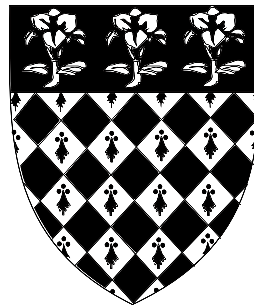
Massively Parallel Computing for Particle Physics

Ian Preston
Magdalen College, Oxford



Thesis submitted in fulfilment of the requirements
for the degree of Doctor of Philosophy

University of Oxford, Michaelmas Term 2010

Massively Parallel Computing for Particle Physics

Ian Preston,
Magdalen College, Oxford

Thesis submitted in fulfilment of the requirements
for the degree of Doctor of Philosophy.

University of Oxford, Michaelmas Term 2010

Abstract

This thesis presents methods to run scientific code safely on a global-scale desktop grid. Current attempts to harness the world's idle desktop computers face obstacles such as donor security, portability of code and privilege requirements. Nereus, a Java-based architecture, is a novel framework that overcomes these obstacles and allows the creation of a globally-scalable desktop grid capable of executing Java bytecode. However, most scientific code is written for the x86 architecture. To enable the safe execution of unmodified scientific code, we created JPC, a pure Java x86 PC emulator.

The Nereus framework is applied to two tasks, a trivially parallel data generation task, BlackMax, and a parallelization and fault tolerance framework, Mycelia. Mycelia is an implementation of the Map-Reduce parallel programming paradigm. BlackMax is a microscopic blackhole event generator, of direct relevance for the Large Hadron Collider (LHC). The Nereus based BlackMax adaptation dramatically speeds up the production of data, limited only by the number of desktop machines available.

This thesis was typeset with LaTeX.

© Ian Preston, 2010.
All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without express permission of the author.
Published at the University of Oxford, Oxford, United Kingdom.

“It’s true. Good things come to those who wait and persevere.”

Charmaine

Thanks

First and foremost I would like to thank my wonderful parents. They have supported me tremendously through thick and thin and made me who I am today. I couldn't ask for better, more supportive parents.

I thank Michael Watt, without whose extreme generosity I would not be studying in Oxford. Magdalen College has been my home during my time in Oxford and they have done a tremendous job of looking after and educating me. My head college tutor, Professor John Gregg, has been both a mentor and a friend and played a crucial role during my time here. I thank Anthony Smith for being such a welcoming president of Magdalen and making my time here as comfortable as possible.

Dr Rhys Newman is an inspiration. He has guided my learning process to become a world class programmer and taught me an immense amount. Professor Rodney Jory showed me what a joy Physics can be to learn and teach and gave me much encouragement on my path. I would like to thank my supervisor, Jeff Tseng, for pulling me through some of the tough times during my DPhil. My undergraduate tutors, Arzhang Ardavan, Giles Barr, Geoff Smith have inspired me for life and given me an excellent foundation to build from. I thank my grandpa, Gordon Preston, for teaching me calculus from first principles when I was 12 and thus showing me how beautiful mathematics can be. I thank my mathematics teachers throughout school for keeping me well supplied with difficult problems. I thank my examiners, Todd Huffman and Tim Marsland, for giving me such an enjoyable and intellectually stimulating viva. I thank Guillaume Kirsch for being a great office mate and for the TeX template for this thesis.

I thank my gorgeous girlfriend, Joanna Bagniewska, for putting up with me during the stressful times during my DPhil, listening patiently to all my ideas and giving me many valuable suggestions. I thank George Economides for all the coffees and lunches, lots of good feedback and advice and generally being awesome. Charlotte Woolley, who was my lovely dance partner for four years, has had a huge impact on my life and will remain a best friend for life. I thank my darling dance partner, Charmaine Yap, for being there when I most needed support and keeping me sane with lots of dancing. I thank Bruce Richardson, who taught me how to dance and thus ignited a fire which will burn for the rest of my life. I thank Ewa Turek for originally giving me the idea to take up dancing. I thank Sean Murphy for excellent comments and advice. I thank Kirsty Simpson for being such a good friend. Last but by no means least, I thank my good friends Alex Gray and Tamasin Graham. They kindly opened their house to me when I had no where else to stay and have done everything they can to help me.

Thank you to all my friends over the years! You have all played a part in helping me reach this point in my life.

Author’s contributions

The work presented here is the result of the effort of six people comprising the Interdisciplinary Grid Development research group: Dr. Jeff Tseng, Dr. Rhys Newman, Chris Dennis, Dr. Guillaume Kirsch, Mike Moleschi and myself. I wrote 70% of the code for JPC. 80% of that (56% of the total) is my original creation. The other 20% is from me correcting errors in pre-existing code. The following elements of JPC are entirely my work:

- Source code compiler
- Basic loop compiler
- Virtual clock
- Snapshot capability
- Remote hard drive
- Ability to store compiled classes between runs
- Ethernet Card
- Fat32 drive emulation

I contributed significantly to the actual implementation of JPC microcodes and debugging their execution.

I was involved in the inital structural design of Nereus and its later testing.

The Java version of Blackmax, and Mycelia, the Nereus implementation of Map-Reduce, are entirely my work.

Contents

1 Introduction **1**
1.1 Motivation 1

2 Architecture **5**
2.1 Introduction 5
2.2 Existing grid computing architectures 7
2.2.1 LHC Computing Grid (G-Lite) 8
2.2.2 Globus Toolkit 8
2.2.3 Condor 9
2.2.4 PlanetLab 9
2.2.5 Eucalyptus 10
2.2.6 BOINC (Berkeley Open Infrastructure for Network Computing) 10
2.2.7 Summary of issues 11
2.3 A solution using Java applets 11
2.3.1 Java performance and suitability for scientific computing 13
2.4 Nereus architecture 15
2.4.1 Nereus client 18
2.4.2 Client communication 19

3 The x86 architecture and virtualization **21**
3.1 x86 architecture 21
3.1.1 Real Mode 24
3.1.2 Protected Mode 25
3.1.3 Interrupts 26
3.1.4 Virtualization 27
3.1.5 Emulation security 31
3.1.6 Direct Virtualization security 32
3.1.7 Virtualizability of x86 34
3.1.8 Existing x86 emulators 37
3.2 Potential speed of a Java x86 emulator 38

4 JPC architecture and implementation **53**
4.1 JPC Memory architecture 56
4.2 Motherboard 58
4.3 PCI 60
4.4 Peripherals 60
4.5 Timing implementations 61
4.6 Block Devices 63

4.7 Network card 65
4.8 Snapshot 66

5 JPC execution and profiling **67**
5.1 JPC Execution 67
5.2 Compiled execution 71
5.2.1 Source code compiler 72
5.2.2 Basic block compiler 73
5.2.3 Basic loop compiler 73
5.2.4 Full loop compiler 73
5.2.5 Storing compiled blocks between runs 74
5.3 Improving JPC's emulation accuracy 75
5.3.1 Platform specification 75
5.3.2 Initial debugging attempts 76
5.3.3 Direct debugging 78
5.4 Profiling JPC 81
5.4.1 Measurements 82
5.4.2 Code 82
5.4.3 Results 83
5.4.4 Possible improvements 88

6 Map-Reduce and a Nereus based implementation, Mycelia **92**
6.1 Introduction 92
6.1.1 Problems handled by a Map-Reduce framework 93
6.1.2 The algorithm 94
6.1.3 Usage examples 95
6.1.4 Distributed implementation 96
6.1.5 Existing implementations 98
6.2 Mycelia - Map-Reduce in Nereus 100
6.2.1 Architecture 100
6.2.2 Data Flow 101
6.2.3 Implementation 103
6.2.4 Consistency, correctness and failure handling 109
6.2.5 Redundancy 110
6.3 Tests 111

7 BlackMax generator **113**
7.1 Introduction 113
7.2 Execution in JPC 113
7.3 Translating BlackMax to Java 114
7.3.1 Single machine timing tests 114
7.4 An alternative possibility - execute in another emulator, NestedVM 115
7.4.1 Running in Nereus 116

8 Conclusion **117**

Bibliography **121**

List of Figures

1.1 Power Usage 2

2.1 A proxy server 13
2.2 Nereus architecture 16
2.3 Application server 18
2.4 Nereus client communications 20

3.1 CPU structure 22
3.2 x86 Instruction Format 23
3.3 Real mode segmentation 24
3.4 Segmentation 25
3.5 Protected mode address translation 26
3.6 Linear Address Translation 27
3.7 Full Virtualization 28
3.8 Hardware-assisted Virtualization 29
3.9 Paravirtualization 30
3.10 Emulation 31
3.11 Emulation in Java 32
3.12 Toy CPU relative speeds 50
3.13 Toy CPU relative speeds with native comparison on Machine A 51
3.14 Toy CPU relative speeds with native comparison on Machine B 52

4.1 PC architecture 54
4.2 Schematic PC architecture 54
4.3 JPC Architecture 55
4.4 A 2 layer memory structure 56
4.5 JPC's 3 layer memory structure 57
4.6 Privilege level changing in JPC 59
4.7 Corrective Drift Timing 63
4.8 Network DOOM between two JPC instances 65

5.1 Codeblock cache structure 71
5.2 The compile chain for x86 code in JPC 72
5.3 The components of a PC's state 75
5.4 Comparing an emulator with a real PC one instruction at a time 77
5.5 Comparing two emulators one instruction at a time 78
5.6 Comparing two emulators one instruction at a time and ignoring interrupts . . . 79
5.7 Timing results of JPC benchmark 1 86
5.8 Timing results of JPC benchmark 1 run natively 86
5.9 Timing results of JPC benchmark 2 87

6.1 Map-Reduce 97
6.2 Mycelia architecture 100
6.3 Job Centre state 102
6.4 Mapreduce data flow 102
6.5 Job structure 103
6.6 Map-Reduce data flow 104
6.7 Mycelia classes 104
6.8 Results storage structure 108
6.9 Worker state 110
6.10 Map-Reduce data scaling 112

List of Tables

2.1 Global desktop grid requirements 7
2.2 Problems with current grid computing architectures 11
2.3 Applet sandbox restrictions 12
2.4 Service tag attributes 19

3.1 x86 emulation vs other virtualization 30
3.2 Virtualizer vulnerabilities 35
3.3 Native benchmark timings 51
3.4 Toy emulator timings 52

4.1 Non-monotonic time emulation 62

7.1 Blackmax timings for Java and native version 115
7.2 BlackMax timings in NestedVM 116

Glossary

Amazon EC2	An online service which provides the ability to run x86 virtual machines in the cloud
API	An Application Programming Interface is the set of interaction points that a program presents to other programs for their use
applet	A Java applet runs untrusted Java code from the internet in a security enforced sandbox
application server	A http server designed to managed a nereus computation, possibly storing input and output data
ATLAS	A Toroidal LHC ApparatuS, general purpose high-energy physics detector built as one of the four LHC experiments
BIOS	The Basic Input/Ouput System contains firmware code that a PC executes first upon being switched on
break point	Execution of a program stops when a breakpoint is reached
CERN	European Organisation for Nuclear Research, located near Geneva, Switzerland
class loader	The part of the Java runtime environment that verifies and loads new classes into the JVM
cloud	Cloud computing is Internet-based computing, whereby resources and services are made available to Internet-connected devices in a model based on the electricity grid
cluster	A groups of computers networked together, often using a fast local area network
compute cloud	A cloud which mainly provides computing power, as opposed to data storage
CPL	The Current Privilege Level an x86 CPU is operating at
CPU	Central Processing Unit, or the processor of a computer executes the instructions of programs run on the computer
DDoS	A distributed denial of service attack uses multiple agents to all simultaneously request the same resource in an attempt to deny other agents access to the same resource

desktop grid	A grid composed of machines that are desktop machines, which are often being simultaneously used for other purposes
DMA	Direct Memory Access allows certain hardware subsystems in a computer to access the memory indepently of the CPU
DOS	An operating system compatible with Microsoft Disk Operating System (MS-DOS)
DPL	The Descriptor Privilege Level of a memory segment on an x86 machine defines the lowest (numerically greatest) privilege level which can access that segment
emulator	A program or electronic device which imitates another program or device
ext2	ext2 is a file system for Linux
far jump	On x86 processors, a far jump is a jump to an instruction in a different segment to the current code segment, but at the same privilege level
FAT32	FAT32 was the default file system for Window versions from 95 to 98
firewall	A program or device that is configured to allow or deny network transmissions based on a set of rules
FLOPS	FLoating point OPerations Per Second
garbage collection	A form of automatic memory management for programs, which involves reclaiming memory taken up by objects that are no longer used by the program
gcc	The GNU Compiler Collection is a set of open source compilers often used for compiling C or FORTRAN programs
GiB	2^{30} bytes, a binary gigabyte
grid	A combination of computing resources, often loosely coupled machines, from multiple administrative domains into one available resource
hash	a mathematical function that converts a large, variable sized amount of data into a datum, usually an integer.
http	Hypertext Transfer Protocol is the basis of communication on the world wide web
hypervisor	A VMM
I/O	Input and output
IDE	The Integrated Drive Electronics interface is an old standard defining how to access attached disk drives on a PC
interface	An interaction point between different components of a system, usually with a prespecified set of access points and protocols

IP	The Internet Protocol used for routing data packets across a packet switched network
JAR	A Java ARchive is used to distribute Java class files and associated data in a format based on the zip format
JIT	A Just In Time compiler compiles and optimises code whilst it is running, normally using information obtained during the execution, such as branching ratios
JVM	Java Virtual Machine
KiB	2^{10} bytes, a binary kilobyte
KVM	The Kernel-based Virtual Machine is a virtualizer included in the Linux kernel. It can act as a paravirtualizer or hardware-assisted virtualizer.
LHC	Large Hadron Collider, a 27 km long proton–proton collider with a center of mass energy of $\sqrt{s} = 14$ TeV and luminosity of $\mathscr{L} = 10^{34}\ cm^{-2}s^{-1}$
MiB	2^{20} bytes, a binary megabyte
microcode	Lower level code that can be combined to make a higher level machine code
MIPS	Microprocessor without Interlocked Pipeline Stages is a RISC ISA originally developed and released by MIPS Computer Systems in 1985
MMU	The Memory Management Unit of a CPU handles accesses to memory, including enforcing memory protection and peforming address translation
NAT	Network Address Translation is the process of modifying network address information in network packet headers by a router to map one IP address space onto another
native code	Machine instructions that can run directly on a physical processor
NTFS	New Technology File System superseded the FAT32 file system as the default for Windows operating systems from 2000 onwards.
page	A 4KiB long section of memory aligned on a 4KiB boundary
parity	The number of ones in the binary representation of a number (odd or even)
PC	A personal computer
PCI	The Peripheral Component Interconnect is a computer bus (communication channel between components) allowing PCI card based peripheral devices to be attached to a PC

port	A virtual data connection used by programs to exchange data directly
proxy server	An intermediary for requests from clients seeking resources from other servers
RAM	Random Access Memory is low latency, temporary storage connected directly to a CPU
register	A small amount of storage in a CPU which can be accessed faster than any other storage
RPL	The Requestor Privilege Level is a field in an x86 segment selector which allows code to request a resource at a more privileged level
sandbox	A security mechanism for separating running programs and controlling their access to resources and information
sector	A subdivision of a track on a disk, usually 512 bytes
socket	A socket is an endpoint of a bidirectional communication flow between processes. It can be used between processes on a single machine or over a network.
SSA	Static Single Assignment form is a representation of computer code where every variable is assigned exactly once
statically compiled language	A programming language that is compiled in advance of execution to the final form that will be executed
system call	A program's request for service from an operating system kernel that it would not otherwise be permitted to access
TCP	Transmission Control Protocol, which is also known as TCP/IP is a member of the Internet Protocol Suite which provides reliable, ordered transmission of a stream of bytes between two machines on a network
UDP	User Datagram Protocol is a stateless transfer protocol in the Internet Protocol Suite
UNIX	An operating system started in 1969 by AT&T
URI	A Uniform Resource Identifier is a string of characters used to identify a name or resource on the Internet
VGA	The Video Graphics Array of a PC controls the visual output to the screen
virtual machine	A software or hardware based, isolated duplicate of a real machine or machine specification
virtualization	The creation of a virtual version of hardware, an operating system or a device. In the context of this thesis, normally a virtual machine representing a computer

VMM	A Virtual Machine Monitor allows one or more virtual machines to be run on a host computer and manages their access to resources
VMotion	A feature of VMWare V-sphere virtualization that enables running virtual machines to be moved between different physical hosts without the virtual machine noticing.
watch point	Execution of a program stops when the value at a watch point is modified or accessed
x86	x86 is a family of instruction set architectures based on the Intel 8086, which was released in 1974
Xen	A virtualizer for the x86 architecture which can act as a paravirtualizer or hardware-assisted virtualizer.
XML	Extensible Markup Language is a set of rules for encoding text documents in a machine readable form

Chapter 1

Introduction

This chapter presents the motivation for developing Nereus, a global Java based desktop compute cloud and JPC, a Java x86 PC emulator capable of safely executing unmodified x86 code.

1.1 Motivation

Scientific research requires an ever increasing amount of computing power to obtain new results [1][2][3][4]. Many scientific developments would be impossible without access to large pools of computing power. Much of the extra computing is required because scientists are creating increasingly large datasets. Modern scientific analyses often consist of a large number of small, independent computations and involve generating large data sets via simulation, or filtering large datasets according to a particular criteria, both of which are highly parallel in nature [5]. The Large Hadron Collider is expected to generate 41 terabytes of data each day at full operational intensity [6]. This data has to be stored and analysed, operations that will require an enormous amount of CPU time. Within a decade, the Square Kilometre Array telescope is expected to create over 17 petabytes of data per day [7]. The computing demands of science are rapidly increasing.

Coupled with this increasing demand for computing is the exponential increase in the power of general desktop machines [8]. Most of the computing power in desktop machines is unused.

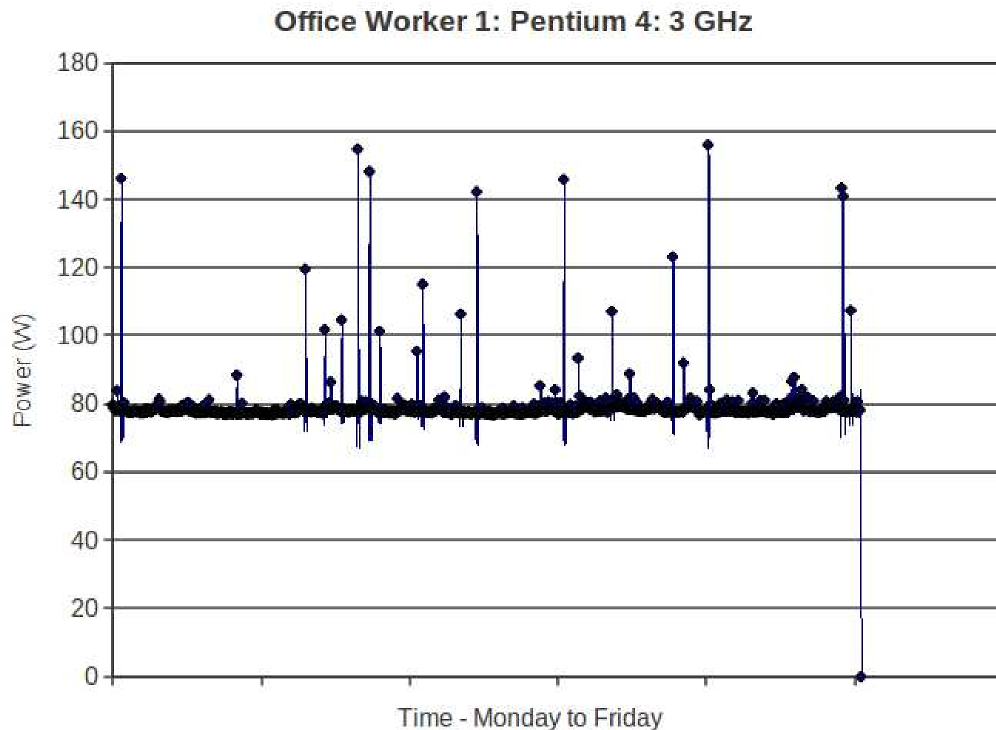


Figure 1.1: Power usage of a typical desktop machine with a person using it for typical office tasks.

A plot of electricity usage for a typical office PC shows the two main usage levels when switched on (shown in Figure 1.1): the first, when the PC is on and the CPU is idle, and the second, when the CPU is in heavy use. The relative sparsity of peaks in this plot illustrates just how idle desktop machines are during typical use. The amount of idle time on each machine will increase rapidly as the number of cores in desktop processors increases.

Scientific computing using a desktop grid approach can be very economical as well as environmentally friendly. Using volunteer desktop grids is at least an order of magnitude cheaper than using cloud computing (the provision of dedicated compute instances in a data centre administered externally, for example Amazon EC2)[9], which itself is cheaper than building a dedicated cluster [10]. A large part of the reduction in energy used by a desktop grid compared to a cluster is due to cooling requirements in a cluster. The cooling costs alone make up roughly half the cost of running a data centre [11]. A desktop grid, because of its distributed nature, does not need any extra cooling.

The most successful attempt so far at harnessing the world's idle desktops is BOINC [12],

which has over 570,000 desktop machines donating their CPU time [12]. However, there are over a billion desktop computers around the world connected to the Internet, representing an untapped computing resource with more than a hundred times the combined power of the world's top 500 supercomputers [13][14]. To achieve more of this potential, several important hurdles must be overcome.

The first obstacle is improving security for donor machines. Security vulnerabilities in grids are a serious potential problem [15][16][17]. Donor machines must be protected from having private data compromised or denial of service attacks. Extensions have been made to BOINC to improve the security for donor machines [18]. However, any grid that executes code without a strong sandbox fundamentally requires donors to trust not only the application developers' intent, but also their ability. A suitable sandbox needs to enforce the necessary security restrictions on code at runtime to eliminate the need for donors to trust application developers.

The second obstacle is the heterogeneity of the desktop grid. There are many different types of operating systems, processors and firewalls, all of which can affect the execution of code. An example of how time consuming it can be to maintain software running across this heterogeneity is the climateprediction.net desktop grid (which uses BOINC). This project has two people employed full time solely to support all of the different platforms that their code might run on [19]. This is clearly not an efficient way to proceed. An ideal grid would execute code in a cross-platform manner, without requiring recompilation for all target platforms.

The third major obstacle stopping current desktop grids from capturing a larger percentage of available machines is the need to install client software, possibly requiring elevated privileges, as well as necessary firewall modifications. Many of the world's corporate desktop users will not have the privileges necessary to install software. BOINC, although requiring root privileges for installation, eliminates the need for firewall modifications by doing all communications using http on the standard port 80 used to browse the internet.

As well as idle desktop computers, there are several other potential sources of large untapped processing power. Set-top boxes are becoming increasingly common in households [20]. These

come equipped with excellent processors and very good, permanent internet connections and could be harnessed for work during most of the day in return for making the tv subscription cheaper. Mobile phones represent another upcoming resource, with modern high end phones containing better processors than recent laptops [21]. The only complication here is that users don't want their mobile battery drained when they are travelling, so any use would have to be restricted to when the phone were plugged into mains power.

Monte Carlo data generation in particle physics is trivially parallel and often CPU-intensive. Particle physics generates large datasets of measurements of independent events. A typical analysis involves searching this large dataset for events that satisfy certain criteria, and is also often CPU-intensive. Both types of problems fall into the larger category of highly parallel CPU-intensive tasks. Such tasks are well suited to a grid of desktop computers where the communication overhead through the Internet is much larger than if the nodes were different cores of a super-computer.

This thesis presents software to enable the creation of a global desktop grid for scientific computing. Chapter 2 presents a Java based desktop grid computing framework that solves these three obstacles and allows the creation of a globally scalable desktop grid to safely run arbitrary untrusted Java bytecode. To run unmodified native code on such a grid requires a virtual machine sandbox for the x86 architecture. Chapter 3 describes the x86 architecture and different ways to create such a virtual machine, specifically the implementation of a system emulator in Java. Chapter 4 introduces the architecture of JPC, an x86 PC emulator in Java. Chapter 5 details the execution process within JPC, methods for improving the emulation accuracy, and benchmarks the current speed of JPC. Chapter 6 describes Mycelia, an implementation of MapReduce, the popular parallel programming paradigm, for Nereus. Chapter 7 investigates the possible methods for utilising Nereus resources for a microscopic black hole event generator, BlackMax. Chapter 8 concludes the thesis and assesses progress towards a global desktop grid.

Chapter 2

Architecture

This chapter presents Nereus [22], a Java desktop distributed computing system for running Java bytecode in a secure and scalable way. Nereus aims to make the world's idle CPU time available for productive use.

2.1 Introduction

There are over a billion desktop PCs around the world connected to the Internet, representing an untapped computing resource with more than a hundred times the combined power of the world's top 500 supercomputers [13][14]. Almost all this processing power is wasted because there is currently no software capable of harnessing a significant fraction of these machines, when they are otherwise idle, in a secure and scalable way.

During the last decade, most supercomputers have moved away from the use of proprietary and high cost components and architectures to the use of standard, off-the shelf components. Currently 88% of the world's top supercomputers are built using typical desktop processors; 52% even use standard Gigabit Ethernet to connect individual machines. The convergence is also reflected in the operating system usage, with 89% of them using Linux as their operating system [13].

The most successful attempt to date at desktop supercomputing is the Berkeley Open Infrastructure for Network Computing (BOINC [12][23]) which is used for projects like SETI@home [24][25][26], climateprediction.net [27][28] and folding@home [29][30]. After 20 years, this effort has achieved a large active install base of $\sim$ 570 thousand PCs [12], totalling around 5 petaFLOPS - more than the current fastest supercomputer, the Cray Jaguar with $\sim$ 2 petaFLOPS [13]. This is roughly 0.035% of the total 1.6 billion Internet connected desktops [31]. However, there are several obstacles to the BOINC project harnessing a larger percentage of available desktop PCs:

- *security* of running code is not enforced
- installation of the client requires *root privileges*
- *lack of portability* increases work required to write and maintain applications

Aside from the lack of economic or other strong incentives, the main factor limiting uptake is security. BOINC runs natively as user-mode code, and thus a donor (person whose computer time is being donated) must trust the BOINC framework and the application developers not to write any malicious (or simply broken) code which might read personal data or compromise the system. The security risks involved have been acknowledged by the BOINC team [32]. This lack of enforced security is enough to stop most people running it and also means that BOINC applications have a long verification cycle for updates; they need to be tested on all possible platforms, for all kinds of problems, deliberate or otherwise. The donor must trust this manual vetting process. The BOINC website advises: “Projects should test their applications thoroughly on all platforms and with all input data scenarios before promoting them to production status”. Such a high latency programming process directly opposes the nature of research code, which is often created under tight time and resource budgets without the luxury of robust software development processes.

Along with the security problems inherent in executing native code, the BOINC client for a particular operating system and processor combination has to be downloaded and installed,

including creating a new user account on the computer to execute the client. Many corporate or institutional desktop users simply will not have the necessary privileges to install the client.

The final problem is portability of code across multiple platforms. Personal conversations with climateprediction.net researchers[19] have indicated that they need two full-time developers just to maintain their application code working across the different platforms it runs on. Whilst this does not affect the donors (assuming their PC's platform is supported), it is a significant barrier for users wishing to create or modify an application.

One excellent feature of BOINC is that all client communications will work behind any firewall that still allows access to the Internet. Given that most corporations have very restrictive firewalls, this is highly desirable. This is achieved by making all communications use the http protocol with the standard port number 80.

A truly successful global desktop grid must have the properties listed in Table 2.1.

Table 2.1: Global desktop grid requirements.

1.	Enforced **security** to protect donors, which removes the need to **trust** application developers
2.	**No installation** necessary
3.	**Portable** to different operating system types and versions
4.	Must work transparently behind firewalls, proxy servers, and NAT boxes

2.2 Existing grid computing architectures

There have been many attempts to create global scale grid computing software. This section discusses some of the larger projects and why they are unsuitable for a truly global desktop grid. The security issues of these and other approaches have been discussed elsewhere [16][15][17][33][34].

2.2.1 LHC Computing Grid (G-Lite)

G-lite [35] is used by CERN as a grid architecture that functionally operates as a large batch processing system. Numerous network ports must be kept open on donor machines, a fact that rules out most of the corporate and university desktops in the world. Installation requires root privileges, and the software needs a specific version of one of three operating systems (Red Hat Linux 3.0, CentOS or Scientific Linux), or something binary compatible, which precludes the use of general office computers without significant investment in IT resources.

The G-lite user guide contains over 160 pages, a formidabe barrier for adoption for many users. In summary the problems with using LCG for a global desktop grid are:

1. it cannot coexist with normal desktop usage - requires complete OS install
2. it needs specific firewall configuration to open ports

In spite of these problems, it is one of the largest grids in the world, with over 100,000 computers connected. This is because LHC related administrations at many large universities around the world have mandated the creation of large clusters specifically for this grid, without which participation in the exciting physics of the LHC would be much more difficult.

2.2.2 Globus Toolkit

Globus [36][37] is an open source grid computing toolkit first released in 1998. It requires a UNIX variant operating system to run [38], and thus will not run on the dominant operating system in the world, Windows. This rules out a very large portion of potential donor computers in the ideal CPU target market. Globus installation requires root privileges and the creation of a new user purely for running Globus. A Globus installation also requires many ports to be opened in any firewalls [39]. Thus, the reasons why Globus is unlikely to become a global desktop grid include:

1. it cannot coexist with normal desktop usage - requires Unix OS

2. it requires root privileges for installation

3. it requires extensive firewall modification

2.2.3 Condor

Condor [40] is a specialized workload management system for CPU intensive jobs. Like other batch systems, Condor provides a job queueing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. Condor can also be made interoperable with resources provided by Globus.

The execution model within Condor could allow various kinds of privilege escalation or denial of service attacks [41][42][15]. Condor also requires a large number of ports to be opened for bidirectional access, using UDP and TCP over IP, in donor computers' and organisational firewalls. Condor is not likely to be globally accepted as a desktop grid because:

1. it requires root privileges for installation

2. it requires root privileges for execution, if security enforcement is desired

3. its security model insufficient for untrusted, global execution

4. it requires extensive firewall configuration

2.2.4 PlanetLab

PlanetLab is a global network of computers designed to be utilised as a network testbed for distruptive Internet technologies [43]. An institution must sign up to PlanetLab before any computers can be added from that institution. All nodes on PlanetLab must be dedicated high-end server machines with x86 based processors and a minimum of 4 cores. All nodes are also required to be external to any institutional firewall. Each node also requires at least one static, publicly routable IP address.

1. cannot coexist with normal desktop usage

2. it requires root privileges for installation

3. requires firewall modification

2.2.5 Eucalyptus

The Eucalyptus framework [44][45] provides an interface identical to Amazon EC2 for creating virtual machines on a compute cluster. It allows organisations to pool their machines together into a private cloud which can transparently spill over into the Amazon EC2 cloud in high usage times and is designed to allow multiple compute clusters to be drawn into a single resource pool [46]. On the client side, Eucalyptus uses hypervisors (currently Xen or KVM) and thus requires root privileges to install. Hurdles to using Eucalyptus on a desktop grid include:

1. cannot coexist with normal desktop usage - requires hypervisor install

2. requires firewall modification

2.2.6 BOINC (Berkeley Open Infrastructure for Network Computing)

The BOINC architecture was discussed in section 2.1.

The obstacles to BOINC capturing more of the available global desktop computing include:

1. code is not run in a secure sandbox

2. code is not portable

3. root privileges are required for installation

2.2.7 Summary of issues

The problems faced by current grid computing solutions which will prevent them from becoming globally ubiquitous desktop grids can be summarised in the following table:

Table 2.2: Problems with current grid computing architectures.

	LCG	Globus	Condor	PlanetLab	Eucalyptus	BOINC
lack of donor security	•		•	•	•	•
root privileges to install	•	•	•	•	•	•
lack of portable execution	•			•		•
requires firewall configuration	•	•	•	•	•	

2.3 A solution using Java applets

The first hurdle for a successful global desktop grid, that of donor security, can be solved by using an independent and trusted sandbox to run the client. A trusted sandbox can run arbitrary code in a safe and secure way. The two most common sandboxes for untrusted code are the Flash [47] and Java applet [48] sandboxes, both of which were released around 15 years ago.

A Java applet is a way of extending the functionality of static html webpages. When an applet is run, it is given space within the page to lay itself out and a sandbox to run its code within. The code (Java bytecode) is downloaded from a server and executes on the client machine.

Flash has had more than double the number of critical security patches released than that of Java in 2004-2010 according to the United States Computer Emergency Readiness Team [49]. A remote code execution vulnerability in Adobe Flash Player was the second most exploited vulnerability in 2009 according to Symantec [50].

Java, as well as being historically more secure, is installed in 79% of browsers [51]. This figure does not include all the other Java enabled devices connected to the Internet such as set top boxes and mobile phones, which could be set to donate only when they are plugged in to main power.

Java applets were introduced in 1995 in the first version of the Java Virtual Machine (JVM) and the Java applet “sandbox” was designed to run untrusted code from the Internet as a means to enhance the web browsing experience. The sandbox restrictions are summarised in Table 2.3. These restrictions are designed to address potential security threats from untrusted code execution [52]. Restrictions 1-3 are to protect the host machine from direct attacks that modify the machine or violate the privacy of the users of the machine. Restriction four is to ensure that new code cannot be loaded which circumvents existing security restrictions. Restrictions five and six prevent distributed denial of service (DDoS) attacks.

Table 2.3: Applet sandbox restrictions.

1.	no access to system properties
2.	no access to local file system
3.	cannot call native code (no access to Java Native interface (JNI))
4.	no custom classloaders - only URL classloaders allowed
5.	outbound networking is limited to TCP connections to the originating server only
6.	inbound networking is completely forbidden (cannot listen for connections on a socket)

Running the grid clients in the applet sandbox actually satisfies all of the requirements for a global grid:

1. Donors can be sure the applet code is contained and cannot harm their system.
2. There is no software installation required to donate (assuming that Java is installed).
3. Java is cross platform, designed for portable execution across multiple platforms, regardless of the underlying hardware or operating system.
4. An applet is downloaded from a webserver. The only requirement of the firewall around the donor is that it can see this webserver.

A global grid satisfying these properties would be a very large step forward, enabling most of the world’s idle CPU time to be safely harvested (compared to the current best attempt of $\sim$0.035%). However, as it stands, such a grid would only be suitable for trivially parallel tasks (tasks that can be split into many completely independent computations).

