queue(q);
spin_unlock_irqrestore(q->queue_lock, flags);
void __blk_run_queue(struct request_queue *q)
@Description - if queue id not empty or stopped removes the plug and calls to request_fn() to process the requests from the queue
blk_remove_plug(q);
blk_queue_stopped(q)
false
true
return;
elv_queue_empty(q)
true
false
!queue_flag_test_and_set(QUEUE_FLAG_REENTER, q)
false
true
let the unplug handling reinvoke the handler shortly if we already got there
queue_flag_set(QUEUE_FLAG_PLUGGED, q);
kblockd_schedule_work(q, &q->unplug_work);
In the case request_fn will call to run_queue() in recursion we will recurse only once cause otherwise QUEUE_FLAG_REENTER will be set and unplug work_queue will treat this case
q->request_fn(q);
queue_flag_clear(QUEUE_FLAG_REENTER, q);

FIGURE 64 Run queue flow

APPENDIX 5.12 Fetch Request

Fetch Request is the lowest level in block layer. It provides the main service for the block device driver; given a queue fetch a request from it. Here are we appeal to the most important mechanisms of the block layer which force the requests in the queue to be issued in some specific order or postponed:

1. The I/O scheduler dispatch_fn() method moves requests from its data structures into a main dispatch queue, or does nothing if it wants to postpone the requests it has.
2. Barrier mechanisms which by encountering special barrier requests force cache-queue flushing/draining, keep the requests in order and prevent the requests that arrived post barrier to be issued until all the requests that arrive before the barrier are completed.

The main drivers function to obtain requests from the queue is blk_fetch_request() it calls to blk_peek_request() which peeks some request from the queue or returns NULL, if the queue is empty or one of the mechanisms (1,2) has decided to postpone the requests. If it is not NULL blk_start_request() is called to physically remove the request from the queue, and start a timeout timer on it for error handling.

blk_peek_request() This function is the core of the mechanism: it calls to __elv_next_request() to peek a request from a dispatch queue (if it is NULL then the function returns NULL) marks it as started, notifies the I/O scheduler about the new started request in flight, and calls to prep_fn() which may be defined by the driver (for example to assign a tag to a request for TCQ).

__elv_next_request() This function tries to peek at the first request in the dispatch queue if it succeeds it calls to blk_do_ordered() that checks whether the request is a barrier and begins a flush/drain (see section 5.13) or if a flush is already started it may return NULL instead of a request if it is out of order (and then try to peek next request once again). If the dispatch queue is empty it calls to scheduler dispatch_fn() to fill the queue with requests (the I/O scheduler may refuse to do it even though it contains requests) and if it does not fill the queue with any request it returns NULL, otherwise it starts the process from the beginning.

APPENDIX 5.13 Barrier

The barrier request is used when the request queue (cache) is flushed. We use the barrier so that no new incoming requests will be served until the flushing

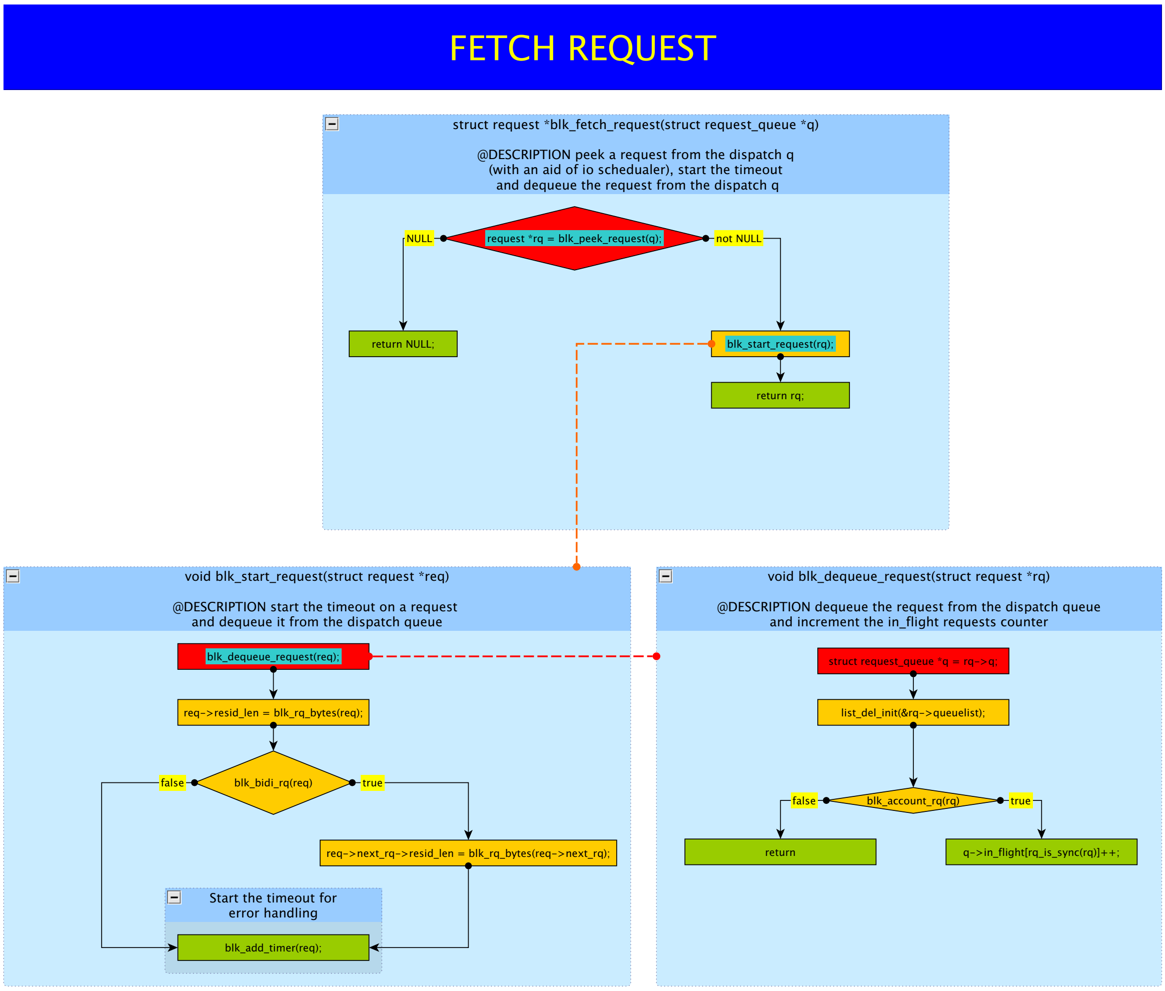

FETCH REQUEST
struct request *blk_fetch_request(struct request_queue *q)
@DESCRIPTION peek a request from the dispatch q
(with an aid of io schedualer), start the timeout
and dequeue the request from the dispatch q
NULL
request *rq = blk_peek_request(q);
not NULL
return NULL;
blk_start_request(rq);
return rq;
void blk_start_request(struct request *req)
@DESCRIPTION start the timeout on a request
and dequeue it from the dispatch queue
blk_dequeue_request(req);
req->resid_len = blk_rq_bytes(req);
false
blk_bidi_rq(req)
true
req->next_rq->resid_len = blk_rq_bytes(req->next_rq);
Start the timeout for
error handling
blk_add_timer(req);
void blk_dequeue_request(struct request *rq)
@DESCRIPTION dequeue the request from the dispatch queue
and increment the in_flight requests counter
struct request_queue *q = rq->q;
list_del_init(&rq->queuelist);
false
blk_account_rq(rq)
true
return
q->in_flight[rq_is_sync(rq)]++;

FIGURE 65 Fetch request flow

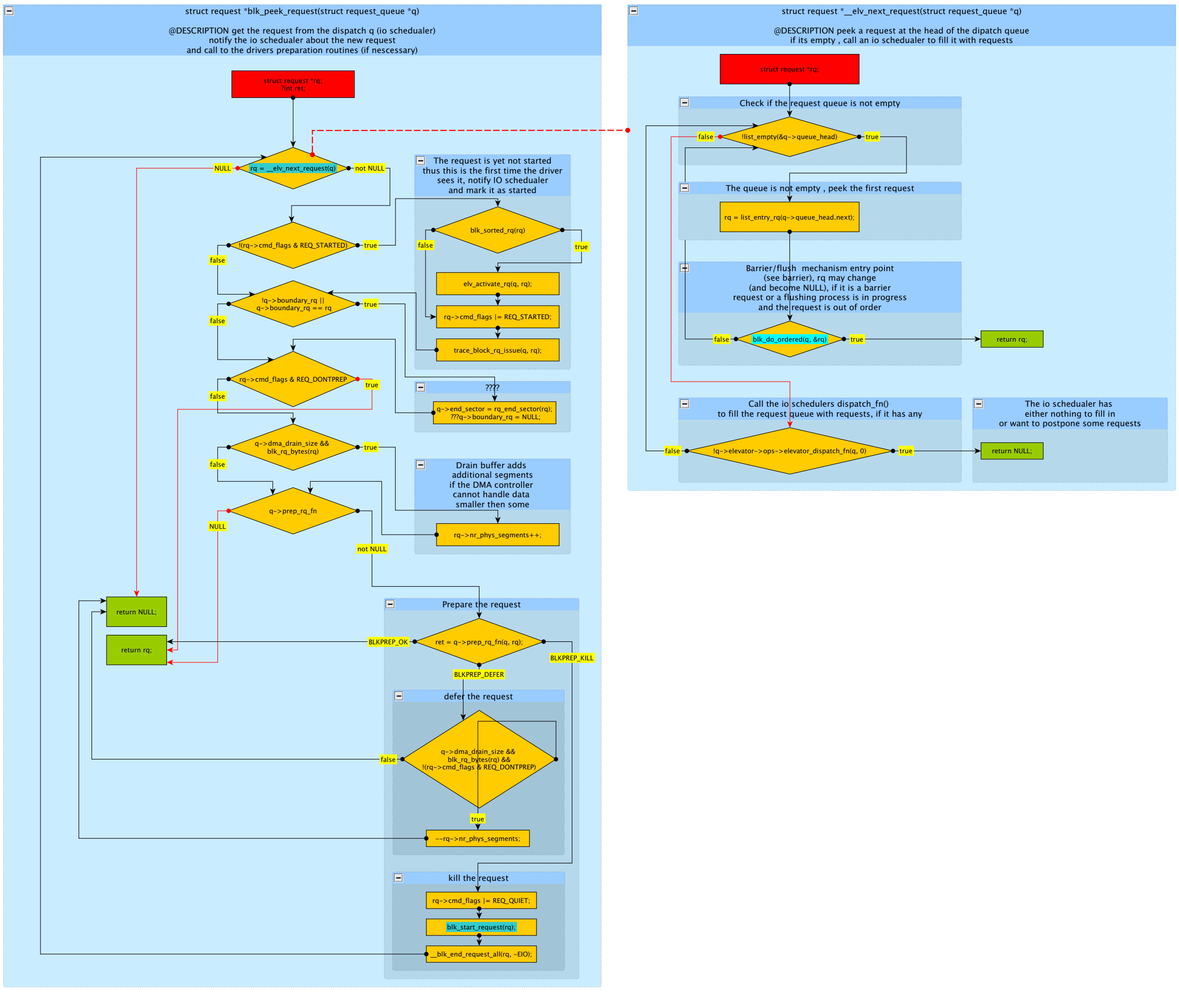

struct request *blk_peek_request(struct request_queue *q)
@DESCRIPTION get the request from the dispatch q (io schedualer)
notify the io schedualer about the new request
and call to the drivers preparation routines (if nescessary)
struct request *rq;
?int ret;
rq = __elv_next_request(q)
NULL
not NULL
The request is yet not started
thus this is the first time the driver
sees it, notify IO schedualer
and mark it as started
blk_sorted_rq(rq)
false
true
elv_activate_rq(q, rq);
rq->cmd_flags |= REQ_STARTED;
trace_block_rq_issue(q, rq);
!(rq->cmd_flags & REQ_STARTED)
true
false
!q->boundary_rq ||
q->boundary_rq == rq
true
false
rq->cmd_flags & REQ_DONTPREP
true
false
????
q->end_sector = rq_end_sector(rq);
???q->boundary_rq = NULL;
q->dma_drain_size &&
blk_rq_bytes(rq)
true
false
Drain buffer adds
additional segments
if the DMA controller
cannot handle data
smaller then some
rq->nr_phys_segments++;
q->prep_rq_fn
NULL
not NULL
return NULL;
return rq;
Prepare the request
ret = q->prep_rq_fn(q, rq);
BLKPREP_OK
BLKPREP_KILL
BLKPREP_DEFER
defer the request
q->dma_drain_size &&
blk_rq_bytes(rq) &&
!(rq->cmd_flags & REQ_DONTPREP)
false
true
--rq->nr_phys_segments;
kill the request
rq->cmd_flags |= REQ_QUIET;
blk_start_request(rq);
__blk_end_request_all(rq, -EIO);
struct request *__elv_next_request(struct request_queue *q)
@DESCRIPTION peek a request at the head of the dipatch queue
if its empty , call an io schedualer to fill it with requests
struct request *rq;
Check if the request queue is not empty
!list_empty(&q->queue_head)
false
true
The queue is not empty , peek the first request
rq = list_entry_rq(q->queue_head.next);
Barrier/flush mechanism entry point
(see barrier), rq may change
(and become NULL), if it is a barrier
request or a flushing process is in progress
and the request is out of order
blk_do_ordered(q, &rq)
false
true
return rq;
Call the io schedulers dispatch_fn()
to fill the request queue with requests, if it has any
!q->elevator->ops->elevator_dispatch_fn(q, 0)
false
true
The io schedualer has
either nothing to fill in
or want to postpone some requests
return NULL;

FIGURE 66 Peek Request flow

operation is completed. Therefore, a barrier request is the one that indicates to the block layer that all the requests before it should be processed before all the proceeding requests. Moreover no new request should be processed until the old requests have been flushed.

To implement the barrier, the block layer introduces the barrier requests which are created by setting a special flag in bio before the submission. The 2 main entry points of the barrier mechanism occur when the bio is submitted (see __make_request() _elv_add_request in section 5.10) and when it is fetched by the driver from out of the queue (see section 5.12). During the submission of a bio the requests fields are set as barrier request (flags). In the add_request the request is pushed into the front of the dispatch queue.

During the time we fetch request and before the returning of a fetched request it is conveyed to the blk_do_ordered() method which checks whether the flush process has not already started. If the process has not started it checks whether the request is a barrier, in which case it starts the flushing by calling to start_ordered(). If the flushing is in progress, it checks whether the request can be processed (in order of flushing ,(for ex. arrived before the barrier) if it is not it postpones it by returning NULL.

For devices/drivers that support tagged queuing the responsibility to produce the requests in order is conveyed to the driver, by assigning appropriate tags.

APPENDIX 5.14 Put Request

Put request complements get request. Its purpose is to free the request data structures. The main functions are:

1. Memory deallocation (free) of memory used by request data structure.
2. I/O scheduler notification (to release private request data)
3. I/O scheduler notification about requests completion.
4. Managing queue congestion / full statuses (together with get request)
5. Completing the I/O batching mechanism (presented in get request).

Request deallocation is the last step in the life of an I/O in the system. put_request uses the same request_list data structure as get_request. When the request is asked to be deallocated the Linux kernel first assures that there are no more references to that request. If there are references the deallocation is stopped. Then the I/O scheduler is notified about the completion of request.

blk_free_request() asks the I/O scheduler to free the memory it allocated for its private fields and frees the request data structure returning the memory to the memory pool. After resource deallocation is completed, the queue state may be changed. The request lists count field (in the original requests direction) is decremented and then checks if the queue is still congested or

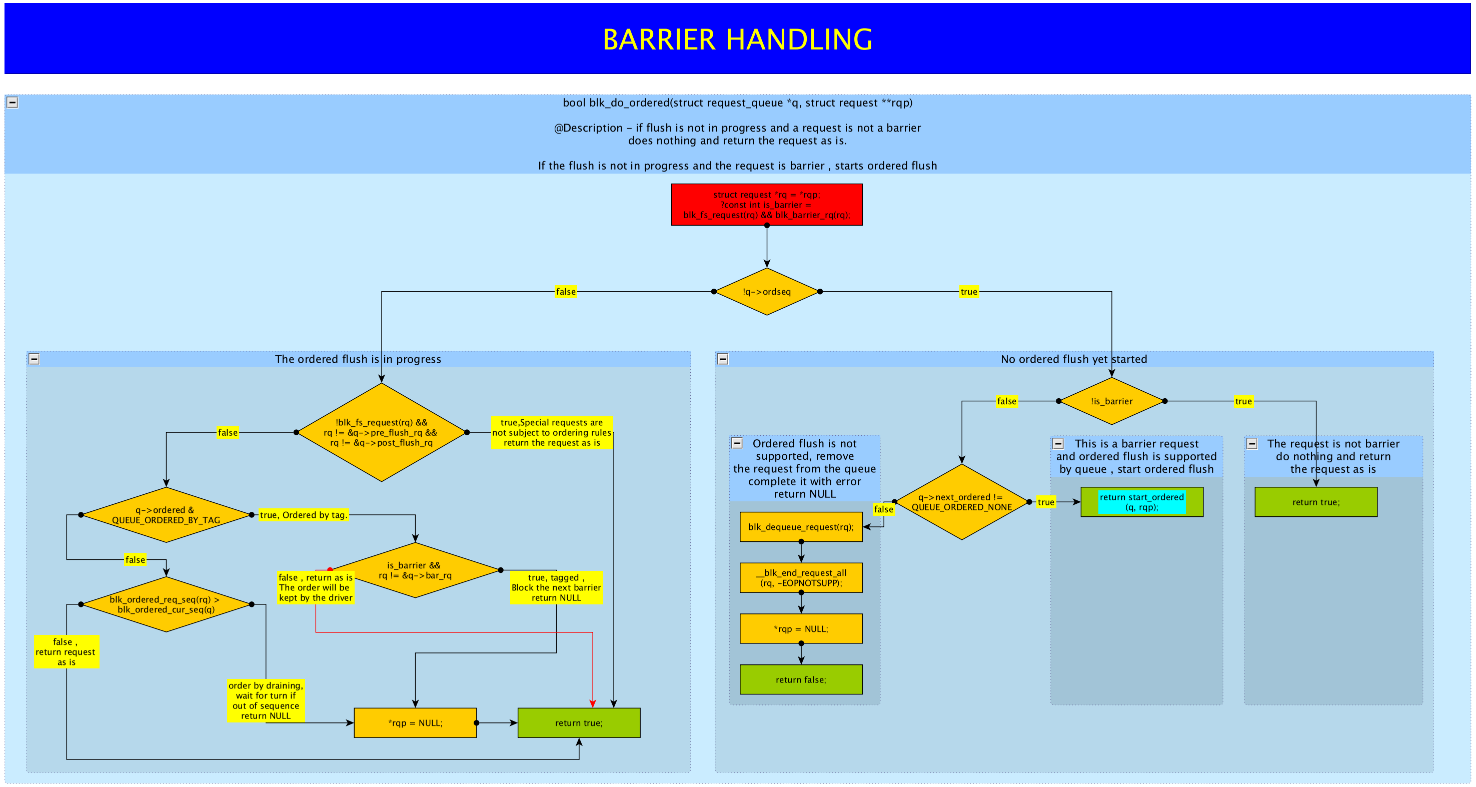


FIGURE 67 Do ordered flow

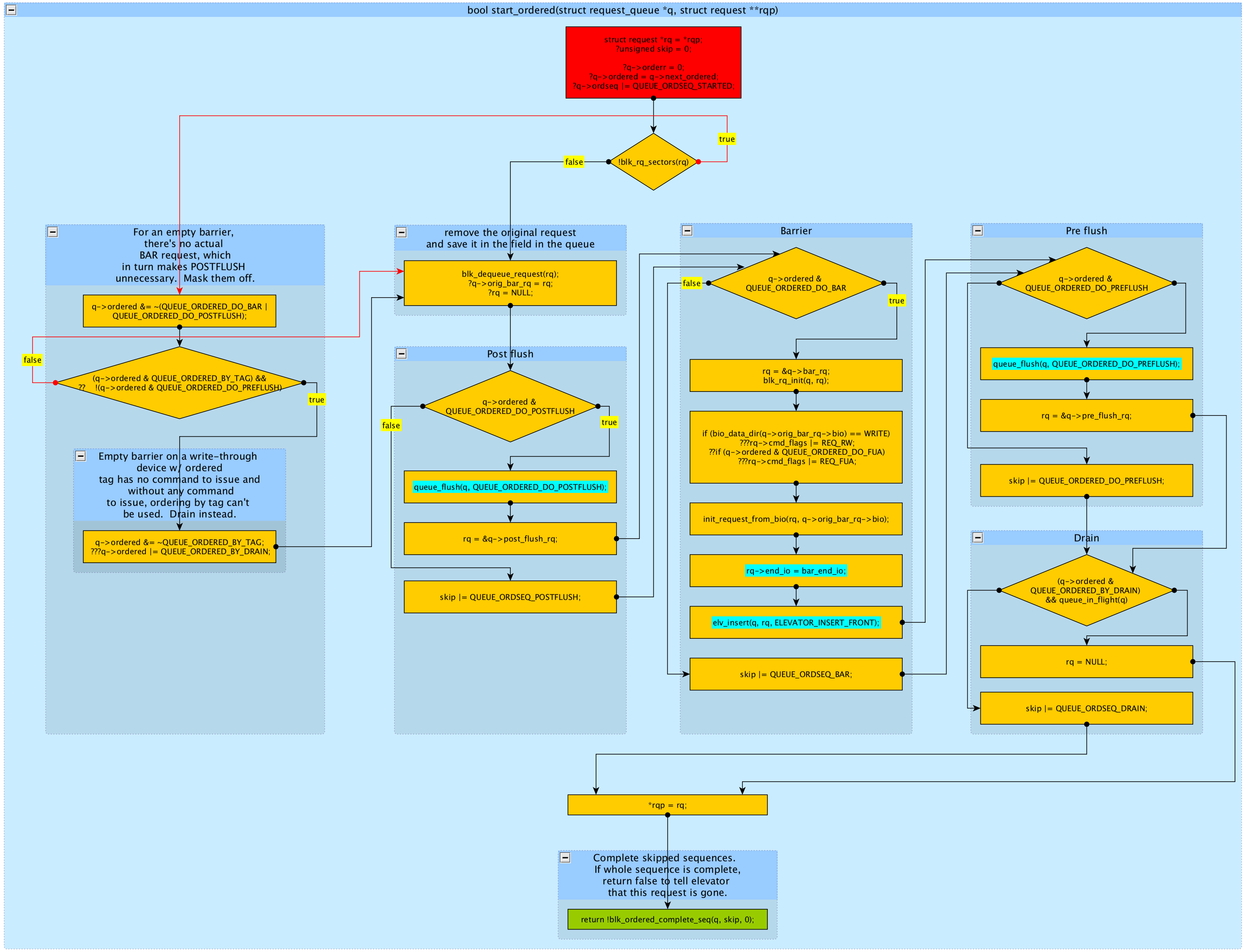

bool start_ordered(struct request_queue *q, struct request **rqp)
struct request *rq = *rqp;
?unsigned skip = 0;
?q->orderr = 0;
?q->ordered = q->next_ordered;
?q->ordseq |= QUEUE_ORDSEQ_STARTED;
!blk_rq_sectors(rq)
true
false
For an empty barrier, there's no actual BAR request, which in turn makes POSTFLUSH unnecessary. Mask them off.
q->ordered &= ~(QUEUE_ORDERED_DO_BAR | QUEUE_ORDERED_DO_POSTFLUSH);
(q->ordered & QUEUE_ORDERED_BY_TAG) &&
?? !(q->ordered & QUEUE_ORDERED_DO_PREFLUSH)
false
true
Empty barrier on a write-through device w/ ordered tag has no command to issue and without any command to issue, ordering by tag can't be used. Drain instead.
q->ordered &= ~QUEUE_ORDERED_BY_TAG;
???q->ordered |= QUEUE_ORDERED_BY_DRAIN;
remove the original request and save it in the field in the queue
blk_dequeue_request(rq);
?q->orig_bar_rq = rq;
?rq = NULL;
Post flush
q->ordered & QUEUE_ORDERED_DO_POSTFLUSH
false
true
queue_flush(q, QUEUE_ORDERED_DO_POSTFLUSH);
rq = &q->post_flush_rq;
skip |= QUEUE_ORDSEQ_POSTFLUSH;
Barrier
q->ordered & QUEUE_ORDERED_DO_BAR
false
true
rq = &q->bar_rq;
blk_rq_init(q, rq);
if (bio_data_dir(q->orig_bar_rq->bio) == WRITE)
???rq->cmd_flags |= REQ_RW;
??if (q->ordered & QUEUE_ORDERED_DO_FUA)
???rq->cmd_flags |= REQ_FUA;
init_request_from_bio(rq, q->orig_bar_rq->bio);
rq->end_io = bar_end_io;
elv_insert(q, rq, ELEVATOR_INSERT_FRONT);
skip |= QUEUE_ORDSEQ_BAR;
Pre flush
q->ordered & QUEUE_ORDERED_DO_PREFLUSH
queue_flush(q, QUEUE_ORDERED_DO_PREFLUSH);
rq = &q->pre_flush_rq;
skip |= QUEUE_ORDERED_DO_PREFLUSH;
Drain
(q->ordered & QUEUE_ORDERED_BY_DRAIN) && queue_in_flight(q)
rq = NULL;
skip |= QUEUE_ORDSEQ_DRAIN;
*rqp = rq;
Complete skipped sequences. If whole sequence is complete, return false to tell elevator that this request is gone.
return !blk_ordered_complete_seq(q, skip, 0);

