Hybrid Runtime Systems

Kyle C. Hale

Abstract

As parallelism continues to increase in ubiquity—from mobile devices and GPUs to datacenters and supercomputers—*parallel runtime systems* occupy an increasingly important role in the system software stack. The needs of parallel runtimes and the increasingly sophisticated languages and compilers they support do not line up with the services provided by general-purpose OSes. Furthermore, the semantics available to the runtime are lost at the system-call boundary in such OSes. Finally, because a runtime executes at user-level in such an environment, it cannot leverage hardware features that require kernel-mode privileges—a large portion of the functionality of the machine is lost to it. These limitations warp the design, implementation, functionality, and performance of parallel runtimes. I make the case for eliminating these compromises by transforming parallel runtimes into

This work was made possible by support from the United States National Science Foundation through grants CCF-1533560 and CNS-0709168, from the United States Department of Energy through grant number DE-SC0005343, and from Sandia National Laboratories through the Hobbes Project, which is funded by the 2013 Exascale Operating and Runtime Systems Program under the Office of Advanced Scientific Computing Research in the DoE's Office of Science.

hybrid runtimes (HRTs), runtimes that run *as* kernels, and that enjoy full hardware access and control over abstractions to the machine. The primary claim of this dissertation is that the hybrid runtime model can provide significant benefits to parallel runtimes and the applications that run on top of them.

I demonstrate that it is feasible to create instantiations of the hybrid runtime model by doing so for four different parallel runtimes, including Legion, NESL, NDPC (a home-grown language), and Racket. These HRTs are enabled by a kernel framework called Nautilus, which is a primary software contribution of this dissertation. A runtime ported to Nautilus that acts as an HRT enjoys significant performance gains relative to its ROS counterpart. Nautilus enables these gains by providing fast, light-weight mechanisms for runtimes. For example, with Legion running a mini-app of importance to the HPC community, we saw speedups of up to forty percent. We saw further improvements by leveraging hardware (interrupt control) that is not available to a user-space runtime.

In order to bridge Nautilus with a "regular OS" (ROS) environment, I introduce a concept we developed called the hybrid virtual machine (HVM). Such bridged operation allows an HRT to leverage existing functionality within a ROS with low overheads. This simplifies the construction of HRTs.

In addition to Nautilus and the HVM, I introduce an event system called Nemo, which allows runtimes to leverage events both with a familiar interface and with mechanisms that are much closer to the hardware. Nemo enables event notification latencies that outperform Linux user-space by several orders of magnitude.

Finally, I introduce Multiverse, a system that implements a technique called *automatic hybridization*. This technique allows runtime developers to more quickly adopt the HRT model by starting with a working HRT system and incrementally moving functionality from a ROS to the HRT.

Keywords: Hybrid Runtimes, Parallelism, Operating Systems, High-Performance Computing

NORTHWESTERN UNIVERSITY

Hybrid Runtime Systems

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Science

By

Kyle C. Hale

EVANSTON, ILLINOIS

August 2016

Thesis Committee

Peter A. Dinda
Northwestern University
Committee Chair

Nikos Hardavellas
Northwestern University
Committee Member

Russ Joseph
Northwestern University
Committee Member

Fabián E. Bustamante
Northwestern University
Committee Member

Arthur B. Maccabe
Oak Ridge National Laboratory
External Committee Member

Abstract

Hybrid Runtime Systems

Kyle C. Hale

As parallelism continues to increase in ubiquity—from mobile devices and GPUs to datacenters and supercomputers—*parallel runtime systems* occupy an increasingly important role in the system software stack. The needs of parallel runtimes and the increasingly sophisticated languages and compilers they support do not line up with the services provided by general-purpose OSes. Furthermore, the semantics available to the runtime are lost at the system-call boundary in such OSes. Finally, because a runtime executes at user-level in such an environment, it cannot leverage hardware features that require kernel-mode privileges—a large portion of the functionality of the machine is lost to it. These limitations warp the design, implementation, functionality, and performance of parallel runtimes. I make the case for eliminating these compromises by transforming parallel runtimes into *hybrid runtimes* (HRTs), runtimes that run *as* kernels, and that enjoy full hardware access and control over abstractions to the machine. The primary claim

of this dissertation is that the hybrid runtime model can provide significant benefits to parallel runtimes and the applications that run on top of them.

I demonstrate that it is feasible to create instantiations of the hybrid runtime model by doing so for four different parallel runtimes, including Legion, NESL, NDPC (a home-grown language), and Racket. These HRTs are enabled by a kernel framework called Nautilus, which is a primary software contribution of this dissertation. A runtime ported to Nautilus that acts as an HRT enjoys significant performance gains relative to its ROS counterpart. Nautilus enables these gains by providing fast, light-weight mechanisms for runtimes. For example, with Legion running a mini-app of importance to the HPC community, we saw speedups of up to forty percent. We saw further improvements by leveraging hardware (interrupt control) that is not available to a user-space runtime.

In order to bridge Nautilus with a "regular OS" (ROS) environment, I introduce a concept we developed called the hybrid virtual machine (HVM). Such bridged operation allows an HRT to leverage existing functionality within a ROS with low overheads. This simplifies the construction of HRTs.

In addition to Nautilus and the HVM, I introduce an event system called Nemo, which allows runtimes to leverage events both with a familiar interface and with mechanisms that are much closer to the hardware. Nemo enables event notification latencies that outperform Linux user-space by several orders of magnitude.

Finally, I introduce Multiverse, a system that implements a technique called *automatic hybridization*. This technique allows runtime developers to more quickly adopt the HRT model by starting with a working HRT system and incrementally moving functionality from a ROS to the HRT.

Acknowledgments

When I first embarked on the road to research, I considered myself an ambitious individual. Little did I know that personal ambitions and aspirations comprise a relatively minor fraction of the conditions required to pursue an academic career. I have found that the incredible amount of support so generously offered by my mentors, my friends, and my family has been far more important than anything else. I cannot hope to thank everyone who has shaped me as an individual and led me to where I am, but I will attempt here to name a few that stand out.