For more general problems, such as Map-Reduce discussed in Chapter 6, the individual units of computation need to be able to communicate with each other. The second applet sand-

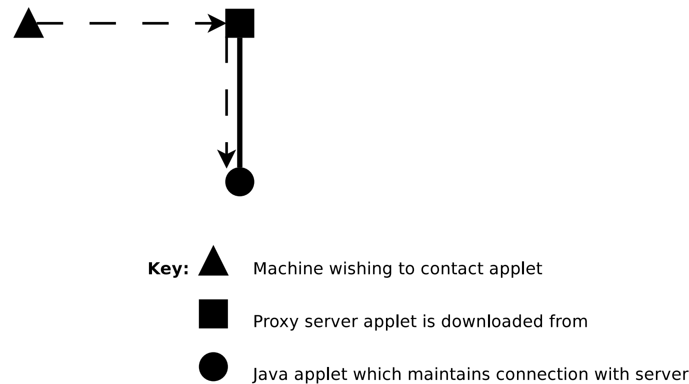


Figure 2.1: How a proxy server can facilitate incoming connections to Java applets.

box restriction listed above means that this communication is not directly possible. However, if a proxy server is run from the location the applet is downloaded from, it can forward communications to other clients. A proxy server is a connection forwarding service that takes incoming connections and forwards them on to the requested server, returning the response.

The receiving applet however cannot directly listen on a port, due to the sixth applet sandbox restriction. To facilitate communications, each client can maintain a connection with its proxy server. Then if the proxy server receives an incoming request to contact a donor, it can use this connection to forward the request, as illustrated in Figure 2.1. This then enables full two-way communication between arbitrary clients in a global applet grid without compromising the security features of the Java applet sandbox. All communications from an applet go through the server which can control the bandwidth used and thus prevent DDOS attacks.

Section 2.4 will discuss Nereus, a Java implementation of a grid satisfying the above requirements.

2.3.1 Java performance and suitability for scientific computing

Java is an interpreted, object-oriented programming language. Java source code files are compiled to Java class files containing Java bytecode. When a Java program is executed its bytecode is interpreted (and possibly dynamically compiled) by a Java Virtual Machine (JVM). Java source code is syntactically similar to C source code. For more information see http://www.java.com.

The suitability of Java for scientific computing has been investigated elsewhere [53][54], concluding that integer and floating point arithmetic in Java is comparable to optimised C and Fortran, but that highly communication intensive applications do not scale well in Java. The non-performance related issues include several floating point limitations: Java only supports Round-to-nearest, Java cannot trap IEEE floating point exceptions and Java only defines one bit pattern for NaNs. However, these should only affect a small minority of users.

Java performance is comparable to that of statically compiled languages C, C++ and FORTRAN [55][56]. Early on in Java's history, the optimising capabilities of commercial JVMs were limited, and thus Java acquired a reputation for being slow. Since then, every aspect of the Java runtime environment has improved. JVMs have a lot more information available to optimise a running Java program than a C compiler for an equivalent C program simply because the JVM compiler executes at runtime, rather than compile time, and can profile the Java code as it runs. Theoretically this gives a lot more scope for optimisation.

All commercial JVMs now have a powerful "just in time" (JIT) compiler. A JIT compiler can use the profiling information of the running program to spend more time optimising the frequently executed parts of the code and get maximum speed for the minimum compilation effort. JIT compilers will typically inline (replace a call to a method (function) with the code in that method) frequently called methods, removing the overhead of the method call and allowing for a broader scope for optimisation in the calling method because of the larger code base to optimise over.

An example of an optimisation possible with a JIT, but not with statically compiled code, is to inline virtual call targets. A virtual call is used in the case of an interface method invocation, where there may be multiple types that implement the given interface and hence different code for the particular method being called. A JVM can profile at runtime and decide that there is only one possible target for a given virtual call (as only one suitable class has been loaded) and completely inline the target's method, removing the call altogether. If at a later stage another possible candidate class for the interface call is loaded, the JVM can dynamically deoptimise the method inline and revert to the correct behaviour. This is not possible with a statically

compiled language.

Object allocation (acquiring a section of memory to store a structure containing data) is now very fast in modern JVMs and certainly is not noticably slower than stack allocation in C or C++ [55]. The Sun JIT compiler, Hotspot, will also do stack allocation of an object whenever it can.

Garbage collection now has parallel, low pause implementations. In concurrent programming, having an automatic garbage collector enables some highly scalable algorithms to be written. Such algorithms are at best impractical and at worst impossible using C++ and malloc/free [55].

Synchronization has also been heavily optimised. Synchronization is the protection of data structures from corruption due to simultaneous accesses from different threads. The simplest form of synchronization is to protect data with a “lock” object which prevents more than one thread from entering a code section at once. A decade ago the acquisition of a lock by a thread (whether or not other threads were trying to access it) was a computationally expensive process. Now this has been optimised to the point where uncontended locks (locks which are only acquired by one thread at runtime) have almost zero cost.

In summary, Java is a highly performant language and very suitable for scientific, CPU-intensive programming. An example of a real scientific program, BlackMax, is discussed in Chapter 7. BlackMax is originally written in C++ and was translated to Java resulting in a speed increase between 24% to 76%.

2.4 Nereus architecture

Nereus is a Java implementation of a computing grid designed to harness the world’s idle desktop computing in a secure and scalable way, enabling users to run Java bytecode on desktop machines in the applet sandbox. To use Nereus, a user modifies their code to use the Nereus API, compiles it to java bytecode, and uploads the result to a web server. A Nereus server can then be instructed to make all of its connected donor applets download the user’s code and

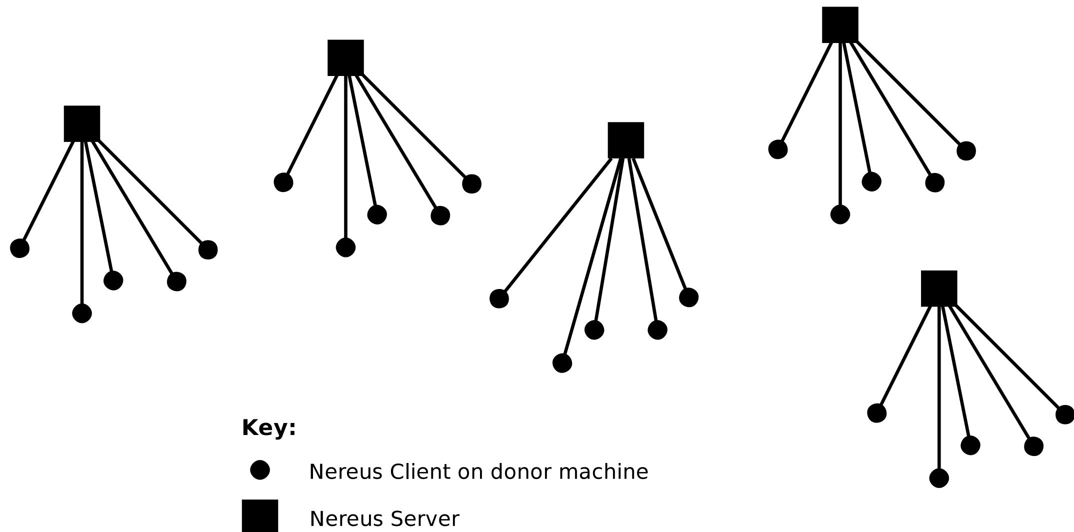


Figure 2.2: The global Nereus architecture: a forest of independent Nereus servers, each providing connectivity to a group of applets.

execute it. In this way, the user's code is executed across a large grid of machines.

A donor connects to a Nereus server by opening a web page which contains the Nereus applet. The Nereus applet runs at minimum thread priority, so that if a machine is in active use, Nereus is essentially unnoticable.

The Nereus Network topology is a forest of trees (Figure 2.2). This is similar to the domain name server topology which serves the Internet and is inherently scalable - more trees (Nereus servers) can always be added to the forest. In current tests and under typical use patterns, a single Nereus server can handle over 1000 simultaneously connected Nereus clients.

Applications suitable for running Nereus are ideally CPU-bound computations with small amounts of inter-node communication. The best latency and bandwidth that an application will typically see is when the donor machines are connected to Nereus servers on a Local Area Network with gigabit, or better, ethernet connections. A program running on a Nereus donor, a *service*, needs to implement the NereusService interface shown in Listing 2.1. A simple way to ensure this is to extend AbstractService and override the init() method. The computation can be done in the init() method, or more threads can be started from there. If the service needs to respond to HTTP requests, then it must override the handleConnection(InetAddress remoteAddress, HTTPRequest request, HTTPResponse response) method.

```java
/**
   The main interface needed to use Nereus client resources. A class which
       implements this interface can be created as the main entry point of a
        Nereid, however almost all implementations will extend
       AbstractService rather than implement this interface directly.
 */
public interface NereusService
{
    /** This will be called by the environment once an instance of this
        class has been loaded and instantiated. The ServiceContainer
        argument provides access to external resources.*/
    public void init(ServiceContainer container);

    public void yield(Container child, boolean isBeforeParentYield);

    public boolean allowedToListen(ServiceContainer child);

    public boolean allowedToDisplayComponent(Container child);

    public boolean allowedToCreateFrame(Container child);

    public boolean allowedToRequestSystemExit(Container child);

    public boolean allowedToRequestClientRestart(Container child);

    /** This will be called by the Nereus client to let the child container
         handle an incoming HTTP request - assuming incoming HTTP is
        permitted on this child and the child is currently listening for
        requests.*/
    public void handleConnection(InetAddress remoteAddress, HTTPRequest
        request, HTTPResponse response) throws IOException;

    public boolean filterConnection(ServiceContainer child, InetAddress
        remoteAddress, HTTPRequest request, HTTPResponse response) throws
        IOException;
}
```

Listing 2.1: The NereusService interface that running code needs to implement

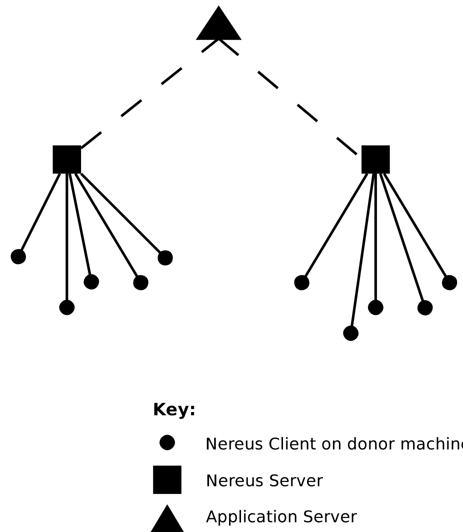


Figure 2.3: How a typical application server interacts with Nereus.

A typical application in Nereus will have its own application server (outside the Nereus network) to coordinate the computation, run by the project coordinator, illustrated in Figure 2.3. Here, an application server is simply an HTTP server that somehow manages the state of a distributed computation on nereus nodes. For example, typically it is useful to make new donor machines contact the appplication server to register themselves and wait for work tasks. It is at the application level that fault tolerance must be built in. Nereus provides access to the computational resources, and different applications using Nereus can implement their own policies for handling disappearance of donor nodes.

2.4.1 Nereus client

The code that a Nereus Client runs is defined by an XML-like document called the Nereus Markup Language (NML) file. The location of this is defined by the Nereus Server the client connects to. The Clients are found and accessed through the Nereus Server they are connected to. A typical NML file is shown in listing 2.2. Similar to the html <APPLET> tag, the service tag and its attributes (see Table 2.4) defines the service (program) that is to be run.

The code running in a Nereus client can be updated by changing the NML file. Each client will check the Nereus server to see if the NML file has changed (currently once a day), at which

Table 2.4: Service tag attributes.

Tag	Use
archive	source JAR(s) file location(s) - can be relative or absolute URLs, semi-colon separated
code	class containing entry point into code
id	a name that is displayed to the donor

```
<service id=main default=true code=org.myapp.MyClass archive="http://www.
    mywebserver.com/MyApp.jar">
 </service>
```

Listing 2.2: Typical NML document

point the old running code is shut down and the new code is downloaded and executed. This may be acceptable for many purposes, however, often code changes need to be propagated faster than within 24 hours. The Command Service was written for this purpose.

The Command Service is an optional base service that can be installed in Nereus clients to facilitate fast code redeployment. The Command Service maintains a connection to the Nereus server to enable contacting the client on demand, which then allows updated/changed code to be pushed out to the clients directly, as well as stopping running code remotely.

The only class loader allowed in an applet is a standard URL class loader pointing to the server the applet was downloaded from. In the case of a Nereus client the classloader can load code from anywhere on the Internet via the Nereus server's proxy mechanism.

There is no local file access allowed within applets, either reading or writing. Hence if an application is to store data locally, it has to store it in RAM. Note that this can't be used as a DDOS because any Java program is allocated a fixed maximum heap size which cannot be exceeded. In the case of an applet this heap size is determined by the browser.

2.4.2 Client communication

Most of the world's Internet-connected PCs are behind some kind of firewall. These will vary from the relatively free firewalls of Internet service providers to the extremely strict firewalls of large financial institutions. For any global computing grid to be successful, the computers need

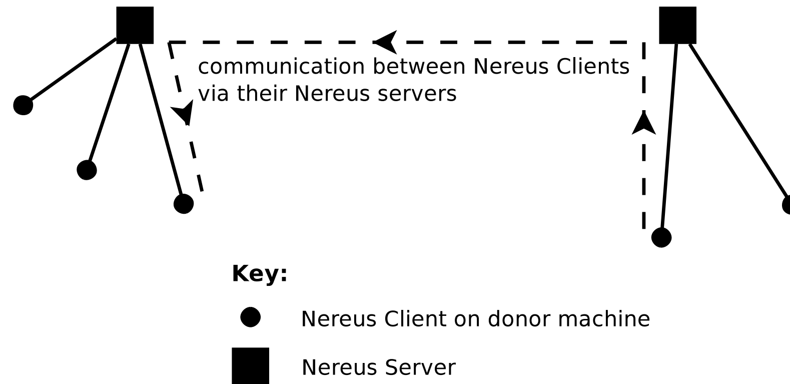


Figure 2.4: Communication between two Nereus Clients proceeds via the connections they each maintain to their Nereus Server.

to be able to connect to the Internet and communicate with each other. The easiest way to allow them to access the Internet is to use a proxy server.

The Nereus Server acts as an http proxy for all the clients connected to it. Applets are only allowed to make connections to the server they came from. The Nereus Server uses standard proxying techniques to forward these connections to the rest of the Internet, including other Nereus clients. Communication between Nereus clients is illustrated in Figure 2.4. This allows the vast majority of the world's PCs to contribute to a global compute cloud. The only requirement is that the donor machines are able to connect to the Nereus servers and the Nereus servers are able to access the Internet.

There are several advantages from this kind of topology. There appears to be a flat network structure. This means that any client can communicate with any other client anywhere in the world. More importantly it means that programmers are not unnecessarily burdened with concerns of the underlying network structure and can more easily focus on the functional structure of their programs.

The Nereus client allows Java bytecode to be run on the donated PCs. As a result of executing Java bytecode the following languages can also be used: Jython (a Java implementation of Python) [57], Clojure (a dialect of Lisp) [58][59], JRuby [60], Rhino (Javascript) [61], Scala [62][63], Groovy [64] and Fortress [65].

Chapter 3

The x86 architecture and virtualization

This chapter describes the complexity of the x86 architecture, defines the different types of virtualization and discusses the virtualizability of the x86 architecture.

3.1 x86 architecture

The x86 instruction set architecture (ISA) is extremely complicated [66][67][68][69][70], due to its three decade evolution during which it maintained backward compatibility (back to the original 8080 in 1974), although binary backward compatibility only goes as far back as the 8086. The x86 instruction length varies from one byte to 15 bytes. There are three main modes of operation, Real Mode, Protected Mode, and Virtual 8086 Mode; two operand sizes (16 and 32 bit); two address sizes (16 and 32 bit) and operation prefices to alter these, as well as two different memory address translation schemes. Modern x86 processors also have 64 bit mode of operation, however JPC initially only aims to implement up to a Pentium II processor (32 bit), which is enough to execute most modern scientific code. There is a 5th mode of operation, System Management Mode (SMM), which is a high privilege CPU operation mode that suspends all normal execution and where special code is run, normally firmware or a hardware-assisted debugger. There are six segment registers, whose interpretation depends on the mode of operation. There are four different "rings", or privilege levels, of execution

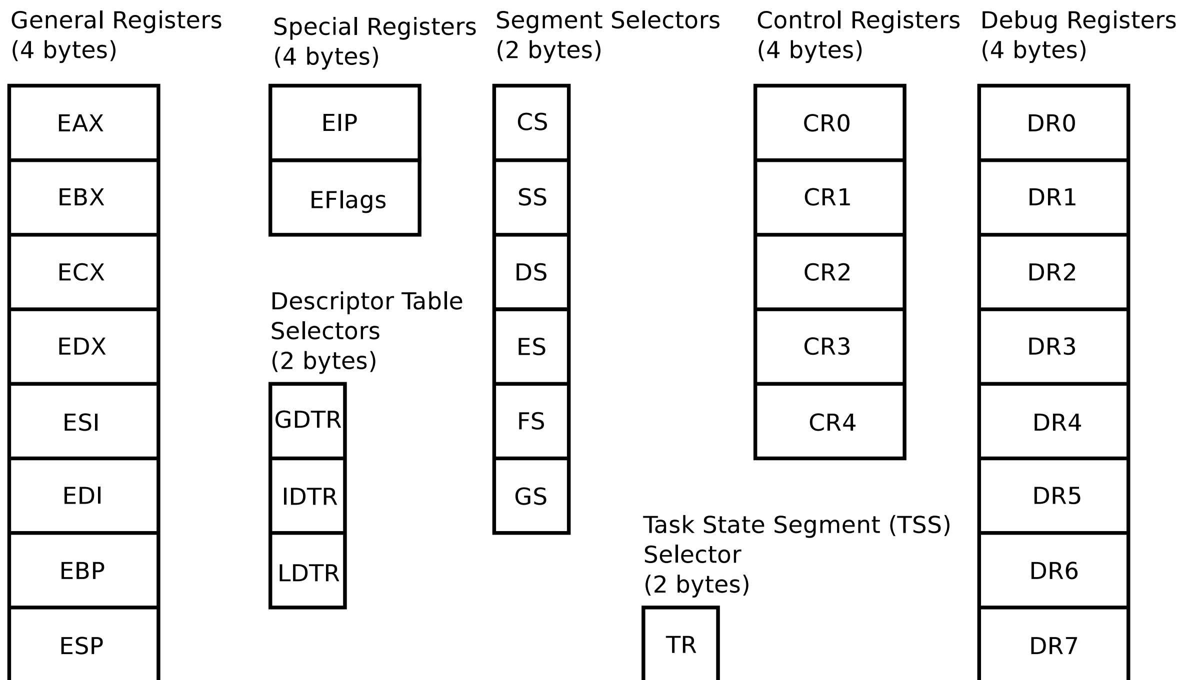


Figure 3.1: The basic internal structure of an x86 CPU

(zero to three). Operating systems normally run in the most privileged level, ring 0, whereas application software normally runs in the least privileged level, ring 3. Rings 1 and 2 are normally not used, although virtualizers sometimes run the guest operating system there. This complexity, combined with documentation that is sometimes incomplete and inaccurate, makes x86 execution very challenging to emulate.

The internal structure of a 32 bit x86 CPU, defined by Intel in [70], is shown in Figure 3.1. An x86 CPU has 8, 32-bit general registers– EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI– and a 32-bit instruction pointer, EIP. The six segment registers are CS, DS, ES, SS, FS, and GS. A *segment* is a window into a continuous section of memory that controls access rights. CS is the code segment, SS the stack segment, and the remaining segments are general data segments. There are three 16-bit segment selectors which point to descriptor tables, the GDTR, IDTR, and LDTR. The mode of operation of the processor is controlled by 5 control registers (CR0 - CR4). The task register selector (TR) points to the task state segment (TSS), which can store the state of a current task, facilitating multitasking. There are also debug registers which are used when the processor is remotely executed, one instruction at a time.

An x86 instruction has the structure shown in fig 3.2. The prefixes can modify the address

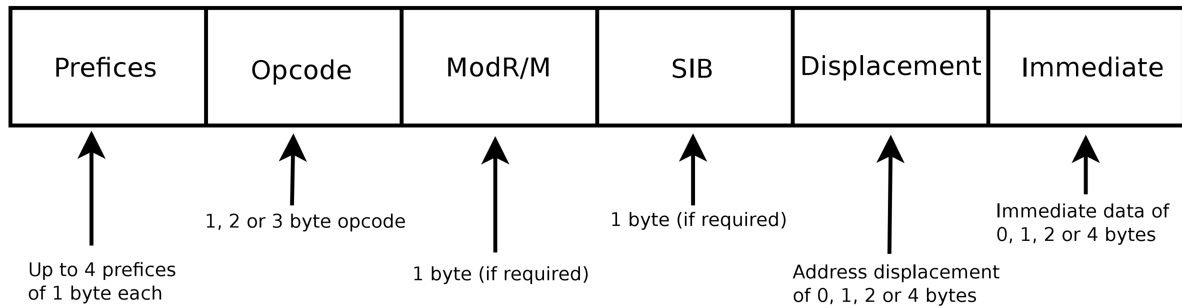


Figure 3.2: The x86 instruction format

size (16 or 32 bit), operand size (16 or 32 bit), make the instruction atomic and make all its effects visible simultaneously (LOCK prefix), make the operation repeat (REP prefix) or override which segment is used. The ModR/M byte is used to select which registers and addressing modes are used. The SIB byte modifies the addressing scheme used. The displacement is an offset which is added to the address used. The immediate byte is used for miscellaneous information required by different opcodes, for example, as the amount to add to EIP in a jump instruction.

The three main modes of operation are Real mode, Protected mode and Virtual 8086 mode. When the computer is turned on, it starts in Real mode with an instruction pointer of 0xFFFF0. This points to the start of the BIOS code which initializes the hardware, possibly with a memory test, before booting from an attached drive. At some stage in this process (in modern operating systems), a *far jump* is executed to switch to protected mode operation. The third mode, Virtual 8086 mode, is used in Protected mode to emulate the operation of a Real mode program, in a sandboxed manner. The reason this is necessary is that in Real mode there is no memory protection; all processes can access any section of memory. The Virtual 8086 Mode sandbox gives each task its own virtual, Real mode compatible, address space, and thus prevents different Virtual 8086 mode tasks from interfering with each other. All modern operating systems use Protected mode, however Real mode is still present and used at the beginning of the boot process for historical and compatibility reasons. Most old software should continue to run on modern hardware. This means that any x86 emulator must emulate all of these modes to be able to boot a modern operating system.

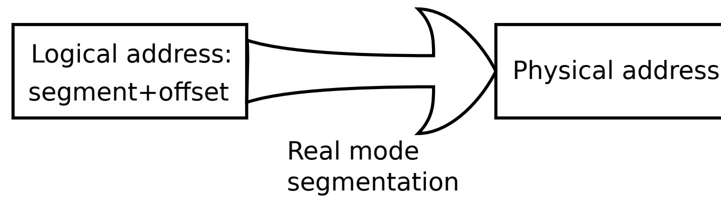


Figure 3.3: Real Mode address translation, which converts a logical memory address loaded by code to a real physical memory address

3.1.1 Real Mode

Real mode is the historical 16 bit operation mode that preceded the 32 bit modes introduced in the 80386. Real mode is characterised by having a 20 bit address space, giving 1 MiB of addressable memory. A segment is assigned a 2 byte selector (the actual value in the segment register) - a number from 0 to FFFF. *Real mode segmentation* uses a segment selector to determine the base of a segment by shifting it left 4 binary places. Then the particular byte in a segment is determined by adding a 2 byte offset to the base, shown in Figure 3.3. For example, the initial address of 0xFFFF0 is obtained from the code segment (CS) having a selector of 0xF000 and the EIP of 0xFFF0, which is the offset into the segment. In practice, slightly more than 1 MiB can be accessed. If a segment with selector 0xFFFF is accessed with an offset of 0xFFFF then this will point to the memory address 0xFFFF0 + 0xFFFF = 0x10FFEF. This gives an extra (64KiB -16B), which is called the High Memory Area. There is a slight proviso here in that an address line called the A20 line has to be set on for this to work, otherwise the higher addresses wrap around to zero.

On the whole, Real mode addressing is quite simple. The only complication comes from switching back to Real mode from Protected mode. This was a trick used by old software to obtain access to more memory. The program would switch to protected mode, load a segment with a base at an address larger than 1MiB, and switch back to Real mode. On the switch back to real mode the segment base is not reloaded (it is cached internally by the CPU in a segment descriptor for speed reasons), and thus the cached version is still used. This allows the program to address memory higher than the 1MiB barrier and is referred to as Extended Memory.

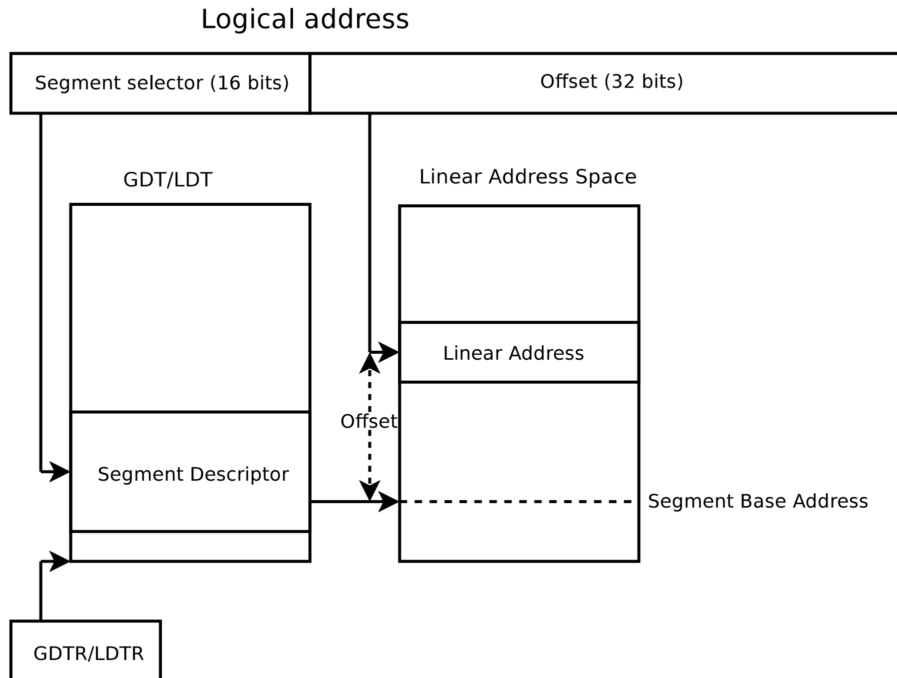


Figure 3.4: The x86 Protected mode segmentation mechanism, the first step of converting a logical memory address to a physical memory address

3.1.2 Protected Mode

Protected mode is a modern, 32 bit mode with full memory protection mechanisms and is used by all modern mainstream operating systems. Memory protection is implemented by introducing a new, indirect method of addressing memory, using *Protected mode segmentation* and *paging*. Segmentation converts a logical address (selector:offset) to a *linear address* and is illustrated in Figure 3.4. When a segment is loaded, the base of the segment is obtained in the following way: The segment selector (the value loaded into the segment register) is used as an index into a lookup table, either the Global Descriptor Table (GDT) or a Local Descriptor Table (LDT), according to bit 2 of the selector. This points to a 64 bit segment descriptor that determines various properties of the segment, including the (linear) base address and protection levels. This is called Protected mode segmentation and is always enabled. It is used mainly by the Windows operating systems. Each segment also has its own limit defined. If an access is attempted with an offset larger than the limit, a General Protection Exception is thrown.

The translation of a linear address to a physical address is called *paging*, and can be turned on or off. The full address translation process, including paging and segmentation, is illustrated

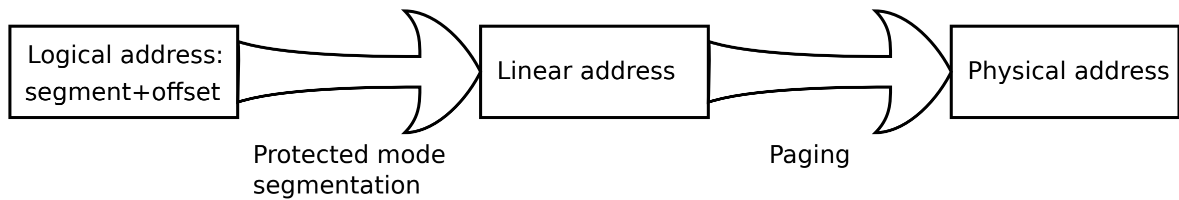


Figure 3.5: Protected Mode address translation, which converts a logical memory address loaded by code to a linear address, which is then further converted to a physical address

in Figure 3.5. If paging is turned off the linear and physical addresses are the same. Fundamentally, paging allows software to control the protection of pages and rearrange them. Paging is used by Linux to implement memory security and isolation without using the features of segmentation. This corresponds to a flat address space - all the segments are set to have a base of 0 and a limit of 4GiB. When a segment is accessed it is read or written via a 32-bit linear address. The 32-bit linear address is split into 3 sections of 10, 10, and 12 bits, shown in Figure 3.6. First, the processor's Control Register 3 (CR3) value is used to find the base of the Page Directory, a list of 1024 32-bit entries. The upper 10 bits of the linear address are used to select an entry in this table, which then points to a Page Table. A Page Table is a list of 1024 32-bit entries, each of which defines the physical location of a 4 KiB page of data. The next 10 bits of the linear address select an entry in the Page Table, which then points to an actual page in memory. The remaining 12 bits of the linear address are used as a direct offset into the selected page.

3.1.3 Interrupts

To further complicate the architecture there are hardware interrupts. An interrupt is a signal from a piece of hardware to the processor that there is some processing that needs to be done. When the processor sees that the flag for an interrupt has been raised then it saves its current state, jumps to a particular section of code to handle the interrupt and then reloads the saved state and resumes execution where it left off. Interrupts have an associated number that determines which handler code the processor jumps to.

The processor, in theory, needs to check if there is a pending interrupt after every single

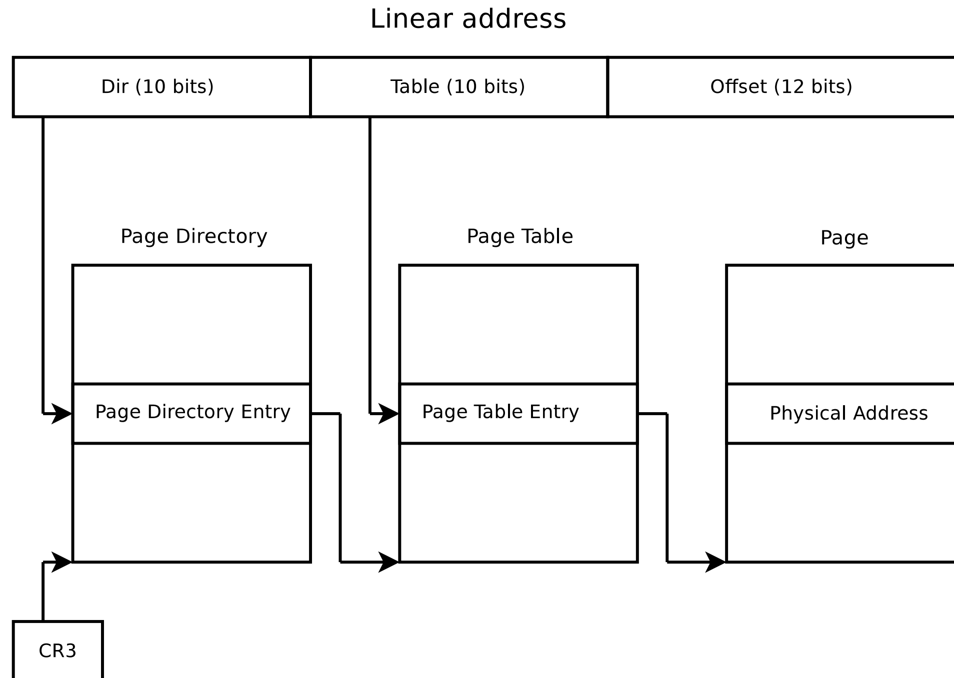


Figure 3.6: The paging process which converts a linear memory adress to a physical memory address

instruction. This adds a large overhead to the execution of a system emulator. In practice it is normally acceptable to check for interrupts only after each *basic block*.

A *basic block* is defined as a sequence of non-branching instructions that terminates with a branch instruction, or a HALT instruction. A *branch instruction* is one that modifies the instruction pointer so that it does not point to the first byte after the end of the current instruction.