FIGURE 68 Start ordered flow

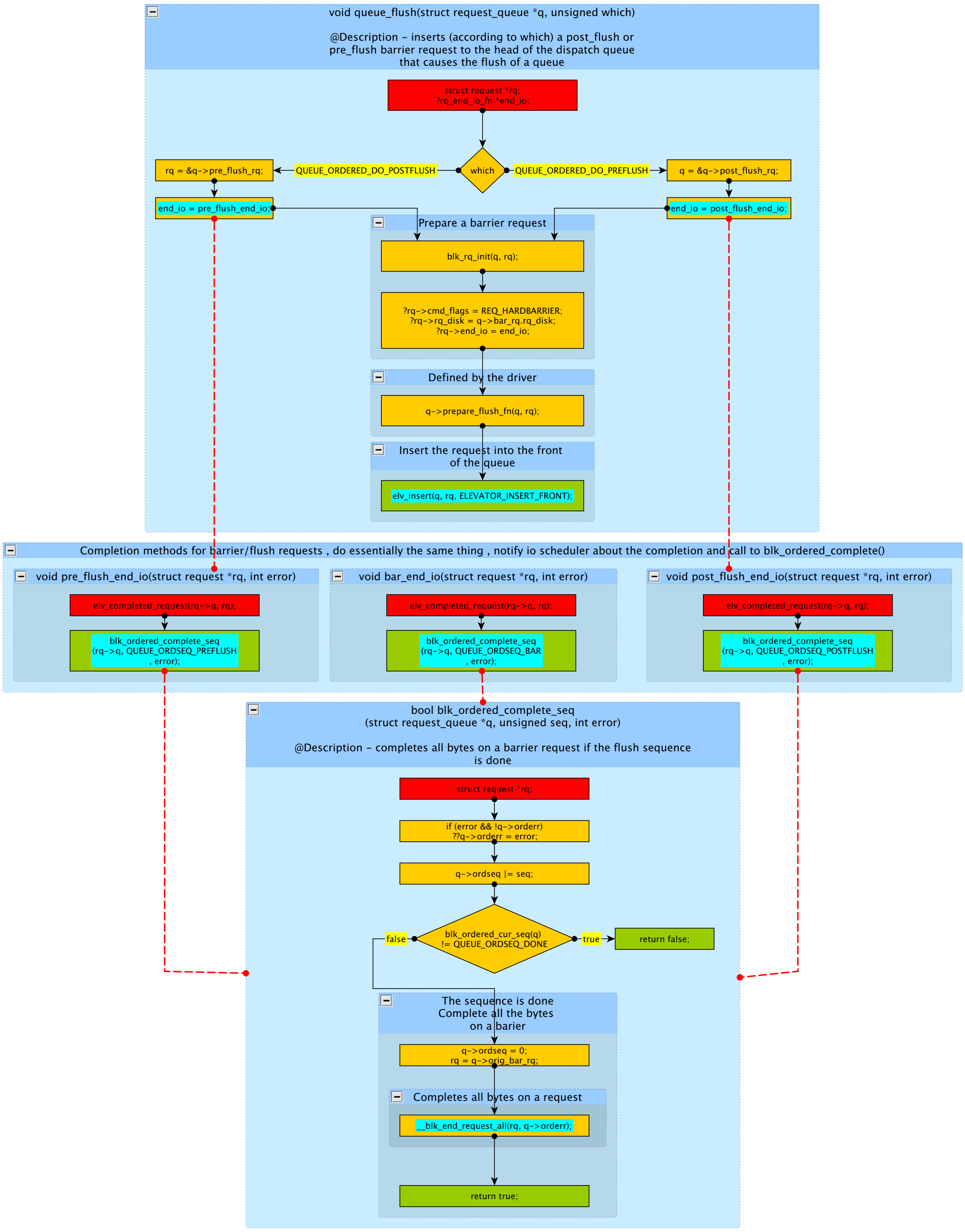

void queue_flush(struct request_queue *q, unsigned which)
@Description – inserts (according to which) a post_flush or pre_flush barrier request to the head of the dispatch queue that causes the flush of a queue
struct request *rq;
?rq_end_io_fn *end_io;
which
QUEUE_ORDERED_DO_POSTFLUSH
QUEUE_ORDERED_DO_PREFLUSH
rq = &q->pre_flush_rq;
q = &q->post_flush_rq;
end_io = pre_flush_end_io;
end_io = post_flush_end_io;
Prepare a barrier request
blk_rq_init(q, rq);
?rq->cmd_flags = REQ_HARDBARRIER;
?rq->rq_disk = q->bar_rq.rq_disk;
?rq->end_io = end_io;
Defined by the driver
q->prepare_flush_fn(q, rq);
Insert the request into the front of the queue
elv_insert(q, rq, ELEVATOR_INSERT_FRONT);
Completion methods for barrier/flush requests , do essentially the same thing , notify io scheduler about the completion and call to blk_ordered_complete()
void pre_flush_end_io(struct request *rq, int error)
elv_completed_request(rq->q, rq);
blk_ordered_complete_seq
(rq->q, QUEUE_ORDSEQ_PREFLUSH
, error);
void bar_end_io(struct request *rq, int error)
elv_completed_request(rq->q, rq);
blk_ordered_complete_seq
(rq->q, QUEUE_ORDSEQ_BAR
, error);
void post_flush_end_io(struct request *rq, int error)
elv_completed_request(rq->q, rq);
blk_ordered_complete_seq
(rq->q, QUEUE_ORDSEQ_POSTFLUSH
, error);
bool blk_ordered_complete_seq
(struct request_queue *q, unsigned seq, int error)
@Description – completes all bytes on a barrier request if the flush sequence is done
struct request *rq;
if (error && !q->orderr)
??q->orderr = error;
q->ordseq |= seq;
blk_ordered_cur_seq(q)
!= QUEUE_ORDSEQ_DONE
false
true
return false;
The sequence is done
Complete all the bytes
on a barier
q->ordseq = 0;
rq = q->orig_bar_rq;
Completes all bytes on a request
__blk_end_request_all(rq, q->orderr);
return true;

FIGURE 69 Flush queue flow

full. If the queue is no longer congested or full then the appropriate flags are cleared. In addition, if a queue is not full anymore and there are some processes in a wait queue that was sent to sleep by get_request one of these processes is woken up and removed from the wait queue.

If there are some starved processes in an opposite direction one of them is also tried to be awaken if the queue is not full anymore in that (opposite) direction. The functions above happens through a combination of freed_request() and __freed_request() functions. The behavior of a process after it is awaken (i.e. it becomes batching etc.) is described in section 5.9.

APPENDIX 5.15 Complete Request

The complicated structure of a request is the main issue for a request completion. Each request contains a list of bios, which in turn are an array of io_vecs and must have the option of being completed partially, since not every driver may have the ability to complete all the bytes in one DMA.

The below sequence is as follows (for a single direction request, bi directional request involves an extra two way wrapper and only adds complexity):

1. Some stacking driver wants to complete a number of bytes on a request (but not to complete the whole request) , and not to remove it even if it is completed.
   (a) Account for the completion of a number of bytes.
   (b) Iterate over requests bios and complete each one (by means of req_bio_endio(), that does actually the same thing as step 1, but for a bio, rather then request, and if all the bytes of a bio were completed call its completion routine).
   (c) Repeat until the request is completed (checked via number of bytes requested), if a partially completed bio is encountered update its io_vecs array index, to point the first uncompleted io_vec and possibly (if some bytes of this io_vec were transferred) update its length and offset.
   (d) Update requests length, sectors etc...
2. The driver wants to complete a number of bytes on a request
   (a) call step 1
   (b) If no uncompleted bios left in the request:
      i. Account for request completion.
      ii. Remove the request by means of blk_put_request() (see section 5.14

Step 1 is completed by a large function update_request(). Step 2 is completed by blk_end_request() that calls for blk_finish_request() (2.2.1 , 2.2.2)

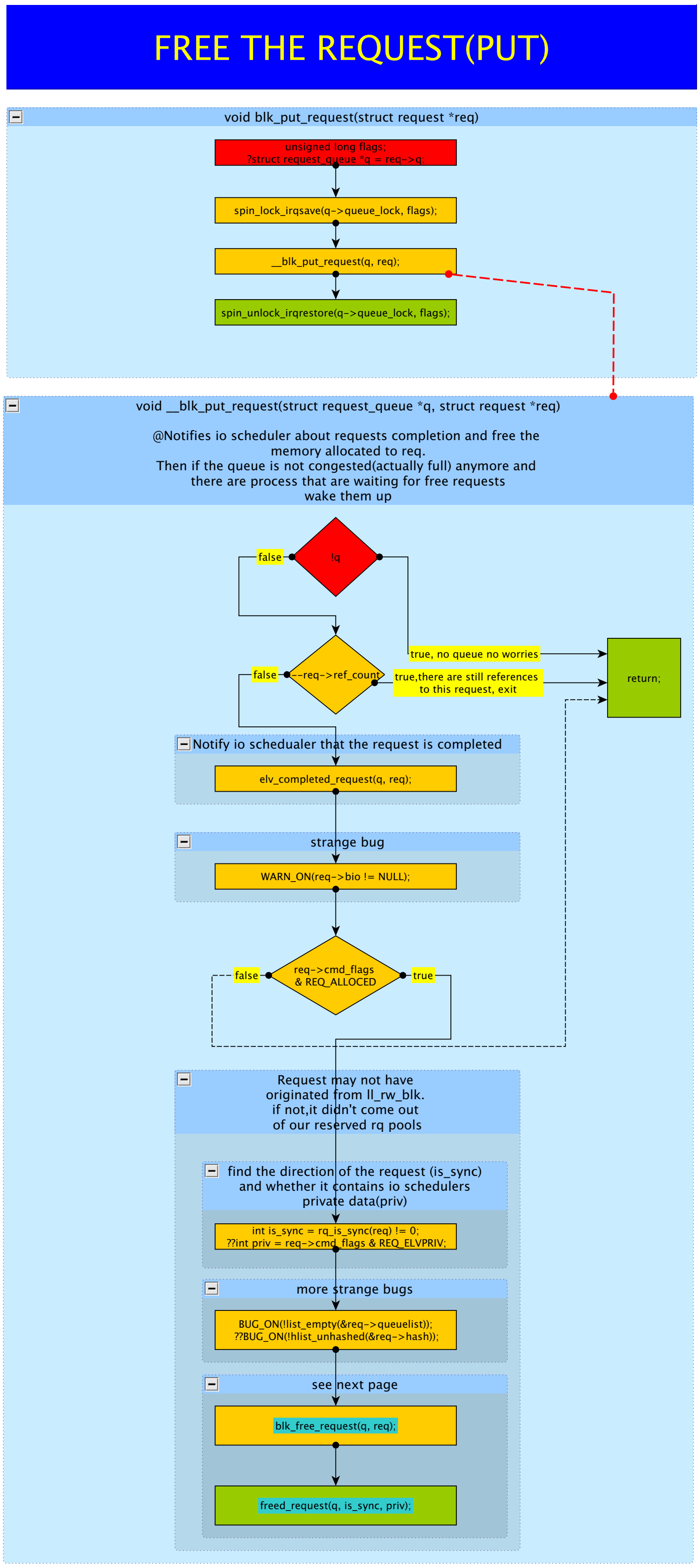

FREE THE REQUEST(PUT)
void blk_put_request(struct request *req)
unsigned long flags;
?struct request_queue *q = req->q;
spin_lock_irqsave(q->queue_lock, flags);
__blk_put_request(q, req);
spin_unlock_irqrestore(q->queue_lock, flags);
void __blk_put_request(struct request_queue *q, struct request *req)
@Notifies io scheduler about requests completion and free the
memory allocated to req.
Then if the queue is not congested(actually full) anymore and
there are process that are waiting for free requests
wake them up
false
!q
true, no queue no worries
false
--req->ref_count
true,there are still references
to this request, exit
return;
Notify io schedualer that the request is completed
elv_completed_request(q, req);
strange bug
WARN_ON(req->bio != NULL);
false
req->cmd_flags
& REQ_ALLOCED
true
Request may not have
originated from ll_rw_blk.
if not,it didn't come out
of our reserved rq pools
find the direction of the request (is_sync)
and whether it contains io schedulers
private data(priv)
int is_sync = rq_is_sync(req) != 0;
??int priv = req->cmd_flags & REQ_ELVPRIV;
more strange bugs
BUG_ON(!list_empty(&req->queuelist));
??BUG_ON(!hlist_unhashed(&req->hash));
see next page
blk_free_request(q, req);
freed_request(q, is_sync, priv);

FIGURE 70 Put request flow

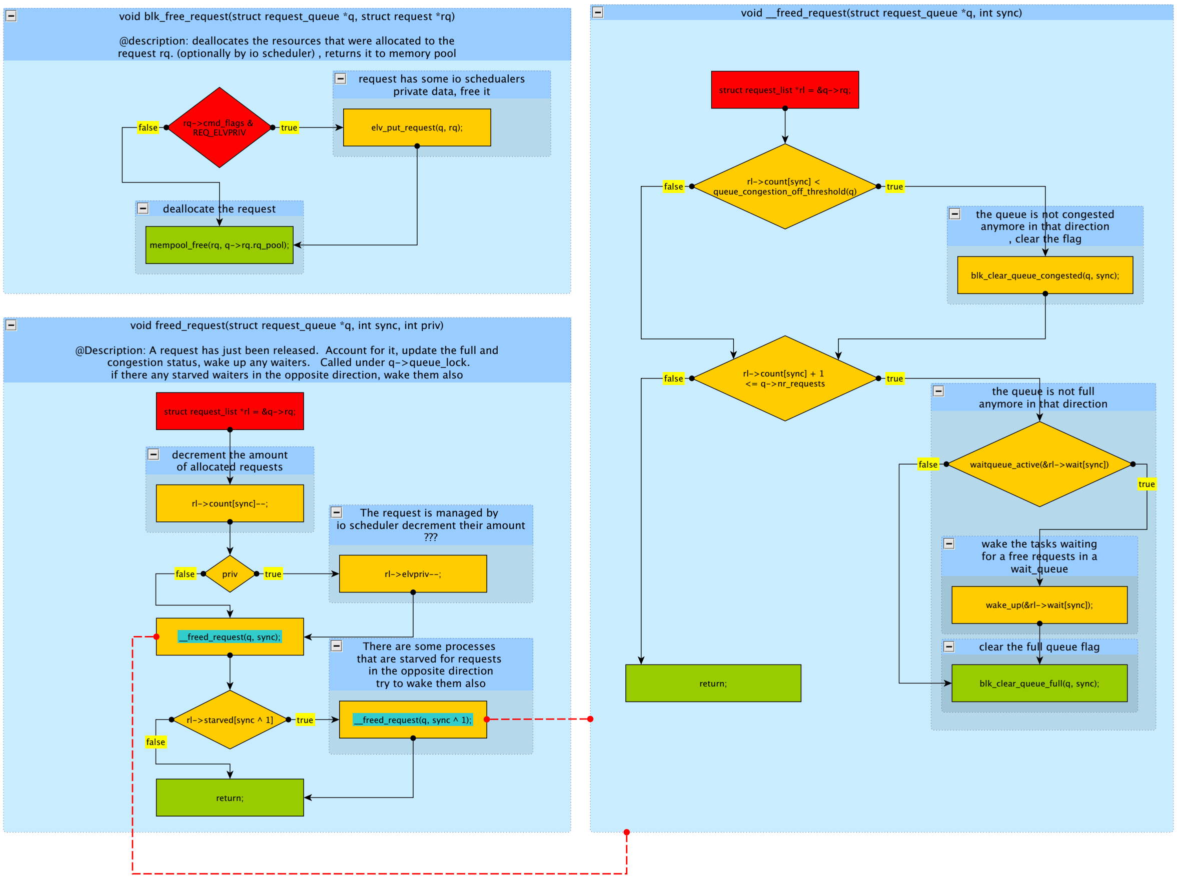

void blk_free_request(struct request_queue *q, struct request *rq)
@description: deallocates the resources that were allocated to the request rq. (optionally by io scheduler) , returns it to memory pool
request has some io schedualers private data, free it
false
rq->cmd_flags & REQ_ELVPRIV
true
elv_put_request(q, rq);
deallocate the request
mempool_free(rq, q->rq.rq_pool);
void freed_request(struct request_queue *q, int sync, int priv)
@Description: A request has just been released. Account for it, update the full and congestion status, wake up any waiters. Called under q->queue_lock.
if there any starved waiters in the opposite direction, wake them also
struct request_list *rl = &q->rq;
decrement the amount of allocated requests
rl->count[sync]--;
The request is managed by io scheduler decrement their amount ???
false
priv
true
rl->elvpriv--;
__freed_request(q, sync);
There are some processes that are starved for requests in the opposite direction try to wake them also
rl->starved[sync ^ 1]
true
false
__freed_request(q, sync ^ 1);
return;
void __freed_request(struct request_queue *q, int sync)
struct request_list *rl = &q->rq;
false
rl->count[sync] < queue_congestion_off_threshold(q)
true
the queue is not congested anymore in that direction , clear the flag
blk_clear_queue_congested(q, sync);
false
rl->count[sync] + 1 <= q->nr_requests
true
the queue is not full anymore in that direction
false
waitqueue_active(&rl->wait[sync])
true
wake the tasks waiting for a free requests in a wait_queue
wake_up(&rl->wait[sync]);
clear the full queue flag
blk_clear_queue_full(q, sync);
return;

FIGURE 71 Free request flow

Additional logic is added inside those functions to treat the tagged request (release the tag for a completed request). The barrier requests are distinguished from the regular requests since they do not have actual bios to complete.

APPENDIX 5.16 Linux Block Layer Usage In Our Systems

We used Linux block I/O layer in our asynchronous mirror product which is described in Chapter 4.

In the asynchronous mirror project we essentially replicated block devices across hosts. Using the block I/O layer for our asynchronous mirror software allows us to support any hardware (SSD, magnetic disk etc.) and any higher layer software. (File systems and other software that use block device directly such as databases etc.)