I first thank my advisor, Professor Peter Dinda, for his encouragement and advice during my studies at Northwestern. I find it inadequate to just call him my advisor, as it understates the role he has played in guiding my development, both intellectually and personally. Peter embodies all the qualities of a true mentor, from his seemingly inexhaustible supply of patience to his extraordinary ability to cultivate others' strengths. Before meeting him, I had never encountered a teacher so genuinely dedicated to advocating for his students and their success. I can only hope to uphold the same standards of mentorship.

I would also like to thank my other dissertation committee members. Nikos Hardavellas has always had an open door for brainstorming sessions and advice. Russ Joseph similarly has always been willing to take the time to help, always with an uplifting note of humor and a smile. Fabián Bustamante has throughout been a source of candid conversation and genuine advice. Barney Maccabe has been an indispensable colleague and has always made me feel welcome in the HPC community.

Two very important people who gave me the benefit of the doubt when I started doing research deserve special thanks. Steve Keckler and Boris Grot not only advised me as an undergraduate, they also helped me discover confidence in myself that I never thought I had. I also owe thanks to Calvin Lin and the Turing Scholars Program at The University of Texas for planting the seeds for my interest in research in computer systems.

To FranCee Brown McClure and all of the other Ronald E. McNair scholars, I could not have made it to this point without you.

I must also thank Kevin Pedretti and Patrick Bridges for helping make my stint in New Mexico an enjoyable one, and for making me feel at home in the Southwest. To Jack Lange I am grateful for being able to enjoy the solidarity of a fellow Texan who also braved the cold Chicago winters to earn a doctorate.

Several members of the Prescience Lab, both former and present, deserve thanks for their assistance, including Conor Hetland, Lei Xia, and Chang Bae.

While the friends that have supported me through the years are too numerous to name, some stand out in helping me survive graduate school. I have Leonidas Spinoulas, Luis Pestaña, Armin Kappeler, and Jaime Espinosa to thank for broadening my horizons and bringing me out of my shell. I am ever grateful to John Rula for being a fantastic roommate, a loyal friend, and a willing participant in what others would consider outlandish intellectual discussions. Office life in graduate school would not have been the same without Maciek Swiech and Marcel Flores, who were both crucial sources of lighthearted antics, fascinating discussions, and delicious Matt Newtons. I will sincerely miss having you all around. I have to thank Jose Rodriguez, Spencer Cook, and Trevor Cook as well for their friendship and for expanding my intellectual horizons to radically different areas.

My family has been instrumental in their constant encouragement and unyielding confidence in me. I am truly lucky to have such unwavering support, and no words are enough to thank them. I am ever grateful to my mother, Mary Catherine Hale, who always seemed to guide me in the right direction and who believed in me even when I could not believe in myself. To my father, Chris Hale, who taught me to be curious and how to appreciate the value of hard work and determination. To my sister Stacy Hale, who colored my imagination and instilled in me a sense of wonder. To my grandmother Catherine Hill, who has always been a voice of comfort, and who incidentally helped me write my first research paper. To my uncle, Mark Hill, who first introduced me to computers and likely brought out the nerd in me.

Finally, I would like to thank Leah Pickett for her unconditional love and support, her profound kindness, and her steadfast encouragement, without which I could not have finished the long journey towards a Ph.D.

Preface

At the time of this writing, the United States Department of Energy (DoE) had begun to take significant steps to catalyze research and development focused on realizing a supercomputer system operating at an ExaFLOP (10^{18} FLOPs), with the intention of having the machine come online roughly in the 2020 time frame. This effort is now colloquially referred to as "Exascale." The work described in this dissertation largely fits within the context of that effort. The DoE determined that focus in various sub-areas in high-performance computing would be necessary to make this goal possible. Researchers in one such area were given the task of exploring new system software directions—mainly at the level of the operating system and runtime system (OS/R)—for the Exascale machine. The work on hybrid runtimes discussed in this text was primarily carried out with funding from the Hobbes project[1] (one of the large multi-institutional projects in the OS/R group) and with funding from the National Science Foundation[2]. Readers aware of other lines of investigation within Hobbes will find that this work lies on the more exploratory side of that effort. I will note that it began as a toy project, and only started in earnest as a

[1]Provided from Sandia National Laboratories and funded by the 2013 Exascale Operating and Runtime Systems Program under the Office of Advanced Scientific Computing Research in the United States Department of Energy's Office of Science, `http://hobbes.xstack.sandia.gov`

[2]United States National Science Foundation, grant number CCF-1533560

research project after a rather fortuitous, ad hoc brainstorming session with my advisor. The earlier work described in the appendices was carried out within the context of the V3VEE project[3], an effort targeted at building a modern virtual machine monitor (VMM) framework for high-performance systems. Both the GEARS and guarded module systems were developed with support from the NSF[4] and the DoE[5].

This dissertation will be of broad interest to those interested in the structure of system software aimed at massive parallelism. While the supercomputing arena has faced the significant challenges of parallelism for some time now, parallelism has already become pervasive in commodity devices, including desktop machines and mobile devices. Contemporary desktop and server machines often employ eight or more CPU cores. High-end servers available on the market now have more than one hundred cores. I expect the trend of increasing levels of parallelism to continue, along with the need for exploratory efforts in system software targeting parallelism.

This text may also be useful as a reference for those interested in low-level benchmarking of kernel components and for those intrigued by the role of the runtime systems in the context of the larger system software stack.