3.1.4 Virtualization

Popek and Goldberg [71] originally defined a virtual machine (VM) as “an efficient, isolated duplicate of a real machine.” This definition has grown to include virtual machines which do not necessarily correspond to any real hardware, for example the Java Virtual Machine (JVM). Virtual machines have been investigated in depth elsewhere [72]. Virtualization is the means by which a virtual machine’s execution and state are managed. Emulation is a special case of virtualization where all the hardware of a machine is simulated in software, typically including the memory, disk drives, and processor. If the simulation is accurate enough, all software,

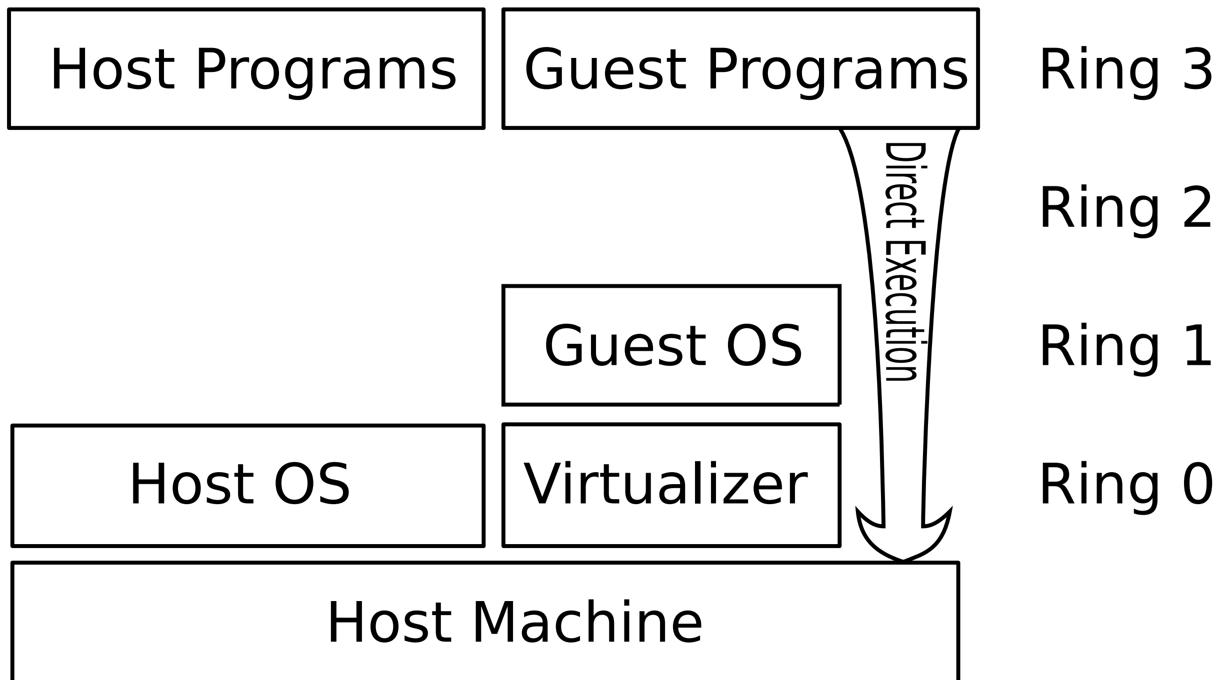


Figure 3.7: Full Virtualization runs most guest code directly on the physical CPU. Each layer in the diagram executes on the layer below it. The guest OS is executed using binary translation by the virtualizer.

including operating systems, can be run unmodified as if executing on a real machine. More general Virtualization comes in several other types which involve simulating all the hardware (excluding most of the CPU functionality) and running most guest code directly on the physical CPU. There are four different categories of virtualization: emulation and 3 types of *direct virtualization*.

1. Emulation
2. Full Virtualization
3. Hardware-assisted Virtualization
4. Paravirtualization

Full Virtualization, illustrated in Figure 3.7, relies on binary translation to trap and emulate certain sensitive, non-virtualizable instructions (see section 3.1.7). VirtualBox [73], VMWare [74] and Microsoft Virtual PC [75] can run as full Virtualizers. In this case, all operating system code is run at a privilege level with greater privilege than application code running in

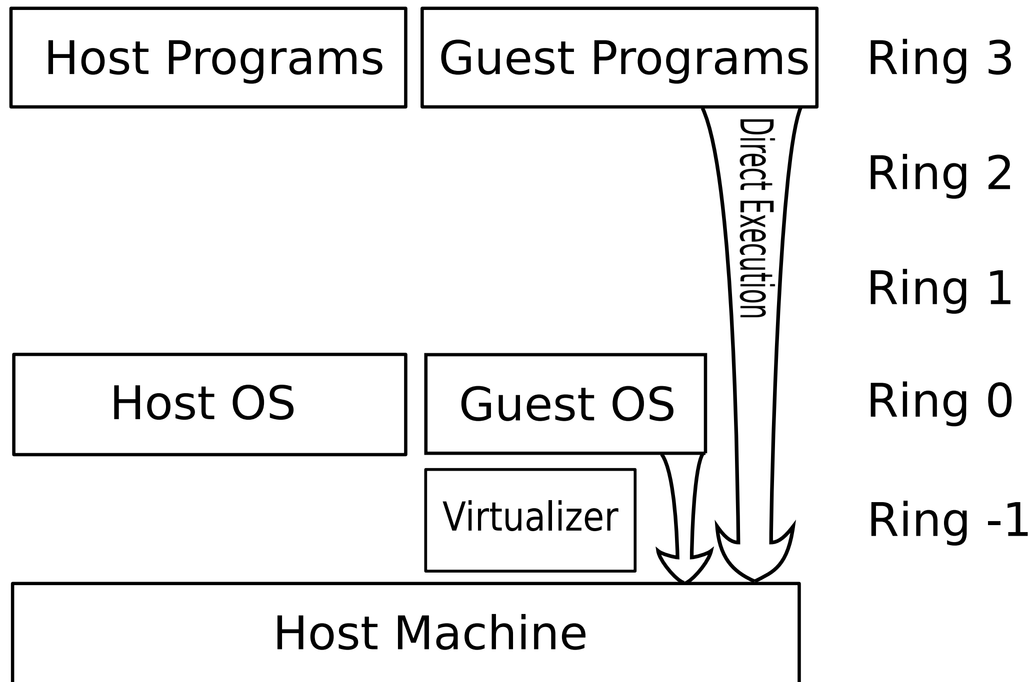


Figure 3.8: Hardware-assisted virtualization runs all guest code directly on the physical CPU. Sensitive or privileged instructions jump to the virtualizer to be emulated.

"ring" 3, but less privilege than the virtualizer running in ring 0, usually ring 1. Any privileged instructions then trap to the virtualizer which can emulate their effect. This process can have a serious speed penalty for commonly executed privileged instructions, but in some cases this can be alleviated by the virtualizer patching in code to directly handle the instruction without a trap. Due to the presence of sensitive non-virtualizable instructions all ring 0 code must be analysed before it is executed. VirtualBox, for example, disassembles the guest ring 0 code and patches in replacements for non-virtualizable instructions [76]. This process has a speed penalty for operating system code, typically 10-20% [77], however application level code is run directly on the real processor.

Hardware-assisted virtualization, illustrated in Figure 3.8, utilises instructions (VT-x and AMD-V) recently introduced by Intel and AMD specifically to aid virtualization. These instructions allow sensitive instructions to be virtualized without using binary translation. Operating system code is run in ring 0, and privileged or sensitive instructions call into the virtualizer which operates with greater privilege than ring 0. VMWare, VirtualBox, Xen [78] and Microsoft Hyper-V [79] can use hardware assisted virtualization.

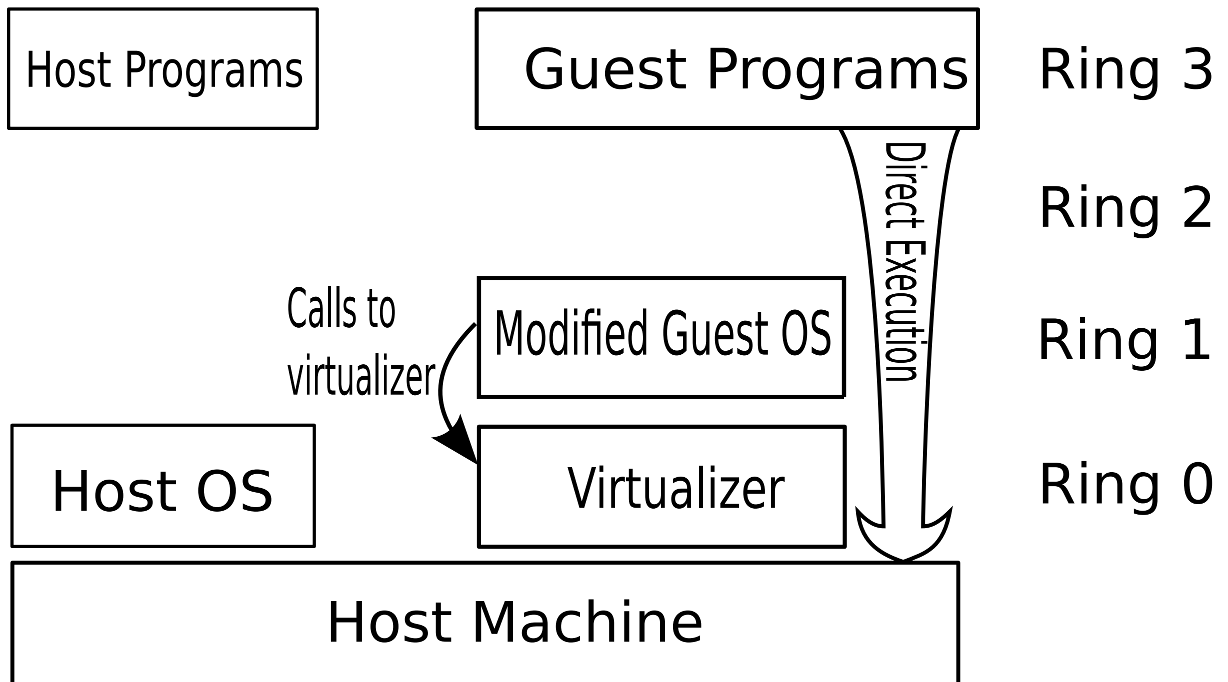


Figure 3.9: Paravirtualization runs most guest code directly on the physical CPU. The guest OS must be modified to replace non virtualizable instructions with calls to the virtualizer.

Table 3.1: x86 emulation vs other virtualization

	Emulation	**Direct Virtualization**
Speed (% native)	0.001*-20**	80-100
Secure	✓	✗
Portable to different hardware	✓	✗

* The first version of JPC was this speed
** JPC currently peaks at this speed

Paravirtualization, illustrated in Figure 3.9, is a software interface to a guest operating system that is similar, but not identical to the underlying physical hardware. It involves modifications of any guest operating systems to call the virtualizer (referred to as a hypercall) for tasks that are normally difficult to virtualize, such as network I/O. The guest operating system runs in ring 1 and the paravirtualizer runs in ring 0. For operating systems where the source code is not available (such as Windows), device drivers can be written that use the paravirtualization interface. Paravirtualization can improve efficiency of certain operations compared to full virtualization. Parallels [80], Xen and VMWare can act as paravirtualizers.

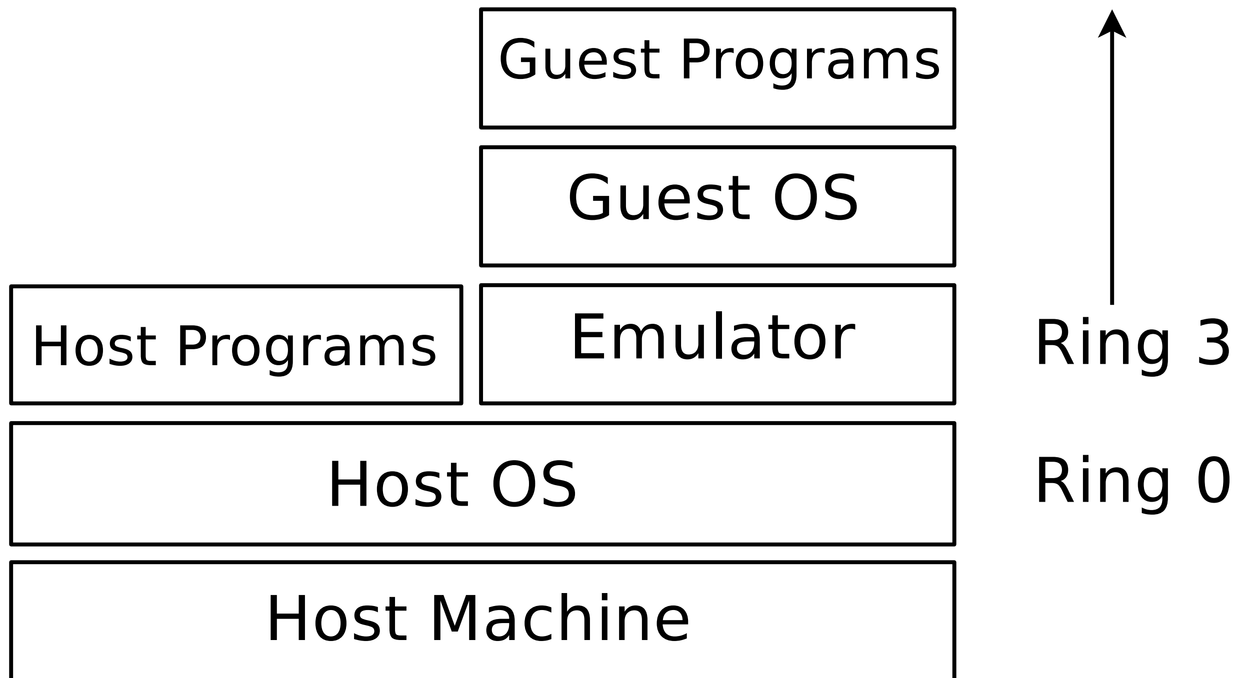


Figure 3.10: Emulation simulates all the hardware, including the CPU, in pure software. Each layer in the diagram executes on the layer below it.

In all three cases of *direct virtualization*, most of the guest application code runs directly on the physical CPU, which typically allows the code to run 90-98% native speed. A consequence of this is that direct virtualizers can only virtualize the physical hardware they are running on. For example, an x86 Virtualizer can only run x86 code on a real x86 processor. The increasing number of ARM, PowerPC and other non x86 based processors in general usage raises the importance of this point.

The main practical differences between emulation and general virtualization are summarised in Table 3.1. Virtualization gives speed of execution at the potential cost of security and portability across different hardware. Emulation, and particularly Java based emulation, enables security and portability at the cost of speed.

3.1.5 Emulation security

In theory, emulation allows complete security of execution to be achieved for guest code, as well as being undetectably different from physical hardware. This is because the guest code is not running directly on the real CPU, but on an emulated CPU. The execution of each instruction

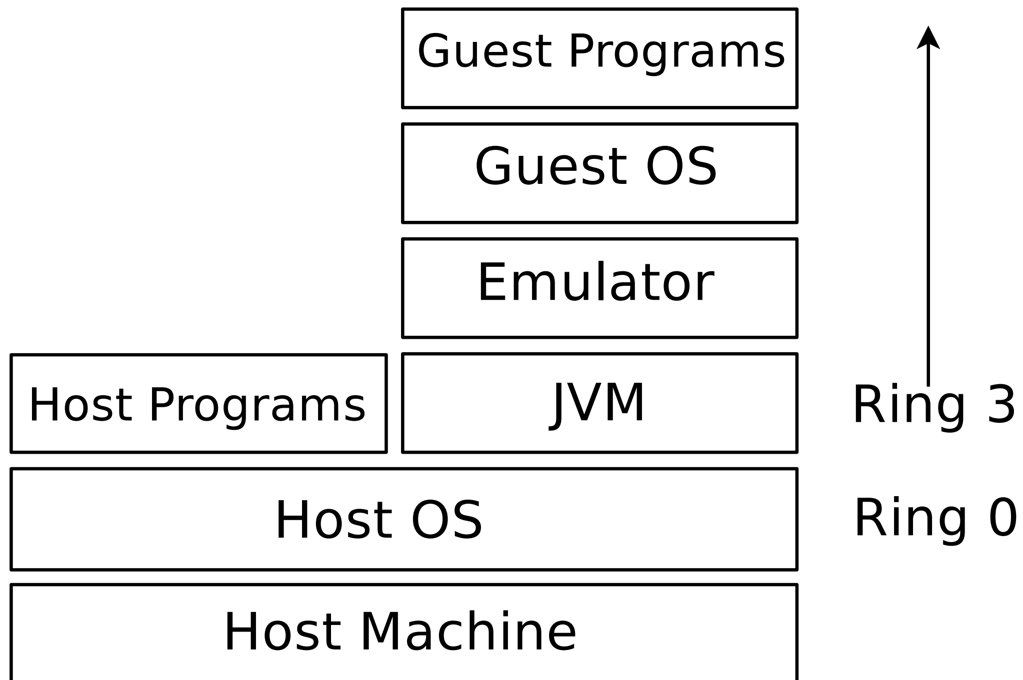


Figure 3.11: A Java emulator has an extra, independent layer of security.

is emulated using virtual CPU registers and memory. A security exploit would have to make use of a bug in the emulator, which could be accessed by guest code running in the emulator, to escape from the emulator sandbox. The emulator itself runs as standard application code with no extra privileges beyond ring 3.

If the emulator itself runs in a virtual machine or managed runtime environment there is a further layer of security. A managed runtime environment like Java eliminates most bugs involving memory management and buffer overflows, which are one of the most common vectors used in program compromises [81]. This extra layer of security is illustrated in Figure 3.11. The Java sandbox has been a trusted method to execute untrusted code (in applets) for over 10 years.

3.1.6 Direct Virtualization security

The security of direct virtualization is dependent on the security features of the physical processor being used. There is no extra layer (independent or otherwise) to protect the other programs on the host machine from those in the virtualizer. The addition of new processor instructions

by Intel and AMD specifically to aid virtualization have also introduced more potential avenues for exploitation.

There are two main kinds of attacks on virtualization. The first, less serious, type has been called ‘red pills’ [82] and enables software running within a virtualizer to determine that it is in fact running in a virtual machine. This is not a danger to other programs running on the host machine. However, this type of exploit can be used by malware to halt malicious behaviour in virtual machines. If malware is running inside a virtual machine, it is probably being analysed by a security researcher and thus should not demonstrate its attack strategies or functionality. The second much more serious type of attack is where a program running inside a virtualizer exploits a security flaw to escape from the virtual machine, so that its execution is no longer controlled by the virtualizer.

A red pill attack essentially amounts to detecting differences in the emulated hardware compared to real hardware. For example, the original red pill example shown by Joanna Rutkowska uses the SIDT instruction [82]. The SIDT instruction stores the contents of the interrupt descriptor table register (IDTR) to the target operand. The notable aspect about this instruction is that it is an unprivileged instruction that can be run in user mode without throwing an exception; thus, there is no way for a virtualizer to lazily emulate this by catching a privilege exception. All that is required by this red pill in advance is knowledge of where virtualizers store the IDTR and compare this to the result obtained.

The power of the red pill approach is that it can target any piece of the hardware. If, for example, the USB operation is not 100% identical with the corresponding real USB device being emulated, then that constitutes a possible red pill. In theory, these kinds of attacks can be eliminated by a virtualizer which correctly emulates all the hardware.

However, there is another kind of red pill attack based on time that is much harder to avoid. A time profile can be built of the execution time of various instructions on a particular real processor. These timings can then be compared to the recorded timing in a virtual machine to determine that the code is being virtualized. This works because of the overhead associated with running a virtual machine, which changes the absolute and relative timings of many instructions.

The most famous of the second, much more dangerous (and controversial), type of attack is the ‘blue pill’ [83][84]. This attack allows code running within a hardware-assisted virtual machine to take control of the host CPU. It uses weaknesses in the operation of the Intel VT-x or AMD-V CPU instructions and was first demonstrated in 2006 by Joanna Rutkowska with an attack through the Microsoft Windows Vista kernel. The blue pill traps a running instance of an operating system by creating a thin hypervisor which virtualizes the rest of the machine. As far as the OS is concerned, nothing has changed. However, now all the OS’s requests for resources are intercepted by the hypervisor. This kind of attack is extremely insidious as the host also cannot detect the exploit [83]. However, this is only one example of an exploit that escapes a virtual machine. The next paragraph discusses just how many such vulnerabilities are reported annually.

Full virtualizers can also be split into two types: type 1 (server) virtualizers run directly on a physical machine without requiring an operating system, whereas type 2 (workstation) virtualizers run as a program within a normal operating system using a kernel, which is loaded into the host. The potential attack surface for both these types are different. For example an attack that escapes a type 2 virtualizer may not be able to affect other concurrent VMs because it is still contained by the host operating system and its security enforcement. A risk report by IBM [85] listed vulnerabilities in the two types with very different frequencies, see Table 3.2. The report also states that there are around 100 vulnerabilities in virtualization code disclosed annually, over half of which are high and medium severity. High severity vulnerabilities tend to be easiest to exploit and provide full control over the attacked system, and thus virtualization vulnerabilities represent a significant security threat [85].

3.1.7 Virtualizability of x86

The virtualizability of an instruction set architecture (ISA) was first discussed by Popek and Goldberg [71]. They define a virtual machine (VM) as an efficient, isolated duplicate of a real machine. By ‘efficient’ it is meant that a statistically significant fraction of the instructions must be executed directly by the real processor and not emulated or interpreted in software.

Table 3.2: Vulnerabilities in virtualization system code for the two types of software virtualizers (server and workstation).

Vulnerability description	**Type 1**	**Type 2**
Vulnerabilities that allow an attacker to escape from a guest virtual machine to affect the host operating system on which the virtualization system is running.	N/A	24.1%
Vulnerabilities that allow an attacker to escape from a guest virtual machine to affect other virtual machines, or the hypervisor itself. In the case of Type 2 virtualizers, these vulnerabilities do not affect the host operating system.	35.0%	3.8%

Any program run in the VM must exhibit identical behaviour compared with the same program run on the real machine, excluding effects caused by varying availability of system resources or timing dependencies. They define a virtual machine monitor (VMM) as the software which provides the VM environment and controls its resources and execution. In the paper they prove this theorem:

> *For any conventional third generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.*

A *third generation computer* is a computer architecture with memory, user and supervisor modes, and a built-in exception trapping mechanism. A trap occurs when an instruction attempts to perform an illegal operation; either execute a privileged instruction in user mode or address memory outside the program's allowed range. When a trap occurs, the current state of the processor is saved and execution control is passed to a prespecified routine. Privileged instructions are those instructions that can only be executed in supervisor mode without trapping. Sensitive instructions come in several types:

1. Instructions that attempt to change or reference the mode of operation of the cpu.
2. Instructions that read or change sensitive registers or memory locations, for example, the clock register or interrupt flags.

3. Instructions that reference the storage or memory protection system, or address translation system.
4. All I/O instructions.

Robin and Irvine [86] discuss the implications of this theorem for virtualization of the x86 architecture, specifically the Intel Pentium architecture. Their conclusion is that the x86 architecture contains 17 sensitive unprivileged instructions and hence is not strictly virtualizable:

1. SGDT - store the value of the GDTR to a target register or memory location.
2. SIDT - store the value of the IDTR to a target register or memory location.
3. SLDT - store the value of the LDTR to a target register or memory location.
4. PUSHF - store the lower 16 bits of Eflags register to the stack
5. POPF - store the top 16 bits of the stack to the Eflags register
6. MOV - where used to load or store CPU control registers
7. LAR - loads the access rights from a memory segment into a target register or memory location
8. LSL - load the segment limit for a memory segment into a target register or memory location
9. VERR - verifies if a memory segment is readable from the current privilege level (CPL)
10. VERW - verifies if a memory segment is writable from the current privilege level (CPL)
11. POP - references the CPL when loading a segment register
12. PUSH - when used to store the CS register, containing the CPL, to the stack
13. CALL - an inter-privilege far CALL references the CPL and DPL
14. JMP - an inter-privilege far JMP references the CPL and DPL
15. INT n - jumps to an interrupt handler routine, but first stores the Eflags register to the stack
16. RET - an inter-privilege level far RET references the privilege levels and access rights of the code and stack segments
17. STR - stores the Task Register to a general register or memory location, thus allowing a task to examine its RPL

SIDT was mentioned earlier for its use in the red pill attack. PUSHF and POPF are other instructions in this category as the Eflags register contains sensitive flags which control the CPU operation like the Nested Task (NT) flag and the direction flag. The LAR, LSL, VERR, VERW are in the third unvirtualizable category for referencing the memory control system. POP, PUSH, CALL, JMP, INT n, RET, STR are also in the third category. These are all discussed in detail in [86].

This means that standard x86 virtualization designed to run unmodified x86 code must either be insecure, use significant emulation of instructions or use hardware-assisted virtualization (new CPU instructions). A virtual machine that is designed to run untrusted code on a network of computers must have bulletproof security. In light of the blue pill attacks on hardware-assisted virtualization, emulation is currently the only way to guarantee this for unmodified x86 code.

If the restriction to unmodified code is relaxed, programs can be recompiled to use a safe subset of the x86 intructions. This is the approach taken by the Google Native Client [87] sandbox.

3.1.8 Existing x86 emulators

There are several existing x86 emulators available, many of which are open source. The three most advanced are Bochs [88], Qemu [89] and DOSbox [90].

DOSBox is largely aimed at the legacy gaming community and thus focuses on peripherals such as sound emulation and historical graphics modes. It also is not accurate enough (there are still CPU emulation bugs) to boot Linux or modern Windows OSs.

Bochs is the most accurate emulator [91] and was released under the LGPL licence in 2000. Its focus is primarily on correct emulation of all the x86 instructions and hardware, as well as portability to different host architectures. Recent advances have also dramatically improved its speed.

Qemu is the most advanced emulator in terms of speed, using trace caching, binary translation and other techniques. It can either be run as a full emulator, or as a virtualizer (if the host CPU is x86). In full emulation mode it is a factor of two or three times faster than Bochs. Both Qemu and Bochs are accurate enough to boot modern Windows and Linux guests, although Qemu is not as accurate as Bochs due to remaining bugs in the CPU emulation.

Whilst Bochs, Qemu and DOSBox are excellent emulators and useful for comparison, they are not suitable for our purposes because they are written in C++. The JPC x86 emulator was designed to be able to run on the Nereus distributed computing network, which only runs Java code in the applet sandbox.

3.2 Potential speed of a Java x86 emulator

A ‘toy CPU’ was written in Java to estimate the speed attainable with a CPU emulator in Java. The toy CPU has 2 registers (*EAX* and *EBX*) as well as the instruction pointer, *IP*, and stack pointer *SP*, all of type *byte*. It has 15 instructions, and a working memory of 128 bytes, with the code starting at zero and the stack starting at 64. Practically this means the program instructions are stored in the first 64 bytes of “memory” and any variables used by the program are stored in the second 64 bytes of “memory”.

The first and slowest test involves an interpreter loop which looks up the current instruction, executes it, and repeats. The code used was an integer numeric benchmark, which has also been used by Cliff Click in the past [92], and is shown in listing 3.1. This algorithm was coded in six different versions of the CPU emulator. At each stage a different optimisation was applied. Version 1 of the emulator used a switch statement to read the instruction from the memory, perform the required action, and increment the *IP*.

Listing 3.2: Toy CPU 1 benchmark code: instructions are interpreted one by one in a lookup dispatch loop.

```
public class ToyCpu1
{
```

```
byte sum = 0;
    for (byte m=1; m < 101; m++)
        for (byte k=1; k < 101; k++)
            for (byte j=0; j < 100; j++)
                for (byte i=1; i < 101; i++)
                    sum += (sum^i)/i;
```

Listing 3.1: Toy CPU Benchmark code

```
    public static final byte HALT = 0;
    public static final byte JUMP = 1;
    public static final byte BNE = 2;
    public static final byte LOAD_SP = 3;
    public static final byte LOAD_EAX_MEM = 4;
    public static final byte LOAD_EAX_IB = 5;
    public static final byte STORE_EAX = 6;
    public static final byte LOAD_EBX_MEM = 7;
    public static final byte LOAD_EBX_IB = 8;
    public static final byte STORE_EBX = 9;
    public static final byte STORE_1 = 10;
    public static final byte INC = 11;
    public static final byte ADD = 12;
    public static final byte MUL = 13;
    public static final byte DIV = 14;
    public static final byte XOR = 15;

    public static void main(String[] args)
    {
        //first 64 bytes are code, second 64 bytes are stack/data
        byte[] mem = new byte[128];
        //write code
        byte b = 64;
        byte[] loop  = new byte[] {LOAD_SP, b, STORE_1, LOAD_SP, b,
                                   LOAD_EBX_MEM, LOAD_SP, (byte)(b+1),
                                   LOAD_EAX_MEM, XOR, DIV, LOAD_EBX_MEM,
                                   ADD, STORE_EAX};
        byte[] jump  = new byte[] {LOAD_SP, b, LOAD_EAX_MEM, INC,
                                   STORE_EAX, LOAD_SP, (byte)(b+2),
                                   LOAD_EBX_MEM, BNE, 3};
        byte[] jump2 = new byte[] {LOAD_SP, (byte)(b+3), LOAD_EAX_MEM, INC,
                                   STORE_EAX, LOAD_SP, (byte)(b+4),
                                   LOAD_EBX_MEM, BNE, 0};
        byte[] jump3 = new byte[] {LOAD_SP, (byte)(b+5), LOAD_EAX_MEM, INC,
                                   STORE_EAX, LOAD_SP, (byte)(b+6),
                                   LOAD_EBX_MEM, BNE, 0};
        byte[] jump4 = new byte[] {LOAD_SP, (byte)(b+7), LOAD_EAX_MEM, INC,
                                   STORE_EAX, LOAD_SP, (byte)(b+8),
                                   LOAD_EBX_MEM, BNE, 0};
        byte[] stack = new byte[] {1, 0, 101, 1, 101, 1, 101, 1, 101};
        System.arraycopy(loop, 0, mem, 0, loop.length);
        System.arraycopy(jump, 0, mem, loop.length, jump.length);
        System.arraycopy(jump2, 0, mem, loop.length + jump.length, jump2.
            length);
        System.arraycopy(jump3, 0, mem, loop.length + 2*jump.length, jump3.
            length);
```

```
        System.arraycopy(jump4, 0, mem, loop.length + 3*jump.length, jump4.
            length);
        System.arraycopy(stack, 0, mem, b, stack.length);

        byte ip = 0, eax=0, ebx=0, sp = b;
        cpu:
        while (true)
        {
            switch (mem[ip])
            {
            case HALT:
                break cpu;
            case JUMP:
                ip = mem[ip+1];
                break;
            case BNE:
                if (eax != ebx)
                    ip = mem[ip+1];
                else
                    ip += 2;
                break;
            case LOAD_SP:
                sp = mem[ip+1];
                ip += 2;
                break;
            case LOAD_EAX_MEM:
                eax = mem[sp];
                ip++;
                break;
            case LOAD_EAX_IB:
                eax = mem[ip+1];
                ip += 2;
                break;
            case STORE_EAX:
                mem[sp] = eax;
                ip++;
                break;
            case LOAD_EBX_MEM:
                ebx = mem[sp];
                ip++;
                break;
            case LOAD_EBX_IB:
                ebx = mem[ip+1];
                ip += 2;
                break;
            case STORE_EBX:
                mem[sp] = ebx;
                ip++;
                break;
            case STORE_1:
                mem[sp] = 1;
                ip++;
                break;
            case INC:
                eax++;
                ip++;
                break;
            case ADD:
```

```
                eax += ebx;
                ip++;
                break;
            case MUL:
                eax *= ebx;
                ip++;
                break;
            case DIV:
                eax /= ebx;
                ip++;
                break;
            case XOR:
                eax ^= ebx;
                ip++;
                break;
            default:
                System.out.println("Illegal opcode: " + mem[ip] + " at ip "
                                  + ip);
                break cpu;
            }
        }
        System.out.println("Sum = " + mem[65]);
    }
}
```

Switch statements, particularly ones with many possible options, are very hard for a JVM to optimise. Version 2 of the toy CPU removed the switch by writing out the algorithm in toy assembly code.

Listing 3.3: Toy CPU 2 benchmark code: lookup dispatch loop is removed by writing instructions out in sequential 'toy assembly code'

```
public class ToyCpu2
{
    ...

    public static void main(String[] args)
    {
        ...

        //eliminate lookup dispatch by writing out toy assembly code
        cpu:

        while (true)
        {
            while (true)
            {
                while (true)
                {
                    sp = mem[ip+1];
                    ip += 2;
                    mem[sp] = 1;
                    ip++;
                    while (true)
```

```
        {
            sp = mem[ip+1];
            ip += 2;
            ebx = mem[sp];
            ip++;
            sp = mem[ip+1];
            ip +=2;
            eax = mem[sp];
            ip++;
            eax ^= ebx;
            ip++;
            eax /= ebx;
            ip++;
            ebx = mem[sp];
            ip++;
            eax += ebx;
            ip++;
            mem[sp] = eax;
            ip++;
            //jump
            sp = mem[ip+1];
            ip += 2;
            eax = mem[sp];
            ip++;
            eax++;
            ip++;
            mem[sp] = eax;
            ip++;
            sp = mem[ip+1];
            ip += 2;
            ebx = mem[sp];
            ip++;
            if (eax != ebx)
                ip = mem[ip+1];
            else
            {
                ip += 2;
                break;
            }
        }
        //jump2
        sp = mem[ip+1];
        ip += 2;
        eax = mem[sp];
        ip++;
        eax++;
        ip++;
        mem[sp] = eax;
        ip++;
        sp = mem[ip+1];
        ip += 2;
        ebx = mem[sp];
        ip++;
        if (eax != ebx)
            ip = mem[ip+1];
        else
        {
            ip += 2;
```

```
                        break;
                    }
                }
                //jump3
                sp = mem[ip+1];
                ip += 2;
                eax = mem[sp];
                ip++;
                eax++;
                ip++;
                mem[sp] = eax;
                ip++;
                sp = mem[ip+1];
                ip += 2;
                ebx = mem[sp];
                ip++;
                if (eax != ebx)
                    ip = mem[ip+1];
                else
                {
                    ip += 2;
                    break;
                }
            }
            //jump4
            sp = mem[ip+1];
            ip += 2;
            eax = mem[sp];
            ip++;
            eax++;
            ip++;
            mem[sp] = eax;
            ip++;
            sp = mem[ip+1];
            ip += 2;
            ebx = mem[sp];
            ip++;
            if (eax != ebx)
                ip = mem[ip+1];
            else
            {
                ip += 2;
                break;
            }
        }
        System.out.println("Sum = " + mem[65]);
    }
}
```

Most code is not self modifying - Intel actively discourages the use of self modifying code, and thus constants which would normally be read from memory can be hardcoded into each instruction. Version 3 of the toy CPU removed these constant reads from the instruction stream and hard coded these into the toy assembly (for example, immediate bytes).