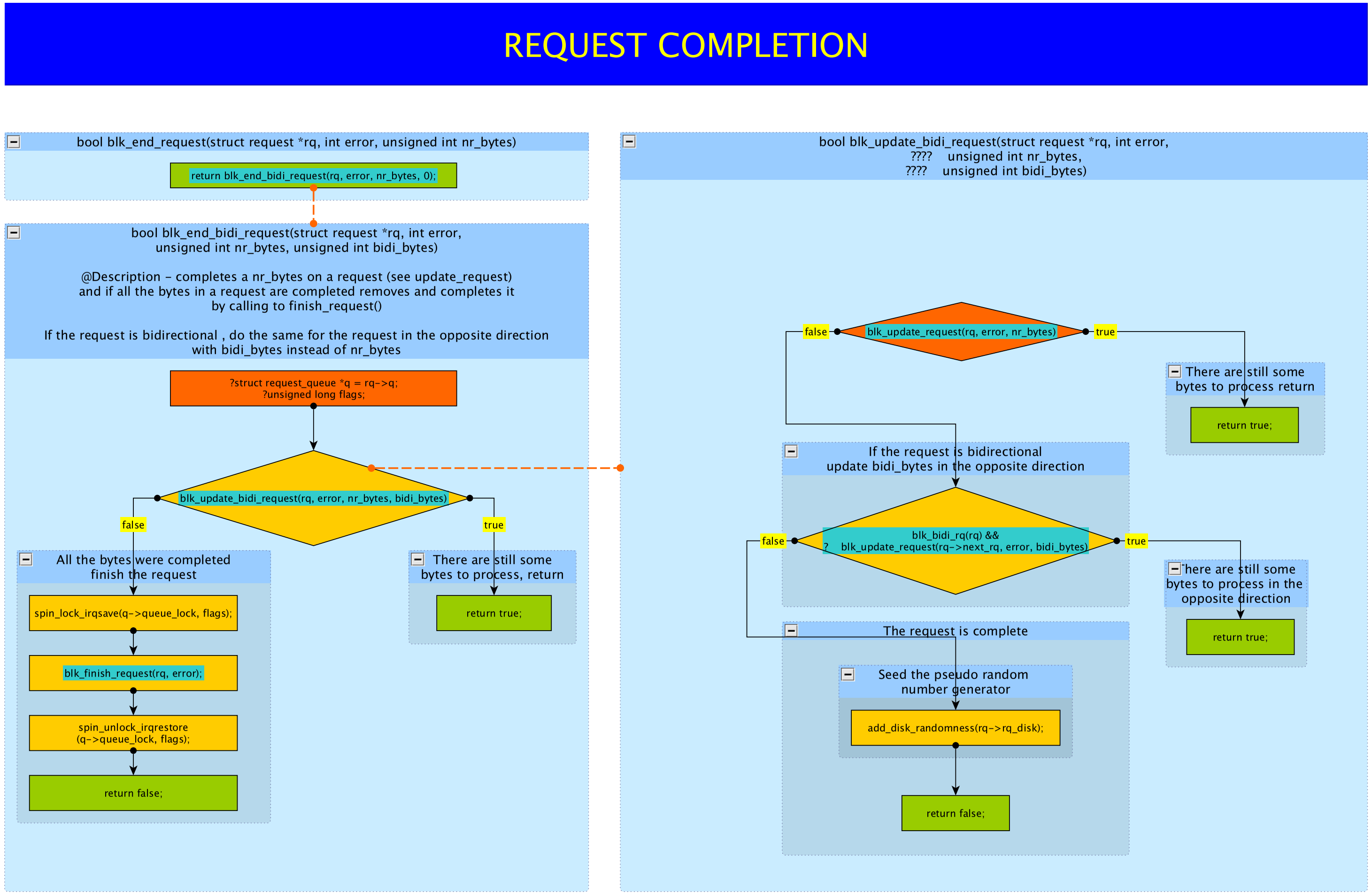

REQUEST COMPLETION
bool blk_end_request(struct request *rq, int error, unsigned int nr_bytes)
return blk_end_bidi_request(rq, error, nr_bytes, 0);
bool blk_end_bidi_request(struct request *rq, int error,
unsigned int nr_bytes, unsigned int bidi_bytes)
@Description - completes a nr_bytes on a request (see update_request)
and if all the bytes in a request are completed removes and completes it
by calling to finish_request()
If the request is bidirectional , do the same for the request in the opposite direction
with bidi_bytes instead of nr_bytes
?struct request_queue *q = rq->q;
?unsigned long flags;
blk_update_bidi_request(rq, error, nr_bytes, bidi_bytes)
false
true
All the bytes were completed
finish the request
spin_lock_irqsave(q->queue_lock, flags);
blk_finish_request(rq, error);
spin_unlock_irqrestore
(q->queue_lock, flags);
return false;
There are still some
bytes to process, return
return true;
bool blk_update_bidi_request(struct request *rq, int error,
???? unsigned int nr_bytes,
???? unsigned int bidi_bytes)
blk_update_request(rq, error, nr_bytes)
false
true
There are still some
bytes to process return
return true;
If the request is bidirectional
update bidi_bytes in the opposite direction
blk_bidi_rq(rq) &&
? blk_update_request(rq->next_rq, error, bidi_bytes)
false
true
There are still some
bytes to process in the
opposite direction
return true;
The request is complete
Seed the pseudo random
number generator
add_disk_randomness(rq->rq_disk);
return false;

FIGURE 72 End request flow

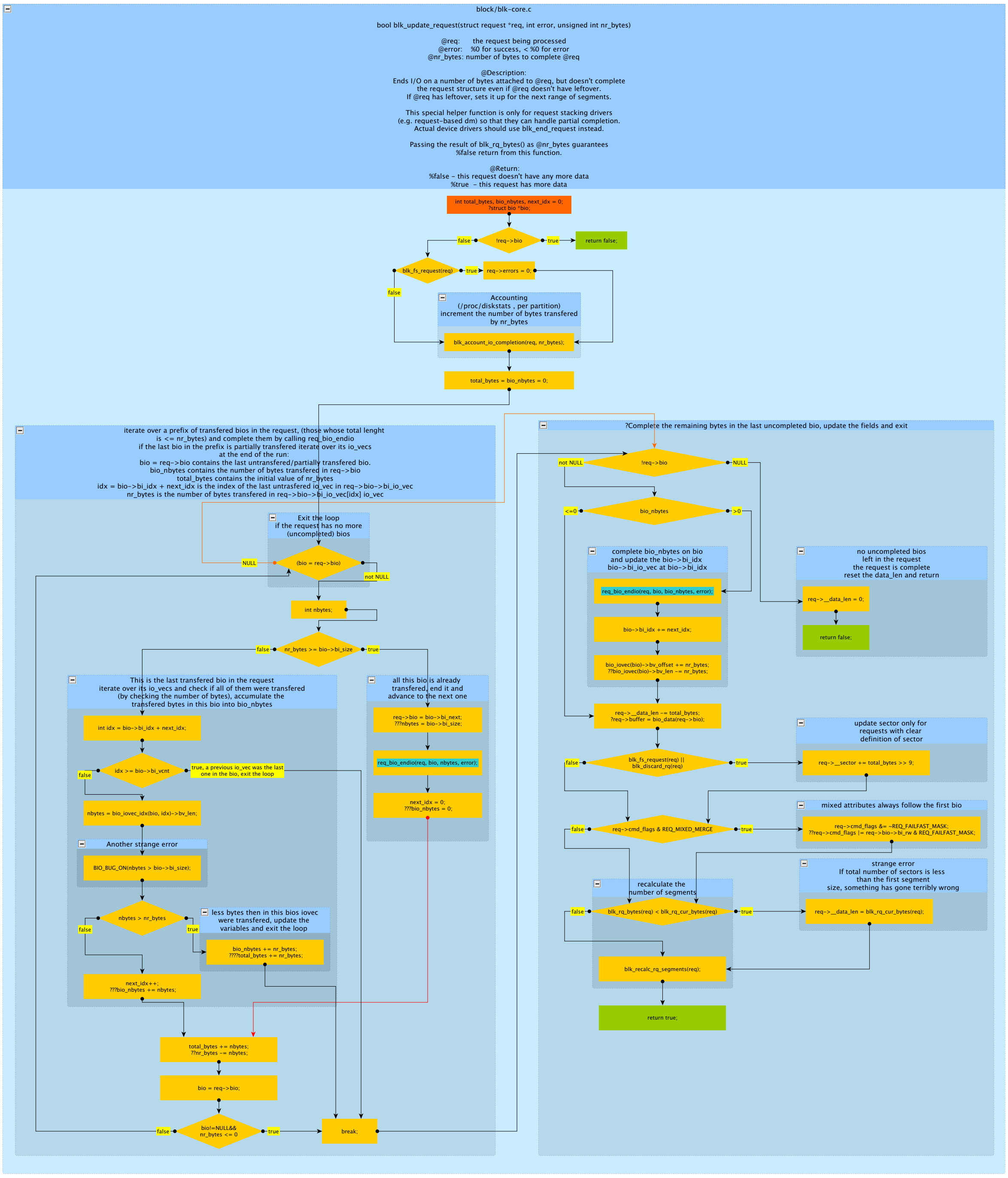

block/blk-core.c
bool blk_update_request(struct request *req, int error, unsigned int nr_bytes)
@req: the request being processed
@error: %0 for success, < %0 for error
@nr_bytes: number of bytes to complete @req
@Description:
Ends I/O on a number of bytes attached to @req, but doesn't complete
the request structure even if @req doesn't have leftover.
If @req has leftover, sets it up for the next range of segments.
This special helper function is only for request stacking drivers
(e.g. request-based dm) so that they can handle partial completion.
Actual device drivers should use blk_end_request instead.
Passing the result of blk_rq_bytes() as @nr_bytes guarantees
%false return from this function.
@Return:
%false - this request doesn't have any more data
%true - this request has more data
int total_bytes, bio_nbytes, next_idx = 0;
?struct bio *bio;
!req->bio
false
true
return false;
blk_fs_request(req)
true
req->errors = 0;
false
Accounting
(/proc/diskstats , per partition)
increment the number of bytes transfered
by nr_bytes
blk_account_io_completion(req, nr_bytes);
total_bytes = bio_nbytes = 0;
iterate over a prefix of transfered bios in the request, (those whose total lenght
is <= nr_bytes) and complete them by calling req_bio_endio
if the last bio in the prefix is partially transfered iterate over its io_vecs
at the end of the run:
bio = req->bio contains the last untransfered/partially transfered bio.
bio_nbytes contains the number of bytes transfered in req->bio
total_bytes contains the initial value of nr_bytes
idx = bio->bi_idx + next_idx is the index of the last untrasfered io_vec in req->bio->bi_io_vec
nr_bytes is the number of bytes transfered in req->bio->bi_io_vec[idx] io_vec
Exit the loop
if the request has no more
(uncompleted) bios
NULL
(bio = req->bio)
not NULL
int nbytes;
false
nr_bytes >= bio->bi_size
true
This is the last transfered bio in the request
iterate over its io_vecs and check if all of them were transfered
(by checking the number of bytes), accumulate the
transfered bytes in this bio into bio_nbytes
int idx = bio->bi_idx + next_idx;
false
idx >= bio->bi_vcnt
true, a previous io_vec was the last
one in the bio, exit the loop
nbytes = bio_iovec_idx(bio, idx)->bv_len;
Another strange error
BIO_BUG_ON(nbytes > bio->bi_size);
false
nbytes > nr_bytes
true
less bytes then in this bios iovec
were transfered, update the
variables and exit the loop
bio_nbytes += nr_bytes;
????total_bytes += nr_bytes;
next_idx++;
???bio_nbytes += nbytes;
all this bio is already
transfered, end it and
advance to the next one
req->bio = bio->bi_next;
???nbytes = bio->bi_size;
req_bio_endio(req, bio, nbytes, error);
next_idx = 0;
???bio_nbytes = 0;
total_bytes += nbytes;
??nr_bytes -= nbytes;
bio = req->bio;
false
bio!=NULL&&
nr_bytes <= 0
true
break;
?Complete the remaining bytes in the last uncompleted bio, update the fields and exit
not NULL
!req->bio
NULL
<=0
bio_nbytes
>0
complete bio_nbytes on bio
and update the bio->bi_idx
bio->bi_io_vec at bio->bi_idx
req_bio_endio(req, bio, bio_nbytes, error);
bio->bi_idx += next_idx;
bio_iovec(bio)->bv_offset += nr_bytes;
??bio_iovec(bio)->bv_len -= nr_bytes;
no uncompleted bios
left in the request
the request is complete
reset the data_len and return
req->__data_len = 0;
return false;
req->__data_len -= total_bytes;
?req->buffer = bio_data(req->bio);
false
blk_fs_request(req) ||
blk_discard_rq(req)
true
update sector only for
requests with clear
definition of sector
req->__sector += total_bytes >> 9;
false
req->cmd_flags & REQ_MIXED_MERGE
true
mixed attributes always follow the first bio
req->cmd_flags &= ~REQ_FAILFAST_MASK;
??req->cmd_flags |= req->bio->bi_rw & REQ_FAILFAST_MASK;
strange error
If total number of sectors is less
than the first segment
size, something has gone terribly wrong
recalculate the
number of segments
false
blk_rq_bytes(req) < blk_rq_cur_bytes(req)
true
req->__data_len = blk_rq_cur_bytes(req);
blk_recalc_rq_segments(req);
return true;

FIGURE 73 Update request flow

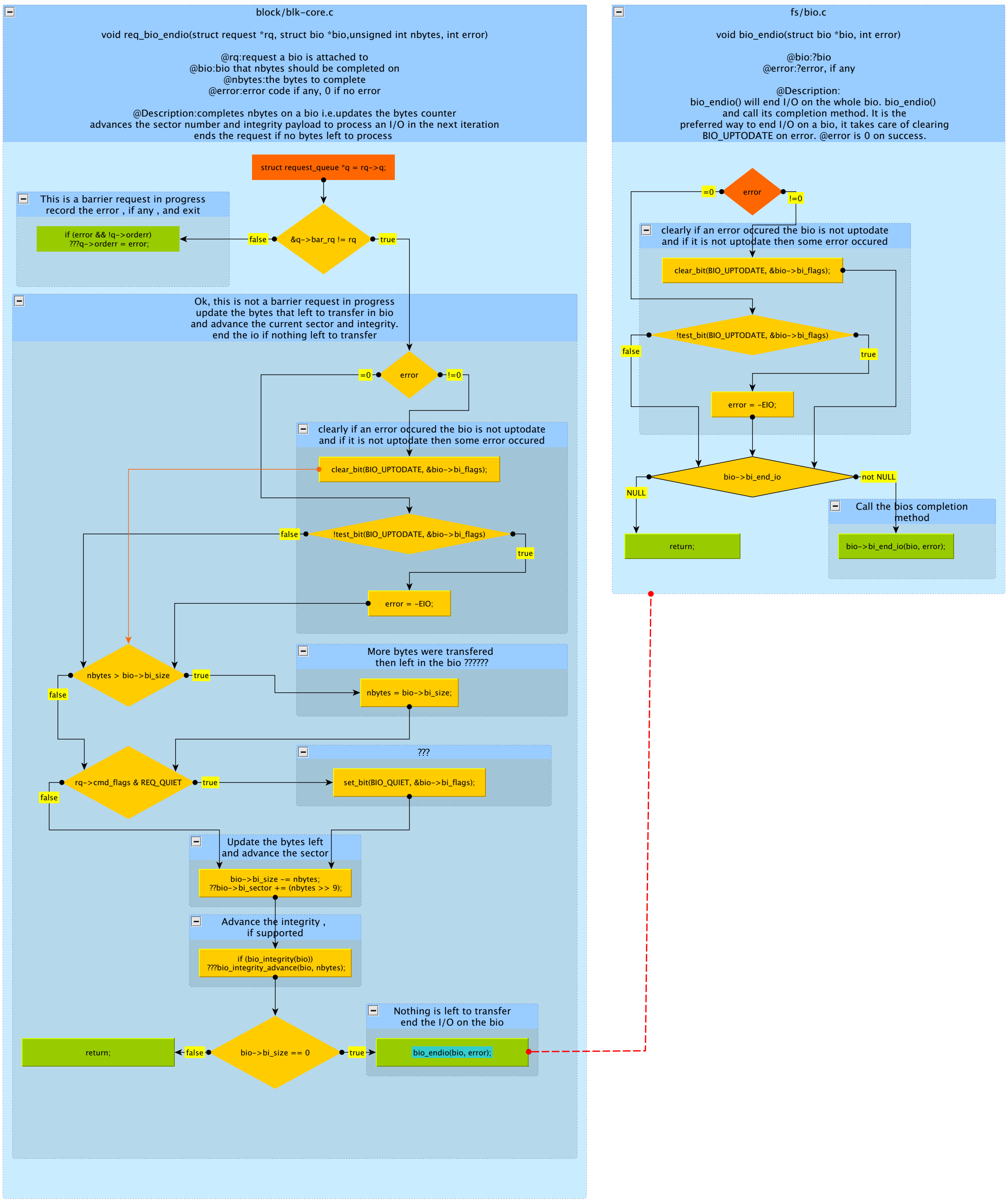

block/blk-core.c
void req_bio_endio(struct request *rq, struct bio *bio,unsigned int nbytes, int error)
@rq:request a bio is attached to
@bio:bio that nbytes should be completed on
@nbytes:the bytes to complete
@error:error code if any, 0 if no error
@Description:completes nbytes on a bio i.e.updates the bytes counter
advances the sector number and integrity payload to process an I/O in the next iteration
ends the request if no bytes left to process
struct request_queue *q = rq->q;
This is a barrier request in progress
record the error , if any , and exit
if (error && !q->orderr)
???q->orderr = error;
false
&q->bar_rq != rq
true
Ok, this is not a barrier request in progress
update the bytes that left to transfer in bio
and advance the current sector and integrity.
end the io if nothing left to transfer
=0
error
!=0
clearly if an error occured the bio is not uptodate
and if it is not uptodate then some error occured
clear_bit(BIO_UPTODATE, &bio->bi_flags);
false
!test_bit(BIO_UPTODATE, &bio->bi_flags)
true
error = -EIO;
nbytes > bio->bi_size
true
false
More bytes were transfered
then left in the bio ??????
nbytes = bio->bi_size;
???
rq->cmd_flags & REQ_QUIET
true
false
set_bit(BIO_QUIET, &bio->bi_flags);
Update the bytes left
and advance the sector
bio->bi_size -= nbytes;
??bio->bi_sector += (nbytes >> 9);
Advance the integrity ,
if supported
if (bio_integrity(bio))
???bio_integrity_advance(bio, nbytes);
return;
false
bio->bi_size == 0
true
Nothing is left to transfer
end the I/O on the bio
bio_endio(bio, error);
fs/bio.c
void bio_endio(struct bio *bio, int error)
@bio:?bio
@error:?error, if any
@Description:
bio_endio() will end I/O on the whole bio. bio_endio()
and call its completion method. It is the
preferred way to end I/O on a bio, it takes care of clearing
BIO_UPTODATE on error. @error is 0 on success.
=0
error
!=0
clearly if an error occured the bio is not uptodate
and if it is not uptodate then some error occured
clear_bit(BIO_UPTODATE, &bio->bi_flags);
false
!test_bit(BIO_UPTODATE, &bio->bi_flags)
true
error = -EIO;
NULL
bio->bi_end_io
not NULL
Call the bios completion
method
return;
bio->bi_end_io(bio, error);

FIGURE 74 End I/O flow

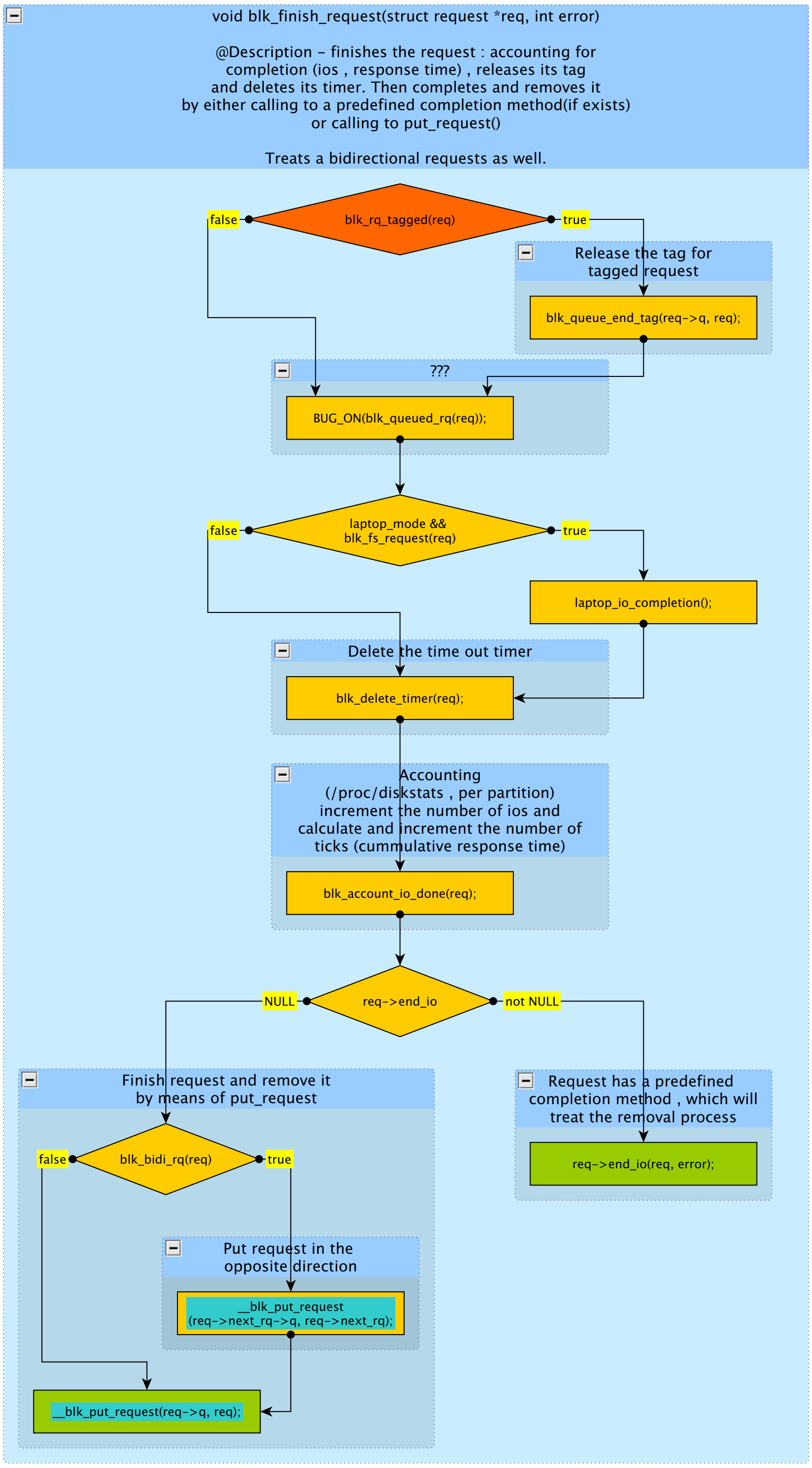

void blk_finish_request(struct request *req, int error)
@Description – finishes the request : accounting for completion (ios , response time) , releases its tag and deletes its timer. Then completes and removes it by either calling to a predefined completion method(if exists) or calling to put_request()
Treats a bidirectional requests as well.
false
blk_rq_tagged(req)
true
Release the tag for tagged request
blk_queue_end_tag(req->q, req);
???
BUG_ON(blk_queued_rq(req));
false
laptop_mode && blk_fs_request(req)
true
laptop_io_completion();
Delete the time out timer
blk_delete_timer(req);
Accounting (/proc/diskstats , per partition) increment the number of ios and calculate and increment the number of ticks (cummulative response time)
blk_account_io_done(req);
NULL
req->end_io
not NULL
Finish request and remove it by means of put_request
Request has a predefined completion method , which will treat the removal process
req->end_io(req, error);
false
blk_bidi_rq(req)
true
Put request in the opposite direction
__blk_put_request (req->next_rq->q, req->next_rq);
__blk_put_request(req->q, req);

FIGURE 75 Finish request flow

APPENDIX 6 LGUEST AND VIRT I/O

In this chapter we introduce Lguest and Virt I/O. Lguest is a small x86 32-bit Linux hypervisor that was developed by Rusty Russell. Lguest allows running Linux under Linux and demonstrating the para-virtualization abilities in Linux.

Lguest was built, at least partially, with academic purposes. The code is very readable and thus lends itself easily for extensions. Lguest has been part of the Linux kernel since Linux 2.6.20.

We have used Lguest in our asynchronous mirror system and in our kernel profiler.

Virt I/O provides an abstraction layer for all virtual drivers under Linux, regardless of hypervisor. We have developed a serial adapter based on virt I/O and used it in our kernel profiler.

The purpose of this chapter is two-fold. First, we wish to describe Lguest to demonstrate how system virtual machines work and how simple System virtual machine (Lguest) can be constructed. Second, we explain Lguest's internal design which is critical for the two systems that we developed using Lguest.

APPENDIX 6.1 Lguest

Lguest is a type 2 hypervisor for Linux. The majority of Lguest code is implemented as a single kernel module lg.ko. Lguest allows Linux kernel to run inside Lguest virtualization environment – just like we run a process in userspace. Lguest is a 32bit product designed for x86 environment. Lguest64 is an experimental 64bit product for x86-64 environment.

Lguest uses para-virtualization. The guest Linux OS uses a set of virtual I/O drivers to communicate with the host OS. The motivation behind the creation of Lguest was Russel's, Lguest's creator, attempt to standardize a set of APIs for hypervisors to support Xen.