The contents of Chapter 3 were published in the proceedings of the 24th International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC) in June of 2015. The contents of Chapter 5 appeared originally in the proceedings of the 12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments in April of 2016. A portion of the work that appears in that chapter was carried out with the help of Madhav Suresh and Conor Hetland, and the design of the hybrid virtual machine

[3]`http://v3vee.org`
[4]NSF grant number CNS-0709168
[5]DoE grant number DE-SC0005343

is primarily due to my advisor, Peter Dinda. The material in Chapter 7 was first published in the proceedings of HPDC 2016, and was co-authored with Conor Hetland. The contents of Appendix B first appeared in the proceedings of the 9^{th} ACM International Conference on Autonomic Computing (ICAC) in September of 2012, and was carried out with the assistance of Lei Xia. The work on guarded modules, which appears in Appendix C, was first published in the proceedings of ICAC in June of 2014.

List of Abbreviations

ABI	Application Binary Interface
ACPI	Advanced Configuration and Power Interface
ADT	Action Descriptor Table
API	Application Programming Interface
APIC	Advanced Programmable Interrupt Controller
AVX	Advanced Vector Extensions (Intel ISA)
BAR	Base Address Register
BIOS	Basic Input/Output System
BSP	Bootstrap Processor
CDF	Cumulative Distribution Function
CMP	Chip Multiprocessor
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture (NVIDIA)
DMA	Direct Memory Access
DOE	Department of Energy
DVFS	Dynamic Voltage and Frequency Scaling

ELF	Executable and Linkable Format
FLOPS	Floating Point Operations Per Second
GDT	Global Descriptor Table
GEARS	Guest Examination and Revision Services
GPGPU	General-Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
HAL	Hardware Abstraction Layer
HPC	High-Performance Computing
HPET	High-Precision Event Timer
HRT	Hybrid Runtime
HVM	Hybrid Virtual Machine
ICR	Interrupt Command Register
IDT	Interrupt Descriptor Table
IOAPIC	I/O Advanced Programmable Interrupt Controller
IOMMU	Input-Output Memory Management Unit
IPC	Inter-Process Communication
IPI	Inter-Processor Interrupt
IRQ	Interrupt Request
IST	Interrupt Stack Table
JIT	Just-in-Time (compilation)
LWK	Lightweight Kernel
MPI	Message Passing Interface
MPSS	Manycore Platform Software Stack (Intel Xeon Phi)
MSI	Message Signaled Interrupt

MSR	Model-Specific Register
MTU	Maximum Transmission Unit
NIC	Network Interface Card
NPTL	Native POSIX Thread Library
NUMA	Non-Uniform Memory Access
OS	Operating System
PC	Program Counter
PCI	Peripheral Component Interconnect
PDE	Partial Differential Equation
PGAS	Partitioned Global Addres Space
POSIX	Portable Operating System Interface
RAM	Random Access Memory
RDMA	Remote Direct Memory Access
ROS	Regular Operating System
RTM	Restricted Transactional Memory (Intel ISA)
SASOS	Single Address Space Operating System
SIMD	Single Instruction Multiple Data
SIPI	Startup Inter-Processor Interrupt
TCP	Transmission Control Protocol
TLB	Translation Lookaside Buffer
TSC	Time Stamp Counter
TSS	Task State Segment
TSX	Transactional Synchronization Extensions (Intel ISA)

UDP	User Datagram Protocol
VM	Virtual Machine
VMM	Virtual Machine Monitor

Dedication

To all who inspired me.

Sine te nihil sum.

Table of Contents

List of Figures

Chapter 1 Introduction

Modern parallel runtimes are systems that operate in user mode and run above the system call interface of a general-purpose kernel. While this facilitates portability and simplifies the creation of some functionality, it also has consequences that warp the design and implementation of the runtime and affect its performance, efficiency, and scalability. First, the runtime is deprived of the use of hardware features that are available only in kernel mode. This is a large portion of the machine. The second consequence is that the runtime must use the abstractions provided by the kernel, even if the abstractions are inadequate. For example, the runtime might need subset barriers, and be forced to build them out of mutexes. Finally, the kernel has minimal access to the information available to the parallel runtime or to the language implementation it supports. For example, the runtime might not require coherence, but get it anyway.

The complexity of modern hardware is rapidly growing. In high-end computing, it is widely anticipated that Exascale machines will have at least 1000-way parallelism at the node level. Even today's high-end homogeneous nodes, such as the one used for several evaluations in this dissertation, have 64 or more cores or hardware threads arranged on top of a complex intertwined cache hierarchy that terminates in 8 or more memory zones with non-uniform access. Today's heterogeneous nodes include accelerators, such as the Intel Xeon Phi, that introduce additional coherence domains and memory systems. Server platforms for cloud and datacenter computing, and even desktop and mobile platforms are seeing this simultaneous explosion of hardware complexity and the need for parallelism to take advantage of the hardware. How to make such complex hardware programmable, in parallel, by mere humans is an acknowledged open challenge.

Some modern runtimes, such as the Legion runtime [21, 189], already address this challenge by creating abstractions that programmers or the compilers of high-level languages can target. Very high-level parallel languages can let us further decouple the expression of parallelism from its implementation. Parallel runtimes such as Legion, and the runtimes for specific parallel languages share many properties with operating system (OS) kernels, but suffer by not *being* kernels. With current developments, particularly in virtualization and hardware partitioning (discussed in Section 8.1.1), we are in a position to remove this limitation. In this dissertation, I make the claim that transforming parallel runtime systems into kernels can be advantageous for performance and functionality. I report on a new model called the *hybrid runtime* (HRT), which is a combination of a parallel runtime, the applications it supports, and a kernel, which has full access to hardware and control over abstractions to that hardware. This dissertation focuses on the claim that the hybrid runtime model can provide significant benefits, both in terms of performance and functionality, to parallel runtimes.

1.1 Runtime Systems

Ambiguities in the naming of system software layers warrant a clear, succinct definition for the term *runtime system*, as much of this dissertation depends on not just what a runtime system *is*, but also on the issues that arise in its design and implementation.