Listing 3.4: Toy CPU 3 benchmark code: removes reads of constants from the instruction stream and hard codes them.

```
public class ToyCpu3
{
    ...

    public static void main(String[] args)
    {
        ...

        //remove reading constants from code stream
        // this removes reads from the code area of memory
        cpu:

        while (true)
        {
            while (true)
            {
                while (true)
                {
                    sp = b;
                    ip += 2;
                    mem[sp] = 1;
                    ip++;
                    while (true)
                    {
                        sp = b;
                        ip += 2;
                        ebx = mem[sp];
                        ip++;
                        sp = (byte)(b+1);
                        ip +=2;
                        eax = mem[sp];
                        ip++;
                        eax ^= ebx;
                        ip++;
                        eax /= ebx;
                        ip++;
                        ebx = mem[sp];
                        ip++;
                        eax += ebx;
                        ip++;
                        mem[sp] = eax;
                        ip++;
                        //jump
                        sp = b;
                        ip += 2;
                        eax = mem[sp];
                        ip++;
                        eax++;
                        ip++;
                        mem[sp] = eax;
                        ip++;
                        sp = (byte)(b+2);
                        ip += 2;
                        ebx = mem[sp];
                        ip++;
```

```
                    if (eax != ebx)
                        ip = 3;
                    else
                    {
                        ip += 2;
                        break;
                    }
                }
                //jump2
                sp = (byte)(b+3);
                ip += 2;
                eax = mem[sp];
                ip++;
                eax++;
                ip++;
                mem[sp] = eax;
                ip++;
                sp = (byte)(b+4);
                ip += 2;
                ebx = mem[sp];
                ip++;
                if (eax != ebx)
                    ip = 0;
                else
                {
                    ip += 2;
                    break;
                }
            }
            //jump3
            sp = (byte)(b+5);
            ip += 2;
            eax = mem[sp];
            ip++;
            eax++;
            ip++;
            mem[sp] = eax;
            ip++;
            sp = (byte)(b+6);
            ip += 2;
            ebx = mem[sp];
            ip++;
            if (eax != ebx)
                ip = 0;
            else
            {
                ip += 2;
                break;
            }
        }
        //jump4
        sp = (byte)(b+7);
        ip += 2;
        eax = mem[sp];
        ip++;
        eax++;
        ip++;
        mem[sp] = eax;
```

```
                ip++;
                sp = (byte)(b+8);
                ip += 2;
                ebx = mem[sp];
                ip++;
                if (eax != ebx)
                    ip = 0;
                else
                {
                    ip += 2;
                    break;
                }
            }
            System.out.println("Sum = " + mem[65]);
        }
}
```

The interleaved IP increment instructions make it more difficult for a JVM's compiler to optimise code, and so Version 4 removed these increments, and only incremented IP immediately before each potential control flow change.

Listing 3.5: Toy CPU 4 benchmark code: removes interleaved increments to the instruction pointer (ip) and only updates ip before a control flow change.

```
public class ToyCpu4
{
    ...

    public static void main(String[] args)
    {
        ...

        //remove interleaved IP increment instructions
        cpu:

        while (true)
        {
            while (true)
            {
                while (true)
                {
                    sp = b;
                    mem[sp] = 1;
                    ip += 3;
                    while (true)
                    {
                        sp = b;
                        ebx = mem[sp];
                        sp = (byte)(b+1);
                        eax = mem[sp];
                        eax ^= ebx;
                        eax /= ebx;
                        ebx = mem[sp];
                        eax += ebx;
```

```
                    mem[sp] = eax;
                    //jump
                    sp = b;
                    eax = mem[sp];
                    eax++;
                    mem[sp] = eax;
                    sp = (byte)(b+2);
                    ebx = mem[sp];
                    if (eax != ebx)
                        ip = 3;
                    else
                    {
                        ip += 21;
                        break;
                    }
                }
                //jump2
                sp = (byte)(b+3);
                eax = mem[sp];
                eax++;
                mem[sp] = eax;
                sp = (byte)(b+4);
                ebx = mem[sp];
                if (eax != ebx)
                    ip = 0;
                else
                {
                    ip += 10;
                    break;
                }
            }
            //jump3
            sp = (byte)(b+5);
            eax = mem[sp];
            eax++;
            mem[sp] = eax;
            sp = (byte)(b+6);
            ebx = mem[sp];
            if (eax != ebx)
                ip = 0;
            else
            {
                ip += 10;
                break;
            }
        }
        //jump4
        sp = (byte)(b+7);
        eax = mem[sp];
        eax++;
        mem[sp] = eax;
        sp = (byte)(b+8);
        ebx = mem[sp];
        if (eax != ebx)
            ip = 0;
        else
        {
            ip += 10;
```

```
                break;
            }
        }
        System.out.println("Sum␣=␣" + mem[65]);
    }
}
```

Version 5 of the toy CPU recognised loops and compiled them as such. This further improves the branch prediction of the JVM and underlying processor for this code.

Listing 3.6: Toy CPU 5 benchmark code: recognises loops and compiles them as such.

```
public class ToyCpu5
{
    ...

    public static void main(String[] args)
    {
        ...

        //recognise loops and loop counters
        for (byte m = 1; m < 101;)
        {
            for (byte k = 1; k < 101;)
            {
                for (byte j = 1; j < 101;)
                {
                    sp = b;
                    mem[sp] = 1;
                    for (byte i = 1; i < 101;)
                    {
                        ebx = i;
                        sp = (byte)(b+1);
                        eax = mem[sp];
                        eax ^= ebx;
                        eax /= ebx;
                        ebx = mem[sp];
                        eax += ebx;
                        mem[sp] = eax;
                        sp = b;
                        i++;
                        mem[sp] = i;
                    }
                    //jump2
                    j++;
                    sp = (byte)(b+3);
                    mem[sp] = j;
                }
                //jump3
                k++;
                sp = (byte)(b+5);
                mem[sp] = k;
            }
            //jump4
            m++;
```

```
            sp = (byte)(b+7);
            mem[sp] = m;
        }
        ip += 54;
        System.out.println("Sum = " + mem[65]);
    }
}
```

Version 6 assumes aggressive optimisations and eliminates unnecessary movement of data into/out of registers.

Listing 3.7: Toy CPU 6 benchmark code: assumes aggressive optimisations and remove unecessary movement of data between registers.

```
public class ToyCpu6
{
    public static byte sum()
    {
        byte[] mem = new byte[128];
        for (byte m = 1; m<101; m++)
        {
            mem[71] = m;
            for (byte k=1; k<101; k++)
            {
                mem[69] = k;
                for (byte j = 1; j<101; j++)
                {
                    mem[67] = j;
                    for (byte i = 1; i < 101; i++)
                    {
                        mem[64] = i;
                        mem[65] += (mem[65] ^ i) /i;
                    }
                }
            }
        }
    }
}
```

The different toy CPUs were timed on two computers: Machine A, a 32 bit Intel Pentium M 760, 2 GHz with 2MiB L2 cache and Machine B, a 64 bit Intel Core 2 Duo (E8400) 3 GHz with 6MiB L2 cache. The results for the two machines are illustrated in Figure 3.12. The difference between toy CPUs 4 and 5 is striking, giving roughly a factor of eight speed up. This is not surprising as this is the step that allows the JVM to see the looping structure, and allows many more optimisations.

For comparison this algorithm was coded in C (see listing 3.8) and compiled with different

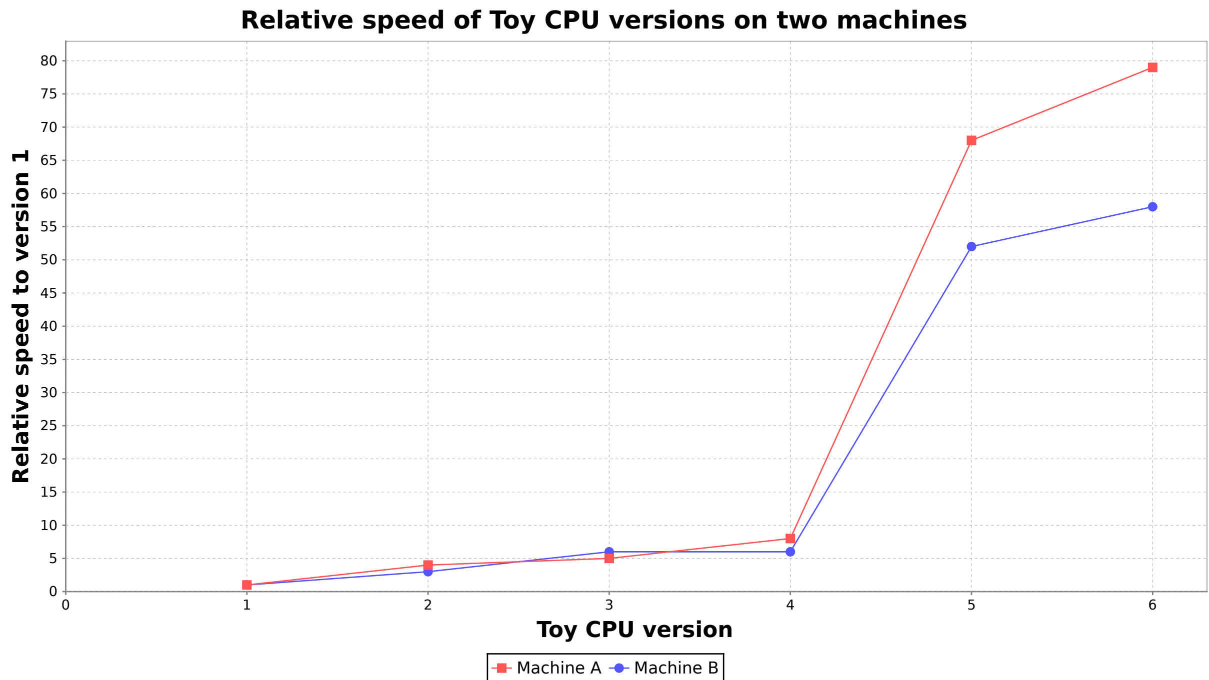


Figure 3.12: The relative speeds of the different toy CPU implementations on Machine A: 2 GHz, 32 bit Intel Pentium M 760, and Machine B: 3 GHz, 64 bit Intel Core 2 Duo (E8400)

levels of optimisation and timed on two different machines described in Table 3.3. On Machine A, the timings were essentially the same until optimisation level 3 was used. On Machine B, the timings gradually improved up to their best at optimisation level 2. The results are shown in Table 3.3.

```
#include <stdio.h>

int main()
{
  signed char sum, i, j, k, m;
  sum = 0;
  for(m=1; m < 101; m++ )
       for(k=1; k< 101; k++ )
            for(j=1; j < 101; j++ )
                 for(i=1; i < 101; i++ )
                      sum += (sum^i)/i;
  return(sum);
}
```

Listing 3.8: C version of Toy CPU Benchmark code

Table 3.3: Native code timings: Machine A is a 32 bit Intel Pentium M 760, 2 GHz with 2MiB L2 cache. Machine B is a 64 bit Intel Core 2 Duo (E8400) 3 GHz with 6MiB L2 cache.

Optimisation Level	**Time (mS)**	
	Machine A	Machine B
O0 (default)	1611±4	932±1
O1	1647±4	769±1
O2	1628±4	748±1
O3	1026±2	748±1
O4	1027±2	748±1

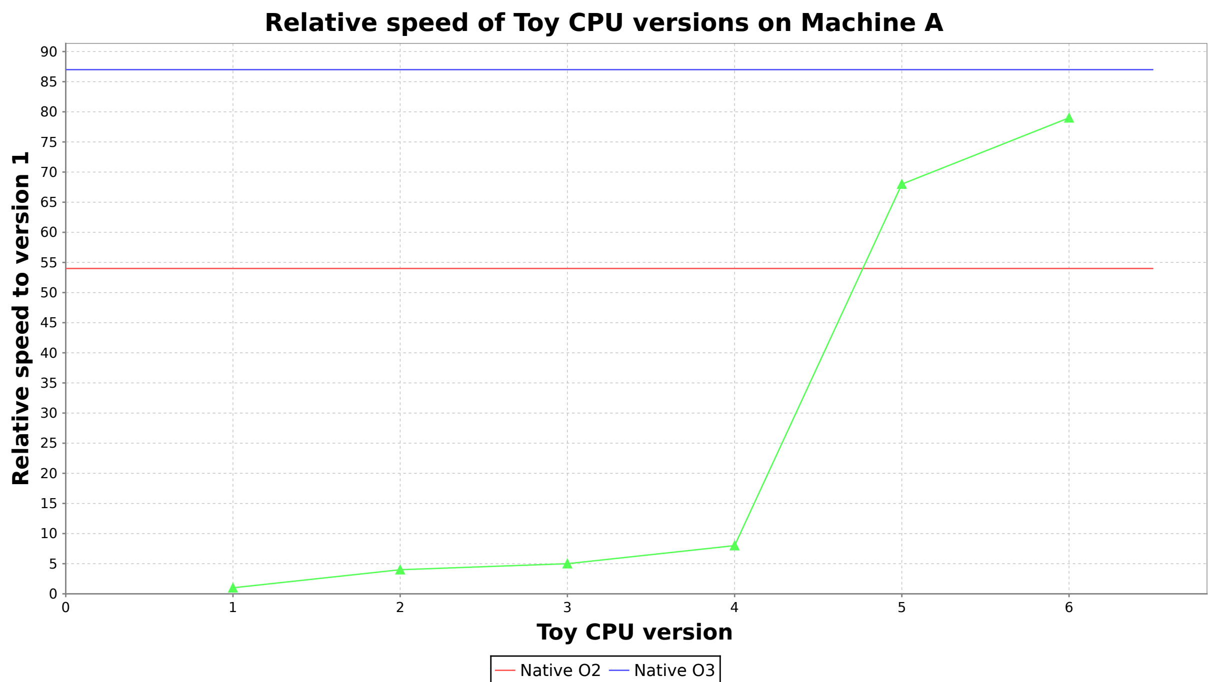


Figure 3.13: The relative speeds of the different toy CPU implementations on Machine A: 2 GHz, 32 bit Intel Pentium M 760

The timings of the toy CPUs and their comparison to native speed are listed in Table 3.4 and illustrated in Figures 3.13 and 3.14.

One can see from these results that a CPU emulator written in Java can perform extremely well. Many scientific projects use the default optimisation level, with some venturing up to level 2. Almost no large projects use optimisation level 3 during compilation. This means that an emulator actually has a very real possibility of being faster than native speed. Thus, an x86 emulator written in Java not only offers significant security and portability benefits, but also the possibility of unexpected performance gains.

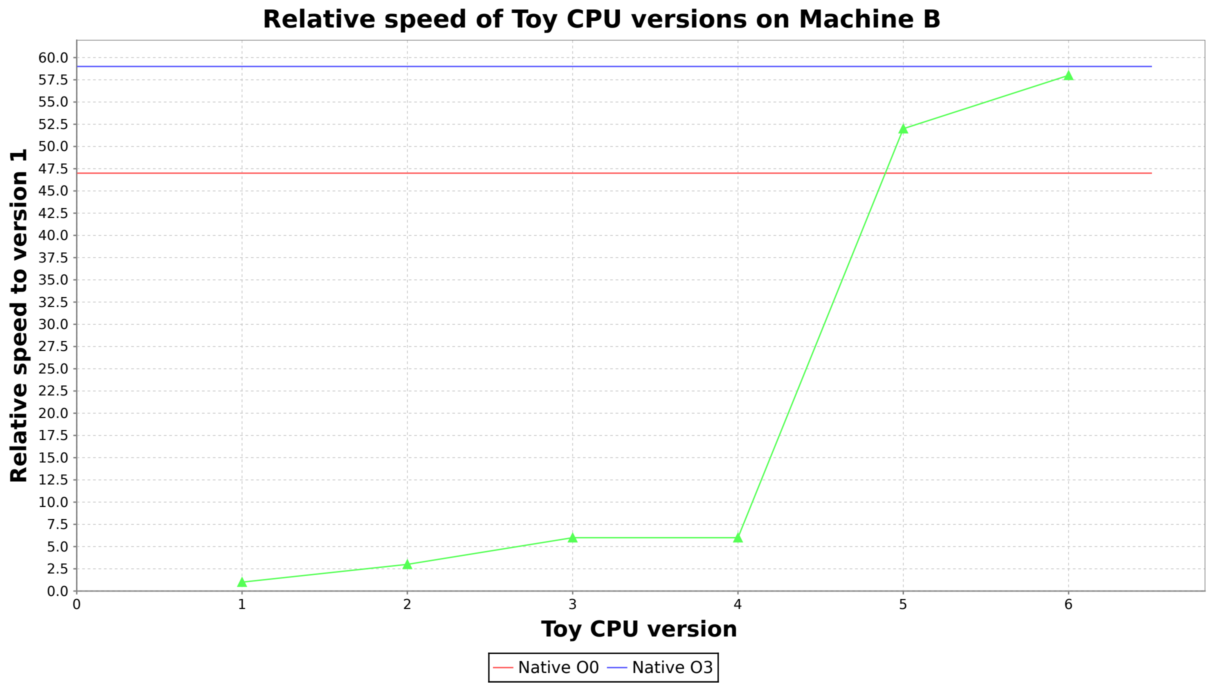


Figure 3.14: The relative speeds of the different toy CPU implementations on Machine B: 3 GHz, 64 bit Intel Core 2 Duo (E8400)

Table 3.4: Toy CPU emulator timings: Machine A is a 32 bit Intel Pentium M 760, 2 GHz with 2MiB L2 cache. Machine B is a 64 bit Intel Core 2 Duo (E8400) 3 GHz with 6MiB L2 cache.

Toy CPU	**Machine A**			**Machine B**		
	Time (mS)	% native O2 speed	% native O3 speed	Time (mS)	% native speed	% native O2 speed
1	89482±9	1.8	1.1	44713±2	2.1	1.7
2	22265±7	7.2	4.6	11962±2	7.8	6.2
3	17342±7	9.2	5.9	6452±4	14.4	11.6
4	10625±4	15.2	9.6	6452±4	14.4	11.6
5	1301±3	124	79	853±1	109	87.7
6	1121±3	143	92	767±1	122	97.5

Chapter 4

JPC architecture and implementation

JPC is a Java based emulator of the Intel x86 processor architecture [93][94]. It is cross platform, operates inside the Java sandbox and uses dynamic compilation to achieve reasonable runtime performance. This chapter describes the structure of JPC and the various compromises in design of speed versus memory usage.

The structure of JPC largely mirrors the structure of an actual computer and its peripherals, for which a typical example is shown in Figure 4.1 and schematically in Figure 4.2. The Northbridge is a high speed interconnect between the cpu and the memory. The Southbridge is a relatively slower interconnect to the peripheral devices, using the PCI Bus, ISA Bus, or the USB Bus. The Southbridge also contains the Realtime Clock, Interrupt Controller and Interval Timer. The hard drives are accessed through the PCI Bus via an IDE Interface.

The actual architecture of JPC is shown schematically in Figure 4.3. Each box represents a hardware component which is emulated using a Java class. The arrows represent lines of control flow, and all originate from the CPU in one way or another. Note that because the North and South bridges are connections, they are not directly represented, but manifest only in the links between components.

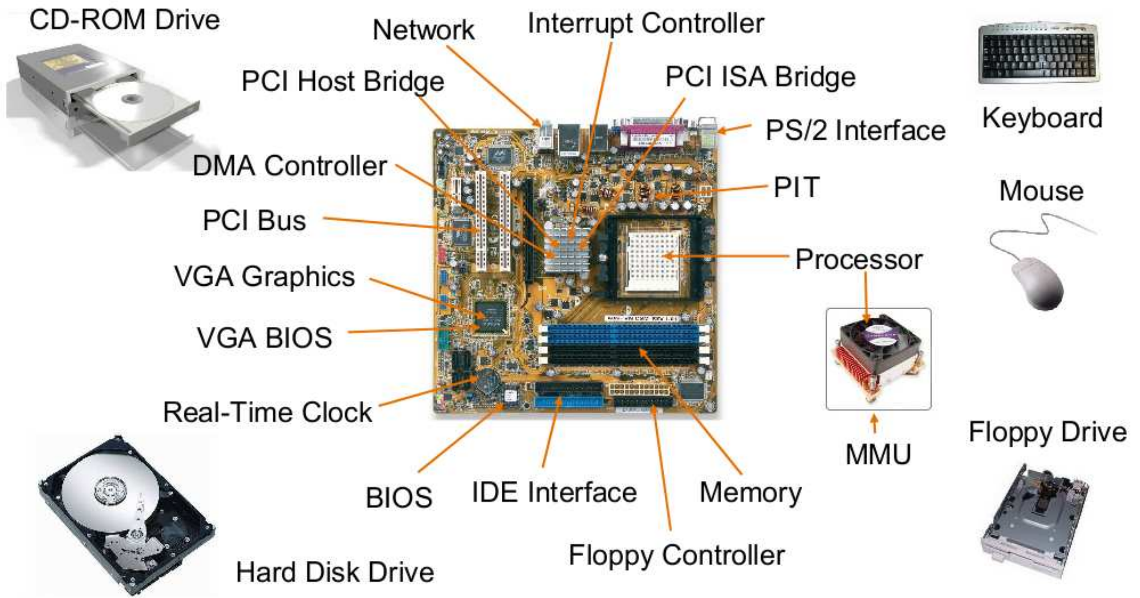


Figure 4.1: A generic x86 PC architecture

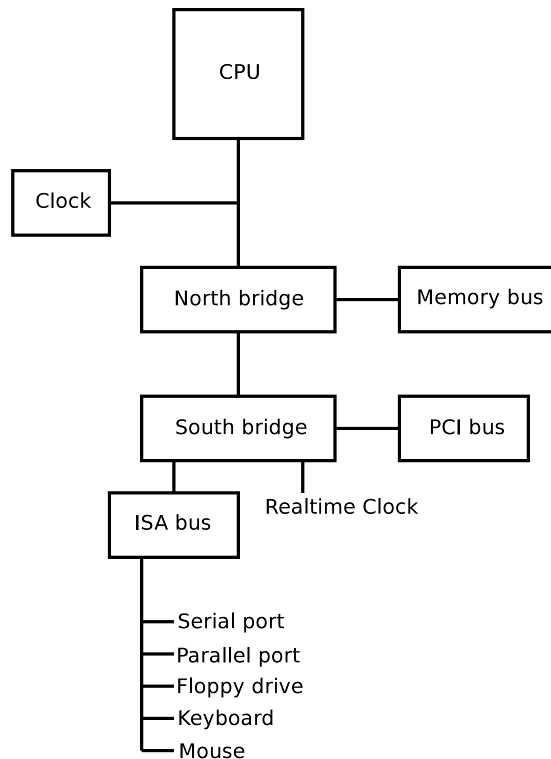


Figure 4.2: A schematic x86 PC architecture

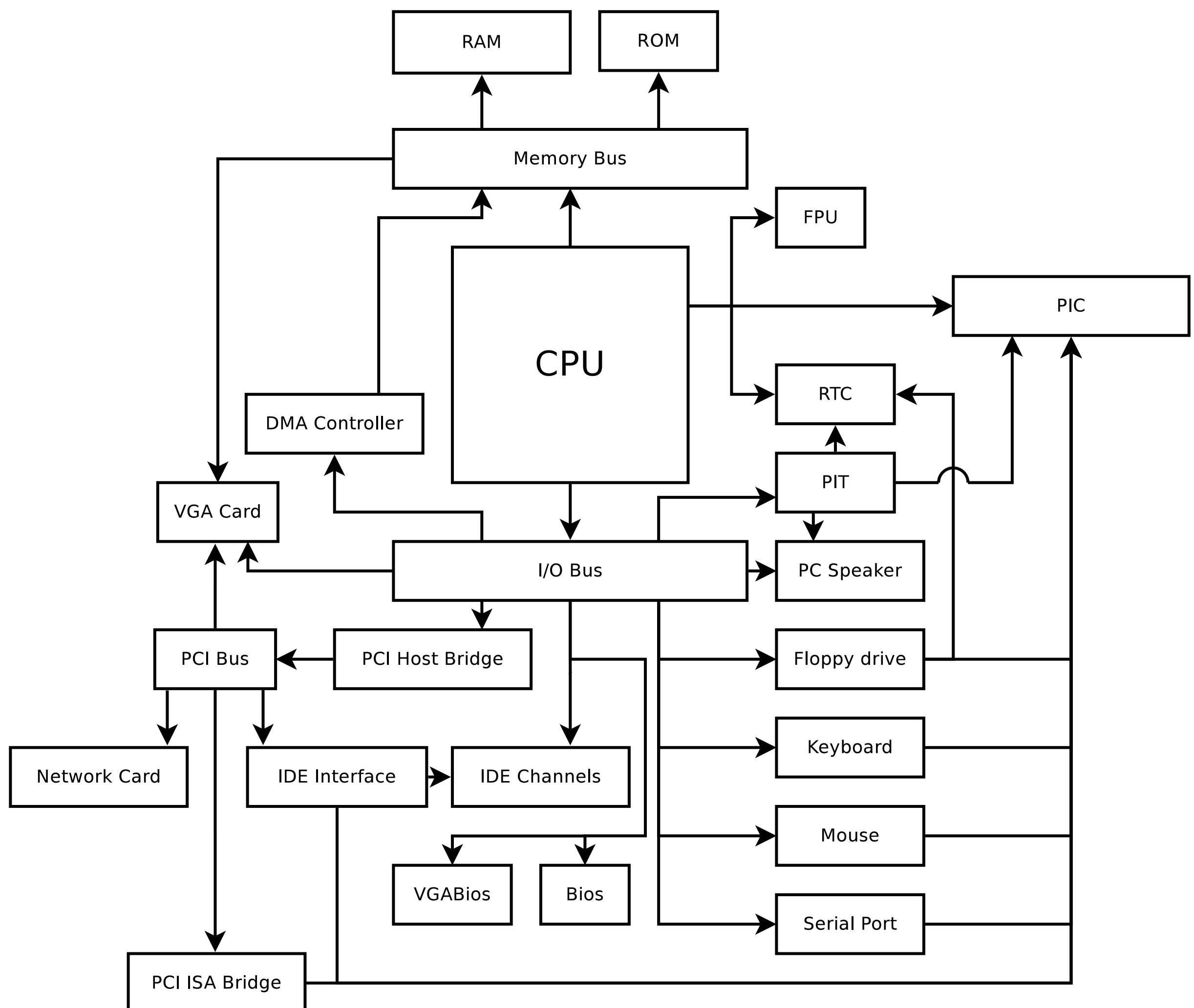


Figure 4.3: The JPC architecture: lines represent control flow and all originate from the CPU. This shows how all the components are connected and how they are accessed.

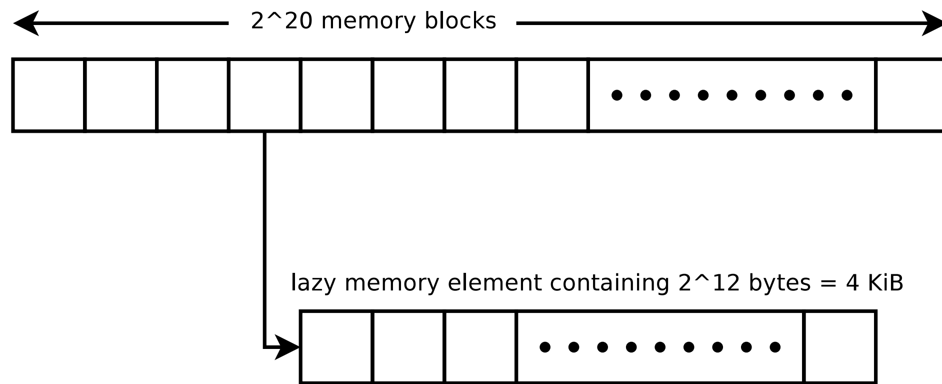


Figure 4.4: A two layer memory structure for physical memory.

4.1 JPC Memory architecture

JPC emulates a 32-bit x86 processor, and thus it has to emulate a full 4 GiB (2^{32} bytes) address space for the memory. The naive approach would be simply to create a byte array for the memory. However this clearly will not work, because as soon as the array is created, it will require more than 4 GiB of memory. The key to solving this requires the realisation that in a typical PC most of the address space is not occupied - only recently have real home PCs started having 4 GiB, or more, of physical memory. This, combined with the fact that most memory is not written to in many circumstances, points to a solution.

The solution JPC uses is a two-tiered structure of byte arrays and lazy initialization, shown in Figure 4.4. Lazy initialization means that a memory page is only allocated if it is written to. The top layer of the structure is a 1 MiB element array where each element is a lazy memory element. When each lazy memory element is allocated it holds a 4 KiB byte array representing a page of memory and arrays representing code blocks. The reason for the choice of 4 KiB granularity is that the Protected mode memory management unit (MMU) of the CPU splits the address space into indivisible 4 KiB pages. This two layer structure in JPC makes the memory subsystem immediately take up around 4MiB of real memory (a 1 MiB element array of references which are 4 bytes), and this increases as memory is written to until a maximum, when (if) the entire memory of the virtual machine has been written to since it was last booted.

Adding further layers to this model would further decrease the memory overhead, but at the cost of speed of access. JPC uses a combination of the two approaches, shown in listing 4.1.

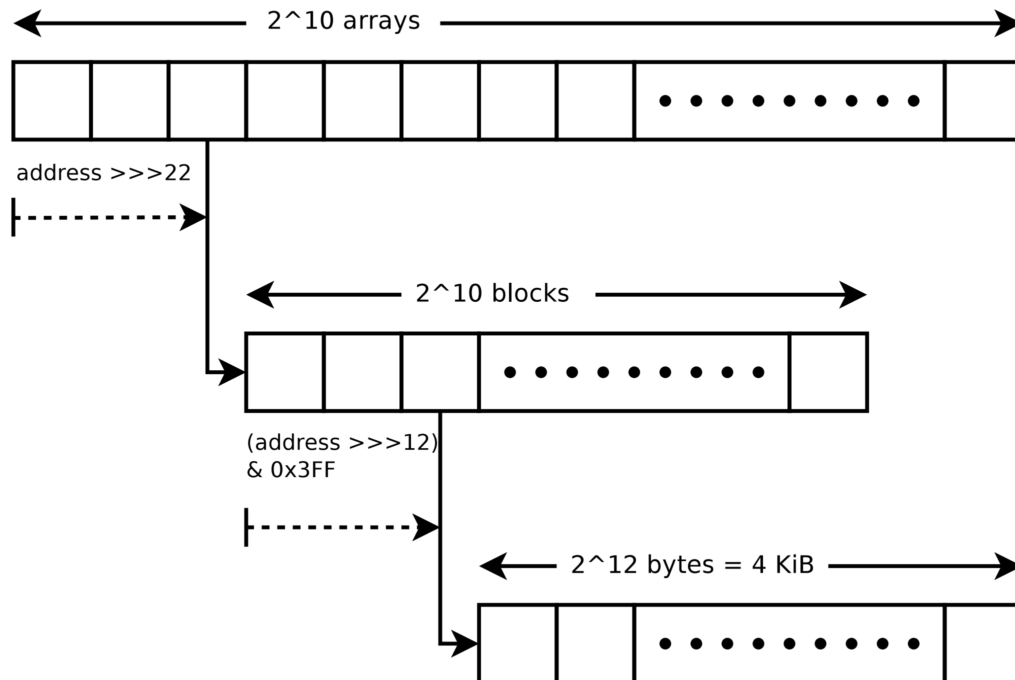


Figure 4.5: A three layer memory structure for ROM and memory mapped I/O.

The *quickIndex* array is big enough to contain all the RAM. The motivation for this structure is the observation that memory in the physical address space comes in three types: RAM, which has very low latency and mapped upwards from zero, ROM, which is rarely accessed and has low latency, and memory mapped I/O, which has much higher latency. The ROM and memory mapped I/O can be at any address. A quick 2 layer structure is used for RAM, and a slightly slower 3 layer structure for the remaining address space.

For typical scientific applications of JPC, less than 256 MiB of virtual memory is sufficient. With the combined address space structure, the memory overhead is $\sim 256\ \text{MiB}/4\ \text{KiB} \times 4\ \text{bytes} + 2^{10} \times 4\ \text{bytes} = 260\ \text{KiB}$. This is a factor of 16 less than the two layer structure. Note that as JPC's memory is written to the amount of actual memory taken up increases as roughly $4\ \text{KiB} \times$ number of pages written to.

Linear address translation (paging) is an expensive process and real x86 processors cache the results of a lookup in a Translation Lookaside Buffer (TLB). The TLB is flushed on a task switch to ensure a task does not gain illegal access to pages of memory or when the CR3 register is written (which is used to define the beginning of the translation process). This normally does not include when user code switches to supervisor (privileged) code and vice versa, hereafter

called a *mode switch*. A mode switch happens in a user level program at the beginning and end of every system call. Operating systems like Linux achieve this mode switch by emulating in software the task switch mechanism, but only saving the absolutely necessary state and thus avoiding the invalidation of the TLB [95].

JPC uses a similar tactic to what a TLB must use to cache linear address translations [96]. The *linear address space* in JPC implements this according to the four possible combinations of read/write and user/supervisor [97], illustrated in Figure 4.6. Four arrays of memory pages are kept, read_supervisor, write_supervisor, read_user and write_user along with two references - the current read array and write array. When a privilege level change (mode switch) occurs, the two references for the current read and write array are switched to the new privilege level. The processor changes mode frequently and thus is it important that changing the TLB contents is fast. In this implementation all that needs to be done on a mode switch is to switch two pointer references, which allows JPC memory access in Protected mode to be almost as fast as in Real mode, despite the extra overhead of paging and more complex segmentation. With this design, the memory performance of JPC is not a bottleneck across the operating systems it can boot.

When a new memory segment is loaded in Protected mode, first the current cache array of translations is checked to see if the relevant page address has already been translated. If the address has not been previously accessed a null pointer is thrown which is then caught and the slow process (see Section 3.1.2) of manually translating the linear address to a physical address is used. The result of the slow translation is then stored in the current cache array for faster future access.