APPENDIX 6.2 The motivation behind Virt I/O and Lguest

In an effort to support Xen hypervisor and future hypervisors, Rusty Russel created Lguest based on the prediction that future hypervisors will require common interfaces for para-virtualization on the Linux kernel. Even though at the time VMWare has already released VMI (Virtual Machine Interface) an open, stable standard was desired. This is a single structure that encapsulates all the sensitive instructions (functions pointers) which a hypervisor might want to override. This was very similar to the VMI , but not identical. Rusty created Lguest as trivial self-contained Linux-on-Linux hypervisor. Lguest lives in the kernel source code. It runs the same OS for the guest and host and is contained in less then ten thousands lines of code. Lguest is small enough to be considered educational hypervisor and a test bed for paravirt_ops.

APPENDIX 6.3 Running Lguest

In order to run Lguest we need to recompile the kernel with Lguest support as documented in documents/lguest/lguest.txt in the kernel tree.

Once the Lguest kernel module has been inserted to the running kernel we can communicate with Lguest kernel module via the special file /dev/lguest.

Now we can create guests by calling ioctl(2) on the file.

The second thing we need in order to run guests is to create the guest boot image. A large variety of boot images exist on the Internet with various version of busybox and other tiny Linux distributions. The boot images can also be edited to include new files needed for the guest.

APPENDIX 6.4 Lguest high level design and file list

Lguest consists of five parts:

1. The guest's "paravirt_ops" implementation.
2. The virtual I/O devices and drivers.
3. The launcher which sets up, runs and services the guest.
4. The host module (lg.ko) which sets up the switcher and handles the kernel side of things for the launcher.
5. The switcher which flips the CPU between host and guest.

APPENDIX 6.4.1 The Guest Code

In order to configure Lguest we need to include CONFIG_LGUEST_GUEST=y into the kernel .config spec. This switch would compile [arch/x86/lguest/boot.c] into the kernel. By including this file the kernel can now run as a guest at boot time. While running as guest, the kernel knows that he cannot do privileged operations. As a guest the kernel "knows" that it has to ask the host to do privileged operations. [arch/x86/lguest/boot.c] contains all the replacements for such low-level native hardware operations.

APPENDIX 6.4.2 The Launcher Code

The file [Documents/lguest/lguest.c] is the launcher code, a simple program that lays out the "physical" (not actually physical but rather what the guest "thinks" is "physical" it is actually process memory on the host that can be swapped.) Memory for the new guest by mapping the kernel image and the virtual devices. It then the special file /dev/lguest to tell the kernel about the guest and control it.

APPENDIX 6.4.3 The Switcher Code

The file [drivers/lguest/x86/switcher_32.S] is the switcher. The switcher is compiled as part of the "lg.ko" module. The switcher resides at 0xFFC00000 (or 0xFFE00000) astride both the host and guest virtual memory, to do the low-level Guest<->Host switch. It is as simple as it can easily be made and is x86 specific. What it does is switch the CPU registers state between guest and host processing.

APPENDIX 6.4.4 The Host Module: lg.ko

This kernel module contains the files:

We'll now start to explore in core details Lguest. The document contains many code samples. Most of them are not fully copied from the original kernel source tree but rather have the important issues cut and pasted here coupled with additional comments not present in the official kernel.

APPENDIX 6.5 The Guest

The guest starts with the kernel booting into "startup_32" in [arch/x86/kernel/head_32.S]. It expects a boot header, which is created by the boot loader (The launcher in Lguest case). When we configure CONFIG_PARAVIRT=y, "startup_32" function preforms the following: it checks the 'BP_hardware_subarch' field that is part of the boot header, if it is set to '1', call "lguest_entry" which is the starting point of the guest and is located at the [arch/x86/lguest/i386_Head.S].

```
__HEAD ENTRY(startup_32)
...
#ifdef
CONFIG_PARAVIRT
...
/*
 * Paravirt-compatible boot parameters.
 * Look to see what architecture we're booting under.
 */
    movl pa(boot_params + BP_hardware_subarch), %eax
    set %eax to 1*/
...
    movl pa(subarch_entries)(,%eax,4), %eax
    subl $__PAGE_OFFSET, %eax
    jmp %eax
subarch_entries:
    .long default_entry
```

TABLE 6 Lguest lg.ko files

drivers/lguest/lguest_user.c	This contains all the /dev/lguest code, whereby the userspace launcher controls and communicates with the guest.
drivers/lguest/core.c	This contains the "run_guest()" routine which actually calls into the Host<->Guest Switcher and analyzes the return, such as determining if the guest wants the host to do something. This file also contains useful helper routines.
drivers/lguest/x86/core.c	This file contains the x86-specific lguest code for future porting of lguest to other architectures
drivers/lguest/hypercalls.c	Just as userspace programs request kernel operations through a system call, the guest requests host operations through a "hypercall". As you'd expect, this code is basically one big C switch statement.
drivers/lguest/segments.c	The x86 architecture support segments. The segment handling code consists of simple sanity checks.
drivers/lguest/page_tables.c	The guest provides a virtual to physical mapping, but we can neither trust it nor use it. This is because the guest does not know what the physical addresses of his memory are, so he can map to places where he shouldn't. The host verifies and converts the mapping here and then points the CPU to the converted guest pages when running the guest.
drivers/lguest/ interrupts_and_traps.c	There are three classes of interrupts: 1. Real hardware interrupts which occur while we're running the guest 2. Interrupts for virtual devices attached to the guest 3. Traps and faults from the guest. Real hardware interrupts are delivered to the host, not to the guest. Virtual interrupts are delivered to the guest, but Lguest makes them look identical to how real hardware would deliver them. Traps from the guest can be set up to go directly back to the guest, but sometimes the host wants to see them first, so we also have a way of "reflecting" them into the guest as if they had been delivered to it directly.

```
    .long lguest_entry
/* normal x86/PC */
/* lguest hypervisor */ /* Xen hypervisor */
     .long xen_entry
     num_subarch_entries = (. - subarch_entries) /
4 .previous
#endif /* CONFIG_PARAVIRT */
```

The routine "lguest_entry" is making an "initialization" hypercall to the host.

```
movl $LHCALL_LGUEST_INIT, %eax
/* put the hypercall index in eax*/
movl $lguest_data - __PAGE_OFFSET, %ebx
/* put the address of lguest_data in ebx */
int $LGUEST_TRAP_ENTRY
/* make a hypercall */
```

The hypercall mechanism will be explained shortly but we can already see that it is using a software interrupt. "lguest_entry" then jumps to the C function "lguest_init()" which is at [arch/x86/lguest/boot.c]. This function is the part of the booting process that is specific to Lguest's guest. It sets the "pv_info" structure (general information about the para-virtualization, like it is Lguest's guest and not to run at the most privileged level etc.), and most importantly to set up all the lguest overrides for sensitive operations, the "pv_ops" interface.

```
__init void lguest_init(void) {
/* We're under lguest. */
    pv_info.name = "lguest";
/* We're running at privilege level 1, not 0 as normal. */
    pv_info.kernel_rpl = 1;
/*
 * We set up all the lguest overrides for
 * sensitive operations.
 * These are detailed with the operations themselves.
 */
/* Interrupt-related operations */
...
    pv_irq_ops.irq_disable = PV_CALLEE_SAVE(irq_disable);
    pv_irq_ops.irq_enable =
        __PV_IS_CALLEE_SAVE(lg_irq_enable);
...
/* Intercepts of various CPU instructions */
    pv_cpu_ops.load_gdt = lguest_load_gdt;
    pv_cpu_ops.cpuid = lguest_cpuid;
...
    pv_cpu_ops.start_context_switch =
        paravirt_start_context_switch;
    pv_cpu_ops.end_context_switch =
lguest_end_context_switch;
/* Page table management */
    pv_mmu_ops.flush_tlb_user = lguest_flush_tlb_user;
    pv_mmu_ops.flush_tlb_single = lguest_flush_tlb_single;
    pv_mmu_ops.flush_tlb_kernel = lguest_flush_tlb_kernel;
...
}
```

We would like to take a look of some interesting implementations of those functions (the size and time limits this document would not allow for the review of all of them. Instead we compensate by extensive documentation of the main features) Before we go into the implementation we need to understand how our guest contact the host to request privileged operations.

APPENDIX 6.5.1 Guest – Host Communication

The first method is called a "hypercall". It is usually preformed using a software interrupt (like we saw at "lguest_entry") that causes a kernel trap. The hypercall mechanism uses the highest unused trap code:

```
#define LGUEST_TRAP_ENTRY 0x1F /* LGUEST_TRAP_ENTRY=31 */
```

Traps 32 and above are used by real hardware interrupts. Seventeen hypercalls are available. The hypercall number is put in the %eax register, and the arguments (when required) are placed in %ebx, %ecx, %edx and %esi. If a return value makes sense, its returned in %eax. The hypercalls are preformed using the "hcall()" function in [arch/x86/include/arm/lguest_hcall.h].:

```
static inline unsigned long
hcall(unsigned long call, unsigned long arg1,
unsigned long arg2,
     unsigned long arg3,
unsigned long arg4)
{
/* "int" is the Intel instruction to trigger a trap. */
    asm volatile("int $" __stringify(LGUEST_TRAP_ENTRY)
/* The call in %eax (aka "a") might be overwritten */
        : "=a"(call)
/* The arguments are in %eax, %ebx, %ecx, %edx & %esi */
: "a"(call), "b"(arg1), "c"(arg2), "d"(arg3), "S"(arg4)
/* "memory" means this might write somewhere in memory.
 * This isn't true for all calls, but it's safe to tell
 * gcc that it might happen so it doesn't get clever. */
        : "memory");
return call;
}
```

An example of a hypercall can be seen in the "lguest_entry" routine, where we made the "int" instruction with the hypercall number (LHCALL_LGUEST_INIT) in %eax, and the hypercall argument (the address of the variable "lguest_data") in %ebx. There is a second method of communicating with the host: via "struct lguest_data", which is in [include/linux/lguest.h]. Once the guest made the initialization hypercall in "lguest entry" (LHCALL_LGUEST_INIT), that tells the host where "lguest_data" is. Both the guest and host use the "lguest_data" to publish information.

Both guest and host need to know the location of "lguest_data" because there are some functions that are called frequently, that are needed to transfer information. If instead of publishing to "lguest_data" we would perform hypercall each time the functions are called and would slow things down. However the guest and host can update fields inside "lguest_data" with a single instruction, and the host would check "lguest_data" before doing anything related to the guest.

Example: we keep an "irq_enabled" (enable interrupts) field inside our "lguest_data", which the guest can update with a single instruction (updating the "lguest_data.irq_enabled" field). The host knows to check there before it tries to deliver an interrupt. After we understand the two ways the guest and the host communicate, let's look at some interesting implementations of the PV_OPS interface. Let us be reminded that all these operations are "sensitive", i.e. operations that the guest does not have the privilege to do, so when the guest needs to perform them he would call one of these overriding functions. We sort these operations by their type:

- Interrupt-related
- CPU instructions (and related operations)
- Page-table related
- Time related.

APPENDIX 6.5.2 Interrupt Related Operations

All interrupts that arrive during the guest run-time cause an immediate switch to the host. The guest doesn't even know that an interrupt has occurred. In other words, the IDT table of the guest is simple and handles all interrupts with a switch to the host. The host controls the guest "real" IDT. The host needs to control the guest IDT as there are some interrupts that the guest cannot handle (like hardware interrupts that are designated to the host OS) or interrupts that the host must be aware of. The guest interrupt descriptor table is described in section 6.10.3.

Usually the OS has a programmable interrupt controller (PIC) as a hardware device. The PIC allows the OS configurations like blocking specific interrupt and prioritizes them. The host cannot let the guest control this device, so the guest's PIC would be virtual and very simple: it would allow only the enabling and disabling of a specific interrupt.

```
/*
 * This structure describes the lguest virtual
 * IRQ controller.
 * the full irq_chip interface is at
 * [/include/linux/irq.h]
 */
static struct irq_chip lguest_irq_controller = {
    .name = "lguest",
/* name for /proc/interrupts */
    .mask = disable_lguest_irq,
/* mask an interrupt source */
    .unmask = enable_lguest_irq,
/* unmask an interrupt source */
};
```

When a guest process would ask the PIC to disable an interrupt, what would actually occur is that the guest tells the host not to deliver this interrupt to him. This is done using the "lguest_data.interrupts" bitmap, so disabling (aka "masking") interrupt is as simple as setting a bit.

```
static void disable_lguest_irq(unsigned int irq)
{
    set bit(irq, lguest_data.blocked_interrupts);
/*  irq == interrupt index */
}

static void enable_lguest_irq(unsigned int irq)
{
    clear_bit(irq, lguest_data.blocked_interrupts);
}
```

During the function "lguest_write_idt_entry()", which writes a descriptor (a gate) to the "interrupt descriptor table" (pointed by the IDTR register), we can find an example for a call to "hcall()" that performs a hypercall to the hypervisor. The IDT tells the processor what to do when an interrupt comes in. Each entry in the table is a 64-bit descriptor: this holds the privilege level, address of the handler, and more. The guest simply asks the host to make the change because the host controls the real IDT.

```
static void lguest_write_idt_entry(gate_desc *dt,
int entrynum,
     const gate_desc *g)
{
/*
 * The gate_desc struct is 8 bytes long:
 * we hand it to the Host in two 32bit chunks
 */

u32 *desc = (u32 *)g;
/* Keep the local copy up to date. */
native_write_idt_entry(dt, entrynum, g);
/* This just memcpy g to dt*/

/* Tell Host about this new entry. */
/*
 *  each descriptor is 8 bytes, so we are going over the
 * entries of the table, each entry is a struct with two
 * unsigned long fields: a,b, which gives 8 bytes.  The
 * entry must be split like that to be compatible with the
 * "hcall()" interface that can handle 4-byte parameters
 */
hcall(LHCALL_LOAD_IDT_ENTRY, entrynum, desc[0], desc[1], 0);
```

The rest of the interrupt related operations are similar and less interesting. The guest can enable/disable interrupts using the calls to "irq_enable()"/"irq_disable()" that change the value of the field "lguest_data.irq_enabled" to X86_EFLAGS_IF/0.

The host checks the value of this field before delivering an interrupt to the guest.

APPENDIX 6.5.3 CPU Instructions for interrupt handling

The "iret" CPU instruction is the interrupt equivalent to the "ret" instruction. Instead of returning from a function call like "ret", "iret" returns from an interrupt or a trap. When we use "int" instruction, it pushes the stack of the return address (the interrupted instruction) and the EFLAGS register, which indicate the processor's current state. The EFLAGS register is pushed because the interrupt may cause flags to change. Actually, in Linux's case, the only relevant flag is IF (interrupt flag): "int" sets it to 0 thereby preventing other interrupts to interfere with the current interrupt handler. The "iret" first pops the IP (return address) and CS (the code segment) to the interrupted instruction, and then pops EFLAGS and returns to the caller atomically. The guest, on the other hand, would need to use two instructions, because the guest is not using IF. The guest does not have the privilege to change it so the guest uses the value at "lguest_data.irq_enabled" instead. The guest needs one instruction to re-enable virtual interrupts, and then another to execute "iret". The problem is that this is no longer atomic: we could be interrupted between the two. The solution to this is to surround the code with "lguest_noirq_start:" and "lguest_noirq_end:" labels. We tell the host, who is responsible for passing interrupts to the guest, that it is NEVER to interrupt the guest between the labels, even if interrupts seem to be enabled. Assembly is used in this case because a help register is needed, so we need to push its value to the stack.

```
ENTRY(lguest_iret)
    pushl %eax
/*
 * a help register is needed to restore EFLAGES
 * - push it
 */
    movl 12(%esp), %eax
/*  This is the location of the EFLAGES value */
    lguest_noirq_start:
/*
 * Note the %ss: segment prefix here.
 * Normal data accesses use the "ds" segment,
 * but that will have already been restored for
 * whatever * we're returning to (such as userspace): we can't
 * trust it.  The %ss: prefix makes sure we use the stack
 * segment, which is still valid.
 */
    movl %eax,%ss:lguest_data+LGUEST_DATA_irq_enabled
/* copy EFLAGS to "lguest_data.irq_enabled" */
    popl %eax
    iret
/* after we update "lguest_data.irq_enabled", we can iret */
lguest_noirq_end:
```

The Intel architecture defines the "lgdt" instruction that loads the Global Descriptor Table (GDT). GDT is a data structure used by the processor in order to define the characteristics of the various memory areas used during program execution, for example, the base address, the size and access privileges like execute and write permissions. These memory areas are called segments in Intel terminology. You tell the CPU where it is (and its size) using the "lgdt" instruction (there is also a similar instruction to load the IDT) and several other instructions refer to entries in the table. There are three entries which the switcher needs, so the host simply controls the entire table of the guest. (The host has the mapping to the physical base address). The guest them asks it to make changes using the "LOAD_GDT_ENTRY" hypercall.

```
/*
 * Notice that this method only LOADS the table to the host,
 * writing a specific  entry in the table is another method
 * called "lguest_write_gdt_entry()"
 */
static void lguest_load_gdt(const struct desc_ptr *desc)
{
    unsigned int i;
/*
 * "desc" parameter, that represents a pointer to a
 * descriptor table, is a struct with two fields.
 * The first is "address" the virtual address of the table
 * and the second is "size", the size of the table in bytes
 */
    struct desc_struct *gdt = (void *)desc->address;
/*
 * This is similar to the
 * LHCALL_LOAD_IDT_ENTRY hypercall we saw
 */
    for (i = 0; i < (desc->size+1)/8; i++)
        hcall(LHCALL_LOAD_GDT_ENTRY, i,
gdt[i].a, gdt[i].b, 0);
}
```

The "lguest_load_tls()" function is presenting the "lazy mode" which is an optimization for a sequence of hypercalls. There are three TLS (thread local storage) entries in the GDT and they are used to allow a multi-threaded application to make use of up to three segments containing data local to each thread (Threads share the same address space, so this is very helpful). Those entries change on every context switch, so we have a hypercall specifically for this case. This hypercall loads three GDT entries in one hypercall, so it saves two extra hypercalls (hypercalls involve context switch and therefore are time consuming operations).

```
static void lguest_load_tls(struct thread_struct *t,
unsigned int cpu)
{
    lazy_load_gs(0);
    lazy_hcall2(LHCALL_LOAD_TLS,
       __pa(&t->tls_array), cpu);
/* Notice the lazy_hcall() rather than hcall() */
}
```

"lazy_mode" is set when delivering hypercalls, that means we are allowed to defer all hypercalls and send them as a batch as lazy_mode. Because hypercalls are reasonably time consuming operations, batching them up so that only one context switch is involved makes sense. The "lazy_hcall1-4()" functions check if we are in "lazy mode", if we are not we call the regular "hcall()" then we use "async_hcall()" that would add the call to a buffer of calls that would be flushed at the next regular hypercall (usually would be the do-nothing LHCALL_FLUSH_ASYNC hypercall). The number on the "laze_hcall" suffix is for the number of arguments that the hypercall uses(1 to 4).

```
static void lazy_hcall1(unsigned long call,
unsigned long arg1)
{
    if (paravirt_get_lazy_mode() == PARAVIRT_LAZY_NONE)
hcall(call, arg1, 0, 0, 0);
```

```
    else
        async_hcall(call, arg1, 0, 0, 0);
}
/*
 * async_hcall() is pretty simple: We have a ring buffer
 * of stored hypercalls which the Host will run though
 * next time we do a normal hypercall.
 * Each entry in the ring has 5 slots for
 * the hypercall arguments, and a "hcall_status"
 * word which is 0 if the call is ready to go, and 255
 * once the Host has finished with it.
 *
 * If we come around to a slot which has not been
 * finished, then the table is full and we just make the
 * hypercall directly.  This has the nice side effect
 * of causing the host to run all the stored calls in the
 * ring buffer which empties it for next time!
 */

static void async_hcall(unsigned long call,
unsigned long arg1, unsigned long arg2,
unsigned long arg3, unsigned long arg4)
{
/* Note: This code assumes we are */
/*  using uni-processor.  */
    static unsigned int next_call;
    unsigned long flags;
/*
 * Disable interrupts if not already disabled:
 * we don’t want an interrupt handler
 * making a hypercall while we’re already doing one!
 */

   local_irq_save(flags);
/*  save the current flags */
   if (lguest_data.hcall_status[next_call] != 0xFF) {
/*
 * Table full, so do normal hcall which will flush table.
 */
/*
 * When the Host handles a hypercall he also handles all
 * the hypercalls that "lguest_data.hcalls" array store
 */
       hcall(call, arg1, arg2, arg3, arg4);
   } else {
       lguest_data.hcalls[next_call].arg0 = call;
       lguest_data.hcalls[next_call].arg1 = arg1;
       lguest_data.hcalls[next_call].arg2 = arg2;
       lguest_data.hcalls[next_call].arg3 = arg3;
       lguest_data.hcalls[next_call].arg4 = arg4;
/*
 * Arguments must all be written before we mark it
 * to go
 */
       wmb();
/*
 * This would generate an assembly instruction
 * guarantees that all previous store instructions
 * access memory before any store instructions
 * issued after the wmb instruction
 */
       lguest_data.hcall_status[next_call] = 0;
/*  mark this call as ready */
       if (++next_call == LHCALL_RING_SIZE)
         next_call = 0;
       local_irq_restore(flags);
/* restore the previous flags */
   }
}
```

Other CPU related operations are: reading and writing to the control registers c0, c2, c3, c4 because sometimes the host needs to know the changes. Furthermore the host does not allow the guest to use the CPUID instruction.

APPENDIX 6.5.4 Page Table Operations

The x86 architecture uses a three-level-paging, i.e in the first level, the "c3" control register points to a page directory at a size of 4kb (1000 entries of 4 bytes each). Each entry points to a second-level page table at a size of 4kb (1000 entries of 4 bytes each). Each entry (called PTE) points to the third-level which is the physical page frame. A virtual memory address would look like this

```
1 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 ...... 0 0 0 0
|<---- 10 bits --->|<---- 10 bits ---->|<--- 12 bits --->|
```

Adding the 12 bit of the offset to the page frame would give the physical address. This virtual memory method creates a 4Gb address space. It turns out that the x86 processors can support a larger address space of 64Gb (i.e 36 bits) using Physical Address Extension (PAE). The addresses are held in 8 byte table entries (to support the extended addresses and some more features, like security). In Lguest design we want the table size would still be a page size (4Kb) so this gives us 512 to table. This gives us half of the PTEs from the former method. Now, we add another, forth, level of a four entries to the first level and the "c3" control register will point to it. The result is a four-level-paging with the hierarchy of: Page-Upper-Directory(PUD)–> Page-Mid-Directory(PMD)–> PTE. A virtual address would look like that:

```
1 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0
|2>|<--- 9 bits ---->|<---- 9 bits --->|<--- 12 bits----->|
```

Linux only supports one method or the other depending on whether the CONFIG_X86_PAE is set. Many distributions turn it on, and not just for people with large amounts of memory; the larger PTE allow room for the NX bit, which lets the kernel disable execution of pages and increase security.