I will use the following definition throughout this text:

Definition 1.1.1. A **runtime system** is a user-level software component that mediates between a user-supplied program and its execution environment (namely, the OS and the hardware). It may or may not implement features of a high-level programming language. It may be invoked explicitly by a user program or implicitly via invocations inserted by a compiler.

This definition deserves some elaboration. Note that I have inserted the term *user-level* here. This is primarily for precision, and to eliminate the inclusion of operating systems in the same category, as they share many attributes with runtimes. One simple way to differentiate them is to realize that—barring the presence of virtualization—there is one operating system present on the machine, but there may be many runtime systems. Examples include the Python runtime system and Java's runtime system. Both of these runtimes mediate interactions with the operating system by issuing system calls based on invocations in the program—for example, opening a file. Both also implement features of the programming languages they support—for example, by managing memory automatically. Both also implement an execution model distinct from the underlying OS.

The last point of Definition 1.1.1 is instructive in understanding the confusion attached to runtime systems, as they are, in most cases, *implicitly* invoked. That is, the programmer need not be aware of the details of the runtime system's operation, or in many cases even its presence. This is the case for the C runtime system, which is primarily invoked via the operating system or via invocations inserted at the binary level by the C compiler. Other runtimes like Legion [21] execute only after being explicitly invoked by the programmer. Using this definition, the underlying machinery that implements a *framework*, such as MapReduce [63], may also qualify as a runtime system. In the case of MapReduce, the *map* and *reduce* functions, supplied by the user, can be thought of as the program, and the underlying machinery that manages the execution of those functions across a cluster can be thought of as a runtime system.

While runtime systems can vary widely in the services they provide, some common examples include memory management, execution models not natively provided by the OS (such as user-level threads), dynamic linking of program components (or modules), event handling, Just-in-time (JIT) compilation, user-level task scheduling, and so on. The nature of such services will become important in Chapter 3, when I discuss parallel runtime systems.

Operating systems can provide similar services, and this can precipitate incompatibilities, some of which I outlined earlier in this chapter. For example, a runtime system implementing a bulk-synchronous, parallel execution model might require deterministic timing between threads, but the OS might not guarantee such operation natively. In another example, a runtime that has high-level semantic knowledge of memory access patterns might be at odds with an operating system that provides an unpredictable, demand-paged virtual memory system.

1.2 Regular Operating Systems

OSes have evolved rapidly over the last half century as computer systems became more programmable. I will provide a brief introduction to modern OSes in Chapter 2. In this section, I will provide some brief background on parts of the operating system architecture that motivate the HRT concept.

Most modern, general-purpose OSes (I will refer to these from this point forward as regular operating systems, or ROS) separate execution into two major privilege modes, *user-mode* and *kernel-mode*. This allows the ROS kernel to form a root of trust that can safely manage the system and keep it running, even in the face of faulty or malicious user-space programs.

A ROS will also typically provide several abstractions that make it easier to program and use the machine. These include virtual memory, processes (for multi-programmed operation), threads, files, system calls, and more. Most ROSes also expose a limited API to applications to make programming more convenient. Such APIs are typically exposed to applications in the form of system calls. In addition to abstractions and APIs, a ROS will often include a varied assortment of device drivers to make them compatible with many systems with diverse hardware. To simplify the base kernel, device drivers and components that may not be necessary to a wide user base are often provided in the form of loadable kernel modules. This is true of most ROSes, including Linux, macOS, and Windows.

Most ROSes are designed to be highly modular, and will include hardware abstraction layers that make them portable across many different systems. Even so, they provide adequate performance to a wide array of applications. However, this portability does come at a cost, and—in addition to other features—contributes to their being suboptimal for many high-performance systems, especially those designed to exploit massive levels of parallelism.

1.3 Lightweight Kernels

In a high-performance computing environment, such as a supercomputer, the execution of the application takes foremost precedence. Large-scale parallel applications can run on these systems for months at a time, and overheads in their execution can become magnified do to their scale both in time and space.

An important example is an application written in the bulk-synchronous parallel model (BSP) [85], wherein processes within the application compute in parallel, perform a communication step to gather results in some manner, then continue. If one node experiences a perturbation (for example, from the OS handling an interrupt from a device), all the other nodes must wait as well. Execution, therefore, proceeds at the pace of the slowest compute node. If this occurs regularly (it often does), and the effect is multiplied across thousands of nodes (it often is), the amount of wasted compute time can become considerable [74, 75, 101].

Nondeterministic perturbations like the interrupt example mentioned above are a hallmark of program execution within a ROS. To tackle this problem, OS researchers in the HPC community answered with a simplified kernel design called a *lightweight kernel* (LWK). LWKs forego much of the functionality present in a ROS that makes it portable and very easy to use. In lieu of convenience, an LWK design will favor raw performance. LWKs do not typically have a broad range of device drivers, nor do they employ complicated hardware abstraction layers or comprehensive system call interfaces. However, they do still supply a limited system call API to the application. This design is intended to limit the kernel to providing only the minimal abstractions necessary to support the desired applications. Such a minimal design not only simplifies development, but also secondarily reduces or eliminates “OS noise” during the application’s execution, e.g. from external interrupts. Notable LWKs that have enjoyed some success include Sandia’s Kitten [129], the Catamount [118] LWK, IBM’s Compute Node Kernel (CNK) [86], and Cray’s Compute Node Linux (CNL).

LWKs work very well for supercomputing sites, as they have a fairly small set of applications they must support, and the hardware present in the system is known ahead of time. However, there is still a lost opportunity for LWKs, as they retain the *overall* architecture of a ROS. That is, they still use many of the same abstractions, and they still relegate the runtime system (and application) to user-space, thus limiting its power.