4.2 Motherboard

The motherboard is a group of core components which enable communication with the peripherals as well as housing the BIOS (both system BIOS, which boots the machine, and VGA BIOS which controls the graphics card). The other components are the I/O port handler, DMA

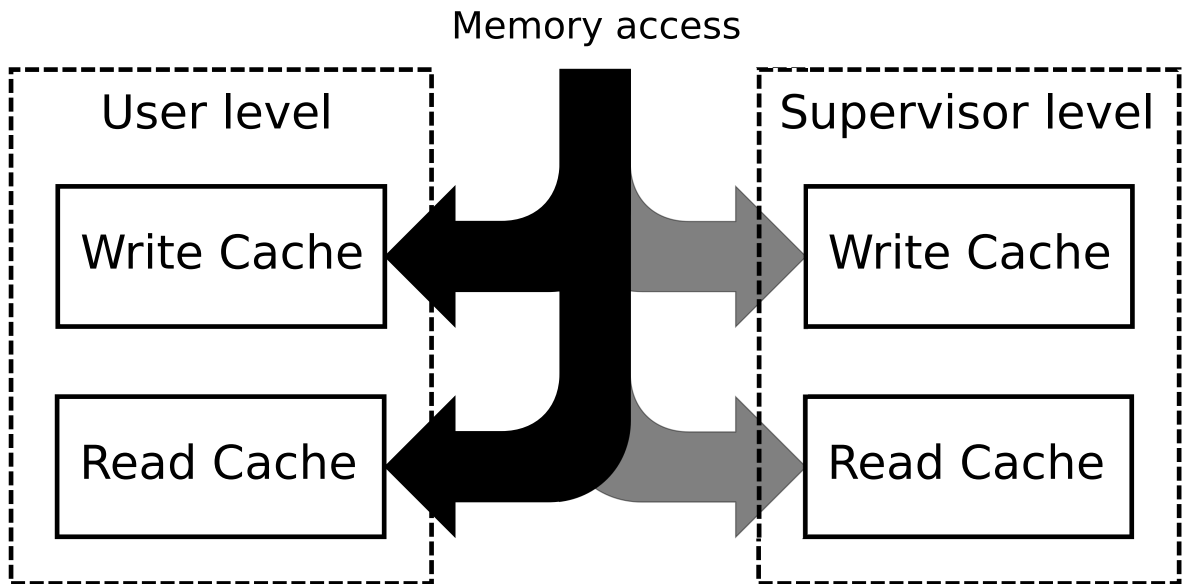


Figure 4.6: The implementation of Privilege level changing and linear address translation caching in JPC

```
public byte getByte(int offset)
{
    return getMemoryBlockAt(offset).getByte(offset & 0xFFF);
}

private Memory getMemoryBlockAt(int i) {
    try {
        return quickIndex[i >>> 12];
    } catch (ArrayIndexOutOfBoundsException e) {
        try {
            return index[i >>> 22][(i >>> 12) & 0x3FF];
        } catch (NullPointerException n) {
            return UNCONNECTED;
        }
    }
}
```

Listing 4.1: Memory access methods

controller, which enables fast ransfers between capable devices, interrupt controller, interval timer and realtime clock.

4.3 PCI

The PCI subsystem includes the VGA card, IDE Controller, and the Ethernet card. A PCI device appears no different to the processor than other peripherals, with communication occurring through port mapped or memory mapped I/O. However, the PCI devices can be configured by the processor using the PCI-Host bridge, allowing dynamic allocation of address space mappings and port mappings to avoid conflicts from hardware assigned values. The PCI ISA Bridge allows PCI devices to raise interrupts on the interrupt controller.

All the current PCI devices in JPC extend the PCIDevice class. New hardware can also be added to JPC by creating a new subclass of PCIDevice.

4.4 Peripherals

The remaining peripherals implemented are the floppy drive, the PC speaker, keyboard and the serial port. All communication with these occurs through the I/O bus. The I/O bus is a communication channel that x86 CPUs provide specific instructions to access (IN and OUT). These instructions are only available to supervisor level code by default. Peripherals can be connected to a number of ports and take appropriate action when one is read from or written to. The serial port, for instance, was particularly useful for initial debugging of the BIOS boot process, as many debug statements are output to it.

The JPC implementation of the peripherals in this section is encapsulated in the IOPortCapable interface, shown in listing 4.2.

```
public interface IOPortCapable
{
    public int[] ioPortsRequested();

    public void ioPortWriteByte(int address, int data);
    public void ioPortWriteWord(int address, int data);
    public void ioPortWriteLong(int address, int data);

    public int ioPortReadByte(int address);
    public int ioPortReadWord(int address);
    public int ioPortReadLong(int address);
```

Listing 4.2: IOPortCapable interface

4.5 Timing implementations

The handling of time in a virtual machine is fraught with difficulties [98][78]. A common problem in virtualization is for the virtual time to fall behind real time, due to multiple virtual machines sharing the same processor. Time in an emulator can end up faster or slower than real time.

Time can be implemented in a number of ways, each with benefits and pitfalls. The most obvious of these is to use wall clock time. The clock in a virtual CPU has to be updated very often to check if any interrupts need to be signalled from expired timers. However, the system call to get the current time is a relatively expensive operation (on the order of microseconds per call on some systems), and therefore calling the system call on every virtual clock update is unsatisfactory.

One solution, which will be called *Corrective Drift Timing*, is to only call the real system clock after a certain number of virtual clock updates and interpolate in the meantime. The problem here is that virtual time is not guaranteed to be monotonically increasing, due to instructions taking different amounts of actual time to execute. For example, consider the following interpolation scheme illustrated in listing 4.3. The virtual time and real time start off synchronized and a drift correction is updated every time the system clock is queried.

One possible sequence of timings is given in Table 4.1 and plotted in plot 4.7, which shows a decreasing virtual time after the last step. This is clearly a problem which could have serious

```
static int LIMIT = 1000;
int counter = 0;
long drift = 0;
long currentTime = System.nanoTime();

public void updateTime()
{
    currentTime += drift;
    if (++counter >= LIMIT)
    {
        long diff = System.nanoTime() - currentTime;
        drift += diff/LIMIT;
        counter = 0;
    }
}
```

Listing 4.3: Corrective Drift Timing

Table 4.1: Virtual time versus real time (arbitrary units).

drift	0	2	4	3	0	-2	-1
virtual time	0	0	2000	6000	9000	9000	7000
real time	0	2000	4000	5000	6000	7000	8000

repercussions for running software. For example, any transaction based software that stores times with each transaction could have the transaction order changed. If the system in question were a banking or database system, this would have potentially severe consequences.

An alternative approach to time emulation, which will be called *Instruction Count Timing* (ICT), is to use a conversion factor from the number of instructions executed. This is determined from the assigned frequency of the virtual CPU and a fixed number of instructions per cycle. The number of instructions executed is monotonically increasing, and hence so is the derived virtual time.

$$\Delta t = \Delta i \times \frac{1}{(\text{CPU frequency}) \times (\text{instructions per cycle})}$$

This approach is entirely deterministic, and thus the virtual PC's execution is deterministic and therefore repeatable. Repeatable execution is highly desirable for debugging purposes. It is also fast as there are no system calls involved. One obvious drawback is that the virtual clock can, and does, get arbitrarily out of synchronization with real time.

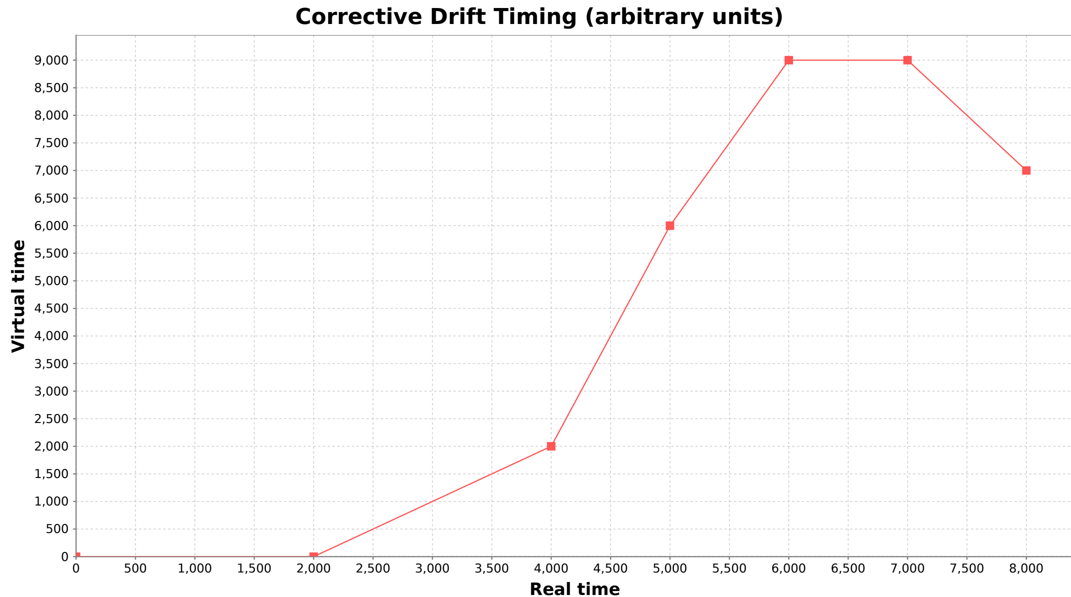


Figure 4.7: A possible timing sequence with non monotonic virtual time.

A better approach turns out to be a combination of the two approaches. The virtual time is updated via the CPU frequency times the number of instructions executed, but the CPU frequency is occasionally increased or decreased according to the difference from the real time. So long as the frequency is defined as a positive number, the virtual time will always increase. JPC currently uses ICT, whilst the CPU emulation accuracy is being improved.

4.6 Block Devices

A block device is an abstraction of a physical disk, like a hard drive, floppy drive, or CDROM drive, which allows arbitrary sectors to be read and written. The basic block devices are implemented to read a flat binary file which holds the disk's data, uncompressed. One could write versions to include compression to reduce the disk image size. Two implementations were found to be particularly useful, a FAT32 hard drive, and remote drive. FAT32 was chosen because it is a format understood by both legacy and current operating systems, and because it is much simpler than NTFS, ext2 or other modern file system formats.

The FAT32 drive, or *tree block device*, is used to get files in and out of the virtual machine quickly and easily. The tree block device acts as a wrapper for a real physical directory. It

reads in the entire subdirectory structure and files and presents them as a valid FAT32 drive to the emulator. By default, writes to the drive are simply cached in memory. However, if output files are required to be retrieved out of the emulator then the drive can be set to continuously synchronize with the underlying directory structure.

The remote block device is a way to present a remote drive image on a web server as a local drive. Without it, the entire disk image needs to be downloaded. Given the aim of running JPC as an applet, and hence being unable to store anything on the local disk, minimising downloaded data is essential. Without a remote drive JPC would be limited to using disk images that would fit in the memory of an applet, typically 512MiB. For particle physics use, the aim is to use JPC to run cernVM [99]. cernVM is a disk image released by CERN specifically designed to run in a virtual machine, and which contains the entire ATLAS software stack. The important point here is that the cernVM disk image is around 5GiB, and thus impossible to use remotely without the remote block device.

The remote disk works by downloading sectors (512 byte sections, aligned on 512 byte boundaries) on demand. As a sector is requested to be read from the disk, a 64 KiB chunk around the sector (64 KiB aligned) is downloaded and cached locally. This size was chosen simply to be a similar size to a typical web page, and thus not cause bandwidth or latency issues. Too small a size would result in sequential reads having to wait for the download of each section, whereas too large would delay the initial disk read that triggered the download. The chosen size allows for good performance when reading sequential sectors (measured by running programs that load large files from disk into memory at startup), and only downloads slightly more data than is strictly necessary in most situations. Caching the sectors locally means that subsequent reads in the same region are very fast. As an added bonus, the startup time for JPC in an applet can be much smaller by avoiding downloading any disk images.

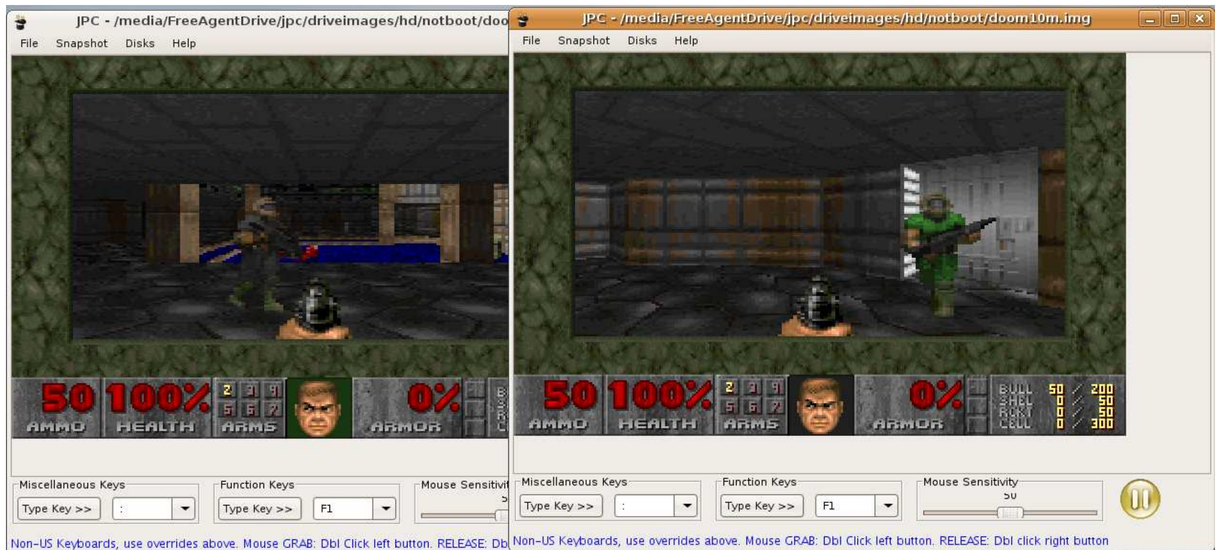


Figure 4.8: Network DOOM between two JPC instances.

4.7 Network card

Emulating a network card is necessary for software running in JPC to be able to have any communication with the Internet, other computers, or even other JPC instances. This involved emulating an ne2000 (Realtek 8029) Ethernet card in software from a relatively well defined hardware specification. This card was chosen because it is supported both by old DOS drivers and modern Linux and Windows drivers. Possible alternatives include a USB modem (which clearly would require implementing a USB port as well) and other network cards such as Intel 82559 Ethernet Controller or AMD PC-Net II Ethernet Controller. These could be implemented in the future to give more connection options.

If a software router were to be implemented as well, this would give another way to get files in and out of JPC at runtime. This is particularly important given that much contemporary software is installed directly from the Internet, after initial operating system installation.

A virtual Ethernet hub was implemented in software. This enables two or more JPC instances to be connected together as if the virtual machines were on the same local area network. As a demonstration of this network DOOM was played between two JPC instances running on the same physical machine, shown in Figure 4.8.

4.8 Snapshot

A snapshot is an efficient way of serializing the state of a JPC instance. It enables JPC to be paused and resumed at will and in different locations and times. This is similar to VMotion, a highly sought after feature of VMWare, which allows live migration of running virtual machines between different physical machines to balance resource usage. In practice VMotion is limited to moving between machines of the same cpu type (simply because it VMWare is a direct virtualizer). If JPC's snapshot ability were extended to allow live migration, it would suffer no such restrictions, allowing complete freedom of movement of JPC instances.

To create a snapshot, the entire state of the virtual machine, including all of the hardware state and memory, is saved to a zip file. This information can then be reloaded when JPC is resumed, typically taking under a second. The entire hardware state excluding memory and hard disks can be saved to under 1 MiB. In practice, the size of the snapshot is largely taken up with the memory state. Note that only the memory that has been written to is saved. This means the virtual machine can be allocated large amounts of memory without worrying about bloating the snapshot.

The power of this technique for grid applications of JPC is that it allows a snapshot to be taken of JPC just before the relevant calculation/program is about to run, which can then be started on multiple machines, thus avoiding the delay of getting through the boot phase.

The saving of the JPC state during a snapshot is complicated by memory mapped objects. These are hardware components that can be directly mapped into memory, and hence written to by any memory operation. For example, the VGA card is mapped into memory at address 0xA0000 and the BIOS is mapped from 0xC0000 to 0xE0000. There is often flexibility as to where exactly in the memory address space each of these is mapped. However, for running programs to function correctly, the location of these objects in memory must be consistent before and after a snapshot, and hence this is additional state information that JPC captures. There is currently no way for JPC to resume any TCP connections that were in progress when a snapshot was made.

Chapter 5

JPC execution and profiling

This chapter summarizes how execution proceeds in JPC and presents methods for improving the emulation accuracy and speed estimation.

5.1 JPC Execution

JPC executes x86 code first by interpreting the instructions, and eventually with compiled methods using dynamic compilation. The basic execution loop is shown in listing 5.1 and is currently single threaded. Currently JPC only emulates a single-core CPU, but eventually it would be beneficial to extend JPC to emulate a multi-core architecture. Such an extension will present many obstacles, such as correctly emulating the concurrent memory access properties and the per core memory cache coherency, not to mention ensuring each core thread of execution is kept in sync with the virtual clock ticks. Of course, a multicore architecture could be more simply emulated using a single real thread, but this would defeat the purpose of enabling concurrent execution of different virtual threads.

The memory repeatedly executes a *basic block* (defined in section 3.1.3) on the CPU and handles any interrupts, until a processor exception occurs. When a processor exception occurs, the exception is handled before returning to the main execution loop. Note that this structure of the memory executing code using the CPU does not correspond to the normal understanding of

```
AddressSpace m; // created during PC construction

while (true)
{
    try
    {
        while (true)
        {
            // execute a basic block
            m.execute(cpu, cpu.getInstructionPointer());
            // process any pending interrupts
            cpu.processInterrupts();
        }
    } catch (ProcessorException e)
    {
        cpu.handleProcessorException(e);
    }
}
```

Listing 5.1: JPC's basic execution loop - execute is called on the memory and the CPU and the instruction pointer are passed in, before handling interrupts

computer operation where a CPU executes code using memory. There is a different version of this execution loop for each processor operation mode. The instruction pointer is fetched from the CPU object, and this is used by the address space to retrieve the relevant codeblock, decoding the raw x86 instructions found there if required. This codeblock is then executed, using the emulated CPU object, and then, any pending hardware interrupts are handled. For 100% accurate emulation, one would have to check for interrupts after every instruction. However, in practice, it is acceptable to check only at the end of *basic blocks*.

When emulating a CPU, it is often beneficial to employ optimisation and simplification techniques similar to those used by the real processors. Modern Intel processors break the Complex Instruction Set Computing (CISC) architecture x86 instruction into one or more simpler microcodes: to load the inputs, do the actual operation, store the output and finally update the CPU flags. For example, an ADD EAX, WORD PTR [EBX] could be decoded to the JPC microcodes in listing 5.2. This instruction takes the 2 byte value (referred to as a "word") at the 32 bit address EBX, adds this to the value in EAX, and stores the result back into EAX. Using a small set of low-level microcodes is much easier to implement and debug (in either a real processor or a processor emulation) than a large set of complex instructions. JPC also uses

microcodes to cache the result of decoding the x86, which is an expensive process. Microcodes also make other optimisations easier for an emulated processor by exposing the structure of the higher level instructions. If, in the future, JPC is expanded to emulate a 64 bit processor, then most of these microcodes will remain intact, with some new versions added.

JPC's microcode set has around 700 microcodes from which x86 instructions are constructed. The microcodes were chosen by imposing 4 steps to each instruction, and if one of these steps is still complex, then it is further broken down into simpler microcodes. These 4 steps are as follows:

1. load input operands
2. perform operation
3. store output operands
4. update flags

A situation where one of these steps needs to be broken down further is when a complex memory address is used. For example the load effective address instruction - LEA ESI, [EBX + 8×EAX + 4], which gets decoded to the microcodes in listing 5.3. Here, there is no operation and no flags to calculate, just load the input, and store the output. JPC currently has 3 virtual registers (*reg0*, *reg1*, *reg2*) as well as a virtual address register *addr*. These virtual registers are used to store the loaded input operands in, which the actual operation (if any) operates on. After the operation the result(s) are stored from the virtual registers to the destination operand(s). The effect of this separation is that the input operands and output operands can be decoded independently of the operation, thus simplifying the code.

Decoding has to be done only once for a given section of x86 code (unless the code is self modifying), and the result can be cached and optimised either in the background, or compiled in advance. The reason this is an effective technique is that x86 code typically has a very high execute/write ratio: every operation decoded is typically executed at least 10,000 times. Thus,

```
LOAD_EAX            load the value in EAX
LOAD_SEG_DS         load the memory segment DS to be read from
ADDR_EBX            set the address to read from in DS
LOAD_MEM_WORD       actually read from the memory segment
ADD                 perform the addition
STORE_EAX           store the result back into EAX
ADD_O16_FLAGS       update the arithmetic flags
```

Listing 5.2: Decoded JPC microcodes for the instruction ADD EAX, WORD PTR[EBX]

```
ADDR_IB             add 4 to the address
4
ADDR_8EAX           add 8EAX to the address
ADDR_EBX            add EBX to the address
LOAD_ADDR           load the resulting address into a temporary
    register
STORE_ESI           store the temporary register value to ESI
```

Listing 5.3: Decoded JPC microcodes for the instruction LEA ESI, [EBX + 8*EAX + 4]

optimising a section of code at runtime can greatly improve the speed of execution. Using microcodes facilitates redundant code removal, particularly arithmetic flag operations. After each arithmetic operation, the processor updates the Eflags register (which stores the current status of the processor, with binary flags for certain properties). The Eflags register stores properties of the previous arithmetic calculation, such as whether the result was zero, of odd or even parity, whether there was an overflow. One of these flags is often used at the end of a block to decide where to jump to, but consecutive arithmetic operations overwrite the flags and hence the earlier updates are redundant. Nevertheless, decoding and optimisation needs to be as fast as possible because they happen during runtime.

The first time a block is executed, a decoded list of microcodes is stored for subsequent execution. Once a block has been executed (interpreted) 1024 times, it is queued for compilation by the JPC compiler, and the resulting compiled block replaces the original interpreted block. The interpretation limit of 1024 was chosen after varying the limit and timing the execution of a cpu benchmark. Eventually, the compiled block is itself compiled by the Java Virtual Machine (JVM) compiler. This flow of optimisation is shown in Figure 5.2.

Currently JPC only compiles code up to the end of a basic block. Once a block has been decoded into microcodes and before it has been compiled it is executed by a microcode inter-

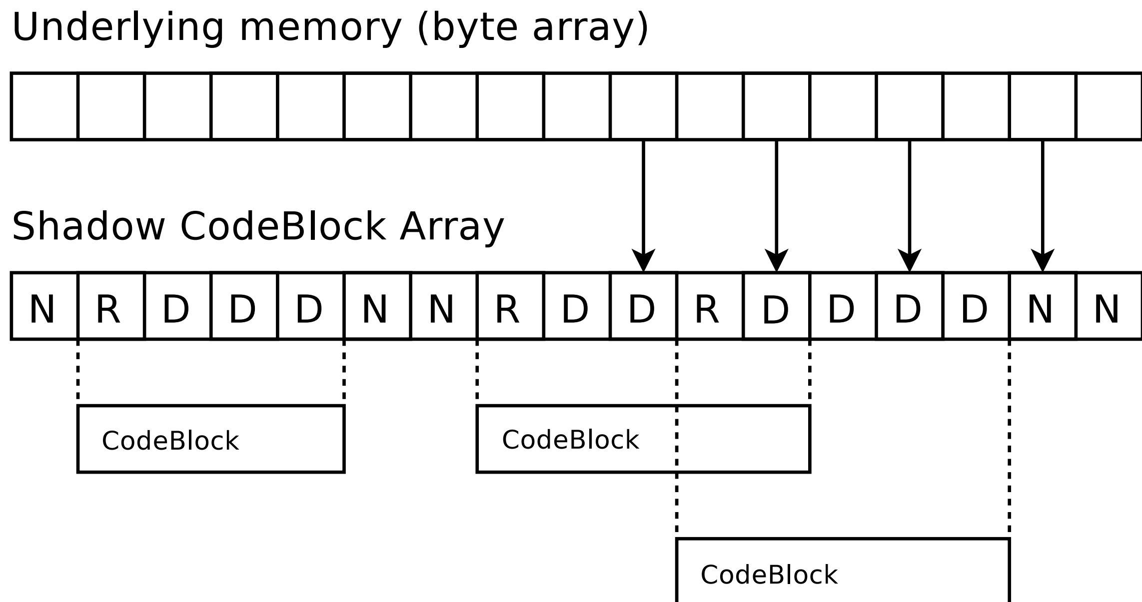


Figure 5.1: The Codeblock structure mirroring the underlying memory. N is a null reference, R is a reference to a CodeBlock and D is a dummy reference to indicate that there is a CodeBlock covering this position.

preter. Decoded/compiled code blocks are stored in a codeblock array mirroring the memory, as illustrated in Figure 5.1, with the corresponding element mirroring the first byte of the code block containing a reference to the decoded/compiled block object. The remaining bytes of the block hold a dummy reference which is used to ensure compiled/decoded blocks are invalidated if a write occurs in the memory area they cover.

5.2 Compiled execution

A code block compiler takes as input the list of microcodes and corresponding x86 instruction lengths in bytes for a basic block, and creates a Java class which performs an equivalent computation to interpreting the list of microcodes. A code block compiler thus corresponds to the middle arrow in Figure 5.2. The reason for compiling a code block is to try and extract the maximum performance out of it, by allowing the JVM to see the structure of the block and its instructions. To execute this class as if it was part of the running JPC process, the class bytes must be loaded into the JVM as if they were part of the JPC application code. Fortunately Java provides this mechanism via the ClassLoader object: all the JPC compilers implement a custom

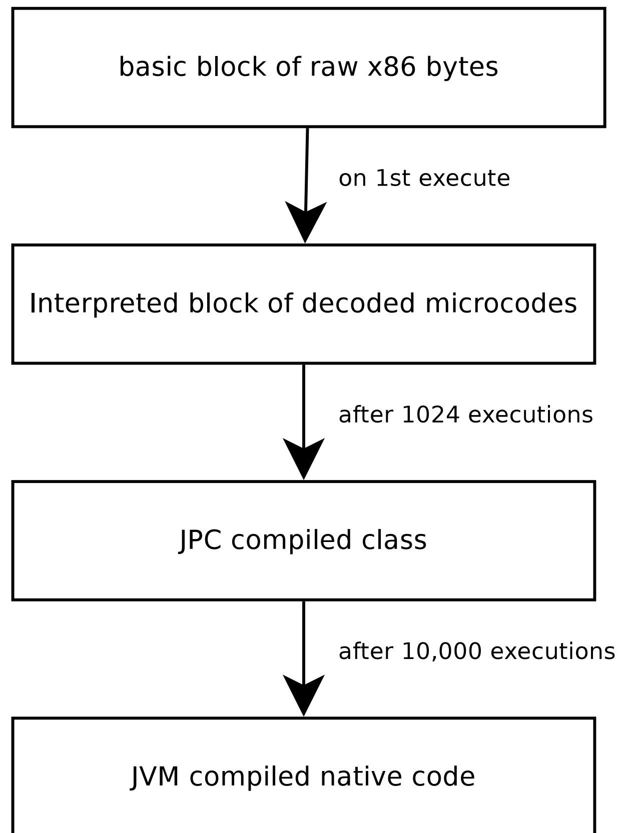


Figure 5.2: The compile chain for x86 code in JPC.

class loader to load in all these new classes.

5.2.1 Source code compiler

The first attempt at a compiler for a basic block was a “source code” compiler. Each microcode had associated Java source code written out by hand. This compiler used the very simple approach of just appending all the source code fragments for the microcodes in a given block, and compiling the resulting Java source file at runtime into a class. This basic approach is easy to implement and the hope was that the Java runtime compiler would be able to optimise the code further. Even with such a naive approach, this compiler still gave an improvement of a factor 2X the speed of the interpreted version. This served as the justification for writing more advanced compilers.

5.2.2 Basic block compiler

The basic block compiler splits up the JPC microcodes into static single assignment (SSA) form [100]. In SSA form each variable is assigned exactly once - existing variables are split into versions. It is often used as an intermediate representation in compilers because it simplifies the properties of variables and improves the results of many optimisations. For example, here it allows the dead code within a basic block to be removed (including redundant copying of unused CPU register variables into local variables) and puts the execution of the block in a form where the JVM compiler can see the sequence of x86 instructions (strictly JPC microcodes) and thus perform further optimisations. This gives roughly a factor of 10X speed increase over the JPC microcode interpreter.

5.2.3 Basic loop compiler

The basic loop compiler was developed to optimise the most common examples of loops. The relevant loops have code blocks that end with a short jump (a jump to a position less than 128 bytes away from the current position) to either the same code block, or another which then returns to the original. After the execution of one of the blocks, the instruction pointer is compared to see if it points to one of the other blocks and if so, that block is directly executed, and so on through other locally connected blocks. This relatively simple step actually affords a factor of 2X speed improvement, hinting at some of the speed increase possible with a full loop compiler.

5.2.4 Full loop compiler

A full loop compiler, which can include jumps (and thus loops), would allow the JVM to see the higher level structure of the code over multiple basic blocks (which are terminated by a jump). This would provide the JVM significantly more scope for optimisations such as variable hoisting, dead code elimination, range check elimination etc.

Section 5.4.3 demonstrates that there is considerable overhead associated with starting a new basic block. This penalty is primarily due to the time consumed by fetching the correct block to execute, setting up all of the local variables for the block, and copying their values from the CPU variables. Theoretically, for tight CPU bound loops, this overhead can be almost completely eliminated with an advanced loop compiler because long periods of time would be spent inside a single 'superblock' or combination of nearby blocks. This strongly suggests that a properly implemented full loop compiler would provide a significant speed increase to JPC. An indication of the potential speed increase is given by the difference between the toy CPU version 4 and 5 in section 3.2, roughly a factor of 8 times.

5.2.5 Storing compiled blocks between runs

JPC is designed to run scientific code on a global grid of Java applets. All of these compilers are very effective at making JPC execution fast, but they only work when JPC is run as a stand alone application, as creating custom class loaders is prevented by the applet sandbox. The prohibition of compiling at runtime within an applet reduces JPC to interpreted performance. However, for a given task, JPC can be run once as a standalone application with all the compiled classes subsequently saved to a Java archive (JAR) file. When JPC is run again with the new JAR file in the classpath, either in the applet or the application, the precompiled classes can still be loaded at runtime, and the execution is in fact faster still, because the overhead of compilation has been removed. This is a perfect fit for the target application of running physics code in JPC using applets.

Java classes are loaded at runtime using only their fully qualified name, for example "org.jpc.emulator.PC". In order for JPC to be able to load a precompiled class for a section of x86 code, the classname must be a deterministic function of the x86 code.

To achieve this, compiled classes are named from a hash of the given block's JPC microcodes and the operating mode of the CPU, for example FAST_PM_1922109533.class. This then allows the applet version of JPC to load classes by computing the hash of the decoded

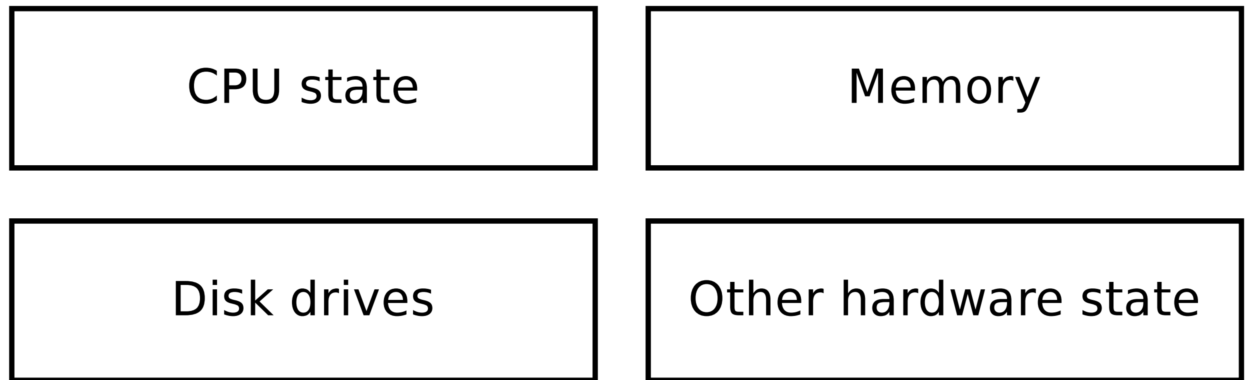


Figure 5.3: The components of a PC's state.

microcodes. To avoid problems caused by hash collisions when a class is loaded, the original microcodes of the compiled class are compared to those of the newly decoded block.

5.3 Improving JPC's emulation accuracy

The state of a virtual PC is made up of the state of the memory, the hard disk, the CPU and the remaining hardware as in Figure 5.3. For ideal execution, the complete state of these would follow that of a real PC with identical hardware attached.

5.3.1 Platform specification

Implementing the Intel specification for the processor execution is not enough, as the specification is often wrong or undefined. For example, according to the Intel specification (Vol 3A, section 7.1.3) an instruction is not allowed to modify the instruction immediately after it, but some old DOS programs depend on this capability to function properly.

A more serious example of an instruction that is incorrectly documented is the AAA - Ascii Adjust after Addition instruction. According to the Intel specification, this instruction behaves as in listing 5.4. However, on a Pentium processor, it behaves according to listing 5.5. Apart from the obvious difference in flag setting, the two give different results if the upper 4 bits of AL are set (`AL & 0xF0 = 0xF0`). For example, with input AX = 0x12FA, the Intel specification gives a resulting AX of 0x1300 and an actual Pentium gives a resulting AX of 0x1400.

```
if (((AL & 0x0F) > 9) || AF)
  {
    AL = AL + 6;
    AH = AH + 1;
    AF = 1;
    CF = 1;
  }
  else {
    AF = 0;
    CF = 0;
  }

  AL = AL & 0x0F;
  OF, SF, ZF and PF are undefined.
```

Listing 5.4: AAA instruction and how it affects AX and the flags according to the Intel specification, N.B. AX is AH:AL

```
if (((AL & 0x0F) > 9) || AF)
  {
    AX = AX + 0x106;
    AF = 1;
    CF = 1;
  }
  else {
    AF = 0;
    CF = 0;
  }

  AL = AL & 0x0F;
  OF = 0;
  SF = 0;
  ZF = (AL == 0);
  PF = parity(AL);
```

Listing 5.5: AAA instruction and how it affects AX and the flags when executed on a Pentium, N.B. AX is AH:AL

5.3.2 Initial debugging attempts

JPC can execute one instruction at a time and allow the relevant state to be examined using the JPC debugger. The JPC debugger is a graphical user interface to JPC that allows the CPU and memory state to be examined at any point, as well as normal debugger features like setting break points and watch points.