The kernel spends a lot of time changing both the top-level page directory and lower-level page table pages. The guest doesn't know physical addresses, and maintains these page tables like usual. The guest also needs to keep the host informed whenever it makes a change. The host will create the real page tables based on the guests' page table. The host page tables are called "shadow" page tables and are covered in section 6.8.6 The next three functions show the difference of handling addresses with PAE and without:

```
/*
 * The guest calls this after it has set a second-level
 * entry (pte), i.e to map  a page into a process'
 * address space.  We tell the Host the top-level and
 * address this corresponds to.
 * The guest uses one page table per process, so we
 * need to tell the Host which one we're changing
 * (mm->pgd).
 * This is used AFTER the entry has been set.
 * This function is responsible
 * only to inform the Host about the change using the
 * hypercall: LHCALL_SET_PTE
 */
static void lguest_pte_update(struct mm_struct *mm,
    unsigned long addr, pte_t *ptep)
/*
 * pte_t is a struct that represents a PTE, it has only
 * two fields of unsigned long: pte_low, pte_high
 */
{
/* make the LHCALL_SET_PTE hypercall to inform the Host
 * about changing the PTE
 */
#ifdef CONFIG_X86_PAE
/*
 * PAE needs to hand a 64 bit page table entry,
 * so it uses two args.
 */
    lazy_hcall4(LHCALL_SET_PTE, __pa(mm->pgd), addr,
        ptep->pte_low, ptep->pte_high);
#else
    lazy_hcall3(LHCALL_SET_PTE, __pa(mm->pgd), addr,
        ptep->pte_low);
#endif
}
/*
 * The guest calls: "lguest_set_pud()" to set a top-level
 * entry and "lguest_set_pmd()"
 * to set a middle-level entry, when PAE is activated.
 *
 * We set the entry then tell the Host which page we changed,
 * and the index of the entry we changed.
 */

#ifdef CONFIG_X86_PAE
static void lguest_set_pud(pud_t *pudp, pud_t pudval)
{
/*
 * put_t is an unsigned long. This action puts pudval
 * (the new entry value)
 * in the place pudp(the entry address)is pointing, i.e it
 * updates the relevant PUD entry
 */
    native_set_pud(pudp, pudval);
    lazy_hcall2(LHCALL_SET_PGD,__pa(pudp)&0xFFFFFFE0,
        (__pa(pudp) & 0x1F)/ sizeof(pud_t));
}

static void lguest_set_pmd(pmd_t *pmdp, pmd_t pmdval)
{
    native_set_pmd(pmdp, pmdval);
/*  #define PAGE_MASK (~(PAGE_SIZE-1))*/
    lazy_hcall2(LHCALL_SET_PMD, __pa(pmdp) & PAGE_MASK,
       (__pa(pmdp) & (PAGE_SIZE - 1)) / sizeof(pmd_t));
}
#else
/*
 * When we are NOT using PAE we need functions that set
 * the top level and second level entries.
 * The top level is handled the same as the middle-level.
 * The guest calls "lguest_set_pmd()" to set a top-level
 * entry when !PAE.
 */

static void lguest_set_pmd(pmd_t *pmdp, pmd_t pmdval)
{
    native_set_pmd(pmdp, pmdval);
    lazy_hcall2(LHCALL_SET_PGD, __pa(pmdp) & PAGE_MASK,
(__pa(pmdp) & (PAGE_SIZE - 1)) / sizeof(pmd_t));
}
#endif
```

There are cases that the kernel changes a PTE directly. This is what "lguest_set_pte()" does. This information of just the PTE has no use for the host because cannot know which page table the PTE changing relates to (each process has a page table). Therefore, the host is being ordered to clear all of his page tables, using the hypercall LHCALL_FLUSH_TLB (clearing the page tables is equivalent to a TLB flush on native hardware when a page entry is changed. Fortunately, these calls are very rare.)

```
static void lguest_set_pte(pte_t *ptep, pte_t pteval)
{
    native_set_pte(ptep, pteval); o
/*  Set the address ptep value as pteval */
    if (cr3_changed)
/*
 * we don't inform the Host if we are still in boot because
 * it makes the boot 8 times slower.
 */
        lazy_hcall1(LHCALL_FLUSH_TLB, 1);
}
```

APPENDIX 6.5.5 Time Related Operations

Another interesting interface type are time related interfaces. Lguest implements some of them. For example, at the "lguest_init()" function, previously introduced, we have this two settings: x86_platform.calibrate_tsc = lguest_tsc_khz; "x86_platform" is from type "struct x86_platform_ops"
(defined at /arc/x86/include/asm/x86_init.h), an interface of platform specific runtime functions:

```
struct x86_platform_ops {
    unsigned long (*calibrate_tsc)(void);
/* get CPU's frequency */
...
};
```

The preferred method to get the time is from the TSC register. The guest does not have access to it, so the Host gets its value from the processor and updates "lguest_data"

```
static unsigned long lguest\_tsc\_khz(void)
{
    return lguest_data.tsc_khz;
}
```

The problem is that TSC is not always stable. In this case the kernel falls back to a lower priority clock: "lguest_clock", where the guest read the time value given to us by the host. The host gets him time from the hardware and updates the "time" field in "lguest_data". This clock is implementing the "struct clocksource" interface which is a hardware abstraction for a free running counter (defined at /include/linux/clocksource.h):

```
static struct clocksource lguest_clock = {
    .name = "lguest" ,
/* clocksource name */
    .rating = 200,
/*
 * rating value for selection (higher is better).
 * 200 is a good rate(A correct and usable clocksource)
 * but is lower than TSC
 */
    .read = lguest_clock_read, /* returns a cycle value */
...
};

static cycle_t lguest_clock_read(struct clocksource *cs)
{
    unsigned long sec, nsec;
/*
 * Since the time is in two parts (seconds and
 * nanoseconds), * we risk reading it just
 * as it’s changing from 99 & 0.999999999
 * to 100 and 0, and getting 99 and 0.
 * As * Linux tends to come apart under
 * the stress of time travel, we must be
 * careful:
 */
    do {
/* First we read the seconds part. */

        sec = lguest_data.time.tv_sec;
        rmb();
/*
 * This read memory barrier,
 * we can't proceed before finishing this
 */
/* Now we read the nanoseconds part. */
        nsec = lguest_data.time.tv_nsec;
        rmb();
/* Now if the seconds part has changed, try again. */
    } while (unlikely(lguest_data.time.tv_sec != sec));
/* Our lguest clock is in real nanoseconds. */
    return sec*1000000000ULL + nsec;
}
```

Similar to the problem the guest has with the PIC, it doesn't has a hardware event device that can be controlled. i.e schedule time interrupts. The solution would be similar to a virtual timer chip that would be controlled by the host. The code below describes our primitive timer chip.

```
static struct clock_event_device lguest_clockevent = {
     .name = "lguest",
/* ptr to clock even name */
     .features = CLOCK_EVT_FEAT_ONESHOT,
/* support only single schedule, not periodic */
     .set_next_event = lguest_clockevent_set_next_event,
    .rating = INT_MAX,
/*  give highest rating to this event */ ...
};

static int lguest_clockevent_set_next_event
    (unsigned long delta, struct clock_event_device *evt)
{
    ...
/* Please wake us this far in the future. */
    hcall(LHCALL_SET_CLOCKEVENT, delta, 0, 0, 0);
    return 0;
}
```

APPENDIX 6.5.6 Patching

Patching is a unique technique used in Lguest. We have demonstrated how the pv_ops structure allows replacing simple native instructions with calls to the host, all throughout the kernel. This mechanism allows the same kernel to run as a guest and as a native kernel, but it slows the execution time because of all the indirect branches. A solution to that is patching. Patching is the system that allows the host to voluntarily patch over the indirect calls to replace them with something more efficient. We patch two of the simplest of the most commonly called functions: disable interrupts and save interrupts. We usually have 6 or 10 bytes to patch into, but the guest versions of those operations are small enough that we can fit comfortably. First we need assembly templates of each of the patch-able guest operations, and these are in i386_head.S.

```
/*
 * We create a macro which puts the assembler code between
 * lgstart_ and lgend_ markers.
 * These templates are put in the .text section:
 * they can't be discarded after boot as we
 * may need to patch modules, too.
 */
.text
/* This appends the name lgstart/lgend to argument */
#define LGUEST_PATCH(name, insns...) \
     lgstart_##name: insns; lgend_##name:;
     \ .globl lgstart_##name; .globl lgend_##name

LGUEST_PATCH(cli,movl $0,
lguest_data+LGUEST_DATA_irq_enabled)
/*disable interrupts*/
LGUEST_PATCH(pushf,movl
lguest_data+LGUEST_DATA_irq_enabled, %eax)
/*  save interrupts */
```

Next we will construct a table from the assembler templates [boot.c] and implement the patch routine:

```
/*
 * This struct holds the start and end
 * addresses of the patching functions
 */
static const struct lguest_insns {
    const char *start, *end;
}
lguest_insns[] = {
/*
 * PARAVIRT_PATCH returns the index (by bytes) of the
 * patched functions on * the "paravirt_patch_template"
 * struct, that is an interface of all the patchable
 * functions (Actually it contains all the paravirt
 * structures, like pv_irq_ops, pv_cpu_ops).
 * So in order get the specific function index we
 * use: pv_irq_ops.
 */
    [PARAVIRT_PATCH(pv_irq_ops.irq_disable)] =
        { lgstart_cli, lgend_cli },
    [PARAVIRT_PATCH(pv_irq_ops.save_fl)] =
        { lgstart_pushf, lgend_pushf },
};
/*
 * Now, our patch routine is fairly simple.
 * If we have a replacement, we copy it in and
 * return how much of he available space we used.
 */

static unsigned lguest_patch(u8 type, u16 clobber,
    void *ibuf, unsigned long addr, unsigned len)
{
    unsigned int insn_len;
    ...
/*
 * type is the patched function index on
 * "paravirt_patch_template"
 */
    insn_len = lguest_insns[type].end -
        lguest_insns[type].start;
    ...
/* Copy in our instructions. */
    memcpy(ibuf, lguest_insns[type].start, insn_len);
    return insn_len;
}
```

We covered all interesting "paravirt_ops" operations. The rest of the "lguest_init" are boot configurations. When the boot configuration is finished, a call is made do "i386_start_kernel" (in head32.c), and proceed to boot as normal kernel would.

APPENDIX 6.6 Drivers

APPENDIX 6.6.1 Virt I/O

Virt I/O is an abstraction layer over devices in a para-virtualized hypervisor (Virt I/O was developed by Rusty Russell in support to lguest). Having different hypervisor solutions with different attributes and advantages (KVM, lguest, User-mode Linux), tax the operating system because of their independent needs. One of the taxes is the virtualization of devices. Rather than have a variety of device emulation mechanisms for each device and each hypervisor, virt I/O provides a common front end (a common term for the guest side) for these device emulations to standardize the interface and increase the reuse of code across the platforms. Using an identical virt I/O driver cross multiple hypervisor is demonstrated in

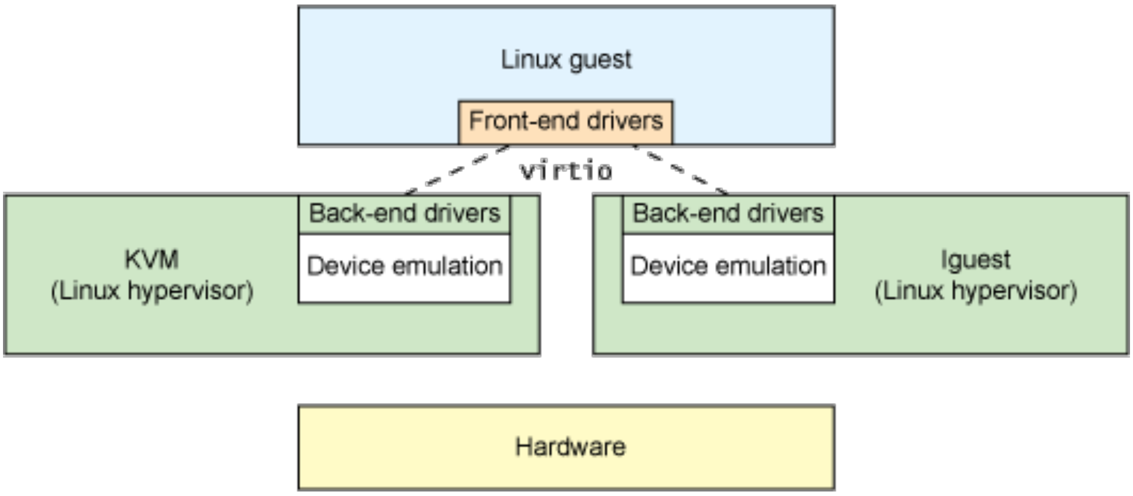


FIGURE 76 Using Virt I/O in multiple hypervisors

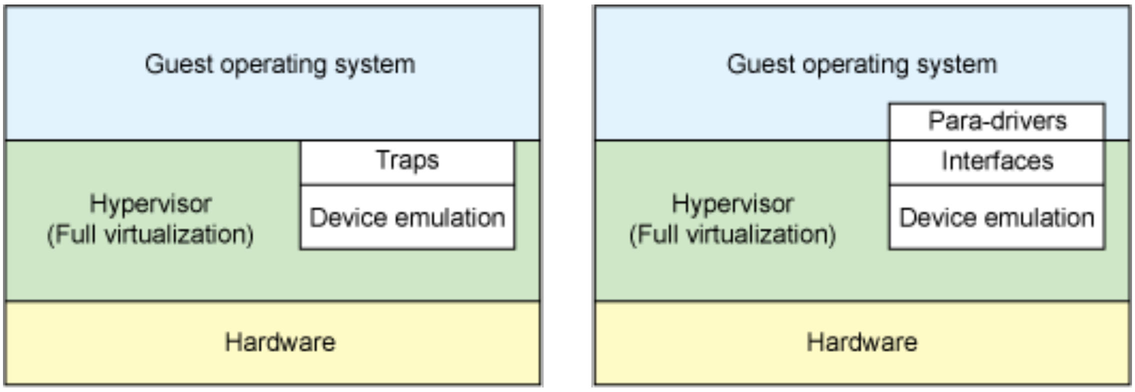


FIGURE 77 Para-virtualization device drivers

figure 76.

In para-virtualization, the guest operating system is aware that it is running on a hypervisor and includes the front end drivers, i.e drivers that know they are dealing with virtual devices. The hypervisor is the one that needs to emulate the "real" devices so it implements the back-end drivers for the particular device emulation (particularly because each hypervisor emulates differently). This is demonstrated in figure 77

These front-end and back-end drivers are where virt I/O comes in. Virt I/O provides a standardized interface for the development of emulated device access to propagate code reuse and increase efficiency. Virt I/O is an abstraction for a set of common emulated devices.

This design allows the hypervisor to export a common set of emulated devices and make them available through an API. The guests implement a common set of interfaces, with the particular device emulation behind a set of back-end drivers. The back-end drivers need not be common as long as they implement the required behaviors of the front end. Thus, the front end drives would be common, allowing the integration to Linux and the back end drives would be implemented by the different hypervisors.

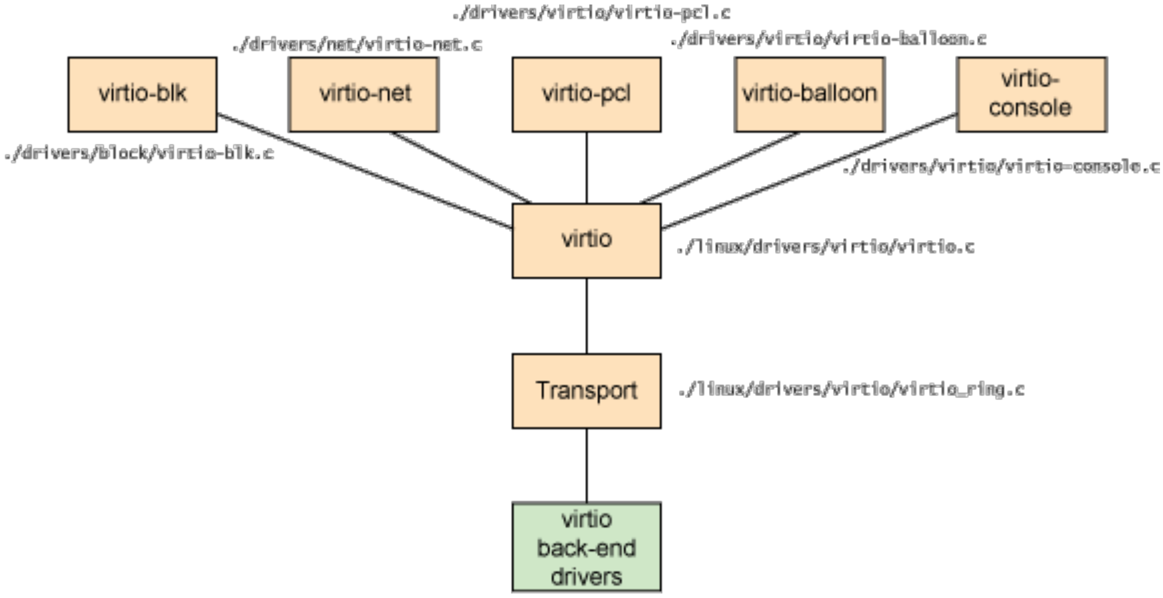
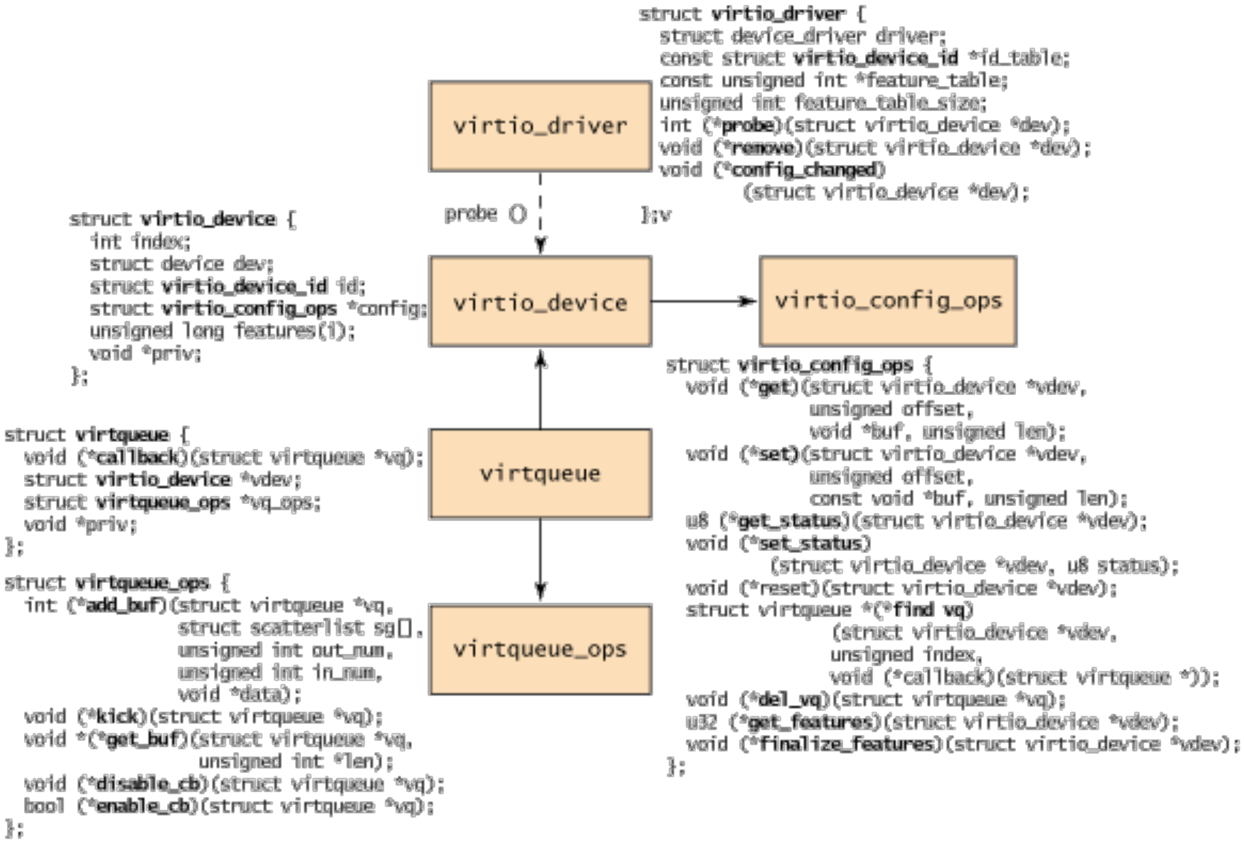

./drivers/virtio/virtio-pcl.c
./drivers/net/virtio-net.c
./drivers/virtio/virtio-balloon.c
virtio-blk
virtio-net
virtio-pcl
virtio-balloon
virtio-console
./drivers/block/virtio-blk.c
./drivers/virtio/virtio-console.c
virtio
./linux/drivers/virtio/virtio.c
Transport
./linux/drivers/virtio/virtio_ring.c
virtio back-end drivers

FIGURE 78 5 Virt I/O drivers

In addition to the front-end drivers (implemented in the guest operating system) and the back-end drivers (implemented in the hypervisor), virt I/O defines two layers called "virtqueues" to support guest-to-hypervisor communication. At the first layer (called "virt I/O") is the virtual queue interface that conceptually attaches front-end drivers to back-end drivers, it defines how they should "speak" to each other via these virtual queues. Drivers can use zero or more queues, depending on their need. For example, the virt I/O network driver uses two virtual queues (one for receiving and one for transmitting), where the virt I/O block driver uses only one. The second layer is the transport interface, i.e what is expected from the implementation of the virtqueues. In Lguest the transport is implemented as rings (called "vrings"). Notice that the virtual queues could be implemented any implementation, as long as both the guest and hypervisor implement it in the same way. Figure 78 demonstrate five front-end drivers listed: block devices (such as disks), network devices, PCI emulation, a balloon driver (for dynamically managing guest memory usage), and a console driver. Figure 79 demonstrate Virt I/O/O driver internal structure. Each front-end driver has a corresponding back-end driver in the hypervisor and each have individual queues. The front-end drivers are the same for all guests. The back end driver is hypervisor specific. As the figure demonstrate the front end drivers approach the back end through "virt I/O" and the transport is responsible moving the data the the other side.