1.4 Issues with Runtimes, ROSes, and LWKs

In this dissertation, I argue that for the specific case of a parallel runtime, the user/kernel abstraction itself, which dates back to Multics, is not a good one. It is important to understand the kernel/user abstraction and its implications. This abstraction is an incredibly useful technique to enforce isolation and protection for processes, both from attackers and from each other. This not only enables increased security, but also reduces the impact of bugs and errors on the part of the programmer. Instead, programmers place a higher level of trust in the kernel, which, by virtue of its smaller codebase and careful design, ensures that the machine remains uncompromised. In this model, which is depicted in

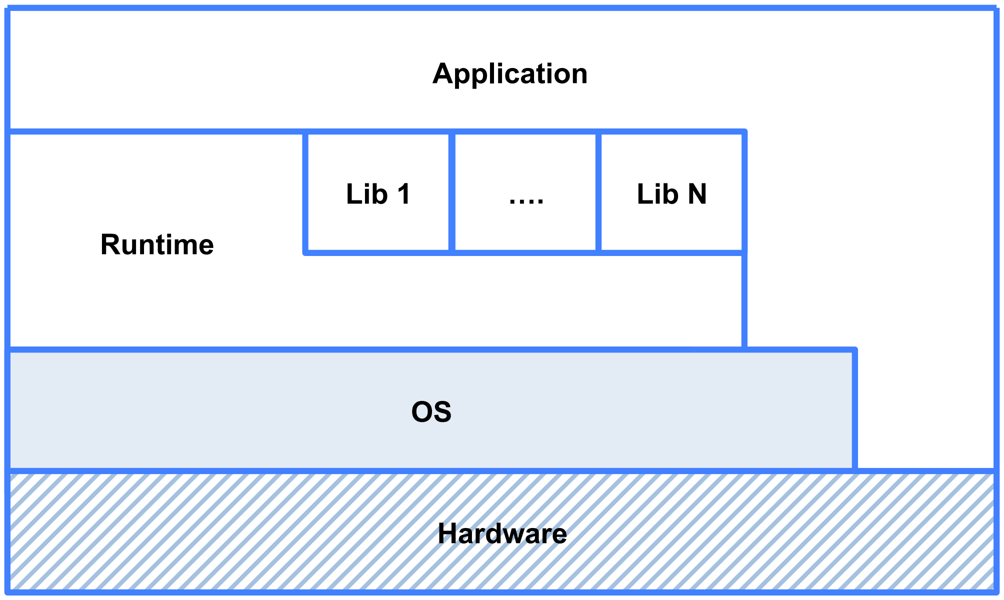


Figure 1.1: Typical system organization.

Figure 1.1, the privileged OS kernel sits directly on top of hardware, and supports runtimes, applications, and libraries which execute at user-level. Because the kernel (lightweight or otherwise) must be all things to all processes, it has grown dramatically over time, as has its responsibilities within the system. This has forced kernel developers to provide a broad range of services to an even broader range of applications. At the same time, the basic model and core services have necessarily ossified in order to maintain compatibility with the widest range of hardware and software. In ROSes, and to a lesser extent LWKs, the needs of parallelism and parallel runtimes have not been first-order concerns.

Runtime implementors often complain about the limitations imposed by ROSes. While there are many examples of significant performance enhancements within ROSes, and others are certainly possible to support parallel runtimes better, a parallel runtime as a user-level component is fundamentally *constrained* by the kernel/user abstraction. In contrast, as a kernel, a parallel runtime would have full access to all hardware features of the machine, and the ability to create any abstractions that it needs using those features.

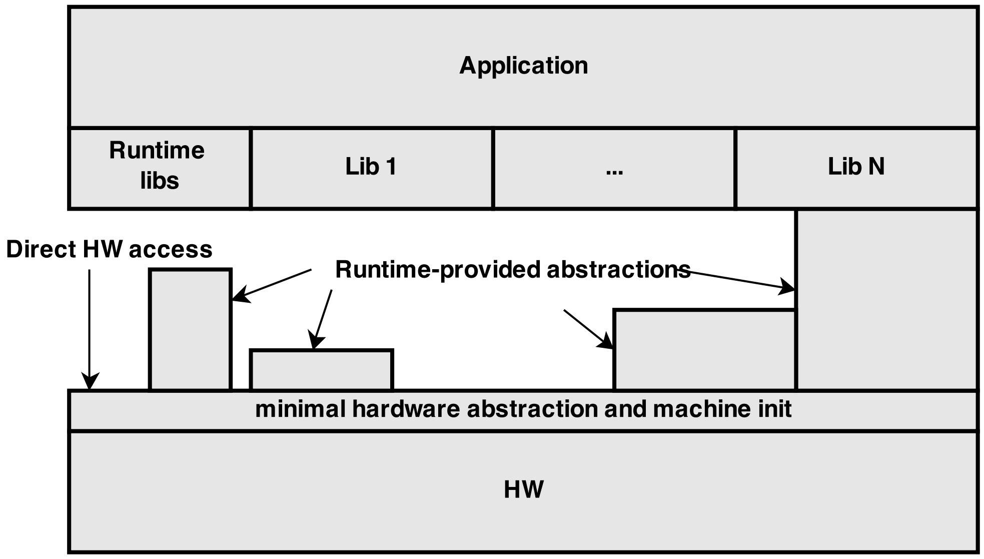

Application
Runtime libs
Lib 1
...
Lib N
Direct HW access
Runtime-provided abstractions
minimal hardware abstraction and machine init
HW

Figure 1.2: System organization in which the runtime controls the machine.

Such a model is depicted in Figure 1.2. Here, the "kernel" is a minimal layer on top of the hardware that provides services to the runtime, essentially as a library would. The runtime can build on those to create its own abstractions and services (to be used by the application), or it can ignore them. If required, the runtime can access the hardware directly. I will show in this dissertation that, in fact, breaking free from the user/kernel abstraction with HRTs can produce measurable benefits for parallel runtimes.