One possible automatic verification method would be to connect JPC to a suitably configured real PC and compare states in lock step (Figure 5.4). Any difference could be detected and the bug fixed before continuing. However, there are many practical difficulties to employing

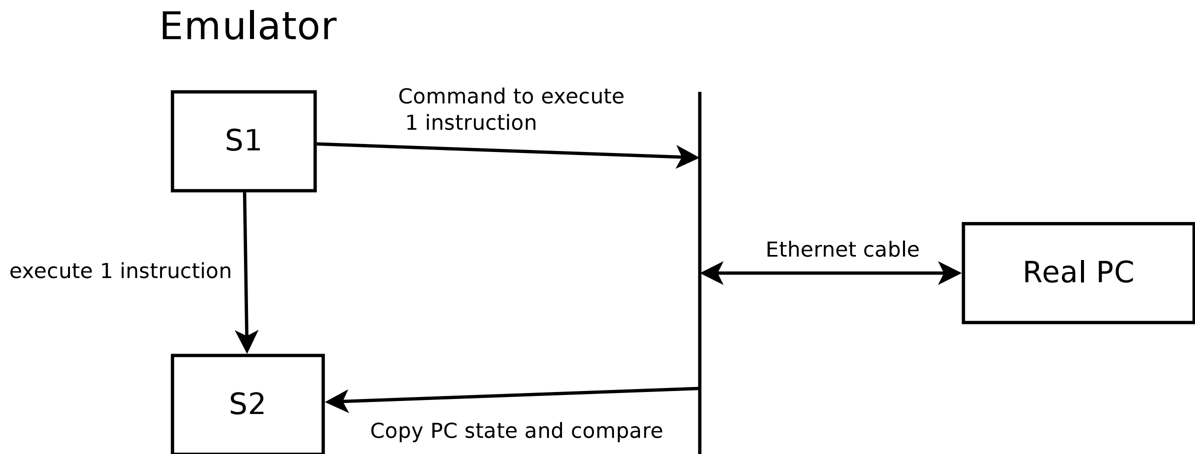


Figure 5.4: Comparing an emulator with a real PC one instruction at a time.

this approach. For example, either the JPC or the real machine's state has to be transfered to the location of the other for comparison. Even if the transfer is done over Gigabit Ethernet this will limit the execution rate to about 1 instruction per second (assuming the PC has at least 100mb of memory, which most modern operating systems need). Booting a modern Ubuntu Linux takes tens of millions of instructions, translating to a comparison time on the order of about 4 months for a single run.

An alternative arrangement is to compare execution in JPC to another x86 emulator that is already sufficiently accurate to run the desired operating system, as illustrated in Figure 5.5. The two obvious candidates are Qemu and Bochs. Qemu was chosen because of its pre-existing gdb interface (a standardised debugging interface), speed and ease of modification. There is no obvious way to compare the state of the hardware, apart from the CPU and the memory. The CPU state amounts to at least 10 four byte registers (depending whether the debug registers, control registers, MMX registers and FPU registers are included). The memory amounts to at least several MiB of data and probably more like several hundred MiB for booting a modern OS. Comparing the memory after each step is very slow. Therefore, initially, the memory state was ignored and only the CPU state was compared after each instruction. This still picked up a large number of errors.

Deterministic execution means repeatable execution, and this is highly desirable whilst JPC is still being debugged. However, interrupts are normally a source of randomness which could

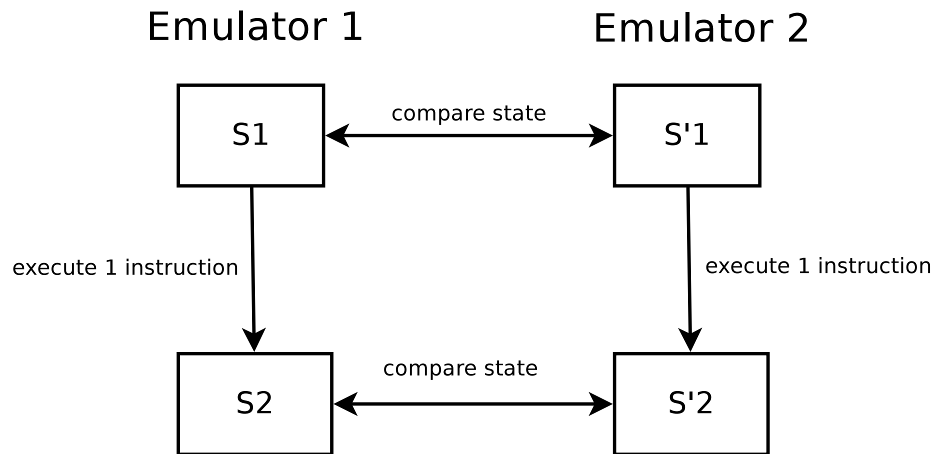


Figure 5.5: Comparing two emulators one instruction at a time.

affect this. Interrupts are very hard to deal with, because they are random hardware sourced events. The exact timing of interrupts is non deterministic for real machines and for emulators other than JPC (this is discussed in section 4.5 - JPC can use deterministic timing that may get out of sync with 'real time' or non deterministic timing that stays synchronized with 'real time'). An interrupt is a signal to the CPU from some other hardware that there is other work to be done, for example a network packet to be read. The physically based randomness of interrupts makes it impossible to single step two machines in parallel. A method to side step this issue is to notice that one of the machines is in an interrupt, via the IF CPU flag (which indicates if the processor is handling an interrupt), and single step that machine until it returns from the interrupt - thereby effectively ignoring the interrupt. This method was used to identify many emulation bugs, but only proved useful for under a minute of native execution as more and more side effects of interrupts accumulate.

The techniques described in this section allowed JPC to reach sufficient accuracy to firstly boot through the BIOS from the BOCHS project, and secondly to boot all versions of DOS, Windows 3.0 and a minimalistic version of command line only Linux (Qemu Linux).

5.3.3 Direct debugging

Once a version of Linux could be booted in JPC, a much more efficient debugging scheme was available. It was then possible to directly test each opcode within JPC on a range of different

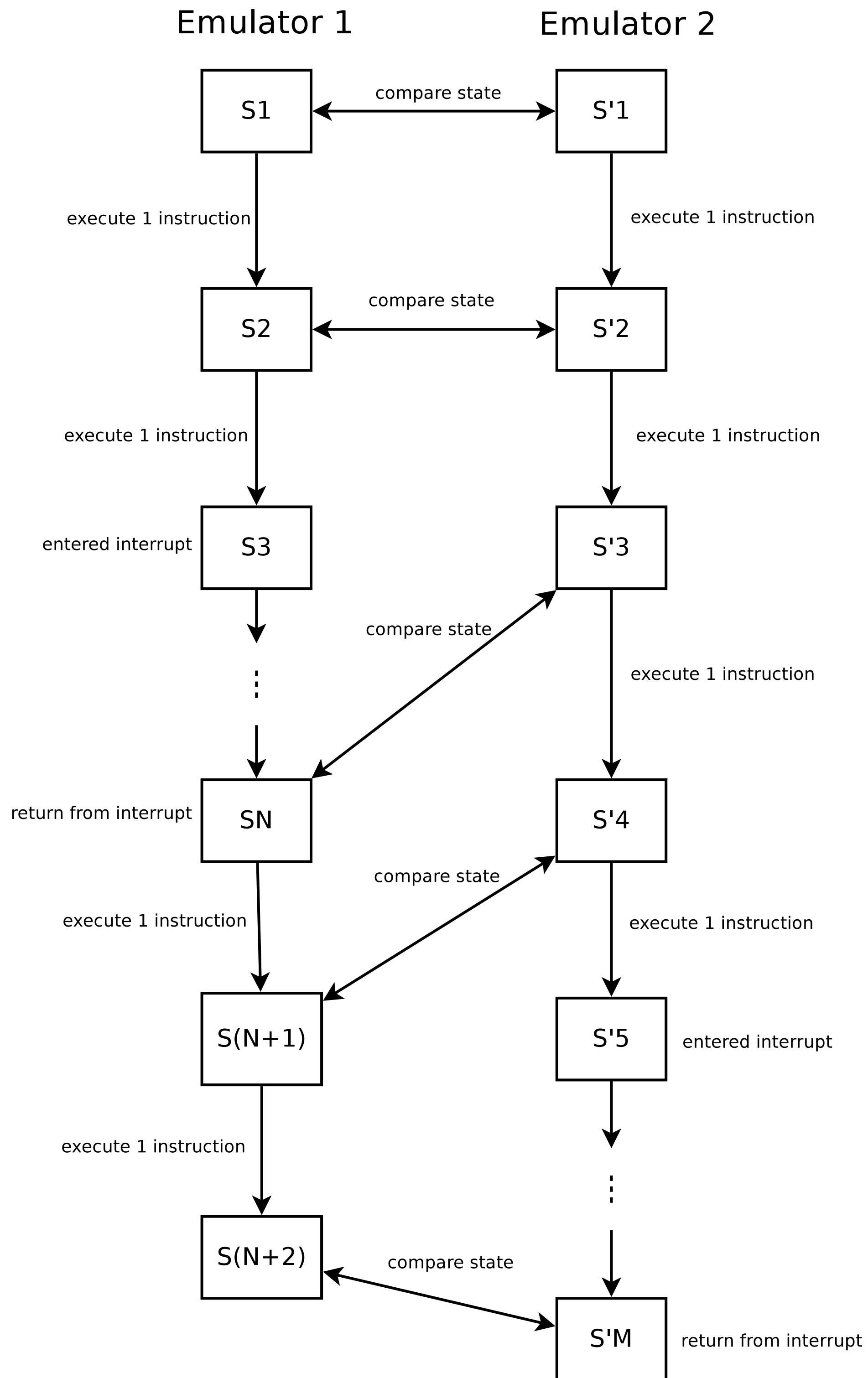


Figure 5.6: Comparing two emulators one instruction at a time and ignoring interrupts.

inputs and print out the CPU state after each one. A C program was written with inlined assembly (a symbolic representation of raw machine code) sections to load desired test values into the CPU registers, perform the instruction, and then extract the final CPU register values. The input CPU state, instructions and the post instruction CPU state are printed to a text file for later comparison. In principle, it is possible to write test cases for all the valid opcodes, run the program inside JPC and on a real machine, and compare the results. It is impossible to test all the possible inputs for every instruction (there are $2^{32^8} \sim 10^{76}$ different possible inputs for a given instruction - or about the same as the number of protons in the universe!), but a well chosen subset of inputs should find the majority of the emulation errors. This method is incredibly fast - it can be run over a representative set of inputs in a minute.

An example C program to test the *add* instruction for one input set is shown in listing 5.6. *r0* and *r1* are the input operands and *iflags* are the input flags. The *asm* call directly inlines assembly code and arranges for output values to be stored in the specified variables (*res* and *flags* in this case). The result of the operation is stored in *res*, and the flags following the operation are stored in *flags*. The *printf* simply formats the inputs and outputs in hexadecimal format and prints them to the console. Note that the l on the end of the add specifies that the add operation must use 32 bit operands.

When this program is run it outputs the following:

```
addl        A=12345678 B=11111111 R=23456789 CCIN=0000 CC=0202
```

A is the first input operand, B is the second input operand, R is the result, CCIN are the flags before the instruction, and CC are the flags after the instruction.

Tests have been written for testing most of the basic instructions including data transfer (e.g. MOV), binary artihemtic (e.g. ADD, DIV), decimal arithmetic (e.g. AAA), logical operations (e.g. AND, XOR), shift and rotate instructions (e.g. SHR, SHL) and byte operations (e.g. SETE). Most of the remaining problems are to do with the complex logic involved in task switching, inter privilege control transfer and switching between different CPU operation modes. However, JPC is now capable of booting some modern graphical Linuxes (Damn Small

```
#include <stdio.h>

int main()
{
  long r0 = 0x12345678;
  long r1 = 0x11111111;
  long iflags = 0;

  long res = r0;
  long flags = iflags;

  asm ("push %4\n\t"\
       "popf\n\t"\
       "addl %k2, %k0\n\t" \
       "pushf\n\t"\
       "pop %1\n\t"\
       : "=q" (res), "=g" (flags)\
       : "q" (r1), "0" (res), "1" (flags));
  printf("%-10s A=%08lx B=%08lx R=%08lx CCIN=%04lx CC=%04lx\n", \
         "addl", r0, r1, res, iflags, flags);
  return 0;
}
```

Listing 5.6: 32-bit ADD opcode test

Linux and Feather Linux).

5.4 Profiling JPC

For the purpose of measuring the speed of JPC, the relevant speed is how fast JPC emulates x86 instructions. There are various other possible speeds to consider, such as I/O speed, or memory access speed within JPC. However, at the moment, JPC is CPU bound (typically running 10-12% native speed, peaking at 20%) and therefore the CPU is the important part to optimise. The next question is how to best measure the execution speed. A relatively long running (of the order of minutes) application that is *virtual CPU bound* is needed. This means it must mainly involve arithmetic and no I/O operations. Due to the fact that the current JPC compiler does not compile loops, any tight loop will be dominated by the effect of the jumps - there is a lookup-dispatch for each block. This means that ideally the application will be a CPU bound, Linux executable with a long inner loop, but not too long for reasons that will become evident later. To ensure that the application is fully CPU bound and not memory bound there should be no, or

minimal, memory accesses. The Java Hotspot compiler sometimes has trouble inlining memory accesses, due to the multiple possible types of memory, which would slow down the benchmark further.

5.4.1 Measurements

Great care must be taken when making time measurements in any virtual machine including JPC and other virtualizers. For example, in JPC one cannot simply time the program as one would on a real system, using the Linux *time* program. Running *time* inside JPC uses JPC's virtual time, which can easily be adjusted to report any value. Measurements of programs running within JPC were done using a stop watch.

One also must be careful of non deterministic VM effects, of both the Java Virtual Machine and JPC itself. Profiling Java programs is surprisingly hard to do in a rigorous fashion [101][102]. There are many non deterministic VM effects that need consideration; for example, Hotspot requires several thousand runs of a code path before it is queued for optimization; garbage collection pauses are not predictable; and compiled code can be dynamically deoptimised and queued for recompilation, which means a return to interpreted performance. Moreover, before the JVM can compile the emulated x86 code, JPC itself must compile the relevant codeblocks, which is set to happen after 1024 executions of a given block. This means that a proper warmup involves running a particular block at least several thousand times. Once the code is warmed up then it must be timed repeatedly to ensure random variation due to the physical machine and remaining JVM effects are quantified.

5.4.2 Code

A small loop shown in listing 5.7, similar to that used for the toy CPU benchmarks in listing 3.1, was used to profile JPC execution. The aim was to estimate both the execution speed of raw instructions and also the fixed cost between basic blocks. This fixed cost consists of the time taken to lookup the new code block to execute, load the relevant registers from the CPU at

```
#include <stdio.h>
int main()
{
  int sum=0, i;
  for(i=1; i < 1000000000; i++ )
    {
      sum += (sum^i)/i;
    }

  printf("%d\n", sum);
  return(0);
}
```

Listing 5.7: Benchmark 1 code - zero unrollings

the beginning of the block and save them back to the CPU at the end of the block. To achieve this, the basic loop of arithmetic was unrolled different amounts and the resulting benchmarks were timed with a stop watch. Unrolling a loop means copying the code in the body of the loop several times. To compensate for unrolling the loop n times, the loop counter is divided by n to make sure the total amount of work is identical. This is illustrated for $n = 5$ in listing 5.8. Note that the results will change with the unrolling, but the instructions executed are the same, so comparisons are still valid, and a good loop compiler would reduce incrementing of loop counters in a similar way. The benchmarks were compiled by gcc with optimisation level 0 to ensure the desired basic block structure. This was then verified using the JPC debugger. The body of the loop was compiled to a single basic block, consisting of 9 x86 instructions. These are shown in listing 5.9.

Note that the constant in the compare (cmp) instruction is 1 billion, the number of loops, and the jump constant is -24, which is to the beginning of this block. The loop was unrolled by factors of 5, 10, 20 and 30 times.

5.4.3 Results

The execution of each of the benchmarks was measured in JPC 10 times on the same physical hardware with the error taken to be the standard deviation of these with a measurement error of

```
#include <stdio.h>
int main()
{
  int sum=0, i;
  for(i=1; i < 200000000; i++ )
    {
      sum += (sum^i)/i;
      sum += (sum^i)/i;
      sum += (sum^i)/i;
      sum += (sum^i)/i;
      sum += (sum^i)/i;
    }

  printf("%d\n", sum);
  return(0);
}
```

Listing 5.8: Benchmark 1 code - five unrollings

```
mov edx, ecx
xor edx, ebx
mov eax, edx
sar edx, 0x1F
idiv ecx
add ecx, 0x1
add ebx, eax
cmp ecx, 0x3B9ACA00
jne 0xFFFFFFE8
```

Listing 5.9: Benchmark 1 code - x86 instructions (24 byte length)

0.5s. The results are shown in Figure 5.7. All the benchmarks took roughly the same time to run natively as can be seen from Figure 5.8, with the default unrolling version taking 13.084±0.009s (measured using the linux *time* command). The execution time for each benchmark in JPC is modelled as follows:

$$T = n(m+c)$$

$$\begin{aligned} T &= \text{total time for benchmark} \\ n &= \text{number of loops} \\ m &= \text{time taken for the work in one loop} \\ c &= \text{a fixed time cost between each loop iteration} \end{aligned}$$

In this model, the product nm is assumed to be fixed across the benchmarks as the total time for the work ignoring looping costs. Then a plot of T vs n with a straight line fit, shown in Figure 5.7, gives two values. The gradient is c and the y-intercept is the 'ideal time' nm for the benchmark if the looping cost was approximately zero. This gives us a value of $c = 0.258 \pm 0.004\ \mu s$ and ideal execution time of $nm = 127 \pm 2s$, translating to 11% native speed.

To verify this result the entire set of measurements was repeated with a different loop body with some extra arithmetic code. The code for this second benchmark is shown in listing 5.10. The timing results for this second benchmark are plotted in Figure 5.9. The value obtained for the loop cost is $c = 0.262 \pm 0.008\mu s$, which is consistent with the previous measurement. It is observed that both benchmark timings increase by more than a factor of 10 beyond a certain level of unrolling. This is because the Java Hotspot compiler has a maximum method length that it will compile (set at 8000 bytes). Once a method passes this limit, despite being executed many times, it will not be compiled to native code and thus will run interpreted in the JVM. This easily results in a factor of 10 slow down compared to normal operation. This is not a concern for normal operation as these are highly artificial sections of code. As shown in [103], most basic blocks in modern x86 execution are only a few x86 instructions long, and 95% of basic blocks are under 16 x86 instructions long.

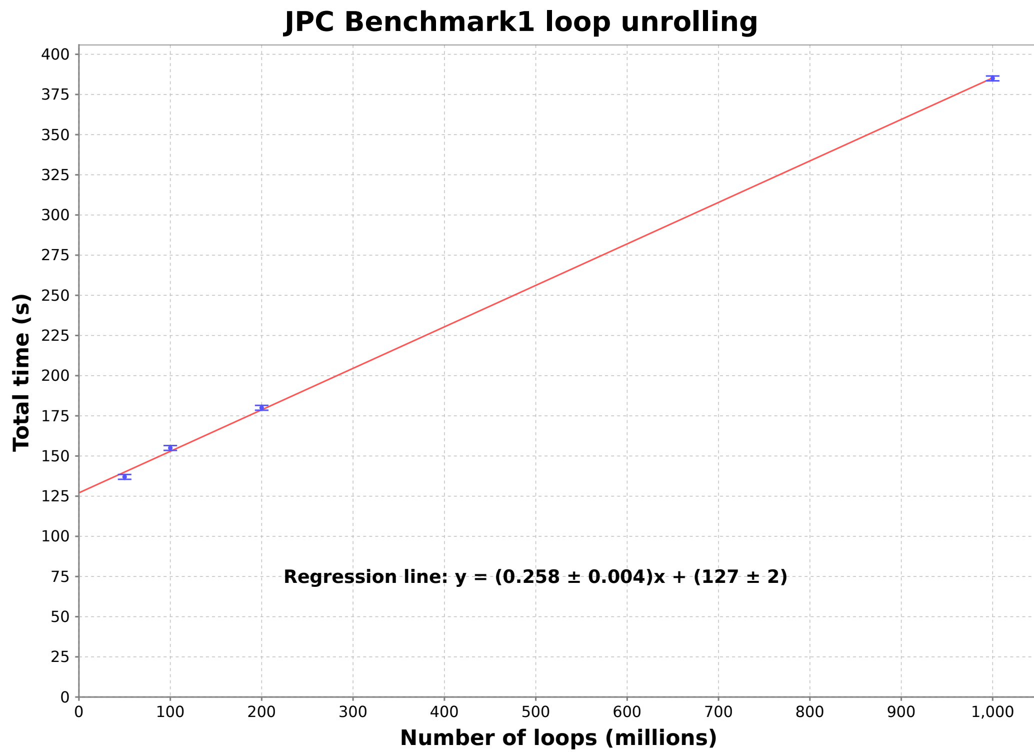


Figure 5.7: Timing results for different loop unrollings of JPC benchmark 1. N.B. the error bars are very small.

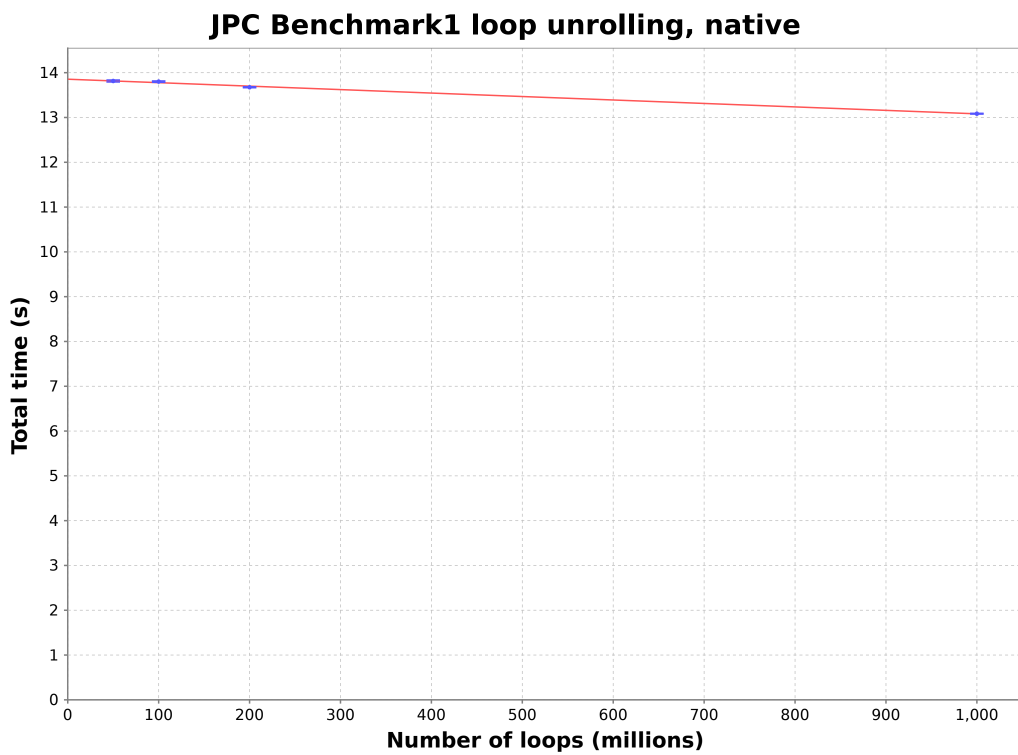


Figure 5.8: Timing results for different loop unrollings of JPC benchmark 1 run natively. N.B. the error bars are very small.

```
#include <stdio.h>
int main()
{
  int sum, i;
  for(i=1; i < 480000000; i++ )
    {
      sum += (sum^i)/i;
      sum *= 3;
      sum -= 10;
      sum &= 1024*1024 -1;
    }

  printf("%d\n", sum);
  return(0);
}
```

Listing 5.10: Benchmark 2 code

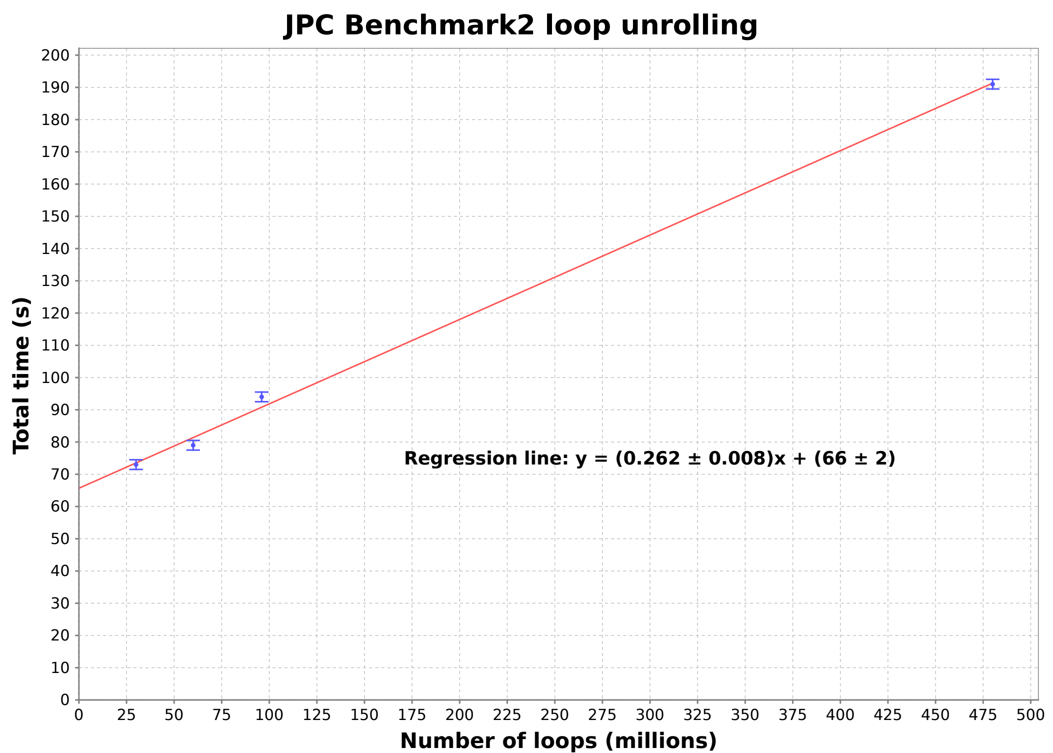


Figure 5.9: Timing results for different loop unrollings of JPC benchmark 2.

```
LOAD0_ECX                reg0 = cpu.eax;
STORE0_EDX               cpu.edx = reg0;
LOAD0_EDX                reg0 = cpu.edx;
LOAD1_EBX                reg1 = cpu.ebx;
XOR                      reg0 ^= reg1;
STORE0_EDX               cpu.edx = reg0;
BITWISE_FLAGS_O32        set flags according to reg0
LOAD0_EDX                reg0 = cpu.edx;
STORE0_EAX               cpu.eax = reg0;
LOAD0_EDX                reg0 = cpu.edx;
LOAD1_IB                 reg1 = (byte) 0x1F;
0x1F
SAR_O32                  reg2 = reg0; reg0 >>= reg1;
STORE0_EDX               cpu.edx = reg0;
SAR_O32_FLAGS            set flags according to reg0
LOAD0_ECX                reg0 = cpu.ecx;
IDIV_O32                 divide and store result:remainder in eax:edx
LOAD0_ECX                reg0 = cpu.ecx;
LOAD1_ID                 reg1 = 1;
0x1
ADD                      reg2 = reg0; reg0 = reg2 + reg1;
STORE0_ECX               cpu.ecx = reg0;
ADD_O32_FLAGS            set flags according to inputs and result
LOAD0_EBX                reg0 = cpu.ebx;
LOAD1_EAX                reg1 = cpu.eax;
ADD                      reg2 = reg0; reg0 = reg2 + reg1;
STORE0_EBX               cpu.ebx = reg0;
ADD_O32_FLAGS            set flags according to inputs and result
LOAD0_ECX                reg0 = cpu.ecx;
LOAD1_ID                 reg1 = 0x3B9ACA00;
0x3B9ACA00
SUB                      reg2 = reg0; reg0 = reg2 - reg1;
SUB_O32_FLAGS            set flags according to inputs and result
EIP_UPDATE               cpu.eip += x86 length of this block;
LOAD0_IB                 reg0 = (byte) 0xE8;
0xE8
JNZ_08                   if the zero flag is not set, add reg0 to EIP
```

Listing 5.11: Benchmark code - JPC microcode instructions and their implementation (37)

5.4.4 Possible improvements

Analysing the code further suggests many more improvements and gives an indication of the attainable speed in the short term.

Listing 5.11 shows the decoded microcodes for the loop body in benchmark 1. These are executed by the interpreter in JPC before the block is compiled. These basically involve loading values into the virtual registers *reg0* and *reg1*, performing an actual operation, and storing the result from *reg0* back out to the appropriate CPU register variable.

Once the block has been executed 1024 times, the JPC compiler will compile it into a new class, which is then able to be compiled by the Java Hotspot compiler to native code. The rough output of the JPC compiler for this block is shown in listing 5.12. This shows that all the redundant and expensive flag operations in the middle of the block have been removed.

A JPC Compiled block is significantly faster than the JPC interpreter, largely because it allows the Java Hotspot compiler to see the structure and relations between instructions, at least within a basic block. Analysis of the code in listing 5.12 shows exactly what corresponds to the fixed lookup cost c calculated earlier. Bytecodes 5, 29 and 130 are loading the needed variables from the CPU to local variables. Bytecodes 176 to 269 correspond to calculating all the remaining flags (not needed for the jump) and storing these flags and the modified CPU registers back in the CPU object. A good JPC loop compiler could remove all these costs.

Assuming that the average basic block length in typical x86 execution is ~ 3 instructions, the minimum speed up from implementing a good loop compiler for JPC can be estimated. It is a minimum because once the Java Hotspot compiler can see the loop structure of the guest x86 code, then it has a lot more scope for optimisation. The typical block would therefore take $t_{now} = c + w$ microseconds to execute. We can estimate w for the typical 3 instruction block by dividing the work time for our 9 instruction benchmark by 3. Therefore

$$t_{now} \sim (0.26 + 0.13/3)\mu S \sim 0.303\mu S$$

The estimated maximum time with a loop compiler that eliminates c is

$$t_{loop} \sim 0.13/3\mu S \sim 0.0433\mu S$$

So the expected speed up is at least

$$\frac{t_{now}}{t_{loop}} \sim 7$$

The other potential area where Hotspot could be given some help is in the method size. The JPC Compiled block for the first benchmark was 650 bytes long, whilst only emulating 9 x86

```
5:   getfield      //Field Processor.ecx
11:  invokestatic  //Method ProtectedMode.reg0_LOAD0_ECX_ecx
19:  invokestatic  //Method ProtectedMode.reg0_LOAD0_ECX_ecx
22:  invokestatic  //Method ProtectedMode.edx_STORE0_EDX_reg0
25:  invokestatic  //Method ProtectedMode.reg0_LOAD0_EDX_edx
29:  getfield      //Field Processor.ebx:I
35:  invokestatic  //Method ProtectedMode.reg1_LOAD1_EBX_ebx
38:  invokestatic  //Method ProtectedMode.reg0_XOR_reg0_reg1
41:  invokestatic  //Method ProtectedMode.edx_STORE0_EDX_reg0
47:  invokestatic  //Method ProtectedMode.reg0_LOAD0_EDX_edx
50:  invokestatic  //Method ProtectedMode.eax_STORE0_EAX_reg0
58:  invokestatic  //Method ProtectedMode.reg0_LOAD0_EDX_edx
61:  ldc           //int 31
63:  invokestatic  //Method ProtectedMode.reg1_LOAD1_IB_immediate
66:  invokestatic  //Method ProtectedMode.reg0_SAR_O32_reg0_reg1
69:  invokestatic  //Method ProtectedMode.edx_STORE0_EDX_reg0
75:  invokestatic  //Method ProtectedMode.eax_IDIV_O32_reg0_eax_edx
83:  invokestatic  //Method ProtectedMode.reg0_LOAD0_ECX_ecx
86:  ldc           //int 1
88:  invokestatic  //Method ProtectedMode.reg1_LOAD1_ID_immediate
91:  invokestatic  //Method ProtectedMode.reg0_ADD_reg0_reg1
94:  invokestatic  //Method ProtectedMode.ecx_STORE0_ECX_reg0
106: invokestatic  //Method ProtectedMode.edx_IDIV_O32_reg0_eax_edx
111: invokestatic  //Method ProtectedMode.reg0_LOAD0_EBX_ebx
116: invokestatic  //Method ProtectedMode.reg1_LOAD1_EAX_eax
119: invokestatic  //Method ProtectedMode.reg0_ADD_reg0_reg1
122: invokestatic  //Method ProtectedMode.ebx_STORE0_EBX_reg0
130: getfield      //Field Processor.eip
133: ldc           //int 24
135: invokestatic  //Method ProtectedMode.eip_EIP_UPDATE_eip_x86length
138: ldc           //int -24
140: invokestatic  //Method ProtectedMode.reg0_LOAD0_IB_immediate
145: invokestatic  //Method ProtectedMode.reg0_LOAD0_ECX_ecx
151: ldc           //int 1000000000
153: invokestatic  //Method ProtectedMode.reg1_LOAD1_ID_immediate
159: invokestatic  //Method ProtectedMode.reg0_SUB_reg0_reg1
165: invokestatic  //Method ProtectedMode.zflag_SUB_O32_FLAGS_reg0
171: invokestatic  //Method ProtectedMode.eip_JNZ_O8_cs_eip_reg0_zflag
176: invokestatic  //Method ProtectedMode.reg2_SUB_reg0
184: invokestatic  //Method ProtectedMode.cflag_SUB_O32_FLAGS_reg2_reg1
189: invokestatic  //Method ProtectedMode.pflag_SUB_O32_FLAGS_reg0
198: invokestatic  //Method ProtectedMode.aflag_SUB_O32_FLAGS_reg0_reg1_reg2
205: invokestatic  //Method ProtectedMode.sflag_SUB_O32_FLAGS_reg0
214: invokestatic  //Method ProtectedMode.oflag_SUB_O32_FLAGS_reg0_reg1_reg2
219: invokevirtual //Method Processor.setOverflowFlag
224: invokevirtual //Method Processor.setSignFlag
229: invokevirtual //Method Processor.setZeroFlag
234: invokevirtual //Method Processor.setAuxiliaryCarryFlag
239: invokevirtual //Method Processor.setParityFlag
244: invokevirtual //Method Processor.setCarryFlag
249: putfield      //Field Processor.eip
254: putfield      //Field Processor.ebx
259: putfield      //Field Processor.edx
264: putfield      //Field Processor.ecx
269: putfield      //Field Processor.eax
274: ireturn
```

Listing 5.12: Benchmark code - JPC compiled block JVM bytecode summary (650 bytes long)

instructions. As can be seen from listing 5.12, only 275 of these are the actual body of the execution. The remainder are set up to handle virtual CPU exceptions. Given that these exceptions are rare it would make sense to put the code for handling them in a separate method. Then if the exceptions do turn out to be common, Hotspot can inline the method and remove the indirection cost. However, with a much smaller execute method, hotspot can do much more inlining before it hits the code size limit. This would not affect the current basic block compiler, but is likely to have a significant effect on a loop compiler.