Virt I/O is now a increasing in popularity and invested effort and it is supported by many companies in the industry (RedHat, IBM, Citrix). Virt I/O is described in greater detail in [Tim].



```
struct virtio_driver {
  struct device_driver driver;
  const struct virtio_device_id *id_table;
  const unsigned int *feature_table;
  unsigned int feature_table_size;
  int (*probe)(struct virtio_device *dev);
  void (*remove)(struct virtio_device *dev);
  void (*config_changed)
          (struct virtio_device *dev);
};v
```

```
struct virtio_device {
  int index;
  struct device dev;
  struct virtio_device_id id;
  struct virtio_config_ops *config;
  unsigned long features(1);
  void *priv;
};
```

```
struct virtqueue {
  void (*callback)(struct virtqueue *vq);
  struct virtio_device *vdev;
  struct virtqueue_ops *vq_ops;
  void *priv;
};
```

```
struct virtqueue_ops {
  int (*add_buf)(struct virtqueue *vq,
                 struct scatterlist sg[],
                 unsigned int out_num,
                 unsigned int in_num,
                 void *data);
  void (*kick)(struct virtqueue *vq);
  void *(*get_buf)(struct virtqueue *vq,
                   unsigned int *len);
  void (*disable_cb)(struct virtqueue *vq);
  bool (*enable_cb)(struct virtqueue *vq);
};
```

```
struct virtio_config_ops {
  void (*get)(struct virtio_device *vdev,
              unsigned offset,
              void *buf, unsigned len);
  void (*set)(struct virtio_device *vdev,
              unsigned offset,
              const void *buf, unsigned len);
  u8 (*get_status)(struct virtio_device *vdev);
  void (*set_status)
          (struct virtio_device *vdev, u8 status);
  void (*reset)(struct virtio_device *vdev);
  struct virtqueue *(*find vq)
                (struct virtio_device *vdev,
                unsigned index,
                void (*callback)(struct virtqueue *));
  void (*del_vq)(struct virtqueue *vq);
  u32 (*get_features)(struct virtio_device *vdev);
  void (*finalize_features)(struct virtio_device *vdev);
};
```

FIGURE 79 Virt I/O driver anatomy

APPENDIX 6.6.2 Adding a Device on Lguest

The guest needs devices to do anything useful. Since the guest cannot access the physical (real) hardware devices. The host could emulate a PCI bus with various devices on it (perhaps using QEMU[Bel05]), but that is a fairly complex burden for the host and sub optimal for the guest, so we have our own simple lguest bus emulation and we use "virt I/O" front-end drivers. Controlling the virtual devices is through an I/O memory region. I/O memory region is used by a device driver to control a device and to pass information between them. Our devices are described by a simplified ID, a status byte, and some "config" bytes (describe the device's configuration), using "lguest_device_desc" structure (in [/include/linux/lguest_device.h].

The I/O memory region is physically placed by the launcher just above the top of the guest's physical memory. The idea is that "lguest_device_desc" structures sit there and the guest can control the devices by these structures.

```
struct lguest_device_desc {
/*
 * The device type: console, network, disk etc.
 * Type 0 terminates.
 */
    __u8 type;
/*
 * The number of virtqueues (first in config array)
 */
    __u8 num_vq;
/*
 * The number of bytes of feature bits. Multiply by 2:
 * one for host features and one for guest acknowledgment.
 */
    __u8 feature_len;
/*
 * The number of bytes of the config array
 * after virtqueues.
 */
    __u8 config_len;
/*
 * A status byte, written by the guest.
 */
    __u8 status;
    __u8 config[0];
};
```

Fairly early in the guest's boot, "lguest_devices_init()" is called to set up the lguest device infrastructure. The guest wants to access the I/O memory region page easily (we currently have the physical address that the Launcher placed this page). For this end, the host map that page frame (as said placed above the guest's physical memory) and store the pointer with a virtual address in "lguest_devices". The mapping is done using the "lguest_map()" function that actually envelopes "ioremap()" and returns a magic cookie (32 bit address that is not guaranteed to be usable directly as a virtual address) that could be passed to accessor functions (with names like readb() or writel()) to actually move data to or from the I/O memory. But on x86 architectures I/O memory is mapped into the kernel's memory space, so accessing the I/O memory is a straightforward pointer dereference, and this pointer is "lguest_devices". Finally "lguest_devices_init()" calls "scan_devices()" that for each device descriptor it finds in the descriptors page it calls using "add_lguest_device()".

```
static
int __init lguest_devices_init(void)
{
    ...
/* Devices are in a single page above top of "normal" mem */
    lguest_devices = lguest_map(max_pfn<<PAGE_SHIFT, 1);
    scan_devices();
    return 0;
}
```

The function "add_lguest_device()" is the core of the lguest bus emulation. It's purpose is to add a new device. Each lguest device is just a virtio device in other words a device that knows how to talk with virtqueues, and a pointer to its entry in the "lguest_devices" page.

```
struct lguest_device {
    struct virtio_device vdev;
/*
 * The entry in the lguest_devices page for this device.
 */
    struct lguest_device_desc *desc;
};
```

First, a "lguest_device" is allocated. Then the "lguest_device" is assigned "virtio_device" (vdev) and it's contained set of routines for querying the device's configuration information and setting its status ("lguest_config_ops"). This process demonstrate why "virtio" is good design: the guest controls the device with these routines, but we supply the implementations to these routines according to our device implementation. This mechanism allows a uniform front-end access to devices.

The most important part in this function is the call to "register_virtio_device()" in [/drivers/virtio/virtio.c] This is virtio code, not lguest code. Let us be reminded that the initial objective was to create a virtual device. The "virtio_device" is not a device that Linux "knows" it is simply a linking layer between the guest and the hypervisor. Therefore, each "virtio_device" struct holds a field of "struct device" that is Linux's representation for a device. "register_virtio_device()" gives this "struct device" a unique name, and causes the bus infrastructure to look for a matching driver. By complete ding this phase the device is now registered with the system. In addition, the function also reset the virtio_device and acknowledges it i.e sets the relevant bit at the status byte of the device (both using a routine we supplied with "lguest_config_ops").

```
static void add_lguest_device(struct lguest_device_desc *d
, unsigned int offset)
{
    struct lguest_device *ldev;
    ldev = kzalloc(sizeof(*ldev), GFP_KERNEL); ...
/*
 * We have a simple set of routines for querying the
 * device's * configuration information and setting
 * its status.
 */
    ldev->vdev.config = &lguest_config_ops; ...
/*
 * register_virtio_device() sets up the generic fields
 * for the struct virtio_device and calls
 * device_register().
 * This makes * the bus infrastructure look for a
 * matching driver.
 */

    if (register_virtio_device(&ldev->vdev) != 0) {
        printk(KERN_ERR "Failed to register lguest dev %u
    type %u\n", offset, d->type);
        kfree(ldev);
    }
}
```

APPENDIX 6.6.3 Device configuration

After adding the device, the next step in using the device is performing the device configuration. The configuration information for a device consists of one or more virtqueue descriptors, a feature bitmap, and some configuration bytes. A virtqueue file descriptor consists of place, size, etc, and will be described in section 6.6.4. The configuration bytes are set by the laucnher and the driver will look at them during setup. All this configuration information is located at the descriptors page frame (located after each device descriptor) and can be retrieved using the functions (located in [lguest_device.c]): "lg_vq()", "lg_features()", "lg_config()". The memory layout looks like the below (here, device 1 has two virtqueues and device 2 has one virtqueue) :

```
first page frame ----------------------------
                 | Guest Physical Memory    |
max page frame   ----------------------------
                 | device descriptor        |
                 | virtqueue 1 descriptor   |
       device 1  | virtqueue 2 descriptor   |
                 | features bitmap          |
                 | config bytes             |
                 ---------------------------- I/O MEMORY
                 | device descriptor        | REGION PAGE
       device 2  | virtqueue 1 descriptor   |
                 | features bitmap          |
                 | config bytes             |
                 ----------------------------
                 |...                       |
max page frame+1 ----------------------------
```

The device can be configured by the guest using "lguest_config_ops". This struct serves as the front end for virt I/O configuration.

```
struct virtio_config_ops {
    void (*get)(struct virtio_device *vdev, unsigned offset,
        void *buf, unsigned len);
    void (*set)(struct virtio_device *vdev, unsigned offset,
        const void *buf, unsigned len);
    u8 (*get_status)(struct virtio_device *vdev);
    void (*set_status)(struct virtio_device *vdev,
u8 status);
    void (*reset)(struct virtio_device *vdev);
    int (*find_vqs)(struct virtio_device *,
        unsigned nvqs, struct virtqueue *vqs[],
       vq_callback_t *callbacks[], const char *names[]);
    void (*del_vqs)(struct virtio_device *);
    u32 (*get_features)(struct virtio_device *vdev);
    void (*finalize_features)(struct virtio_device *vdev);
};
```

The configuration process starts with configuring the devices features. The guest takes the features bitmap the host offers using "lg_get_features()", and copies the driver supported features into the "virtio_device" features array (it has a this field for this purpose). Therefore, as far as the front-end devices specific features are configured. Once the device features are all sorted out the guest calls "lg_finalize_features()". This features allows the guest to infrom the host which of the device features it understands and accepts. This concludes the configuration of the device features. Setting the configuration bytes is achieved using the "lg_get()" and "lg_set()" routines. Controlling the device status is possible using "lg_get_status()" and "lg_set_status()".

APPENDIX 6.6.4 Virtqueues

Virtqueues are the messaging infrastructure that virtio needs: Virtqueues provide means for the guest device driver to registers buffers for the host to read from or write into (i.e send and receive buffers). Each device can have multiple virtqueues: for example the console driver uses one queue for sending and another for receiving.

Fast shared-memory-plus-descriptors virtqueue is already implemented in the Linux kernel and are called *"virtio ring"* or *"vring"* (aptly named because it is implemented as a ring buffer). Vring implementation can be found under *drivers/virtio/virtio_ring.c*. An extensive explanation about virt I/O can be found in Rusty's paper [Rus08].

The Launcher decides the exact location in the memory for the vring virtqueue (they are always placed in pages above the device descriptor page) and the size of the virtqueues. When a device is added "lg_find_vqs()" is called (it is part of the "lguest_config_ops") and it uses "lg_find_vq()" to find the virtqueues that the launcher had placed for the device to read the virtqueues sizes.

Now that the guest's kernel knows the size and the address of the virtqueues it tells [virtio_ring.c] with the function "vring_new_virtqueue()" to set up a new vring at this location and size. After this phase the virtqueue is set and the host and guest can communicate. A guest launcher of a 3rdparty hypervisor can pick any implementation of virtqueues but lguest uses the vring implementation.

When the "virtio_ring" code wants to notify the host about any event, for example, he put information in the buffer that the host needs, it calls "lg_notify()" that makes a hypercall (LHCALL_NOTIFY). It needs to hand the physical address of the virtqueue so the host knows which virtqueue the guest is using.

APPENDIX 6.6.5 Virt I/O summary

Virt I/O(virtio) is an abstraction layer that separates the device and driver implementation between the guest and host. Using virtqueues the device driver interface would be uniform regardless of the device functionality. (Since the implementation of virtqueues in the hypervisor, virtqueues are hypervisor dependent.)

When the guest process starts with the launcher, it places lguest device descriptors, virtqueue descriptors and configuration, at the page above the guest's memory. The launcher also places the virtqueues at the pages above it. Upon the guest startup the guest knows where his top page frame is, using the routine "add_lguest_device()". The routine "add_lguest_device()" is compiled to the boot code to create the virtio device infrastructure for the guest: the devices and the virtqueues.

Using virtqueues infrastructure the guest does nothing to adjust itself to lguest or any other hypervisor in any way: The I/O memory region was created by the launcher via the routine "add_lguest_device()" that was compiled to it. The launcher finds the devices, configures them using configuration routines (like "lg_find_vqs()") that are lguest implementations to a guest interface and created the virtqueue (vring).

The virtio driver control the device using the configuration memory region (mostly using "lg_set_status()") and moving data to the back end using the virtqueues, all done via the virtio infrastructure. The hypervisor reads the status changes and uses his privilege to control the real device, and also send/receive data to the real device. (In order that different hypervisors can translate request differently and perhaps improve efficiency but the guest would act the same regardless of hypervisor.)

APPENDIX 6.7 The Launcher

The launcher is the host user space program that sets up and runs the guest. To the host Lguest kernel module, the launcher *is* the guest.

APPENDIX 6.7.1 Setting and Running the guest Using "/dev/lguest"

The interface between the host kernel and the guest launcher is a character device typically called /dev/lguest that is created by the host kernel. The launcher is reading this device to run the guest, and writing it, to set up the guest and serve it. The code responsible for this is located at [drivers/lguest/lguest_user.c]. All the work performed by the kernel module is triggered by the read(), write() and close() routines:

```
static const struct file_operations lguest_fops = {
.owner = THIS_MODULE,
.release = close,
.write = write,
.read= read,
};

static struct misc device lguest_dev = {
.minor = MISC_DYNAMIC_MINOR,
.name = "lguest",
.fops = &lguest_fops,
};
```

/dev/lguest is a misc char device. All misc char devices share the major number 10 and differentiate themselves based on the minor number. It is registered using "lguest_device_init" which calls "misc_register".

```
int __init lguest_device_init(void) {
return misc_register(&lguest_dev);
}
```

The "lguest_fops" operations are performed on a structure called "struct lguest". The "struct lguest" contains information maintained by the host about the guest.

```
struct lguest {
struct lguest_data lguest_data;
struct lg_cpu cpus[NR_CPUS];
u32 pfn_limit;
/*
 * This provides the offset to the base of
 * guest-physical memory in the Launcher.
 */
void *mem_base;
unsigned long kernel_address;
struct pgdir pgdirs[4];
...
struct lg_eventfd_map eventfds;
const char *dead;
}
```

The key fields in struct lguest are

lguest_data the guest specific data saved on the host

lg_cpu virtual cpu specific information

mem_base, kernel address, pg_dirs memory information for the guest

lg_eventfd_map file descriptors for virtqueues.

The /dev/lguest file structure ("struct file") points to the "struct lguest" using the field "private_data" pointer. When the launcher preforms a read/write he needs to give as an argument, the "struct file", so accessing "struct lguest" is just dereferencing the "private_data" pointer field.

We will examine how the struct lguest data is updated using I/O operations.

```
static ssize_t write(struct file *file,
    const char __user *in, size_t size, loff_t *off)
{
/*
 * Once the guest is initialized, we hold
 * the "struct lguest" in the file private data.
 */

    struct lguest *lg = file->private_data;
    const unsigned long __user *input =
        (const unsigned long __user *)in;
    long req;
    struct lg_cpu *uninitialized_var(cpu);
/* The first value tells us what this request is. */
    if (get_user(req, input) != 0)  return -EFAULT;
    input++;
...
    switch (req) {
    case LHREQ_INITIALIZE:
        return initialize(file, input);
    case LHREQ_IRQ:
        return user_send_irq(cpu, input);
    case LHREQ_EVENTFD:
        return attach_eventfd(lg, input);
  default:
    return -EINVAL;
    }
}
```

The launcher sets up the guest using a write with the LHREQ_INITIALIZE header that calls the "initialize()" function. The Launcher supplies 3 pointer sized values (in addition to the header):

TABLE 7 write LHREQ_INITALIZE pointers

pointer	Use
Base	The start of the guest physical memory inside the launcher memory.
pfnlimit	The highest physical page frame number the guest should be allowed to access. The guest memory lives inside the Launcher, so it sets this to ensure the guest can only reach its own memory.
start	The first instruction to execute (that value of "eip" register in x86)

Calling write with LHREQ_INITALIZE allocates "struct lguest" and initiates it with the function's arguments. It also allocates and initializes the guest's shadow page table. Finally, it sets the field "private data" of the file structure (the first argument of "write()") to point "struct lguest".

One interesting field of "struct lguest" is "cpu", something like a virtual CPU. The "cpu" filed of struct lguest contains the CPU id (in our case always 0, could be changed when there is a full support at SMP) and the "regs" field that holds the CPU registers. When "initialize()" is called, "initialize" calls "lg_cpu_start()" which then allocates a zeroed page for storing all the guest's CPU registers so that when there is a switch from the host to the guest the hardware CPU will update it's internal registers with the value of this page. (This is covered extensively in section 6.9). Then it calls "lguest_arch_setup_regs()" (located in [/drivers/lguest/core.c]) that initiates the registers.

```
/*
 * There are four "segment" registers which the guest
 * needs to boot: The "code % segment" register (cs)
 * refers to the kernel code segment __KERNEL_CS, and
 * the * "data", "extra" and "stack" segment registers
 * refer to the kernel data segment  __KERNEL_DS.
 * The privilege level is packed into the lower bits.
 * The guest runs at privilege level 1 (GUEST_PL).
 */

    regs->ds = regs->es = regs->ss =
        __KERNEL_DS|GUEST_PL;
    regs->cs = __KERNEL_CS|GUEST_PL;
/*
 * The "eflags" register contains miscellaneous flags.
 * Bit 1 (0x002) is supposed to always be "1".
 * Bit 9 (0x200) controls whether interrupts are
 * enabled.
 * We always leave interrupts enabled while
 * running the guest.
 */

    regs->eflags = X86_EFLAGS_IF | 0x2;
/*
 * The "Extended Instruction Pointer" register says where
 * the guest is running.
 */
    regs->eip = start;
```

Once our guest is initialized, the launcher enables it to run by reading from /dev/lguest. The "read" routine is very simple and includes some checking like that a write of LHREQ_INITIALIZE was already done, and that the guest is not dead (using the "dead" field of the "cpu" struct). The most function called by the initialization process is "run_guest()" [drivers/lguest/core.c]. "run_lguest()" runs the guest until something interesting happens such as a signal from the launcher (an interrupt has received) or that the guest wants to notify something to the host (so the guest must stop in order that the host would receive the message). We would return to to "run_guest()" in section 6.8. Every time that the hypervisor returns the control to the launcher, the launcher would run the guest using this read.

The second write operation to /dev/lguest is interrupt related. The launcher is responsible for managing the guest's I/O, in other words managing its virtqueues. When the Launcher have an interrupt that it needs to send to the guest (for example the Launcher received a network packet) the launcher preforms a write to the LHREQ_IRQ header and an interrupt number to /dev/lguest. This write would command the host to send this interrupt to the guest.

The last reason for writing to /dev/lguest is related to guest notifications. The launcher is responsible for handling virtqueues. When the guest does some change in a virtqueue, the launcher needs to know about it. The guest can only communicate with the host (via hypercalls). The solution is that each virtqueue is attached to an event file descriptor (an event fd is useful when we what to poll(2) for some event). This attachment would be a mapping of the virtqueue's physical address to the event fd. The mapped file descriptor would be stored in the "eventsfd" field in "struct lguest". When the guest makes something that the launcher needs to be informed about it would make the "LHCALL_NOTIFY" hypercall with a parameter of the physical address of the virtqueue we want to notify. The host finds the respective event fd and signal it. Now the virtqueue know he has information to pass to the host. Notice that each virtqueue has a launcher thread that listens to this event fd.

The launcher add an event fd using a write with the LHREQ_EVENTFD header and two arguments: the physical address of the virtqueue we want to notify and an event fd. This write calls "attach_eventfd()" that adds this specific mapping to the existing map. At this time the launcher does not block readers from accessing the map when we inserting the new event fd. Because this process takes place currently only during boot, Blocking is not really a problem. Nevertheless, Rusty implemented a solution using Read-Copy-Update (RCU) synchronization mechanism the Linux kernel provides.

The final piece of the "lguest_fops" interface code is the "close()" routine. It reverses everything done in "initialize()". This function is usually called because the launcher exited.

APPENDIX 6.7.2 The Launcher Code

The launcher code is executed in userspace.

The launcher malloc a big chunk of memory to be the guest's "physical" memory and stores it in "guest_base". In other words, guest "physical" == part of the launcher process virtual memory.

Like all userspace programs, the launcher process starts with the "main()" routine. The launcher "main()" function does the following:

- Figure from the the command line arguments, the amount of memory to allocate and allocate this number of pages and an extra of 256 pages for devices (One page is for the device page as referred at the previous chapter and the rest are needed for the virtqueues). All the allocated pages are zeroed.
- Setup command line devices: network device, block device and random number generator device. This code is very boring and wouldn't be presented here. It contains the setup of: the network device using a TUP device and two virtqueues of sending and receiving; the virtual block device that is implemented using reading and writing to a file with a single virtqueue for scheduling requests; and the random number generator that have a single virtqueue transferring chars from " /dev/random".
- Setup a console device. This is always done, regardless of command line arguments. This code contains the console setup with two virtqueues for input and output.
- Load the kernel image (supports both ELF file and bzImage). Map the initrd image if requested (at top of physical memory) and set the boot header (a "struct boot_params" that start at memory address number 0): "E820" memory map (in Lguest case it is simple - just one region for all the memory but it can be complex in other hypervisors), the boot protocol version (2.07 supports the fields for Lguest), the "hardware_subarch" value (set to 1: remember the start of section 6.5).
- Initialize the guest

Algorithm 16 Initialize the guest

```
open /dev/lguest
write to the new file descriptor with the LHREQ_INITIALIZE header {(see
  "initialize()" section 6.7.1).}
```

- Run the guest using "run_guest()". This is not the same "run_guest" function described earlier that is called when reading from /dev/lguest. This is a userspace function (that never returns) that is responsible to handling input and output to the host, while the other "run_guest" is kernel space (host code) that actually runs the guest. The launcher "run_guest" function runs the userspace representation of the guest, serves its input and output, and finally, lays it to rest.

```
static void __attribute__((noreturn)) run_guest(void)
{
    for (;;) {
        unsigned long notify_addr;
        int readval;
/* We read from the /dev/lguest device to run the guest. */
        readval = pread(lguest_fd, &notify_addr,
            sizeof(notify_addr), cpu_id);
/* One unsigned long means the guest did HCALL_NOTIFY */
        if (readval == sizeof(notify_addr)) {
            verbose("Notify on address %#lx\n", notify_addr);
            handle_output(notify_addr);
        }
/* Anything else means a bug or incompatible change. */
 else err(1, "Running guest failed");
    }
}
```

The "read" operation start the kernel space "run_guest()" (in [drivers/lguest/lguest_user.c]). This function runs the guest, but the running could be stopped for several reasons. One of the reasons is a notification of the guest to the host (using LHCALL_NOTIFY). In this case, if the notification address is mapped to an event fd (see section 6.7.1) so the event fd is signaled (using "eventfd_signal()"
in [fs/eventfd.c]) we stay in the host in order to run the guest again. If not, the host returns to the launcher with the "sizeof (unsigned long)" and handle the notification with "handle_output()". An example of the use of notifications is the launcher handling (using "handle_output()") notifications to device descriptors – "the guest updated the device status" (for instance during boot time the guest set the DEVICE_OK status bit which means that the device is ready to use).

APPENDIX 6.8 The Host

We will now describe the host code that handles the guest.