1.5 Hybrid Runtimes

Hybrid runtimes (HRTs) can alleviate the issues faced by ROSes and LWKs by treating the runtime system *as* a kernel, with full access to typically restricted hardware, and complete control over the abstractions over that hardware. An instance of an HRT consists of a runtime system combined with a lightweight kernel framework, called an *Aerokernel*, that provides services to the runtime, much like a library OS. These services can be used, ignored, augmented, or replaced by the runtime developer. I will refer to a general model where a runtime and an Aerokernel framework are combined as the *hybrid runtime model* (or the HRT model). The Aerokernel framework simply serves as a starting point, with familiar interfaces, so a runtime developer can more easily adopt the hybrid runtime model. Unlike within a ROS environment, the hybrid runtime executes in kernel-mode, and unnecessary abstractions are avoided by default.

The primary claim of this dissertation is that the hybrid runtime model can provide significant benefits to parallel runtimes and the applications that run on top of them. I will expand on this claim in Chapter 3 and then substantiate it in subsequent chapters by providing performance evaluations of systems that support the HRT model.

The long term vision for hybrid runtimes is for them to:

- outperform their ROS counterparts for parallel runtimes.
- support creation on demand rather than booting them manually.
- promote experimentation with unconventional uses of privileged hardware.
- be easy to construct (even starting from an existing runtime system).
- operate with low overhead in virtualized environments.
- work in tandem with a ROS environment.

To the user, booting an HRT would look like starting a process from a ROS, but with special meaning—the HRT would take over some portion of the machine and use it solely for the runtime and application that it supports. That is, the HRT would not be used in the same manner that a ROS is, e.g. for management of the system. Rather, it would be used like a “software accelerator”. This environment would be used only when performance is critical or unconventional functionality is required, and it could look very different from the software running on the ROS. Ideally, constructing such an HRT would not be tantamount to writing an OS kernel, and experimentation would be rather easy for the runtime developer. HRTs will mesh well with virtualization, avoiding overheads that arise from the heavy-weight operation of a ROS. Furthermore, an HRT would enjoy a rather symbiotic relationship with the ROS by borrowing more complex functionality from the latter when required. I will describe steps taken toward these goals in Chapters 3–7.

1.6 Benefits and Drawbacks of HRTs

Hybrid runtimes can enjoy considerably increased performance over their ROS-based counterparts. I will show in subsequent chapters how we can achieve up to a 40% speedup on the HPCG mini-app [67, 97] within the Legion runtime ported to the HRT model.

While performance is, of course, an important factor, HRTs provide other benefits as well. They allow runtime developers more flexibility in designing the codebase, avoiding the a priori imposition of mechanisms and policies that may be at odds with the needs of the runtime. HRTs also allow runtime developers to experiment with kernel-level privileges by taking advantage of hardware features that are usually unavailable at user-level. The hope is that this will lead to radically different parallel runtime designs that also perform well.

While HRTs offer some important benefits, their structure has some deleterious effects that render them unsuitable for general-purpose use. First, because HRTs by default have no notion of privilege separation, their deployment on raw hardware can create security risks. A faulty application or runtime built as an HRT could be exploited to take over an entire machine. In Chapter 5, I will describe deployment methods we have explored to mitigate this risk at an operational level. Ultimately, however, the security and protection of an HRT is under control of the runtime developer. Securing such a codebase may actually be simpler than creating a secure, general-purpose ROS. While we have addressed some security issues with particular deployment methods, making HRTs bullet proof is beyond the scope of this dissertation, and will be explored further in future work.

Another potential drawback for HRTs is that they could be difficult to construct because of their low-level nature and tight connection to kernel-level programming. While runtime developers are often sophisticated programmers, they may not be familiar with the idiosyncrasies of kernel development. I will discuss solutions to address this difficulty in Chapter 4. Similarly, debugging an HRT comes with its own set of complications due to its kernel-mode operation. While we have not undergone significant work in this area, its importance cannot be underestimated. Creating production quality HRTs, even in a bleeding edge domain like supercomputing, will require sophisticated debugging facilities. I will describe avenues that could lead to such debugging tools in Chapters 5 and 7.

1.7 Systems to Support and Evaluate the HRT Model

My work on HRTs has primarily focused on building real systems that realize the benefits that the HRT model can provide. These systems center around a small kernel framework named *Nautilus*.

1.7.1 Nautilus

Nautilus is an example of an *Aerokernel*, a very thin kernel framework intended to enable hybrid runtimes. I will introduce Nautilus in detail in Chapter 3, but in this section I will guide the reader through a cursory examination of its structure.

Two essential mechanisms allow Nautilus to enable the creation of hybrid runtimes:

1. It provides a default set of *fast* services that will be familiar to runtime developers.

2. It relinquishes full control of the machine to the runtime system.

Nautilus runs on commodity x64 hardware and on Intel's Xeon Phi accelerator card. It is a Multiboot2-based kernel that supports NUMA systems and many-core machines.

1.7.2 Hybrid Virtual Machines

While Nautilus can run on bare metal (for example, on a server or a supercomputer node), and on commodity virtual machine monitors (VMMs) such as KVM, VMware, and QEMU, it still requires that the runtime using it be ported. As will be discussed further in Chapter 7, this porting process is no small endeavor. It would therefore be advantageous for a runtime developer to be able to leverage ROS functionality (e.g. from Linux). A hybrid virtual machine (HVM) enables this by partitioning a single VM among two environments. One consists of a ROS environment, such as Linux, and the other is a hybrid runtime. With an HVM, functionality not present in Nautilus (or another Aerokernel) can be provided by the ROS. This is done with communication facilitated by the VMM.

Hybrid virtual machines are intended to make it easier for a runtime developer to start adopting the HRT model. We implemented support for HVMs in our Palacios VMM. HVMs are introduced in detail in Chapter 5.

1.7.3 Nemo Event System

One common abstraction that many runtimes support is that of an *asynchronous software event*. If a runtime (or application) uses such events heavily, the inadequacy of the event system's implementation could be a serious issue. In Chapter 6, I will introduce and specialized event system within Nautilus called *Nemo* that leverages the benefits of the HRT model to reduce event signaling latencies by several orders of magnitude. Portions of this speedup come from the simplicity and fast implementation of Nautilus primitives, while other portions come from the ability to leverage restricted hardware (control over interrupts) directly.