A fully optimised JPC should thus be able to execute code at speeds exceeding 70% of native speed. Once even 50% is reached, more possible uses become attractive and economically feasible.

Chapter 6

Map-Reduce and a Nereus based implementation, Mycelia

The map and reduce constructs from functional programming are used to apply the same algorithm to a large number of chunks of data and collect and combine the results at the end. Map-Reduce is a distributed programming paradigm and associated interface, based on the functional programming concepts, which is used to analyse large datasets in parallel using many physical machines. Many problems have been adapted to use Map-Reduce, and particle physics analyses could also benefit. This chapter gives an introduction to the Map-Reduce paradigm and algorithm, followed by a description of Mycelia, an implementation of the Map-Reduce algorithm to run on a Nereus desktop cloud.

6.1 Introduction

Map-Reduce is a programming paradigm, and associated interface, for processing large data sets. The Map-Reduce interface was popularised by Google in a seminal 2004 paper [104], by Jeffrey Dean and Sanjay Ghemawat. This paper introduced Google's implementation, Google MapReduce, which the company used for many tasks, including the creation of a graph of links on the Internet, which is crucial to the company's search engine. Map-Reduce has been adopted

by many other companies for processing very large datasets. Yahoo use an open source implementation, Hadoop [105], which they have used to win the TeraSort benchmark competition.

The idea was inspired by the map and reduce functions from functional programming. Functional programming treats computation as the evaluation of mathematical functions and avoids side-effects, state and mutable data (i.e. calling a function with the same arguments will always return the same result - there is no possibility for other state to change the result). It is exactly this lack of side effects that allows a map function to be evaluated on different sections of input data in parallel.

The Map-Reduce interface allows a large class of algorithms that need to be run over very large datasets to be written without any knowledge of distributed programming. This enables the user to focus on the algorithm implementation and leave all the details of parallelisation, distribution, failure handling and result collection to the Map-Reduce framework. The computation can then be sped up by taking advantage of a large number of machines.

6.1.1 Problems handled by a Map-Reduce framework

A Map-Reduce implementation typically runs on hundreds to thousands of machines, and thus machine failures are likely and must be handled gracefully, maintaining correctness of the computation. A failure is any condition that prevents a machine from participating in a computation. Failures can be caused by hardware faults in any of the major components (hard drive, power supply, processor, RAM) in the contributing PCs, although the main sources are the hard drives, motherboard and the network. A study by Google [106] indicated a typical disk drive failure rate of 2-10% per annum. A larger study by Gartner [107] indicated annualized desktop PC failure rates between 5 and 15%.

Network failures are even more unpredictable as they may or may not represent a hardware failure in one of the contributing machines. A network failure is any condition that causes a machine to be inaccessible over the network. This could be a network card failure in a PC, a router failure on the network, or simply a bandwidth overload of a segment of the network.

Annualized failure rates as high as 15% suggest that for a cluster of around 1000 PCs, a failure should be expected roughly every 2 days. Writing distributed code that can handle failures and still get correct results is non-trivial. This is one of the large benefits of a well tested Map-Reduce implementation, which abstracts the user away from the unreliable underlying machines and network.

The general problem of parallelising a computation is an even more complicated problem than handling failures. Once a problem has been coded to conform to a Map-Reduce framework it is implicitly parallel and can automatically make use of a large Map-Reduce cluster without any extra effort from the user.

Many different types of problems are a good fit for a Map-Reduce framework. The basic requirements for suitability are that most of the processing time is spent applying the same code to different input data, possibly with an end phase where the results are simplified. These will roughly correspond to the Map and Reduce phases of the translated algorithm, which are defined in the next section.

6.1.2 The algorithm

Logically, Map-Reduce is defined by two functions, map and reduce:

$$\begin{aligned} map : (K_1, V_1) &\longrightarrow list(K_2, V_2) \\ reduce : (K_2, list(V_2)) &\longrightarrow list(V_2) \end{aligned}$$

The map function takes key-value pairs in one domain, (K_1, V_1), and outputs lists of key-value pairs in another domain, (K_2, V_2). The reduce function takes a key and all the values associated with that key and outputs a possibly smaller list of values. The order of the intermediate list values supplied to the reduce function is not specified. The reduce function should not depend on it if deterministic operation is desired. The order that intermediate keys are processed by the reduce function is also not specified. However, all parallel implementations

investigated impose a certain order on the processing of intermediate values, by interposing a sort phase which will be discussed in the next section. In this case the reduce function comes in a "left" and "right" form, where the reduce function is applied to the list of values in ascending or descending order. Mycelia, the Nereus implementation discussed below, is restricted to associative reduce operations where the order of processing values is irrelevant. An associative reduce function R satisifes the following,

$$R(K, R(K, list_1) + R(K, list_2)) = R(K, list_1 + list_2)$$

where $+$ mean concatenation of lists. In words, this means that reducing two lists, joining the resulting lists and reducing that final list gives the same result as first joining the two lists and doing a single reduce.

6.1.3 Usage examples

A simple example is a word count function that counts the occurrence of words across multiple large text files. The easiest way to implement this is shown in listing 6.1 and was used to benchmark the Nereus implementation of Map-Reduce and investigate its scaling capabilities. This gives a histogram of the word counts for all the words across all the input documents.

Other text based examples include distributed grep (searching for lines matching a supplied pattern), processing web request logs to create page access frequency tables, web page link graphs, and distributed sort (this relies on a sort phase being implemented).

One of Google's uses of Map-Reduce is to reverse web link graphs. This uses a map function that takes an html page as input and outputs a (target, source) pair for every URL link in the page. The reduce function is the identity function. This gives output of the form (target, list(source)).

A simple particle physics example phrased as a Map-Reduce job is event selection. The input would be a large sample of data from a particle collider such as the LHC. The map function might take key-value pairs of (event number, event data) and output the same pair if the

```
map(URL key, String value)
{
// key is a URL to a document to read
// value is the contents of the document
   for (word w: value)
      addPair(w, ``1'');
}

reduce(String key, list(Integers) value)
{
// key is a word
// value is a list of word counts
   int result = 0;
   for (Integer v: values)
      result += v;
   // output a list with result as the only element
   addPair(key, new list(result));
}
```

Listing 6.1: A word count implementation

```
map(run number, raw data of events) = list(mass bin, number in bin)

reduce(mass bin, list(number in bin)) = (mass bin, total number in bin)
```

Listing 6.2: A mass histogram implementation

event satisfies some criteria, for example, selecting events with at least two leptons with large transverse momentum. In this case the reduce function would be the identity function.

A more complex particle physics example could be to compile a mass histogram, shown in listing 6.2.

Another non-textual application is rendering a film frame by frame. Here the map function would take the scene file defining a frame and render it. The reduce function would take frames corresponding to a contiguous section of time and link them together, assuming they were sorted into chronological order.

6.1.4 Distributed implementation

A general Map-Reduce job consists of 4 phases: a map phase, sort phase, reduce phase and a commit phase. Figure 6.1 shows the logical data flow for a Map-Reduce job with the input split

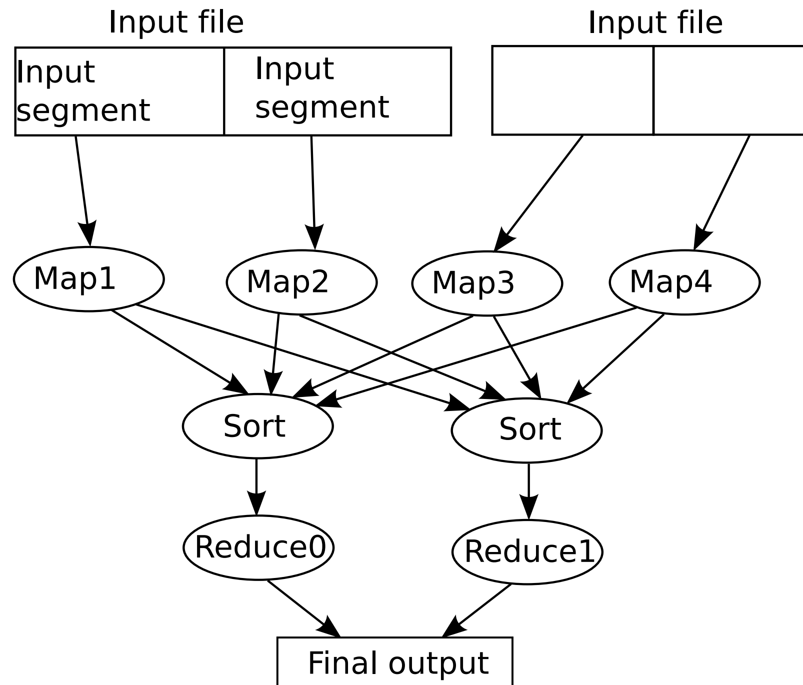


Figure 6.1: Map-Reduce

into 4 blocks, and the intermediate data split into two partitions.

At the beginning of the map phase, the input data for the Map-Reduce job is split into blocks which can be handled by independent invocations of the map function. These are generally similar sized blocks although there is no restriction on this. Each application of the map function parses a key-value pair from its input block and outputs intermediate key-value pairs into temporary storage.

The intermediate data is partitioned into a number of sets which can be handled independently in the remaining phases, thus simplifying the computation. The number of sets the data is divided into, R, is an integer specified by the user at the beginning of a Map-Reduce job. Intermediate keys are assigned to a partition by a partition function. The default implementation normally uses $hash(key) \bmod R \equiv r$ to select keys for a partition r. The aim for a partition function is to distribute the keys roughly evenly over the number of partitions in order to enable load balancing by the framework.

Most implementations of Map-Reduce include a non-trivial sort phase before the reduce phase. The sort phase arranges the keys in each partition in ascending order. To specify an order a binary relation on the keys must be specified, or in Java terms a Comparator. Most

implementations use a default relation which is an extension of alphabetical order as most uses involve only textual keys.

In the reduce phase, the reduce function is applied to all the keys in the input partition, and the results are stored locally. A typical reduce function consolidates all the values for a particular key into one output value.

The collection phase is not normally specified separately from the reduce phase, however here it is defined as the phase where the results of the reduce phase are committed to trusted storage. This storage could be a distributed filesystem or even a single file server.

6.1.5 Existing implementations

The most famous implementation of Map-Reduce is Google's own version. However this is not available (in source or binary) outside of Google, and only published results are available [104].

One of the most popular open source implementations is Hadoop [105][108]. This is written in Java and used by major companies like Yahoo, Baidu, Facebook, Twitter and Amazon [109]. Yahoo uses Hadoop for creating their search indices and supporting their advertising systems. Similarly, Baidu uses Hadoop for analysing search logs, and data mining of web pages. Facebook uses Hadoop for analysing internal logs. Twitter uses Hadoop to analyse tweets, and log files. Amazon uses Hadoop to build their product search indices. Yahoo holds the biggest publicly known Hadoop cluster with 4000 nodes [109]. As well as a Map-Reduce implementation, Hadoop also contains a distributed filesystem, HDFS. This is used by default to store and read the output/input of a Map-Reduce job.

GridGain [110] is another open source Java implementation of Map-Reduce. Gridgain does not have a distributed file system, and has some limitations: rather than the conventional reduce phase with many reduce tasks, there is a single reducer for a Map-Reduce job, and a single output file. This will not scale well for data intensive applications, of which Particle Physics analysis is our prime example. For CPU bound jobs without a complicated reduce phase GridGain works well.

Twister is a Java implementation of iterative Map-Reduce, i.e., many Map-Reduce jobs chained together [111]. This implementation's emphasis is on efficiency gains compared to Hadoop via caching of static data between map invocations, which pays off when many similar Map-Reduce jobs are chained together. Twister does not have a distributed filesystem, and does not implement fault tolerance.

Most of these implementations are successful in their own niche. However, to enable truly massively parallel computations on the scale of millions of nodes it is desirable to have a Map-Reduce implementation that runs on a desktop grid such as the Nereus network. In order to run in Nereus, it must firstly be Java (or JVM bytecode). Secondly, it must fit inside the applet sandbox to run on the Nereus clients. This means that there can be no file access, which rules out Hadoop, and all communication must be done over http, using the Nereus servers as proxies. It is likely the software needs to be modified to use the Nereus API and thus the source code should be available.

So the requirements for a Nereus Map-Reduce implementation are:

1. Written in Java
2. Fit inside the applet sandbox
3. Available source

As well as these requirements, the environment provided by a desktop cloud is quite different from the clusters typically used for running Map-Reduce implementations. Clients on a desktop cloud could be affected not only by the usual hardware and network failures, but also by temporary failures of another kind when the machine is powered off or in active use. The Nereus client runs with low thread priority because it is designed to run in the background. The implication of this is that if a user of the PC the Nereus client is running or starts doing some CPU intensive work, then the Nereus client essentially stops running, because it is never scheduled time on the CPU. This is equivalent to a node dropping off the network completely. Furthermore, because Nereus is a desktop grid, it is expected that donors will occasionally kill their browser window or turn off their machines, thus removing their CPU from the Nereus

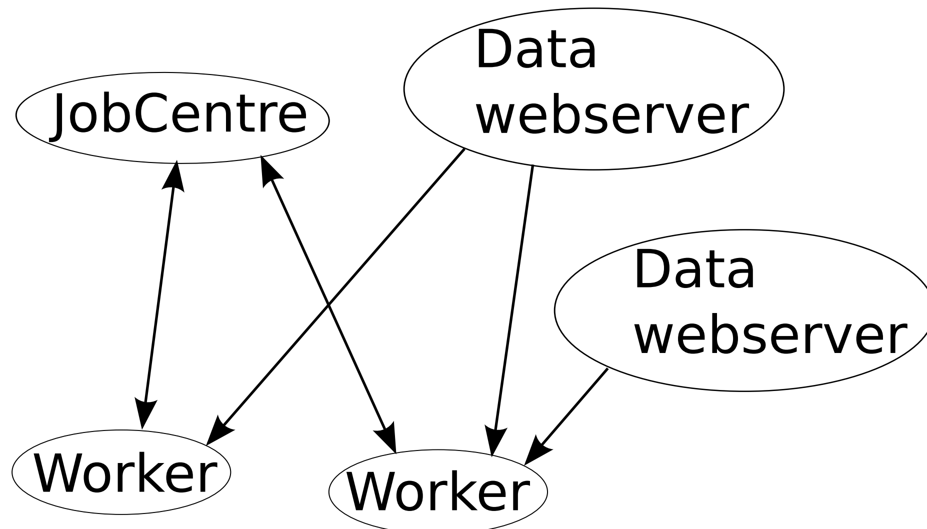


Figure 6.2: Mycelia architecture

cloud. The Nereus network can be very dynamic, and this must be handled by a Map-Reduce implementation. The above mentioned factors were the motivation to develop Mycelia.

6.2 Mycelia - Map-Reduce in Nereus

6.2.1 Architecture

Mycelia was implemented with a simple flat client-server architecture, which would satisfy the three requirements. The architecture is shown in Figure 6.2. The Job Centre is the center of the computation. It handles the allocation of work to the workers and ensures that all parts are successfully completed. It is a separate http server outside of the nereus network that communicates with the nereus clients via http.

Most Map-Reduce implementations have an additional property, a sort phase. In this phase, which happens between the map phase and the reduce phase, all the key-value pairs for a given key are collected together. This makes the reduce function easy to apply by simply passing it an iterator to the list of values for each key. Mycelia omits this step as the typical particle physics analysis does not need it. It can always be added at a later stage if necessary.

There are some important consequences of implementing Map-Reduce for Nereus. The first

is that there is no disk storage directly attached, and thus all intermediate results need to be stored in memory. At the time of writing, most Nereus clients will be given roughly 1 GiB of memory. This means that the maximum total data storage in intermediate stages is about 1 GiB$\times$number of clients. For example, a job with a terabyte of output would need at least 1000 clients (which would need 1 or 2 nereus servers). As a comparison, Google has claimed to sort a terabyte of data using Google MapReduce on approximately 1,800 machines [104]. However, Google also claim to be able to sort a petabyte using only 4,000 machines. The amount of memory allocated to applets is expected to increase rapidly, along with the amount of memory available on desktop PCs.

6.2.2 Data Flow

There are several overall phases to the execution of a MapReduceJob. The conceptual flow of this data is shown in Figure 6.4. The first phase is the Map phase, where the map function is applied to all the input data. This is followed by the Reduce phase which collects and aggregates all the output data from the map phase. The final phase, Collect, is where the output data from the reduce phase is stored to a trusted filesystem. This is illustrated in Figure 6.3. The input data for a MapReduce job is assumed to come from a group of standard webservers.

A MapReduce job consists of a Map job and a Reduce job, which are each composed of tasks (Figure 6.5). When a MapReduce job is started the Job Centre splits the Map job into Map tasks which are then distributed amongst the workers. Each worker stores the output from its Map task locally in memory. The Job Centre keeps track of which workers store the output of which tasks, which is used when a worker fails. When all the Map tasks have been completed, the Job Centre transitions to the Reduce phase. Here the Reduce job is split into Reduce tasks which are distributed amongst the workers. A Reduce task involves the worker being assigned a set of intermediate keys for which the associated lists of values must be processed. To do this it must contact every other worker in the system to ask for any such values. The results of a Reduce task are stored locally in memory on the worker. When all the Reduce tasks are complete the Job Centre starts transferring the results to the specified output file system, or this

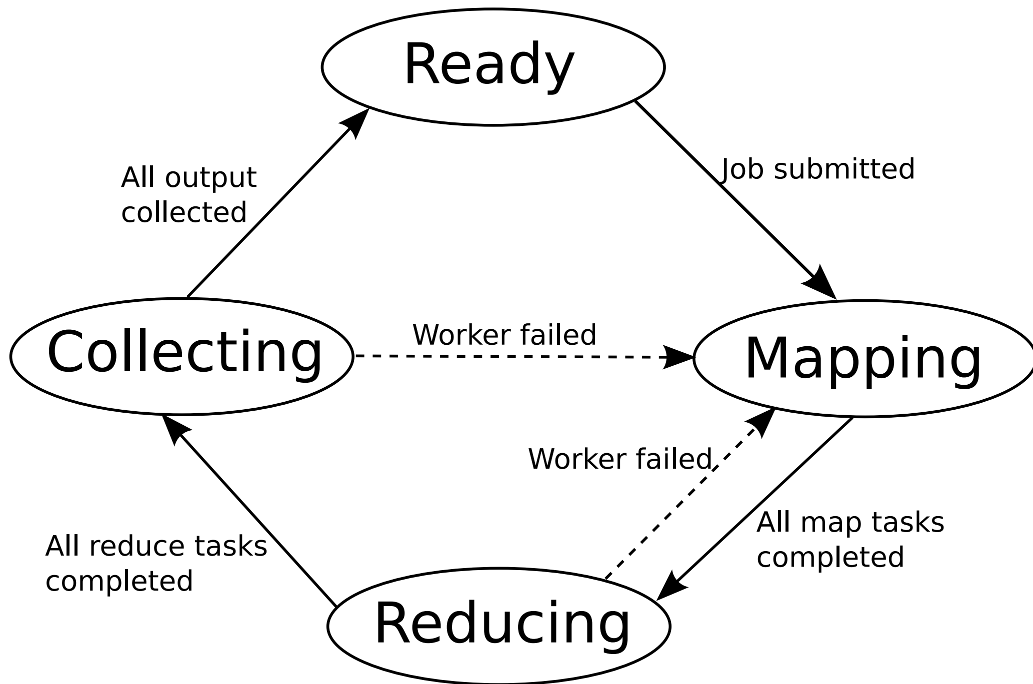


Figure 6.3: Job Centre state diagram

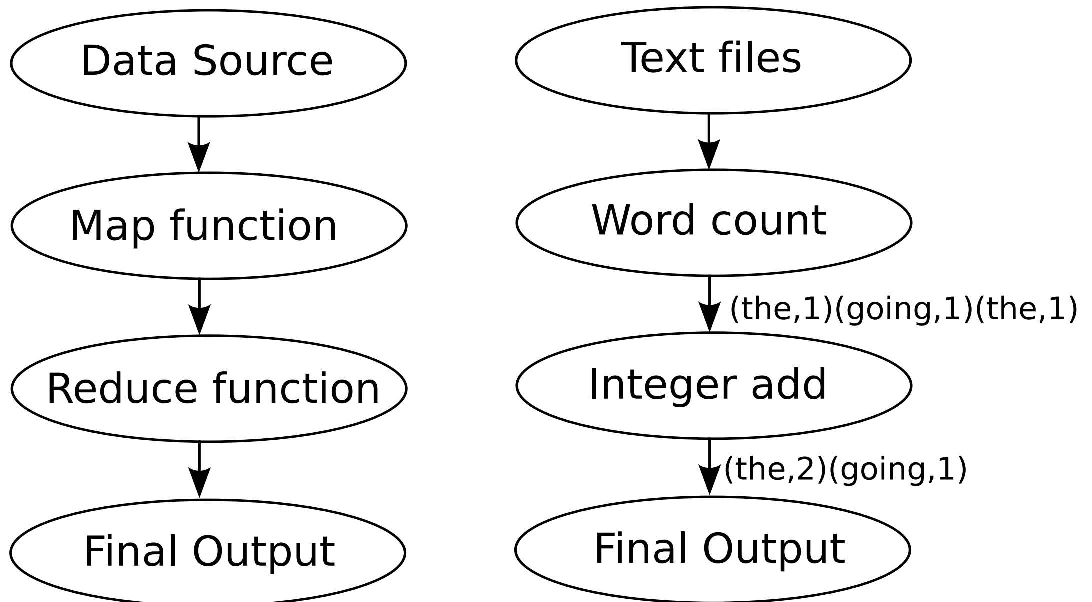


Figure 6.4: Conceptual data flow in Map Reduce

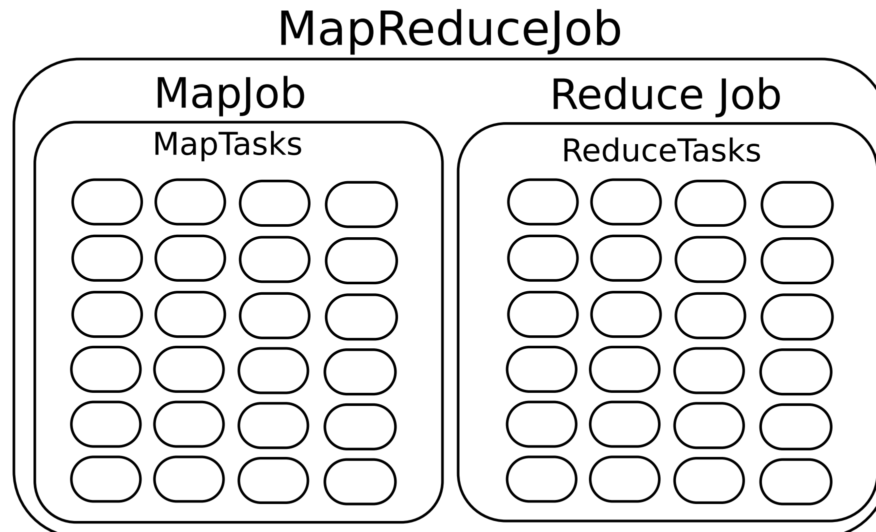


Figure 6.5: MapReduceJob structure in Mycelia

can be done directly by the workers on completion of each task. The data flow throughout this process is illustrated in Figure 6.6.

6.2.3 Implementation

The interaction of the key classes in Mycelia during a MapReduce job is shown in fig 6.7. It shows how the data flows and is encoded and decoded at various points. The two key interfaces which a user must implement to use Mycelia are the Mapper and Reducer interfaces. The Mapper interface is defined in listing 6.3. A simple example of a Mapper which can be used to count the number of occurences of each word in the input is shown in listing 6.1. This simply outputs a (word, 1) pair for every word in the input. The Reduce interface is shown in listing 6.5. A summation Reducer can be used to add up the 1's in the mapper output to get a total word count for each word. This is shown in listing 6.6.

The majority of users will only use these two interfaces, assuming that they are using previously defined input and output types. If not then they will also need to implement a *pair reader* for their desired types. A *pair reader* takes the raw input data as a stream of bytes and parses

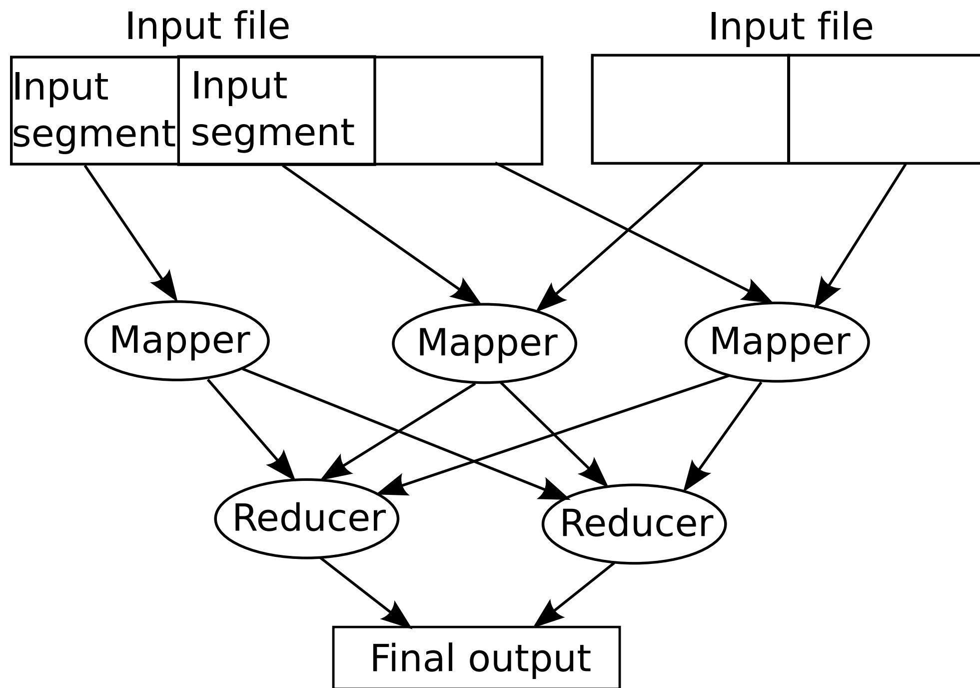


Figure 6.6: Detailed dataflow during MapReduce job

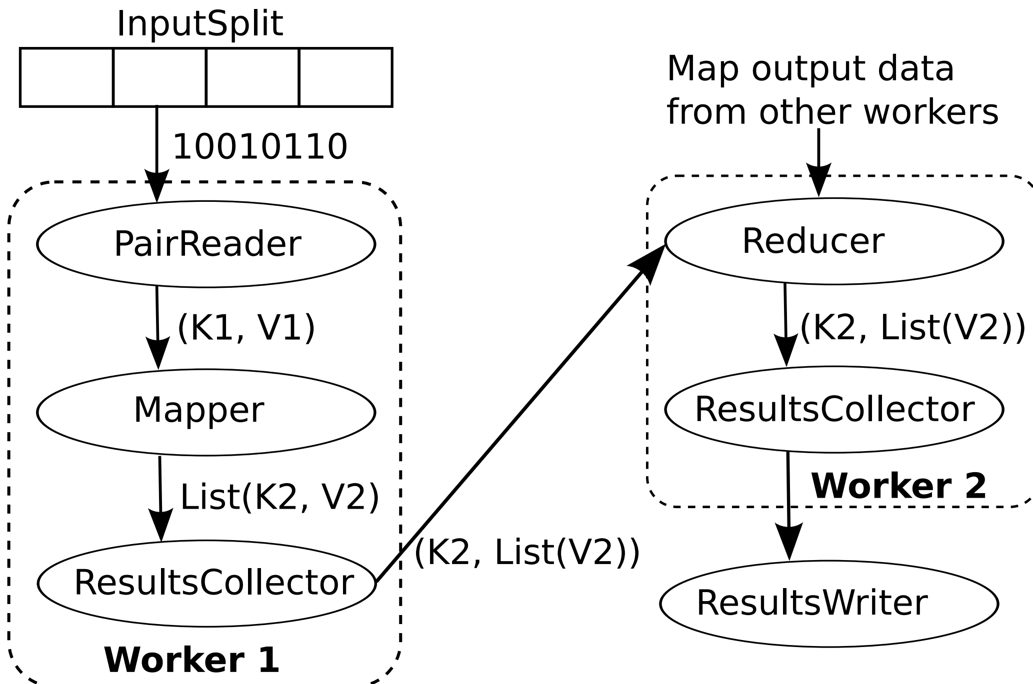


Figure 6.7: Class usage in Mycelia

```
public interface Mapper<K1, V1, K2 extends Serializable, V2 extends
    Serializable>
{
    /**The Map function of the Map-Reduce
     *
     * @param taskID the task id of the form m34:42
     * @param key read from the pair reader
     * @param value read from the pair reader
     * @param output the results collector to write output to
     */
    void map(String taskID, K1 key, V1 value, ResultsCollector<K2> output);

    /**A way to add further configuration options to this mapper
     *
     * @param options map of further parameters to configure this mapper
     */
    void configure(Map<String, String> options);
}
```

Listing 6.3: Mapper interface

```
public class WordCount implements Mapper<Integer, String, String, Integer>
{
    public void map(String taskID, Integer key, String value,
        ResultsCollector<String> output)
    {
        StringTokenizer st = new StringTokenizer(value);
        while (st.hasMoreTokens())
            output.addPair(taskID, st.nextToken(), new Integer(1));
    }

    public void configure(Map<String, String> options) {}
}
```

Listing 6.4: WordCount Mapper

```
public interface Reducer<K2 extends Serializable, V2 extends Serializable>
{
    /** The Reduce function of the Map-Reduce
     *
     * @param taskID of the form r123:45
     * @param key key from pair reader
     * @param values iterator over values from pair reader
     * @param output a results collector to deposit output into
     */
    void reduce(String taskID, K2 key, Iterator<V2> values,
        ResultsCollector<K2> output);

    /**
     *
     * @param options map of further parameters to configure this reducer
     */
    void configure(Map<String, String> options);
}
```

Listing 6.5: Reducer interface

```
public class IntegerAdd implements Reducer<String, Integer>
{

    public void reduce(String taskID, String key, Iterator<Integer> values,
         ResultsCollector<String> output)
    {
         int result = 0;
         while(values.hasNext())
              result += values.next();
         output.addPair(taskID, key, new Integer(result));
    }

    public void configure(Map<String, String> options) {}
}
```

Listing 6.6: Summation Reducer

```
public interface ResultsCollector<K>
{
    public void addPair(String taskID, K key, Serializable value);

    public List<K> getKeysFor(String job, int reducetask, int R);

    public List<StreamHolder> getValuesForKey(String job, K key);

    public boolean validateTask(String jobid);
}
```

Listing 6.7: ResultsCollector

this into (Key, Value) pairs that are then fed into the map function. Most of the difficulty for a pair reader is handling key-value pairs that overlap a boundary of the input split. In this case, the pair reader simply keeps reading past the boundary until the end of the current key-value pair. The corresponding pair reader for the next section of the input split ignores the data up to the first start of a key-value pair. This ensures that no double counting occurs in the input.

ResultsCollector is an interface for temporary storage systems which store the intermediate results from the map tasks until they are fed into the reduce tasks, and also for storing the reduce results until they are uploaded. The main methods in the interface are shown in listing 6.7. The addPair() method is used by the map task to store its output. Reduce tasks then access the data (automatically) via the getKeysFor() and getValuesForKey() methods. Importantly, before a Map or Reduce task's output becomes visible, the worker checks with the server to verify that no other worker has completed the same task. Upon receiving a positive response the output from the task is then atomically committed to the data store which is externally readable. The current implementation uses a MemoryResultsCollector. This stores all the intermediate data in memory until the end of the current job. Internally, a results collector stores the data in the structure shown in Figure 6.8. This structure makes management of, and access to, the data quite easy. As with all Nereus applications, the workers must store all their temporary data in memory. This limits the amount of data that can be processed by a fixed number of workers to that whose intermediate data will fit in the collective memory of all the workers. Typically this is 512 MiB to 4096 MiB per worker. As a result of this CPU-intensive jobs are more suited to the Nereus implementation of Map-Reduce.

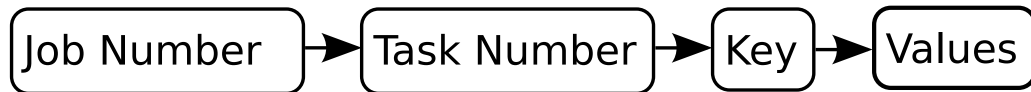


Figure 6.8: ResultsCollector data structure: each arrow represents a map

```
public interface InputSplit
{
public void setInput(URI[] inputs, int m);

    public int numberOfTasks();

    public long getStart(int task);

    public long getEnd(int task);

    public URI getSource(int task);
}
```

Listing 6.8: InputSplit interface

The way the input data is split into chunks, one for each map task, is determined by the InputSplit. This simple interface is shown in listing 6.8. The input split is given the set of URI's, each of which is an input file, and a suggested number of map tasks M. The input split then determines the actual number of map tasks, the input URI for each task, and the start and end positions in that file.

There are two example input splits implemented. The first, FileSplit, simply gives one entire input URI to each map task. This ignores the requested number of maps M and instead has a number of map tasks equal to the number of input files. This is useful in certain cases where the input files have a header on each file before the relevant data which will be parsed to key value pairs, e.g., particle physics n-tuple files.

The second input split provided is the EvenSplit. The even split aims to split the entire input data into M evenly sized chunks of data, where M is the suggested number of map tasks. In practice this is not possible. First the total size S of the input data is calculated. Then $T = S/M$ is the ideal number of bytes per task (rounded down). Each input file is split into chunks of size T and if there is any left over, this remainder is allocated to another map task. This means the actual number of map tasks is between M if everything divides perfectly, and $M+I$, where I is the number of input URIs. This is useful when there are lots of key-value pairs in each input

file, and there is no other extraneous data in the files.

6.2.4 Consistency, correctness and failure handling

Failures occur in any large system, and even more so in a distributed system. In Nereus we are assuming donors can come and go at will. Therefore, any application must be tolerant to failures. In any distributed system, the shared mutable state needs to be managed carefully to ensure correct operation. In Map-Reduce there are many different agents all reading and writing data at different times and at different places. The first step to ensuring consistency is the Job Centre. Having a central server that keeps track of all the workers and data and directs the work is a crucial simplifying step, but one that ultimately limits the scalability.