APPENDIX 6.8.1 Initialization

Examining Lguest init function we see

```
static int __init init(void)
{
    int err;
/* Lguest can't run under Xen, VMI or itself.  */
    if (paravirt_enabled()) {
        printk("lguest is afraid of being a guest\n");
        return -EPERM;
    }
/*
 * First we put the Switcher up in very high
 * virtual memory.
 */
    err = map_switcher();
    ...
/* /dev/lguest needs to be registered. */
    err = lguest_device_init();
    ...
/* Finally we do some architecture-specific setup. */
    lguest_arch_host_init();
/* All good! */
    return 0;
}
```

The switcher is a few hundred bytes of assembler code which actually changes the CPU registers (perform context switch) to run the guest, and then changes back to the host when a trap or interrupt occurs, it is the heart of the virtualization process. The Switcher's code ("switcher.S") is already compiled somewhere in the memory. The Switcher code (currently one page) and two pages per CPU (information needed for the switch - see section 6.9) must be at the same virtual address in the guest as the host since it will be running as the switchover occurs. We need to decide where in the virtual memory that those pages would be placed both in the host and the guest. It turns out that if we place the switcher pages at a high virtual address it will be easier to set the guest's page tables to map this address to the physical location of the Switcher pages. This address is 0xFFC00000 (4MB under the top virtual address.

```
static __init int map_switcher(void)
{
    int i, err;
    struct page **pagep;
/*
 * Map the Switcher's code in to high memory.
 */
...
/*
 * Now we reserve the "virtual memory area" we want:
 * 0xFFC00000 (SWITCHER_ADDR). We might not get it in
 * theory, * but in practice it's worked so far.
 * The end address needs +1 because
 * __get_vm_area allocates an extra guard page,
 * so we need space for that.
 */
    switcher_vma = __get_vm_area(TOTAL_SWITCHER_PAGES *
PAGE_SIZE, VM_ALLOC, SWITCHER_ADDR, SWITCHER_ADDR
         + (TO-TAL_SWITCHER_PAGES+1) * PAGE_SIZE);
/*
 * This code actually sets up the pages we've allocated
 * to appear at SWITCHER_ADDR.  map_vm_area() takes the
 * vma we allocated above, the kind of pages we're
 * mapping (kernel pages), and a pointer to our array of
 * struct pages ("switcher_pages"). It increments that
 * pointer, but we don't care.
 */

    pagep = switcher_page;
    err = map_vm_area(switcher_vma,
PA-GE_KERNEL_EXEC, &pagep);
...
/*
 * Now the Switcher is mapped at the right address.
 * Copy to this virtual memory area
 * the compiled-in Switcher code (from <arch>_switcher.S).
 */
...
    memcpy(switcher_vma->addr, start_switcher_text,
        end_switcher_text - start_switcher_text);
    printk(KERN_INFO "lguest: mapped switcher at %p\n",
         switcher_vma->addr);
/* And we succeeded... */
   return 0;
...
}
```

After the switcher is mapped and the "/dev/lguest" device is registered (using "lguest_device_init()"), the "init()" function needs to do some more i386-specific initialization using "lguest_arch_host_init()". First, it sets up the Switcher's per-cpu areas. Each CPU gets two pages of its own within the high-mapped region(struct lguest_pages). These pages are shared with the guests. Much of this is initialized by "lguest_arch_host_init()", but some depends on what guest we are running (which is set up in "copy_in_guest_info()".

APPENDIX 6.8.2 Chapter:Lguest:Section:IDT

). These pages contain information that the Host needs when there is a switch back to him like: the address and size of the host GDT, and the address of the host IDT, and also information the the guest needs when it is running such as the address and size of the guest GDT, the address of the guest IDT and the guest TSS (task state segment).

Finally, "lguest_arch_host_init()" turn off "Page Global Enable". PGE is an optimization where page table entries are specially marked to show they never change. The host kernel marks all the kernel pages this way because it is always present, even when userspace is running. Lguest breaks this assumption: without the rest of the host kernel knowing, we switch to the guest kernel. If this is not disabled on all CPUs the result will be weird bugs.

APPENDIX 6.8.3 Running the Guest

The main guest "run guest" run loop is called in the kernel when the launcher reads the file descriptor for /dev/lguest.

```
int run_guest(struct lg_cpu *cpu, unsigned long __user *user)
{
/* We stop running once the Guest is dead. */
    while (!cpu->lg->dead) {
       unsigned int irq; bool more;
/* First we run any hypercalls the Guest wants done. */
       if (cpu->hcall) do_hypercalls(cpu);
    }
/*
 * It's possible the Guest did a NOTIFY hypercall
 * to the Launcher.
 * pending notify is the notification address.
 * If there's a mapping for it to an event fd,
 * write to the fd, else return with
 * "sizeof(cpu->pending_notify)" so the Launcher
 * would handle it (using "handle_output()").
 */

    if (cpu->pending_notify) {
/*
 * Does it just needs to write to a registered
 * eventfd (ie. the appropriate virtqueue thread)?
 */
        if (!send_notify_to_eventfd(cpu)) {
           if (put_user(cpu->pending_notify, user))
return -EFAULT;
           return sizeof(cpu->pending_notify);
        }
    }
/* Check for signals */

    if (signal_pending(current)) return -ERESTARTSYS;
/*
 * Check if there are any interrupts which can
 * be delivered now: if so, this sets up the
 * handler to be executed when we next
 * run the Guest.
 */

    irq = interrupt_pending(cpu, &more);
    if (irq < LGUEST_IRQS)
try_deliver_interrupt(cpu, irq, more);
/*
 * All long-lived kernel loops need to check  with this
 * horrible thing called the freezer.
 * If the Host is trying to suspend,
 * the freezer stops us.
 */

    try_to_freeze();
/*
 * Just make absolutely sure the Guest is still alive.
 * One of those hypercalls could have been fatal,
 * for example.
 */

    if (cpu->lg->dead) break;

/*
 * If the guest asked to be stopped, we sleep.
 * The guest's clock timer will wake us.
 */

    if (cpu->halted) {
        set_current_state(TASK_INTERRUPTIBLE);
/*
 * Just before we sleep, make sure no interrupt snick
 * in which we should be doing.
 */
```

```
        if (interrupt_pending(cpu, &more) < LGUEST_IRQS)
            set_current_state(TASK_RUNNING);
        else continue;
        schedule();
    }
/*
 * OK, now we're ready to jump into the guest.
 * First we put up the "Do Not Disturb" sign:
 */

    local_irq_disable();
/* Actually run the guest until something happens. */
    lguest_arch_run_guest(cpu);
/*
 * Now we're ready to be interrupted or
 * moved to other CPUs
 */
    local_irq_enable();
/* Now we deal with whatever happened to the Guest. */
    lguest_arch_handle_trap(cpu);

/* Special case: Guest is 'dead' but wants a reboot. */

    if (cpu->lg->dead == ERR_PTR(-ERESTART))
return -ERESTART;

/* The guest is dead => "No such file or directory" */

    return -ENOENT;
}
```

The procedure "lguest_arch_run_guest()" is the i386-specific code that setups and runs the guest. When this function is called interrupts are disabled and we own the CPU. The first thing the function does is to disable the SYSENTER instruction. This instruction is an optimized way of doing system calls because it uses hard-coded code segment descriptors to describe the target code segment (where the service routine is), i.e access to the GDT (probably cached), protection check, etc... is not needed, so a switch to kernel mode could be done by this instruction. This instruction can't run by the guest because it always jumps to privilege level 0. A normal guest won't try it because we don't advertise it in CPUID, but a malicious guest (or malicious guest userspace program) could, so we tell the CPU to disable it before running the guest. We disable it by giving the hard-coded code segment selector (The register is called MSR_IA32_SYSENTER_CS) the value 0, so every memory reference would cause a fault.

The next thing "lguest_arch_run_guest()" runs the Guest using "run_guest_once()". It will return when something interesting happens, and we can examine its registers to see what it was doing. This is a switcher related code and is be covered in section 6.10. Next we give back the MSR_IA32_SYSENTER_CS register its default value - __KERNEL_CS, the kernel code segment descriptor so the Host can do system calls using SYSENTER (i.e switching to kernel mode).

Finally, we check if the Guest had a page fault. If it did we must read the value of the "cr2" register (holds the "bad" virtual address) before we re-enable interrupts that could overwrite cr2. This concludes the work of "lguest_arch_run_guest()".

Now "run_guest()" reenables interrupts. Next it calls "lguest_arch_handle_trap()" to see why the guest exited. For example, the guest intercepted a Page Fault (trap number 14). The guest could have two reasons for a page fault. The first one is the "normal" reason: a virtual page is not mapped to a page frame. In this case we inform the guest with the value of "cr" and he takes care of it. The second reason is that the address is mapped, but because the shadow page tables in the host are updated lazily the mapping is not present there. In this case the host fixes the page table and does not even inform the guest of the page fault.

APPENDIX 6.8.4 Hypercalls

The first thing "run_guest()" does is preform all the hypercalls that the guest wants done, using a call to "do_hypercalls()". Hypercalls are the "fast" way for the guest to request some thing from the host. There are normal and asynchronous hypercalls and the do_hypercalls function handles both types.

```
void do_hypercalls(struct lg_cpu *cpu)
{
...
/* Look in the hypercall ring for the async hypercalls */
     do_async_hcalls(cpu);
/*
 * If we stopped reading the hypercall ring
 * because the guest did a NOTIFY to the Launcher,
 * we want to return now. Otherwise we do
 * the hypercall.
 */
if (!cpu->pending_notify) {
do_hcall(cpu, cpu->hcall);
cpu->hcall = NULL;
}
}
```

The "do_hcall()" procedure is the core hyper call routine: where the guest gets what it wants or gets killed or, in the case of LHCALL_SHUTDOWN, both. This is a big switch statement of all the hypercalls and their handle. Examples for hypercalls are LHCALL_SENT_INTERRUPTS, LHCALL_SET_CLOCKEVENT and LHCALL_NOTIFY.

There are also architecture specific handling (this was taken out from the original implementation for future porting to other platforms) using "lguest_arch_do_hcall()" (called from "do_hcall()"), for example LHCALL_LOAD_GDT_ENTRY and LHCALL_LOAD_IDT_ENTRY.

Asynchronous hypercalls are handled using "do_async_hcall()". We look in the array of the guest's struct "lguest_data" to see if any new hypercalls are marked "ready".

We are careful to do these in order: obviously we respect the order the guest put them in the ring, but we also promise the guest that they will happen before any normal hypercall (Which is why we call this before calling for a "do_hcall()").

The final note about hypercalls was already mentioned in section 6.5.2 the KVM hypercalls. This mechanism was originally designed to replace the software interrupt mechanism. The motivation for the change was to be able to support live migration and SMP. KVM_HYPERCALL is a "vmcall" instruction, which generates an invalid opcode fault (fault 6) on non-VT CPUs, so the easiest solution seemed to be an "emulation approach". In the new approach if the fault was really produced by an hypercall (is_hypercall() does exactly this check), we can just call the corresponding hypercall host implementation function. But these invalid opcode faults are notably slower than software interrupts so a "patching (or rewriting) approach" was implemented. Using the "patching approach" every time we hit the KVM_HYPERCALL opcode in while running the guest code trap number 6 is marked so "lguest_arch_handle_trap()" calls "rewrite_hypercall()" that patch the instruction to the old "int 0x1f" opcode ("0xcd 0x1f"), so next time the guest calls this hypercall it will use this faster trap mechanism.

APPENDIX 6.8.5 Interrupts and Traps

We will reexamine the interrupt handling part of "run_guest()" routine. First we check, using "interrupt_pending()". If there are any pending interrupts, i.e interrupt we got from the launcher (virtqueues interrupt using LHCALL_NOTIFY hypercall) or a clock event interrupt from the guest (using LHCALL_SET_CLOCKEVENT hypercall). It returns the first pending interrupt that isn't blocked (disabled) by the guest. It is called before every entry to the guest, and just before we go to sleep when the guest has halted itself. If there are any interrupts, the "run_guest()" calls "try_deliver_interrupt()". This function diverts the guest to running an interrupt handler. If there are no "problems" (interrupt enabled, the interrupt has a handler) a call to "set_guest_interrupt()" is made. The "set_guest_interrupt()" routine delivers the interrupt or trap. The mechanics of delivering traps and interrupts to the guest are the same, except some traps have an "error code" which gets pushed onto the stack as well. The caller tells us if this is one. The interrupts are registered on the interrupt descriptor table (IDT). "lo" and "hi" are the two parts of the Interrupt Descriptor Table for this interrupt or trap. The interrupt is split into two parts for backword compatibility reasons. Old versions of gcc didn't handle 64 bit integers very well on 32bit systems. Afterward receiving the interrupt the host sets up the stack just like the CPU does for a real interrupt. Therefore, for the guest the situation is the same as normal OS (without hypervisor). After the guest issue the standard "iret" instruction will return from the interrupt normally. Therefore, under Lguest the guest deals with interrupts like normal Linux OS.

```
static void set_guest_interrupt(struct lg_cpu *cpu, u32 lo
, u32 hi, bool has_err)
{
    unsigned long gstack, origstack;
    u32 eflags, ss, irq_enable;
    unsigned long virtstack;

/*
 * There are two cases for interrupts:
 * one where the guest is already
 * in the kernel and a more complex one where
 * the guest is in userspace.
 * We check the privilege level to find out.
 */

    if ((cpu->regs->ss&0x3) != GUEST_PL) {

/*
 * The guest told us their kernel stack with the
 * SET_STACK hypercall: both the virtual address and
 * the segment.
 */
        virtstack = cpu->esp1;
        ss = cpu->ss1;
        origstack = gstack = guest_pa(cpu, virtstack);
        push_guest_stack(cpu, &gstack, cpu->regs->ss);
        push_guest_stack(cpu, &gstack, cpu->regs->esp);
    } else {
virtstack = cpu->regs->esp;
        ss = cpu->regs->ss;
        origstack = gstack = guest_pa(cpu, virtstack);
    }

/*
 * Remember that we never let the guest actually disable
 * interrupts, so the "Interrupt Flag" bit is always set.
 * Thus, we copy this bit from the guest's
 * "lguest_data.irq_enabled" field into the eflags word.
 * We saw the guest copy it back in "lguest_iret".
 */

    eflags = cpu->regs->eflags;
    if (get_user(irq_enable,
 &cpu->lg->lguest_data->irq_enabled) == 0
         && !(irq_enable & X86_EFLAGS_IF))
              eflags &= ~X86_EFLAGS_IF;

/*
 * An interrupt is expected to push three things on the stack:
 * the old "eflags" word, the old code segment, and the old
 * instruction pointer (the return address).
 */

    push_guest_stack(cpu, &gstack, eflags);
    push_guest_stack(cpu, &gstack, cpu->regs->cs);
    push_guest_stack(cpu, &gstack, cpu->regs->eip);

/*
 * For the six traps which supply an error code,
 * we push that, too.
 * When trapping them, set_guest_interrupt() would be called
 * with "has_err" == true
 */

    if (has_err)
push_guest_stack(cpu, &gstack, cpu->regs->errcode);

/*
 *  Now we’ve pushed all the old state, we change the stack,
 * the code segment and the address to execute.
 */

    cpu->regs->ss = ss;
    cpu->regs->esp = virtstack + (gstack - origstack);
    cpu->regs->cs = (__KERNEL_CS|GUEST_PL);
    cpu->regs->eip = idt_address(lo, hi);

/*
 * There are two kinds of interrupt handlers: 0xE is an
 * "interrupt gate" which expects interrupts to be
 * disabled on entry.
 */
```

```
    if (idt_type(lo, hi) == 0xE)
        if (put_user(0, &cpu->lg->lguest_data->irq_enabled))
            kill_guest(cpu, "Disabling interrupts");
}
```

In the "set_guest_interrupt()" function all references to ss refer to the stack segment that is copied around. We copy return address from interrupt to match the situation the CPU did return from interrupt. In "set_quest_interrupt()" the host made the kernel (interrupt handling) stack and the CPU appear to be at the exact state as if a "real" interrupt occurred. When the Guest would be run next it would start the interrupt handling.

Now that we've got the routines to deliver interrupts, delivering traps (like page fault) is very similar. The last thing that the "run_guest()" run loop does is call
"lguest_arch_handle_trap()" and tries to understand why the Guest exited.
The "lguest_arch_handle_trap()" function is implemented as a big switch statement to cover all possible traps and how to handle them. Each trap that is not handled by the host needs to be delivered to the Guest. Delivering the trap to the guest is done by calling "deliver_trap()" that checks if the trap can be delivered, if so it calls "set_guest_interrupt()" that makes the trap handler start the next time the Guest would run.

APPENDIX 6.8.6 The Page Tables

We use two-level page tables for the Guest, or three-level with PAE (as discussed in section 6.5.4. The Guest keeps page tables, but the host maintains the page tables ones, that are called "shadow" page tables. That is a very guest oriented name as these are the real page tables the CPU uses, although we keep them up to date to reflect the guest's.

Keeping the page table up to date is the most complicated part of the host code. There are four interesting parts to this

1. Looking up a page table entry when the guest faults
2. Setting up a page table entry when the guest tells us one has changed
3. Switching page tables
4. Mapping the Switcher when the guest is about to run

Looking up a page table entry when the guest faults is implemented using "lguest_arch_handle_trap()" (called in "run_guest()"), Lguest handle trap number 14 which is a page fault in the Guest. There are two possible causes for this trap. the first possible cause is the address is mapped on the guest but is not mapped on the shadow table. The host only set up the shadow page tables lazily (i.e. only as they're needed) so we get trap 14 for this reason frequently. The second possible cause is that there was a real fault (this address has a mapping in the shadow table but the page is not in memory). When this trap occurs and the page does not exist in the host memory, The host calls to "demand_page()" (in [/drivers/lguest/page_tables.c]) that quietly fix the shadow tables. If the page fault occurred because the shadow page table was not updated the return to the guest without the guest knowing about the fault. If this was a real fault the host needs to tell the guest, give him the value of the cr2 register (the faulting address) and calls "deliver_trap()".

Setting up a page table entry when the guest tells us one has changed occurs when the Guest asks for a page table to be updated (asking is after updating his own table as explained in section 6.5.4). Lguest implements "demand_page()" that will fill in the shadow page tables when needed, so we can simply dispose shadow page table entries whenever the guest tells us they've changed. When the guest tries to use the new entry it will fault and demand_page() will fix it up. To update a top-level/mid-level entry the guest hypercalls LHCALL_SET_PGD/LHCALL_SET_PMD that results in the host calling "guest_set_pgd()"/"guest_set_pmd()" (in [/drivers/lguest/page_tables.c]) removing the relevant entry. Updating a PTE entry (by calling "guest_set_pte()" as a result of LHCALL_SET_PTE hypercall) is a little more complex. The host keeps track of several different page tables in the field "cpu->lg->pgdirs" (the guest uses one for each process, so it makes sense to cache at least a few). Each of these have identical kernel parts: i.e every mapping above PAGE_OFFSET is the same for all processes. So when the page table above that address changes, the host updates all the page tables, not just the current one. This is rare so it is worth doing. The benefit is that when the host has to track a new page table, he can keep all the kernel mappings. This speeds up context switch immensely.

Switching page tables i.e changes the pointer to the top-level page directory: the cr3 register is implemented using the LHCALL_NEW_PGTABLE hypercall. Switching page tables occurs on almost every context switch. A call to "guest_new_pagetable()" (in [/drivers/lguest/page_tables.c]) would first try to find this page on the host's page table cache. If "guest_new_pagetable" finds the new page table in the host's cache it just updates the "cpu->cpu_pgd" field, else it calls "new_pgdir()" (in [/drivers/lguest/page_tables.c]). In "new_pgdir" the host creates a new page directory. The host randomly chooses a page from the cache to remove and instead puts in the new page. If the page table is reused (and not a newly allocated one) the host releases all the non kernel mapping using "flush_user_mapping()" The kernel mapping stays the same. When a kernel mapping is changed it is necessary to change all the tables. In this case all the shadow tables are clearer. This causes a lot of traps fixed by "demand_page()". Fortunately changing kernel mapping is very rare.

Mapping the switcher when the Guest is about to run . The switcher code and two pages for each CPU (This two pages make the "struct lguest_pages") needs to be shared with the guest and host. These pages must have the same virtual addresses in the host and the guest and are allocated at the top of the virtual memory: 0xFFC00000 (SWITCHER_ADDR). The host already has these addresses mapped to the switcher page frames of the current guest, it just needs to make the guest's last PGD entry (The top level entry that maps SWITCHER_ADDR and above) of the shadow table to map this addresses to the same place. After that the host and this guest will share these pages. Sharing the pages is implemented using a call to "map_switcher_in_guest()" (in [/drivers/lguest/page_tables.c]) that is called from "run_guest_once".

APPENDIX 6.8.7 Segments & The Global Descriptor Table

The GDT is a table of 8-byte values describing segments. Once set up, these segments can be loaded into one of the 6 "segment registers" in i386 architecture. GDT entries are passed around as "struct desc_struct"s, which like IDT entries are split into two 32-bit members, "a" and "b". The GDT entry contains a base (the start address of the segment), a limit (the size of the segment - 1), and some flags. The arrangement of the bits is complex.

The first part of the base is stored in the first 16 bits. Then the first part of the limit is stored in the next 24 bits. Then we have 8 bits of flags, 4 bits (second part of limit) another 4 bits of flags and finally a 2$^{\text{nd}}$part of the base address (extra 8 bits)

As a result, the file [/drivers/lguest/segment.c], that deals with segment in lguest (all functions in this section are from this file), contains a certain amount of magic numeracy.

The host never simply uses the GDT the guest gives it. The host keeps a GDT for each CPU, and copy across the guest's entries each time it wants to run the guest on that CPU. Loading a new GDT entry is with "load_guest_gdt_entry()" (invoked from LHCALL_LOAD_GDT_ENTRY hypercall). This process updates the CPU's GDT entry and mark it as changed. "guest_load_tls()" is a fast-track version for just changing the three TLS entries. Remember from section 6.5.3 that this happens on every context switch, so it is worth optimizing. The load three GDT entries in one hypercall (LHCALL_LOAD_TLS) is such optimization.

After loading an entry, it needs to be fixed using "fixup_gdt_table()". There are two fixes needed. The first is related to the privilege level. Segment descriptors contain a privilege level. The Guest is sometimes careless when it loads the entry and leaves the privilege level as 0 even though the guest running at privilege level 1. If so, the host fix it. The second fix related to the "access" bit. Each descriptor has an "accessed" bit. If the host don't set it now, the CPU will try to set it when the guest first loads that entry into a segment register. But the GDT isn't writable by the Guest (running at privilege level 1), so it must be done by the host.

APPENDIX 6.9 The Switcher

The file [drivers/lguest/x86/switcher_32.S] is the switcher (compiled as part of the "lg.ko" module). The switcher code which reside at 0xFFC00000 (or 0xFFE00000) astride both the host and guest. The switcher does the low-level Guest<->Host switch. It is as simple as it can be made and very specific to x86.