1.7.4 Multiverse Toolchain

To adopt the hybrid runtime model, runtime system developers must currently port their system from its current OS environment to work with an Aerokernel (such as Nautilus). While Nautilus is designed to make this easier (e.g, by providing familiar UNIX-like interfaces), porting a complex user-space code to run with the privilege of a kernel is not a trivial task. Debugging can be quite difficult, and errors challenging to identify. While the HVM alleviates this issue slightly by allowing the developer to leverage existing ROS functionality, the difficulty of a manual port still acts as an obstacle.

To address this issue, I explore a concept called *automatic hybridization*, whereby a Linux user-space runtime is *automatically* transformed into a hybrid runtime simply by rebuilding the codebase. We developed an initial implementation of automatic hybridization called Multiverse, which carries out this transformation for runtimes using the Nautilus Aerokernel. Multiverse will be introduced in detail in Chapter 7.

1.8 Results

Our work on hybrid runtimes has shown considerable promise for the HRT model. Initial experiments with the Legion runtime coupled with an example application and the HPCG mini-app show up to 40% performance improvement simply by porting to the HRT model; this does not include the utilization of privileged-mode hardware features.

Extensive microbenchmarks of Nautilus facilities show significant improvements over Linux, for example with thread creation, context switching, and concurrent programming abstractions like condition variables.

The HVM deployment model of HRTs can leverage ROS functionality with very little overhead. Interactions between a ROS and HRT can occur with latencies on the order of hundreds of nanoseconds. This is within the realm of a typical system call in a ROS.

The Nemo event system can reduce asynchronous software event latency by several orders of magnitude, and when leveraging restricted hardware features, can approach the limits of the hardware capabilities.

Finally, Multiverse can transform a complex, Linux user-space runtime into a hybrid runtime and leverage ROS functionality with little to no overheads in performance. This makes Multiverse a good tool for a runtime developer to begin exploring the HRT model.

1.9 Outline of Dissertation

The remaining text of this dissertation will focus on the motivation for hybrid runtimes, followed by their design, merits, and implementation. Several systems will be introduced that demonstrate their promise.

In Chapter 2, I will provide perspectives on regular operating systems (ROSes) by describing their design, merits, and limitations.

In Chapter 3, I give the detailed case for hybrid runtimes, I outline their benefits, and I describe initial results showing speedups that result from porting runtimes to the HRT model.

Chapter 4 comprises a description of the design and implementation of an Aerokernel framework called Nautilus that is intended to support the construction of high-performance hybrid runtimes. In that chapter I will guide the reader through an extensive series of microbenchmarks measuring the performance of the light-weight primitives that make up Nautilus.

In Chapter 5, I introduce the concept of a hybrid virtual machine (HVM), which simplifies the construction of hybrid runtimes by leveraging a logically partitioned virtual machine to facilitate communication between an HRT and a legacy OS.

Chatper 6 focuses on a low-latency, asynchronous software event system called Nemo, which is implemented as a part of Nautilus.

Chapter 7 introduces the concept of *automatic hybridization* as a method of further simplifying the creation of HRTs. I will describe Multiverse, a system that implements automatic hybridization for Linux user-space runtimes.

Finally, in Chapter 9, I will summarize the research and software engineering contributions of the work that comprises this dissertation and outline a series of next steps in the investigation of hybrid runtimes and specialized, high-performance operating systems.

Appendix A gives a list of OS issues identified by the developers of the Legion runtime. These served as one motivation for initiating the HRT research effort.

Appendix B contains a description of a system called Guest Examination and Revision Services (GEARS) that enables *guest-context virtual services* within a virtual machine. While unrelated to hybrid runtimes, this work runs in a similar vein to HRTs, as it blurs the lines between software layers that are typically viewed as being quite rigid.

Appendix C comprises work on *guarded modules*, which are a type of guest-context virtual services that are protected from the higher-privileged operating system within which they execute. This work is an expansion of the GEARS system.

Chapter 2

Regular Operating Systems

The operating system (OS) is one of the most powerful and essential elements of a modern computer system. While a “regular OS” (ROS) serves many purposes, one of its primary objectives is to provide an execution environment that makes it easier to program the machine. ROSes come in many flavors, and architectures vary widely. Widely used ROSes include Linux, Windows, and Apple’s OSX (recently renamed macOS). This chapter is primarily concerned with a common feature of any ROS, the *kernel*. Throughout the chapter, I will provide background in operating systems necessary to understand the contributions of this dissertation. Section 2.1 introduces the concept of an OS kernel and discusses its salient features. Section 2.3 discusses abstractions typically provided by ROS kernels. Section 2.4 gives a cursory examination of how a typical ROS operates at runtime. Section 2.5 introduces lightweight kernels (LWKs) and contrasts their design with ROSes. Section 2.6 discusses virtualization and its relation to operating systems concepts. Finally, Section 2.7 concludes the chapter with a discussion of the limits of ROSes and LWKs. Readers intimately familiar with OS concepts and implementation may skip ahead to that section.

2.1 ROS Kernels

The roles that a typical ROS kernel plays can be placed into three primary categories:

- Providing abstractions
- Virtualizing resources
- Managing resources

These three roles share the common goal of making a computer system easy to use. In this context, the first role is probably the most organic in that any large, reasonably written program will be constructed using some of these abstractions. Some common abstractions in modern operating systems include pages (an abstraction of memory), files (an abstraction of stored data), and processes (an abstraction of executing instructions; a *thread* plays a similar role). Given a modern CPU, it would be quite difficult (if not impossible) to write software to operate the CPU without creating abstractions. Note that providing abstractions (and interfaces) is also within the purview of *libraries*, and an OS that *only* provides such things can essentially be thought of as a library. Such *library OSes* are the subject of considerable research, and will be discussed in Chapter 8.