The Job Centre maintains a list of active workers connected to the server. New workers can be added to this list at any time. This enables the pool of workers to dynamically change. A worker failure during a map reduce job is detected when another worker, or the server, attempts to contact it. This worker is then removed from the list of active workers. If that worker had completed (and verified) some tasks, then the Job Centre requeues these tasks for submission to other workers. As a result, the Job Centre may have to revert to the map phase until those tasks are finished and it can return to the reduce phase of the job (as illustrated in Figure 6.3).

The Job Centre also keeps track of which tasks have been completed, and which worker completed each task. This information is used to tell a worker doing a reduce task which workers to contact for input data. Clearly this information is also needed when a worker fails, to know which tasks to rerun, as mentioned above. As well as having a Job Centre which stores all this information, the correctness of the results depends on carefully managing the state of the workers and controlling their reading and writing of data.

A Map-Reduce worker, once registered with the Job Centre, can be in one of 3 states: READY, WORKING, REPORTING. This is illustrated in fig 6.9. Initially, the worker is in the READY state. When the worker is assigned a map or reduce task by the Job Centre, the worker moves to the WORKING state. Once the task is complete the worker moves to the

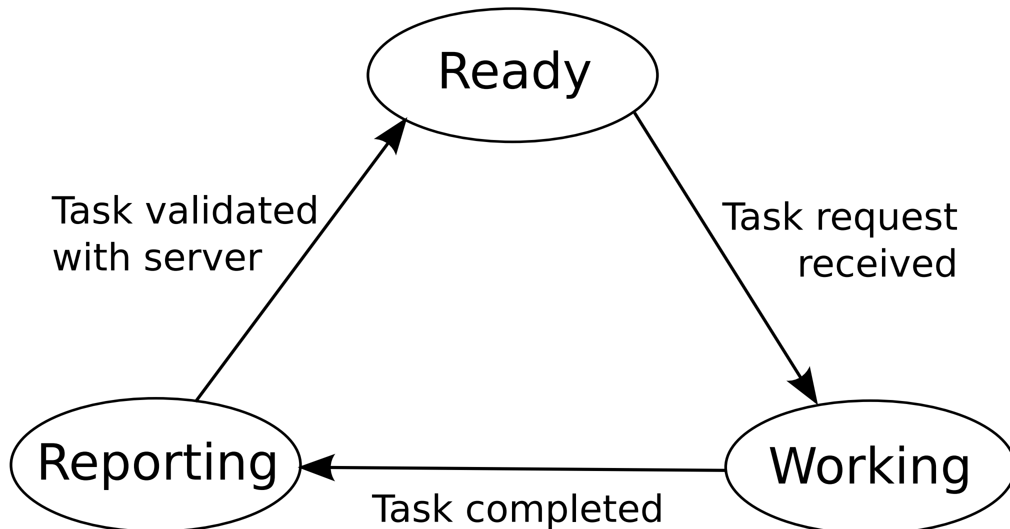


Figure 6.9: Worker state diagram

REPORTING state, where it asks the Job Centre if it should commit the task's output. Task output is committed to a local store in memory on each worker. When a response is received the worker commits the data and then returns to the READY state, waiting for further instructions. A worker will only accept work tasks if it is in the READY state. It will serve requests for data it is storing when it is in any state, except the REPORTING state. This rules out any possibility of it being told by the Job Centre to commit a task's data, and then being asked by another worker for its output before the new data has been committed. This combined with the Job Centre tracking which worker is storing which task's output means that the job is always in a consistent state.

A worker is considered to have failed if an attempt to contact it by the Job Centre or by another worker (which subsequently succeeds in contacting the Job Centre) fails. A worker is allowed up to 5 failures in a given job before it is completely removed from the active worker pool by the Job Centre. Upon any failure of a worker, the Job Centre requeues any tasks that had been completed by that worker.

6.2.5 Redundancy

As demonstrated in the seminal map-reduce paper [104], the total time taken for a Map-Reduce job to complete can be reduced by as much 44% if the last few tasks are multiply submitted. This

is because there will always be a few workers who are slow for various reasons (for example a slower network connection or the host computer has become busy with other work).

Therefore in Mycelia, towards the end of a map job or reduce job the remaining tasks are multiply submitted. Throughout a job the average time taken for a task to be completed (the task duration) is tracked. Once there are free workers, and at least one task has been returned, the remaining unfinished tasks are requeued roughly once every task duration until the job's completion. However, with this mechanism in place, one needs to be careful about double counting results. This is handled by the verification mechanism discussed in the previous section. Essentially, the first worker, and only the first worker, to finish a task is permitted by the Job Centre to commit its output data.

6.3 Tests

The architecture was tested for correctness, failure handling and scaling properties using a distributed word count job.

The input data for the distributed word count was varying numbers of copies of the text of Ulysses by James Joyce [112]. One copy is just over 1.5 MiB of data and counting instances of the characters "the" in 64 copies of this, takes about 3 minutes using grep in Linux on a 2 GHz Pentium with 2GiB of RAM (there are 828096 "the"s).

The results of the distributed grep throughout all the tests were compared with the results of running a single threaded Linux grep process over the same input data. In all cases, even with worker failure, the results were correct.

When workers failed or parts of the network failed, the Job Centre rescheduled any work completed by those nodes and eventually completed the map reduce job.

The scale tests were done to determine the scaling properties of the architecture. The amount of input data to be analysed was varied whilst keeping the number of workers fixed and measuring the total time to complete a map reduce job. This was found to be linear within the range of

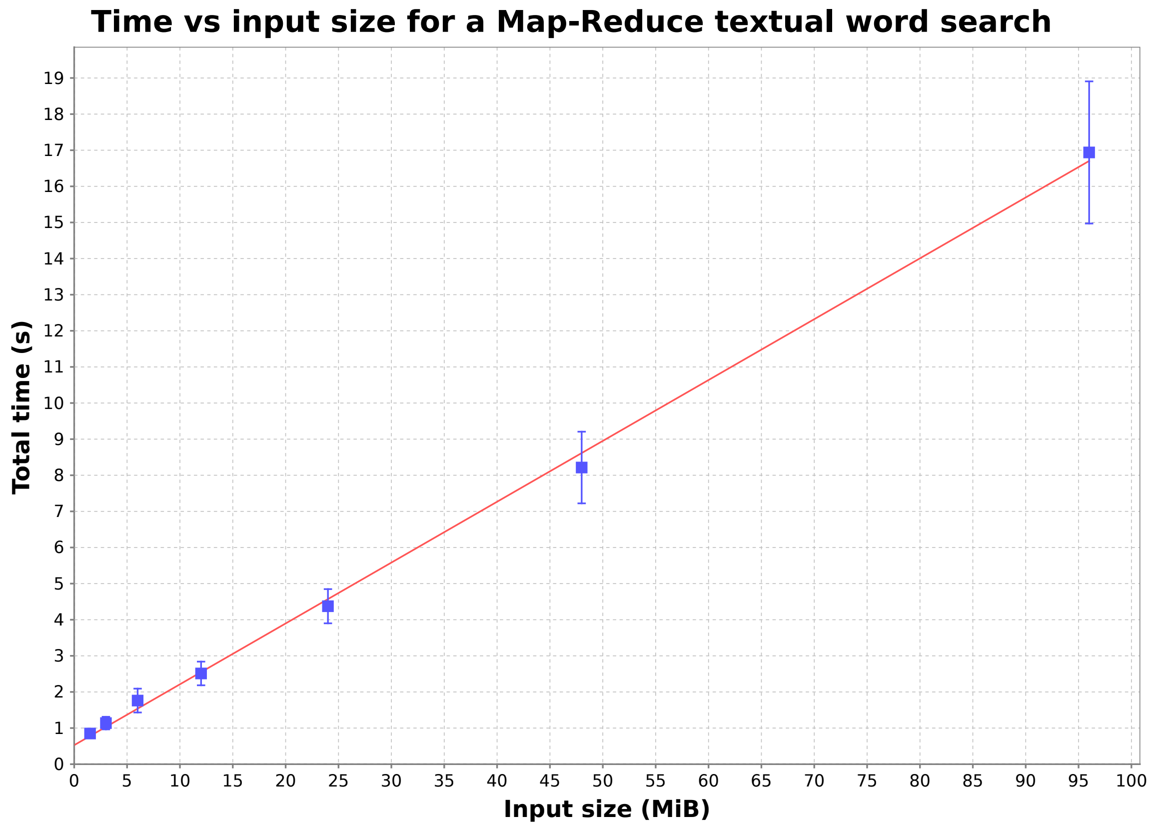


Figure 6.10: Map-Reduce data scaling properties

input sizes tested and within the error. The results are shown in graph 6.10. Each measurement was repeated 10 times and the standard deviation was taken as the uncertainty. These results demonstrate that Mycelia is an accurate Map-Reduce implementation with good initial scaling features.

Chapter 7

BlackMax generator

This chapter discusses massively parallel black hole event generation using the global Nereus cloud. The generator chosen, BlackMax [113][114][115], was written in C and had to be converted to JVM bytecode. The usage of JPC and NestedVM were investigated before directly translating the source code to Java.

7.1 Introduction

Blackmax is an event generator that simulates the production and decay of microscopic blackholes at the LHC. The generator includes many physical possibilities such as blackhole rotation and recoil due to Hawking radiation. Output files from BlackMax are in the Les Houche Accord format and thus can be directly fed as input to Herwig or Pythia for parton shower evolution and hadronization following the decay.

7.2 Execution in JPC

Blackmax was run in JPC in a small Linux distribution, ttyLinux. The numerical results are incorrect due to remaining bugs in the floating point unit (FPU) emulation in JPC. These errors

eliminated the usage of JPC as a viable platform for executing BlackMax, regardless of the execution speed. The floating point unit has yet to be throroughly tested as it was not needed until this point.

7.3 Translating BlackMax to Java

The Blackmax source code was translated to Java in a literal fashion, function by function, without any optimisation. The result was debugged and validated by comparing the output with the output from the native version with the same input parameters and same random seed.

Ideally, the output should be bit-for-bit identical (using a fixed random seed as input). However, this was not possible due to a bug found in Blackmax which affected the control flow.

The difference between two double precision floating point numbers is used to decide whether the blackhole should emit another particle via Hawking decay. The problem is that for the last potential decay, the difference between the two numbers is outside the accuracy of a double precision number (less than 1 part in 10^{17}). However, the Java and the C versions give binary identical results when the control is not changed by this bug. A temporary solution to fix this bug is to test if the difference between the two numbers is greater than the smallest valid difference within double precision - 2^{-53}, although ideally the calculation is rephrased so that two very similar numbers are not compared.

7.3.1 Single machine timing tests

The execution time of the Java version of Blackmax with a typical set of parameters was compared on 3 different machines to that of the native version. The parameters included four extra dimensions, one splitting dimension, 14 TeV collision energy for the the two protons, Planck mass of 1 TeV, and tensionless non-rotating blackholes. The Java version was much faster than the native version on all 3 machines.

Table 7.1: Blackmax code timings for Java and native versions (5000 events).

	Java version (s)	C version (s)
Machine A	68±1	120±1
Machine B	131±1	146±1
Machine C	45±1	56±1

During the timing of Blackmax the structure of the computation must be considered. There is a fixed initial phase before any events are generated, then a roughly constant amount of time per event (excluding dynamic compilation effects). Previous work [116] has focused on profiling and optimising the initial phase, which calculates the cross section used in the event generation. However, this phase takes approximately 1 second on a typical PC and, given that the remaining event generation takes of the order of $\frac{1}{50}$ second per event, is not relevant to the total running time, where thousands of events are generated.

The native gcc compiled C version (with optimisation level 2) and the Java version with the default client compiler were profiled on three different machines: Machine A - a 32-bit laptop with 2 GiB of RAM and a 2GHz Intel Pentium M 760 processor, Machine B - a 32-bit desktop with 3 GiB of RAM and a 3 GHz Intel Pentium 4 processor with hyperthreading and Machine C - a 64-bit server with 8 GiB of RAM and a 3 GHz Intel Core2 Duo (E8400) processor. As can be seen from the times in Table 7.1, the Java version of BlackMax significantly outperforms the native version on all platforms.

7.4 An alternative possibility - execute in another emulator, NestedVM

Another possible route was discovered to enable running of native code within a Java applet - a Java based emulator of the MIPS instruction set, NestedVM. BlackMax was recompiled to MIPS machine code and profiled using NestedVM to assess this approach for feasibility whilst JPC is not accurate enough.

BlackMax is a small project, of the order of a six thousand lines of C code and one thousand

Table 7.2: MIPS emulator timings on a typical office desktop pc (machine B from section 7.3.1) using Java version 6.14.

Running mode	Time	
	10 events	100 events
native	20s	43s
interpreted NestedVM	49m11s	N/A
compiled NestedVM	11m	34m51s

lines of FORTRAN 90 and FORTRAN 77 code. This should have made the recompilation to MIPS process straightforward. The custom compilation toolchain required, however, would be very hard to integrate into the build process for larger projects.

NestedVM implements some standard optimisation techniques. However, as Table 7.2 illustrates, the slowdown is still far from being acceptable. The Blackmax code, executing in NestedVM, ran 33 to 50 times slower than when executing natively. This, coupled with the difficulty of recompilation, suggests that this is not a viable option for BlackMax.

7.4.1 Running in Nereus

An application server for BlackMax was written to distribute generation jobs, collect the results and store the output.

A job was run on this server using Nereus to provide the workers with an output of 10 million blackhole events. Each worker generates 10,000 events at a time, and zips the output before posting it back to the BlackMax server. Each output file was approximately 1.5 MiB (8 MiB before compression). The input data for each task was approximately 9 MiB, largely consisting of the parton distribution functions. This job was repeated 10 times and the resulting average time was 2861 ± 103s. An average single machine, similar to Machine A, would take around 136000s to do this, based on Table 7.1. The ratio of the two times is close to 48. The number of worker PCs connected at this time was 52. The donor machines in this case were in Perth, Australia, whilst the input data and the BlackMax server were in Oxford, UK. Using Nereus brought the generation time down from around 38 hours to 48 minutes, 48 times faster and quick enough to run over a lunch break.

Chapter 8

Conclusion

This thesis presented methods to run scientific code safely on a global-scale desktop grid.

Chapter 2 explored current grid computing solutions and examined the reasons why a significant percentage of the available desktop computers have not been harnessed. It then introduced the distributed computing framework, Nereus, which enables the world's idle desktop PCs to be harnessed in a safe and scalable way. Nereus is written in Java and allows JVM bytecode to be executed safely on donor PCs. A Nereus desktop grid provides computational power in a more energy efficient and economical way than a dedicated cluster.

Nereus addresses several technical barriers to realizing a global-scale desktop grid including donor security, code portability, installation and privilege requirements and thus is a viable candidate for a global desktop grid. Security for donor PCs is ensured by running all code in the Java applet sandbox, which has been an industry tested solution for safely running untrusted code for over a decade. Restricting running code to JVM bytecode automatically ensures portability of code, as the JVM has been ported to all major operating systems and processor architectures. The Nereus client software, by running as an unprivileged Java applet, does not require any installation or escalated privileges.

Chapter 3 detailed the intricacies of the x86 architecture and virtualization. The four main types of virtualization were described: full virtualization, paravirtualization, hardware-assisted

virtualization and emulation. The security of the different types of virtualization was discussed, before concluding that an emulator and more specifically an emulator written in an interpreted language, such as Java, is the most secure way to sandbox untrusted code. The execution speed of a Java based emulator was demonstrated to be comparable to, and sometimes faster than, native code using a Toy CPU emulation based on a simple x86-like instruction set.

The architecture of JPC, a pure Java x86 emulator, was described in Chapter 4. JPC enables native x86 code to be run on a JVM. The architectural decisions to balance speed of execution against memory usage and simplicity of code were discussed. Useful abstractions for scientific computing such as remote harddisks and the snapshot capability were described.

Chapter 5 explained the execution process in JPC including the path from raw x86 bytes through decoding to interpreted JPC microcodes, to compiled JPC classes, and eventually via the JVM's JIT compiler to native code. Some methods for improving the emulation accuracy were discussed along with their relative difficulty of implementation. Finally, JPC was benchmarked to estimate the current speed and the potential speed increase from a full loop compiler. At the time of writing, JPC can boot DOS and some Linux distributions, including two with graphical desktops and executes code at up to 11% native speed. At this speed it is too slow to usefully run scientific code. However, further developments in speed will come with the implementation of an advanced loop compiler. Such a compiler should increase the optimal speed of execution by a factor of 6-8X current performance, bringing it up to between 50 and 80% native speed. At this speed, general code execution in JPC becomes very desirable. Other future developments for JPC include improving the emulation accuracy to be able to execute modern Windows OSs and typical graphical Linux distributions like Ubuntu Linux. A large amount of computing in particle physics is done using Scientific Linux, and thus sufficient emulation accuracy to boot this operating system in JPC would greatly expand the possibilities for scientific use.

Even without JPC, Nereus shows great potential for scientific computing. Two applications that just utilise Nereus were developed in this thesis. Chapter 6 described the Map-Reduce parallel programming paradigm for analysing large datasets. The challenges to writing a Map-

Reduce framework to utilise unreliable computing resources was discussed. Mycelia, a MapReduce implementation for Nereus was then introduced to handle these challenges. Mycelia was tested for accuracy and scaling ability. Mycelia provides an easy way for scientists to utilise distributed Nereus resources without knowing distributed computing techniques or software development methodologies.

Chapter 7 discussed the second application for Nereus, a blackhole event generator called BlackMax. BlackMax was translated to Java to run directly in Nereus and resulted in a 24% to 76% speed increase compared to the original native version. An application server was written to handle the dissemination of BlackMax jobs to the Nereus desktop grid and collection of results. Mycelia and BlackMax thus demonstrate two ways in which Nereus and Java improve the simplicity and efficiency of large scale scientific computing.

Future research could include improving the accuracy and speed of JPC and investigating temporary data storage on Nereus clients. JPC's accuracy can be improved by developing more systematic verification methods for comparing JPC to real processors. The speed of JPC could be improved by implementing more advanced compilers that can detect loops and short jumps, or implementing a multicore architecture to enable JPC to utilise multiple computation threads.

It is appropriate, at the end of this thesis, to consider the original dream of grid computing espoused by Foster et al [117]:

> The grid will connect multiple regional and national computational grids to create a universal source of computing power. The word "grid" is chosen by analogy with the electric power grid, which provides pervasive access to power and, like the computer and a small number of other advances, has had a dramatic impact on human capabilities and society. We believe that by providing pervasive, dependable, consistent, and inexpensive access to advanced computational capabilities, databases, sensors, and people, computational grids will have a similar transforming effect, allowing new classes of applications to emerge.

Nereus and a fully compatible and optimised JPC have the potential to offer a very cost effective,

energy-efficient form of globally distributed computing for particle physics research. They represent a significant step towards realising this compelling, revolutionary vision.

Bibliography

[1] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury & Steven Tuecke. "The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets". *Journal of Network and Computer Applications*, **23**, 3 (2000) 187–200. ISSN 1084-8045.

[2] W. T. C. Kramer, A. Shoshani, D. A. Agarwal, B. R. Draney, G. Jin, G. F. Butler & J. A. Hules. "Deep scientific computing requires deep data". *IBM Journal of Research and Development*, **48**, 2 (2010) 209–232.

[3] S. L Pallickara, M. Pierce, Q. Dong & C. Kong. "Enabling Large Scale Scientific Computations for Expressed Sequence Tag Sequencing over Grid and Cloud Computing Clusters". In "PPAM 2009 EIGHTH INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING AND APPLIED MATHEMATICS Wroclaw, Poland", (2009).

[4] R. T Kouzes, G. A Anderson, S. T Elbert, I. Gorton & D. K Gracio. "The changing paradigm of data-intensive computing". *Computer*, **42**, 1 (2009) 26–34.

[5] L. R Scott, T. Clark & B. Bagheri. "Scientific parallel computing". Princeton University Press (2005).

[6] CERN. "WLCG Worldwide LHC Computing Grid". http://lcg.web.cern.ch/LCG/public/ (2010).

[7] I. Gorton, P. Greenfield, A. Szalay & R. Williams. "Data-intensive computing in the 21st century". *Computer*, **41**, 4 (2008) 30–32.

[8] E. Mollick. "Establishing Moore's law". *Annals of the History of Computing, IEEE*, **28**, 3 (2006) 62–75.

[9] D. Kondo, B. Javadi, P. Malecot, F. Cappello & D. P Anderson. "Cost-benefit analysis of cloud computing versus desktop grids". In "Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on", page 1–12 (2009).

[10] D. P Anderson. "Volunteer computing: the ultimate cloud". *Crossroads*, **16**, 3 (2010) 7–10.

[11] J. G Koomey. "Estimating total power consumption by servers in the US and the world". Citeseer (2007).

[12] Boinc. "BOINC". http://boinc.berkeley.edu/ (2010).

[13] top500.org. "TOP500 Supercomputing Sites". http://www.top500.org/ (2010).

[14] N. J Taylor. "Public grid computing participation: An exploratory study of determinants". *Information & Management*, **44**, 1 (2007) 12–21.

[15] S. Santhanam, P. Elango, A. Arpaci-Dusseau & M. Livny. "Deploying virtual machines as sandboxes for the grid". In "Proceedings of the 2nd conference on Real, Large Distributed Systems-Volume 2", page 12 (2005).

[16] A. R Butt, S. Adabala, N. H Kapadia, R. J Figueiredo & J. A.B Fortes. "Grid-computing portals and security issues* 1". *Journal of Parallel and Distributed Computing*, **63**, 10 (2003) 1006–1014.

[17] A. Marosi, P. Kacsuk, G. Fedak & O. Lodygensky. "Using Virtual Machines in Desktop Grid Clients for Application Sandboxing". *CoreGRID Technical Report*.

[18] A. Marosi, G. Gombas, Z. Balaton, P. Kacsuk & T. Kiss. "Sztaki desktop grid: Building a scalable, secure platform for desktop grid computing". *Making Grids Work*, page 365–376.

[19] Myles Allen. "Private conversations" (2008).

[20] Francesco Venturini. "The Race to Dominate the Future of TV." (2011).

[21] Samsung. "Samsung Galaxy S II".

[22] Rhys Newman & Ian Preston. "Nereus-V: Massively Parallel Computing of, by, and for the Community" (2009).

[23] D. P Anderson. "BOINC: A system for public-resource computing and storage". In "proceedings of the 5th IEEE/ACM International Workshop on Grid Computing", page 4–10 (2004).

[24] "SETI@home". http://setiathome.berkeley.edu/ (2010).

[25] D. P Anderson, J. Cobb, E. Korpela, M. Lebofsky & D. Werthimer. "SETI@ home: an experiment in public-resource computing". *Communications of the ACM*, **45**, 11 (2002) 56–61.

[26] E. Korpela, D. Werthimer, D. Anderson, J. Cobb & M. Leboisky. "Seti@ home-massively distributed computing for seti". *Computing in Science & Engineering*, **3**, 1 (2002) 78–83.

[27] "Climateprediction.net | The world's largest climate forecasting experiment for the 21st century." http://climateprediction.net/ (2010).

[28] D. Stainforth, J. Kettleborough, A. Martin, A. Simpson, R. Gillis, A. Akkas, R. Gault, M. Collins, D. Gavaghan & M. Allen. “Climateprediction. net: Design principles for public-resource modeling research”. In “Proceedings of the 14th IASTED International Conference on parallel and distributed computing systems”, (2002).

[29] “Folding@home - Main”. http://folding.stanford.edu/ (2010).

[30] S. M Larson, C. D Snow, M. Shirts & V. S Pande. “Folding@ Home and Genome@ Home: Using distributed computing to tackle previously intractable problems in computational biology”.

[31] Google. “Google - public data”. http://www.google.com/publicdata?ds=wb-wdi&met=it_net_user&dl=en&tdim=true (2010).

[32] “SecurityIssues - BOINC”. http://boinc.berkeley.edu/trac/wiki/SecurityIssues (2010).

[33] E. Garcia, H. Guyennet, F. Hantz & J. C Lapayre. “Security in GRID Computing”. *Advances in enterprise information technology security*, page 20.

[34] M. A Habib & M. T Krieger. “Security in Grid Computing”. *Seminar aus Netzwerke und Sicherheit: Communication Infrastructure*.

[35] EGEE. “gLite”. http://glite.web.cern.ch/glite/ (2010).

[36] I. Foster. “Globus toolkit version 4: Software for service-oriented systems”. *Journal of Computer Science and Technology*, **21**, 4 (2006) 513–520.

[37] “The Globus Alliance”. http://www.globus.org/ (2010).

[38] “Installation Guide - Globus Toolkit”. http://www.globus.org/toolkit/docs/latest-stable/admin/install/ (2010).

[39] “Firewall - Globus”. http://dev.globus.org/wiki/FirewallHowTo (2010).

[40] J. Frey, T. Tannenbaum, M. Livny, I. Foster & S. Tuecke. “Condor-G: A computation management agent for multi-institutional grids”. *Cluster Computing*, **5**, 3 (2002) 237–246.

[41] B. Beckles. “Implementing privilege separation in the Condor® system”. In “UK e-Science All Hands Conference”, volume 2005 (2005).

[42] Bruce Beckles. “Security concerns with Condor: A brief overview” (2004).

[43] PlanetLab. “Technical Contact’s Guide” (2011).

[44] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff & D. Zagorodnov. “The eucalyptus open-source cloud-computing system”. In “Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid”, page 124–131 (2009).

[45] "Eucalyptus | Your environment. Our industry leading cloud computing software." http://www.eucalyptus.com/ (2010).

[46] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff & D. Zagorodnov. "Eucalyptus: an open-source cloud computing infrastructure". In "Journal of Physics: Conference Series", volume 180, page 012051 (2009).

[47] Adobe. "Adobe Flash Player". http://get.adobe.com/flashplayer/ (2010).

[48] L. Koved, A. J Nadalin, D. Neal & T. Lawson. "The evolution of Java security". *IBM Systems Journal*, **37**, 3 (2010) 349–364.

[49] US-Cert. "Technical Cyber Security Alerts". http://www.us-cert.gov/cas/techalerts/ (2010).

[50] Symantec. "Internet Security Threat Report: Volume XV: April 2010". Whitepaper (2010).

[51] Statowl. "Java Market Share and Usage Statistics". http://www.statowl.com/java.php (2010).

[52] Gary McGraw. "Securing Java : getting down to business with mobile code". Wiley Computer Pub., New York, [2nd ed.]. edition (1999). ISBN 9780471319528.

[53] Brian Amedro, Vladimir Bodnartchouk, Denis Caromel, Christian Delbe, Fabrice Huet & Guillermo Taboada. "Current State of Java for HPC". *Rapport technique*.

[54] Mark Bull. "Using Java for scientific computing" (2006).

[55] Cliff Click. "Java vs. C Performance....Again. | Azul Systems: Run, Scale, Simplify Your Java Application Platform". http://www.azulsystems.com/blog/cliff-click/2009-09-06-java-vs-c-performanceagain (2009).

[56] "Which programming languages are fastest? | Computer Language Benchmarks Game". http://shootout.alioth.debian.org/u32/which-programming-languages-are-fastest.php (2010).

[57] "The Jython Project". http://www.jython.org/ (2010).

[58] "Clojure - home". http://clojure.org/ (2010).

[59] R. Hickey. "The Clojure programming language". In "Proceedings of the 2008 symposium on Dynamic languages", page 1 (2008).

[60] "JRuby.org :: Home". http://jruby.org/ (2010).

[61] "Rhino - JavaScript for Java". http://www.mozilla.org/rhino/ (2010).

[62] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman & M. Zenger. “The Scala Language Specification”. *Programming Methods Laboratory, EPFL. Version*, **2**.

[63] “The Scala Programming Language”. http://www.scala-lang.org/ (2010).

[64] “Groovy - Home”. http://groovy.codehaus.org/ (2010).

[65] Eric Allen, David Chase, Joe Hallet, Victor Luchangco, Jan-William Maessen, Sukyoung Ryu, Guy L. Steele Jr. & Sam Tobin-Hochstadt. “The Fortress Language Specification” (2008).

[66] Intel Corp. “Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 1: Basic Architecture”.

[67] Intel Corp. “Intel 64 and IA-32 Architectures software developer’s manual volume 2A: Instruction set reference, A-M”.

[68] Intel Corp. “Intel 64 and IA-32 Architectures software developer’s manual volume 2B: Instruction set reference, N-Z”.

[69] Intel Corp. “Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 3A: System Programming Guide Part 1”.

[70] I. Corporation. “IA-32 Intel Architecture software developer’s manual”. *Intel Corporation*, **1** (2001) 68.

[71] G. J Popek & R. P Goldberg. “Formal requirements for virtualizable third generation architectures”. *Communications of the ACM*, **17**, 7 (1974) 421.

[72] James E. Smith & Ravi Nair. “Virtual Machines: Versatile Platforms for Systems and Processes”. Morgan Kaufmann (2005). ISBN 9781558609105.

[73] J. Watson. “Virtualbox: bits and bytes masquerading as machines”. *Linux Journal*, **2008**, 166 (2008) 1.

[74] “VMware Virtualization Software for Desktops, Servers & Virtual Machines for Public and Private Cloud Solutions”. http://www.vmware.com/.

[75] “Windows Virtual PC: Home Page”. http://www.microsoft.com/windows/virtual-pc/.

[76] “VirtualBox User Manual, Version 4.0.4” (2011).

[77] E. Stahl & M. Anand. “A Comparison of PowerVM and x86-Based Virtualization Performance”.

[78] P. R Barham, B. Dragovic, K. A Fraser, S. M Hand, T. L Harris, A. C Ho, E. Kotsovinos, A. V. Madhavapeddy, R. Neugebauer, I. A Pratt et al. “Xen 2002”. *University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-553, Jan*.

[79] "Virtualization with Hyper-V". http://www.microsoft.com/windowsserver2008/en/us/hyperv-main.aspx.

[80] "Parallels Optimized Computing". http://www.parallels.com/uk/.

[81] Sean Heelan. "Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities". MSc, University of Oxford (2009).

[82] J. Rutkowska. "Red Pill... or how to detect VMM using (almost) one CPU instruction". *Invisible Things*.

[83] J. Rutkowska. "Introducing Blue Pill". *The official blog of the invisiblethings. org. June*, **22**.

[84] J. Rutkowska. "Subverting VistaTM Kernel For Fun And Profit". *Black Hat Briefings*.

[85] "IBM X-Force 2010 Mid-Year Trend and Risk Report" (2010).

[86] C. E Irvine & J. S Robin. "Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor". Storming Media (2000).

[87] Google. "Native Client". http://code.google.com/p/nativeclient/ (2010).

[88] "Bochs: The Open Source IA-32 Emulation Project". http://bochs.sourceforge.net/.

[89] F. Bellard. "QEMU, a fast and portable dynamic translator" (2005).

[90] "DOSBox, an x86 emulator with DOS". http://www.dosbox.com/.

[91] L. Martignoni, R. Paleari, G. F Roglia & D. Bruschi. "Testing CPU emulators". In "Proceedings of the eighteenth international symposium on Software testing and analysis", page 261–272 (2009).

[92] Cliff Click. "Fast bytecodes for funny languages" (2008).

[93] Rhys Newman & Chris Dennis. "Everything Java: JPC, a Fast x86 PC Emulator" (2007).

[94] Rhys Newman & Chris Dennis. "Everything Java Technology... but Better and Faster: The Evolution of JPC" (2008).

[95] "Context Switch Definition". http://www.linfo.org/context_switch.html (2011).

[96] Tom Shanley. "Protected Mode Software Architecture". PC System Architecture Series. Addison-Wesley (1996).

[97] Rhys Newman. "Private Communication - JPC MMU designed by Dr Rhys Newman" (2007).

[98] VMWare Inc. "Timekeeping in VMware virtual machines". *White papers. Latest revision: 12 Aug*.

[99] "CernVM". http://cernvm.cern.ch/cernvm/.

[100] "Static single assignment form". http://en.wikipedia.org/wiki/Static_single_assignment_form (2010).

[101] A. Georges, D. Buytaert & L. Eeckhout. "Statistically rigorous Java performance evaluation". *ACM SIGPLAN Notices*, **42**, 10 (2007) 76.

[102] ibm. "Robust Java benchmarking, Part 1: Issues". http://www.ibm.com/developerworks/java/library/j-benchmark1.html (2010).

[103] D. Mihocka & S. Shwartsman. "Virtualization Without Direct Execution or Jitting: Designing a Portable Virtual Machine Infrastructure". In "1st Workshop on Architectural and Microarchitectural Support for Binary Translation in ISCA-35, Beijing", (2008).

[104] J. Dean & S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". *To appear in OSDI*, page 1.

[105] "Apache Hadoop". http://hadoop.apache.org/ (2010).

[106] Eduardo Pinheiro, Wolf-Dietric Weber & Luiz André Barroso. "Failure Trends in a Large Disk Drive Population". *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, page 17–28.

[107] Gartner. "Gartner Says Annual Failure Rates of PCs Are Improving, but Manufacturers Can Do Better". http://www.gartner.com/it/page.jsp?id=493841.

[108] D. Borthakur. "The hadoop distributed file system: Architecture and design". *Hadoop Project Website*.

[109] "PoweredBy - Hadoop". http://wiki.apache.org/hadoop/PoweredBy (2010).

[110] "GridGain". http://www.gridgain.com/ (2010).

[111] "Twister: Iterative MapReduce". http://www.iterativemapreduce.org/ (2010).

[112] James Joyce. "Ulysses". http://www.gutenberg.org/files/4300/4300.txt.

[113] D. C Dai, C. Issever, E. Rizvi, G. Starkman, D. Stojkovic & J. Tseng. "The BlackMax Manual".

[114] D. C Dai, G. Starkman, D. Stojkovic, C. Issever, E. Rizvi & J. Tseng. "BlackMax: A black-hole event generator with rotation, recoil, split branes, and brane tension". *Physical Review D*, **77**, 7 (2008) 76007.

[115] D. C Dai, C. Issever, E. Rizvi, G. Starkman, D. Stojkovic & J. Tseng. "Manual of BlackMax, a black-hole event generator with rotation, recoil, split branes, and brane tension". *Arxiv preprint arXiv:0902.3577*.

[116] D. M Gingrich. "Monte Carlo event generator for quantum black hole production and decay in proton-proton collisions". *Arxiv preprint arXiv:0911.5370*.

[117] I. Foster & C. Kesselman. "The grid: blueprint for a new computing infrastructure". Morgan Kaufmann (2004).