APPENDIX 6.10 The Switcher Pages

In Lguest runtime each CPU has two pages allocated at the top of the virtual memory: 0xFFC00000 (SWITCHER_ADDR). These pages are visible to the guest when it runs on that CPU. These pages contain information that the guest must have before running and also information regarding the host that is needed for the switch back. These two pages are abstracted with the "lguest_pages" struct:

```
struct lguest_pages {
/* This is the stack page mapped read-write in guest */
    char spare[PAGE_SIZE - sizeof(struct lguest_regs)];
    struct lguest_regs regs;
/*
 * This is the host state & guest descriptor page,
 * read only in guest.
 */
    struct lguest_ro_state state;
} __attribute__((aligned(PAGE_SIZE)));

struct lguest_regs {
/*
 * Manually saved part.
 * They're maintained by the Guest
 */
    unsigned long eax, ebx, ecx, edx;
    unsigned long esi, edi, ebp;
    unsigned long gs;
    unsigned long fs, ds, es;
    unsigned long trapnum, errcode;
/* Trap pushed part. */
    unsigned long eip;
    unsigned long cs;
    unsigned long eflags;
    unsigned long esp;
    unsigned long ss;
}
```

```
struct lguest_ro_state {
/*
 * Host information we need to restore when
 * we switch back.
 */
    u32 host_cr3;
    struct desc_ptr host_idt_desc;
    struct desc_ptr host_gdt_desc;
    u32 host_sp;
/*
 * Fields which are used when guest is running.
 */
    struct desc_ptr guest_idt_desc;
    struct desc_ptr guest_gdt_desc;
    struct x86_hw_tss guest_tss;
    struct desc_struct guest_idt[IDT_ENTRIES];
    struct desc_struct guest_gdt[GDT_ENTRIES];
}
```

The "lguest_pages" pages, have to contain the state for that guest, so the host copies the state in the switcher memory region (SWITCHER_ADDR) just before it runs the guest using "run_guest_once()". The copy is done using the function "copy_in_guest_info()" that calling is called by "run_guest_once". Copying the guest state is the first thing that "run_guest_once()" does. First, this function saves the "host_cr3" field on the "state" page. Second it calls "map_switcher_in_guest()". Then it updates the IDT/GDT/TLS in the "state" page that were changed in the guest since it last ran. After "copy_in_guest_info()" finishes, all the information that the guest (and switcher) needs is on its designated CPU pages (the two CPU pages an the SWITCHER_ADDR region). Finally we get to the code that actually calls into the switcher to run the guest: "run_guest_once()":

```
static void run_guest_once(struct lg_cpu *cpu,
struct lguest_pages *pages)
{
/*
 * This is a dummy value we need for GCC's sake.
 */
    unsigned int clobber;
/*
 * Copy the guest-specific information into this CPU's
 * "struct lguest_pages".
 */
    copy_in_guest_info(cpu, pages);
/*
```

```
 * Set the trap number to 256 (impossible value).
 * If we fault while switching to the Guest
 * (bad segment registers or bug),
 * this will cause us to abort the Guest.
 */
    cpu->regs->trapnum = 256;
/*
 * Now: we push the "eflags" register on the stack,
 * then do an "lcall". This is how we change from using
 * the kernel code segment to using the dedicated lguest
 * code segment as well as jumping into the Switcher.
 * The lcall also pushes the old code segment (KERNEL_CS)
 * onto the stack, and then the address of this call.
 * This stack layout happens to exactly match the stack
 * layout created by an interrupt...
 */
    asm volatile("pushf; lcall *lguest_entry"
/*
 * This is how we tell GCC that %eax ("a") and %ebx ("b")
 * are changed by this routine. The "=" means output.
 */
        : "=a"(clobber), "=b"(clobber)
/*
 * %eax contains the pages pointer.
 * ("0" refers to the 0-th argument * above, ie "a").
 * %ebx contains the physical address of the Guest's top
 * level page directory.
 */
: "0"(pages), "1"(__pa(cpu->lg->pgdirs[cpu->cpu_pgd].pgdir))
/*
 * We tell gcc that all these registers could change,
 * which means we don't have to save and restore them
 * in the Switcher.
 */
        : "memory", "%edx", "%ecx", "%edi", "%esi");
}
```

The "lcall" asm instruction pushes the IP and CS registers to the stack and put in them the instruction operands. These operands are abstracted in "lguest_entry" and match Intel's "lcall" instruction:

```
static struct {
    unsigned long offset;
    unsigned short segment;
} lguest_entry;
```

This structure is initialized in "lguest_arch_host_init()" with the address of the start of the switching code (at SWITCHER_ADDR after the mapping, not the compiled code address) and the LGUEST_CS segment selector.

APPENDIX 6.10.1 The Host to Guest Switch

The file [drivers/lguest/x86/switcher.S] contains the low-level code which changes the CPU to run the Guest code, and returns to the host when something happens. This is the heart of the Lguest.

The entry point to the switcher is "switch_to_guest". Register %eax has the "struct lguest_pages" (the register page and state page) that the guest can see. Register %ebx holds the Guest's shadow page table.

```
ENTRY(switch_to_guest)
/*
 * 1) Save host registers: Segment registers, stack base
 * pointer  - push them to the stack (the Host's stack).
 * 2) Save the host's current stack pointer - at field
 * "host_sp" in the "state" page ...
 * There are now five steps before us:
 * Stack, GDT, IDT, TSS and last of all,
 * the page tables are flipped.
 *
 * Stack: Our stack pointer must be always valid or else
 * we'll get NMI %edx does the duty here as we juggle,
 * %eax is "lguest_pages", the Guest's stack
 * lies within (at offset LGUEST_PAGES_regs)
 */
    movl %eax, %edx
    addl $LGUEST_PAGES_regs, %edx
    movl %edx, %esp

/*
 * GDT and IDT: The addresses of the Guest's tables
 * that the host placed in the  "state" page
 * (see "copy_in_guest_info()")
 * needs to be loaded to the CPU
 */

    lgdt LGUEST_PAGES_guest_gdt_desc(%eax)
    lidt LGUEST_PAGES_guest_idt_desc(%eax)

/*
 * TSS: The TSS entry which controls traps must be loaded
 * up with "ltr" now, after we loaded the GDT.
 * The reason is that the GDT entry that TSS uses Changes
 * type when we load it
 */

    movl $(GDT_ENTRY_TSS*8), %edx
    ltr %dx

/*
 * The Host's TSS entry was marked used; clear it again
 * our return.
 * PAGE TABLES: Once our page tables is switched,
 * the Guest is alive! The Host fades
 * as we run this final step. Remember that %ebx stores
 * the physical address of the  Guest's top-level page
 * directory
 */
    movl %ebx, %cr3

/*
 * Because the Guest's page table is mapping the
 * "register page" with the same virtual address
 * like the Host, the stack pointer(%esp) is under
 * all the Guest regs (the register page) so we can
 * simply pop off all.  To make this more obvious
 * let's look at the current state of the register
 * page ("struct lguest_regs")
 */

// 0 -------------------------
// | STACK                   |
// | .                       |
// | .                       |
// %esp -----------------------
// | eax, ebx, ecx, edx      |
// | esi, edi, ebp, gs       |
// | fs, es, ds              |
// | old trapnum, old errcode |
// | eip                     |
// | cs,                     |
// | eflags,                 |
// | esp,                    |
// | ss                      |
// 4K -------------------------

    popl %eax
    popl %ebx
    popl %ecx
    popl %edx
    popl %esi
    popl %edi
    popl %ebp
    popl %gs
    popl %fs
    popl %ds
    popl %es
/*
 * We increase the stack pointer by two words so the
 * stack would contain only the information needed
 * for the "iret".  It leaves the trap number and
 * its error above the stack.
 * They would be updated before switching back to Host
 */
// now the register state is
// 0 -------------------------
// | STACK                    |
// | .                        |
// | .                        |
// %esp -----------------------
// | old trapnum, old errcode |
// | eip                      |
// | cs,                      |
// | eflags,                  |
// | esp,                     |
// | ss                       |
// 4K -------------------------
    addl $8, %esp

// now the register state is
// 0 -------------------------
// | STACK                    |
// | .                        |
// | .                        |
// | old trapnum, old errcode |
// %esp -----------------------
// | eip                      |
// | cs,                      |
// | eflags,                  |
// | esp,                     |
// | ss                       |
// 4K -------------------------
```

```
/*
 * although there was no "call" or "int" instruction
 * that got us here, the  "iret" instruction is
 * exactly what we need.  It make the code jump to the
 * return address (%eip), switch privilege From Switcher’s
 * level 0 to Guest’s 1(CS, EFLAGS)
 * and because we decrease the priv level it
 * also pops the stack state (ESP, SS) of
 * the Guest before it finished his run
 * (look at the "iret" instruction specification).
 * The five registers are the stack’s slots so "iret"
 * fits perfectly (actually I assume that it was the "iret").
 * Notice that the NT flag in EFLAGS flag)
 * so there is no task switch.
 */
   iret
// Interrupts are back on. We are guest
```

APPENDIX 6.10.2 The Guest to Host Switch

We already demonstrated that the guest traps to the host with a hypercall that is a software interrupt. When an interrupt occurs the processor pushes the 5 state registers (the same registers from the host to guest switch) to the stack which TSS hold (ss0 and esp0). The function "lguest_arch_host_init()" (in [drivers/lguest/x86/core.c]) assigns "esp0" the bottom of the "registers page", just were we need it. So after an interrupt the "registers page" would look like this:

```
0 -------------------------
| STACK                   |
| .                       |
| .                       |
| old trapnum, old errcode |
esp -------------------------
| eip,                    |
| cs,                     |
| eflags,                 |
| esp,                    |
| ss                      |
4K -------------------------
```

Now the processor would approach the guest's interrupt table, to the relevant gate. The interrupt handlers push the trap number (and error) to the stack and jump to the switcher code that returns to the host. So before the jump to the switch code the "registers page" would look like this:

```
0 -------------------------
```

```
| STACK                   |
| .                       |
| .                       |
| old trapnum, old errcode |
esp ------------------------
| new trapnum, new errcode |
| eip,                    |
| cs,                     |
| eflags,                 |
| esp,                    |
| ss                      |
4K ------------------------
```

There are two paths to switch back to the host, yet both must save Guest state and restore host so we put the routine in a macro:

```
#define SWITCH_TO_HOST
/* We save the Guest state: all registers first       \
 * Laid out just as "struct lguest_regs" defines */   \
    pushl %es;                                        \
    pushl %ds;                                        \
    pushl %fs;                                        \
...
    pushl %ebx;                                       \
    pushl %eax;                                       \
/* find the start of "struct lguest_pages" so*/       \
/* we could access easily to its fields*/             \
    movl %esp, %eax;                                  \
    andl $(~(1 << PAGE_SHIFT - 1)), %eax;             \

/* Save our trap number: the switch will obscure it*/ \
/* (In the host the guest regs are not mapped here)*/ \
/* %ebx holds it safe for deliver_to_host */          \
    movl LGUEST_PAGES_regs_trapnum(%eax), %ebx;       \
/* Return to the host page tables */                  \
    movl LGUEST_PAGES_host_cr3(%eax), %edx;           \
    movl %edx, %cr3;                                  \
/* Switch to host’s GDT, IDT. */                      \
    lgdt LGUEST_PAGES_host_gdt_desc(%eax);            \
    lidt LGUEST_PAGES_host_idt_desc(%eax);            \
/* Restore the host’s stack where its saved regs lie*/\
    movl LGUEST_PAGES_host_sp(%eax), %esp;            \
/* Last the TSS: our host is returned */              \
    movl $(GDT_ENTRY_TSS*8), %edx;                    \
    ltr %dx;                                          \
/* Restore now the regs saved right at the first.*/   \
/* notice that we changed the stack pointer. */        \
/* this values are from the host's stack */            \
    popl %ebp;                                         \
    popl %fs;                                          \
    popl %gs;                                          \
    popl %ds;                                          \
    popl %es;                                          \
```

The first way to reach this switch when the guest has trapped (Which trap was it information has been pushed on the stack). We need only to switch back, and the host will decode why the Guest came home, and what needs to be done.

```
return_to_host: SWITCH_TO_HOST
/*
 * Go back to "run_guest_once()" to see that
 * we left the stack layout exactly match the stack
 * layout created by an interrupt.  So iret is what
 * we need to  come back to "run_guest_once()",
 * i.e return to the host code
 */
iret
```

The second path is an an interrupt with some external causes (an interrupt unrelated to the guest, could be an interrupt for the host, another guest etc). First, we must return to the host using "SWITCH_TO_HOST". However this is insufficient. We can not ignore this interrupt and we need to make the host call this interrupt handler.

```
deliver_to_host: SWITCH_TO_HOST
/*
 * But now we must go home via that place where that
 * interrupt was supposed to go if we had not been
 * ensconced, running the Guest.
 * Here we see the trickiness of run_guest_once(): The
 * host stack is formed like an interrupt With EIP, CS
 * and EFLAGS layered. Interrupt handlers end with
 * "iret" and that will take us back to
 * run_guest_once().
 * But first we must find the handler to call!
 * The IDT descriptor for the host has two bytes for
 * size, and four for address. So after that %edx
 * will hold the IDT address.
 */
    movl (LGUEST_PAGES_host_idt_desc+2)(%eax), %edx

/*
 * We now know the table address we need.
 * Also, SWITCH_TO_HOST saved the trap's number inside
 * %ebx. This put in %edx the handler's address.
 */
    leal (%edx,%ebx,8), %eax
    movzwl (%eax),%edx
    movl 4(%eax), %eax
    xorw %ax, %ax
    orl %eax, %edx
/*
 * We call the handler now:
 * its "iret" drops us home.
 */
    jmp %edx
```

APPENDIX 6.10.3 The Guest's IDT Gates

We define the IDT gates of the guest in order to make the right switch between the two options ("return_to_host" or "deliver_to_host"). When we create the Guest's IDT at the switch pages ("struct lguest_ro_state") with the function "copy_traps()" "copy_traps()" is called from "copy_in_guest_info()" which is called just before calling "run_guest_once()" We use the array "default_idt_entries". In the "default_idt_entries" array each entry contain an interrupt handler address. Hence, each entry will contain: a jump to "return_to_host" or "deliver_to_host" depending on the interrupt generated. The only interrupt that is not handled by both is NMI (none maskable interrupt).

In this case we use "handle_nmi" that just returns and hopes to stay alive. Notice that before jumping the handler push the trap number (and error) to its designated place in the registers page (the stack).

The Switcher initiates "default_idt_entries" with the following division:

– Interrupts handled with "return_to_host" : 0, 1, 3–31 (processor detected exceptions: divide error, debug exception, etc), 128 (system calls).
– Interrupts handled with "deliver_to_host" : 32–127, 129–255 (hardware interrupts)
– Interrupts handled with "handle_nmi" : 2 (none maskable interrupt)

APPENDIX 6.11 Lguest technologies in our system

Lguest provides an easy to extend infrastructure for system virtualization. Unlike KVM Lguest code base is relatively small and easy to understand and modify.

APPENDIX 6.11.1 Lguest as a kernel development platform

We have used Lguest as a platform for kernel profiling and code coverage system discussed in chapter 3 and published in [AKZ11a].

In order to use Lguest as a debugging platform we had to augment hypercalls to support profiling operations. Our current work on the LgDb system (to eliminate the need for code injections) is using Linux kernel debugger connected over serial port implemented on top of Virt I/O.

APPENDIX 6.11.2 Lguest serialization

As we have seen a running guest in the launcher consists of memory, register inside Lguest struct, and virt queues status. By saving the contents of the memory and adding interface to save and load the lguest and virt-queues one can develop infrastructure for serializing (saving and loading) guests.

We have developed this infrastructure and use it in two separate projects; our asynchronous replication of virtual machine discussed in chapter 4 and published in [AMZK11].

APPENDIX 7 PPPC IMPLEMENTATION DETAILS

We present here several design details and state machines not covered in chapter 7.

APPENDIX 7.1 PPPC System Overview

Peer-2-Peer Packet cascading is system designed to provide audio and video streaming clients with the capability to receive data from other clients and relay them to clients PPPC system is divided to PPPC router and PPPC Driver. The PPPC router contains two logical components the Coordinating Server (also called CServer) and Distributing Server (also called DServer).

PPPC driver installed on a client workstation (any standard PC) and consists of thin client software that handles the reception and relay of the packets, and also “feeds" them to any Media Player. The client do not interact with a media player, it only delivers packets to the media player. **Coordinating Server** (CServer), is a command and control system in charge on all PPPC drivers listening to a single stream. CServer is responsible for all the decisions in the system. For example, for a given client, from which client it should receive data, and to which client should it transfer data, how should the tree be rebuilt after a new client arrived, what to do if a client in the middle of the tree was disconnected, what happens when any given client reports he has problems with receiving stream from his parent. **Distributing Server** (DServer) is a data replication and relay system. DServer receives a multicast (data-only) stream and encapsulates the data in PPPC format (recognized by PPPC driver). DServer delivers the encapsulated packets to roots of PPPC clients’ trees (root clients). The CServer decides who are the root clients.

APPENDIX 7.1.1 Detailed description of the DServer, CServer and theirs components

One instance of the server handles one media stream. Multiple instances of the server are possible in order to handle more than one stream. Parts (entities) within the Server communicate with each other by TCP enabling them to run on several computers.

APPENDIX 7.2 Distress Logic

APPENDIX 7.2.0.1 Actions taken on Distress state

No distress No action is taken. If connected to DServer for re-request, disconnect.

Parent distress Send parent distress messages to children. Initiate parent distress timer for 15 seconds Resend message every 2 seconds.

Light distress Send parent distress messages to children. Resend message every 2 seconds. Go to DServer to complement missing packets.

Light+parent distress Send parent distress messages to children. Resend message every 2 seconds. Disconnect from DServer(if connected).

Major distress Send parent distress messages to children. Resend message every 5 seconds. Go to DServer to receive stream. Disconnect from parent. Declare parent to be a bad parent.

Major + parent distress Send parent distress messages to children. Resend message every 5 seconds. Go to DServer to receive stream. Disconnect from parent.

APPENDIX 7.2.0.2 State machines on distress state

No distress During no distress state 3 events are possible

Parent distress message arrived - move to parent distress state.

Packet rate went below DISTRESS_MIN_PACKET_RATE - move to light distress state.

Packet rate went below MIN_PACKET_RATE = move to major distress state.(This should happen very rarely, because client will pass through light distress state before moving directly to major distress state!)

parent distress During parent distress state 4 things can happen

Timer expired - parent distress timer (initiated when entering the state) has expired. Return to no-distress state.

Parent distress message arrived - parent has sent another parent distress message. Re-install timer.

Packet rate drops below DISTRESS_MIN_PACKET_RATE - enter light+parent distress state.

Packet rate went below MIN_PACKET_RATE - enter major+parent distress state.

Light distress There are 3 possible events.

Parent distress message arrived - enter light+parent distress state.

Packet rate goes above DISTRESS_MIN_PACKET_RATE - enter no distress state.

Packet rate went below MIN_PACKET_RATE - enter major distress state.

Light + parent distress There are 4 possible events.

Timer expired - parent distress timer (initiated when entering the state) has expired. Return to light distress state.

Parent distress message arrived - parent has sent

another parent distress message. Re-install timer.

Packet rate goes above DISTRESS_MIN_PACKET_RATE - enter parent distress state.

Packet rate went below MIN_PACKET_RATE - enter major+parent distress state.

Major distress There is only one event which that has to do with major distress. even if packet rate increases back to reasonable limits, we have decided to go to DServer.

Connected to DServer - leave major distress state.

Other events are ignored.

Major + parent distress There is only one event which that has to do with major distress. Even if packet rate increases back to reasonable limits, we have decided to go to DServer.

Connected to DServer - leave major distress state.

Other events are ignored.

APPENDIX 7.2.0.3 Leaving Distress state

Once a client connects to a new parent or connects to DServer the client automatically leaves all previous distress states and removes all pending distress timeouts.

Race condition While connecting to a new parent all distress states conditions still remains(old packet rate is still low) Therefore we ignore distress state for two seconds after moving to new connection.

APPENDIX 7.3 The algorithm

APPENDIX 7.3.1 Multi thredaed algorithm structure

The algorihtm fundamentals are presented in section 7.6.2. But the running algorithm is slightly more complex.

The running algorithm is a multi threaded program with 3 threads.

1. The main thread is described in 7.6.2.
2. The algorithm interface thread is a thread tasked with network I/O that is responsible for sending edges to clients.
3. The rehabilitation thread is responsible for removing bad parents status from clients.

The threads structure

Main thread (same as 7.6.2 brought for completeness)

Algorithm 17 Real time graph analysis

```
Read subnet to autonomous systems and autonomous systems to autonomous systems family files. Store information in a map.
Create global data structure spawn interface and parents rehabilitation thread and interface thread.
while Main thread is alive do
    if There are new requests then
        Handle new request, touch at most 100 containers.
        Inform interface thread when you are done.
    else
        if there are dirty containers then
            Clean at most 100 containers
            Inform interface thread when you are done
        else
            Wait for new request
        end if
    end if
end while
```

Algorithm interface thread This simple thread ensure algorithm won't hang on I/O operations. The main function is call is described in algorithm 18

Algorithm 18 Interface thread - main function

```
Require: message queue to report data. new edges queue.
while interface thread is alive do
    wait for signal for main thread.
    SendEdgesToInterface()
end while
```

Parents Rehabilitation thread This thread is responsible for checking punished clients and applying to them rehabilitation logic so that they can accept clients again.

The rehabilitation thread is described in algorithm 19.

Algorithm 19 Parents Rehabilitation thread - main function

```
Require: punished clients queue.
while thread is alive do
    if punished clients queue is empty then
        Sleep for REHABILITATION_TIME - 1 seconds.
    else
        Check head of punished clients queue and calculate that client's rehabilitation time.
        Sleep until rehabilitation time of that client.
        Rehabilitate client.
    end if
end while
```

ACRONYMS

AES	American Encryption Standard
ALSA	Advanced Linux Sound Architecture
ARC	Adaptive Replacement Caching
AS	Autonomous System
AVM	Application Virtual Machine (also PVM)
BSD	Berkley Software Distribution
CDN	Content Delivery Network
CFG	Control Flow Graph
CFS	Completely Fair Scheduling
CFQ	Completely Fair Queueing
CLR	Common Language Runtime
COM	Component Object Module
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
CPUID	CPU Identification (x86 Instruction)
DCOM	Distributed Component Object Module
DES	Data Encryption Standard
DMCA	Digital Millennium Copyright Act
DR site	Disaster recovery site
DRM	Digital Rights Management
DRP	Disaster Recovery Protection
EGEE	Enabling Grids for E-sciencE
EGI	European Grid Initiative
GDT	Global Descriptor Table
GNU	GNU's Not UNIX
GPL	GNU PUBLIC LICENSE
GPU	Graphical Processing Unit
GPGPU	General Purpose GPU
I/O	Input / Output
IaaS	Infrastructure as a Service
IP	Internet Protocol (networking)
IP	Intellectual Property (Trust)
IR	Intermediate Representation
IRQ	Interrupt Request Queue
ISA	Instruction Set Architecture
JDL	Job Description Language (in EGI)
JIT	Just-In-Time (Compiling)
JVM	Java Virtual Machine
IDT	Interrupt Descriptor Table
IPVS	IP Virtual Server

KDB	Kernel DBugger
KVM	Kernel-based Virtual Machine
LLVM	Low Level Virtual Machine
LRU	Least Recently Used
LVM	Logical Volume Manager
LVS	Linux Virtual Server
NIST	National institute for Standards and Technology
NMI	Non-Maskable Interrupt
OS	Operating System
OSS	Open Source Software
P2P	Peer-2-Peer
PaaS	Platform as a Service
PAE	Physical Address Extension
POSIX	Portable Operating System Interface
PPPC	Peer-2-Peer packet cascading
PS3	(Sony) PlayStation 3
PVM	Process Virtual Machine (also AVM)
QEMU	Quick Emulator
QoS	Quality of Service
RAID	Reliable Array of Independent Disks
RDP	Remote Desktop Protocol
RPC	Remote Procedure Call
RPO	Recovery Point Objective
RTO	Recovery Time Objective
SaaS	Software as a Service
SATA	Serial ATA
SCSI	Small Computer Serial Interface
SOHO	Small Office Home Office
SMP	Symmetric Multiprocessing
SSD	Solid State Disk
SuS	Single Unix Specification
SVM	System Virtual Machine
TCG	Trusted Computing Group
TCG	Tiny Code Generator (part of KVM)
TCP	Transfer Control Protocol
TLS	Thread Local Storage
TSS	Task Stack Segment
TPM	Trusted Platform Module
UDP	User Datagram Protocol
V4L	Video For Linux
V4L2	Video For Linux 2
VDI	Virtual Desktop Infrastructure

VFS	Virtual File System
VIP	Virtual IP
VM	Virtual Machine
VMM	Virtual Machine Monitor (Hypervisor)
VMM	Virtual Memory Manager (OS context)
VPN	Virtual Private Network

INDEX