The second role of ROS kernel is to virtualize resources, such as memory or the CPU itself. *Virtual* resources provide a logical view of the machine that is easy to reason about from the application programmer's perspective. For example, consider a single program running on a machine. Excepting some esoteric hardware architectures, that program essentially sees memory as a contiguous series of addressable bytes. When multiple programs run on the machine, however, their usage of memory must be coordinated. Instead of a complex solution that requires the applications to be aware of this rationing of memory, the ROS kernel can instead provide a virtual abstraction of memory. Each application can then logically see memory as the expected contiguous array of bytes (using virtual addresses), while the ROS kernel partitions the physical memory behind the scenes.

The ROS kernel will also often virtualize the CPU in a similar fashion. When a CPU is virtualized, an application can be written in a way that ignores the presence of other programs in the system. In such a multi-programmed system, the ROS kernel can run several programs by multiplexing a single CPU between many *processes*, the common abstraction that an OS uses to represent a running program. Multi-programmed systems, and the virtualization mechanisms used to realize them, are not without drawbacks. A program that must contend for some resource with another program will necessarily be less efficient than one that has dedicated access to that resource. The software mechanisms used to implement virtualized resources also comes with associated overheads, though these are often mitigated by competent software design and by efficient hardware support (e.g. hardware paging for virtual memory). Finally, a program that requires some amount of determinism may be stymied by the policies that a ROS kernel employs when it virtualizes some resource. For example, a program with strict timing requirements may not be able to operate correctly on a system where the ROS kernel allows interrupts to perturb program execution at irregular intervals.

The third role of a ROS kernel is to manage resources in the system. This role is also brought about by the need to support more than one running program. There are many resource in a typical computer system that the ROS must manage. These include external devices, memory, the CPU, files, and many more. A ROS kernel that employs sound design principles will often manage such resources in ways that separate the underlying *mechanisms* that support their management from the *policies* that determine *how* they are managed. For example, a kernel will typically have a general mechanism for scheduling processes, but will support many scheduling policies, such as simple round-robin scheduling or completely fair scheduling. In order to manage resources in the system effectively, a ROS must also provide *protection*. It must protect itself from applications, it must protect applications from each other, and it must protect sensitive components of the system from erroneous or malicious behavior. The mechanisms that a ROS uses to achieve this protection will be discussed in Section 2.2.

A ROS will also provide a programming interface to the application so that it can invoke OS services with *system calls*. System calls are essentially function calls with an associated set of semantics determined by the OS. In a ringed architecture (also discussed in Section 2.2), system calls will typically initiate a transition the CPU into a higher privileged mode of operation, namely kernel mode or supervisor mode.

These roles will be important to understand, as hybrid runtimes (HRTs) organize them in a different way than a ROS kernel does. We will see these differences in Chapter 3. Readers interested in a more detailed treatment of the roles of traditional OSes are directed to classic texts in the subject [9, 186, 178, 12, 179].

2.2 ROS Kernel Architecture

Probably the most common ROS kernel architecture falls into the *monolithic kernel* category. This architecture has most of the complexity of the operating system implemented inside the kernel itself. Components within a purely monolithic kernel tend to be highly integrated. Contrasted with monolithic kernels are *microkernels*. In a microkernel design, the size of the trusted kernel is minimized because it only implements basic low-level

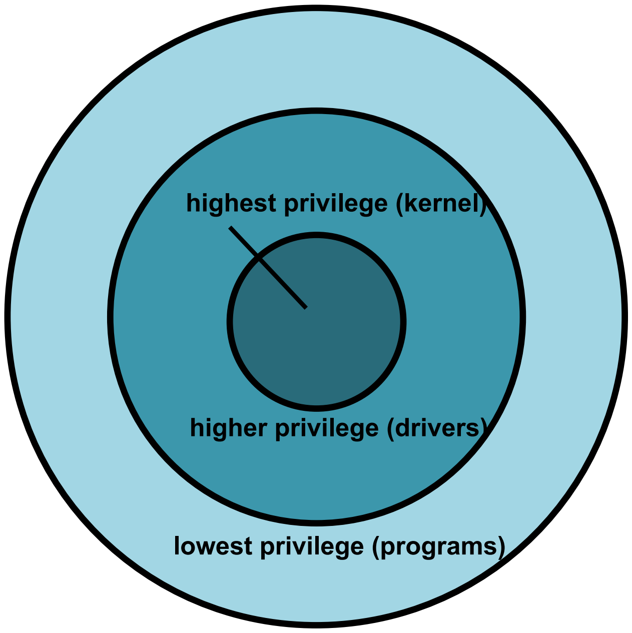

highest privilege (kernel)
higher privilege (drivers)
lowest privilege (programs)

Figure 2.1: A ringed OS architecture. The kernel has highest privilege and applications operate in the lowest privilege level.

services. More complicated functionality is relegated to modular user-space components that communicate via inter-process communication (IPC). This has several purported benefits, including reliability from the curtailment of errors that are more likely to arise in a complex kernel, flexibility from the ability to use interchangeable components, and extensibility from the modular design. Microkernels enjoy a more elegant design (and are thus more attractive from a software engineering standpoint), but they historically suffered from performance issues due to heavy-weight IPC mechanisms and architectural effects arising solely from their design (such as poor locality). Examples of both monolithic kernels and microkernels can be found in Chapter 8. Some successful ROSes, such as Linux, take a hybrid approach that blends aspects of both kernel architectures.

2.2.1 Ringed Organization

In order to implement protection, commonly available CPUs provide facilities that allow the hardware to be placed into different operating modes. For example, in the x86 architecture, the CPU can be placed into four different privileged rings (ring 0 through ring 3, with the lowest ring enjoying highest privileges). In such a system, the kernel operates at ring 0 (kernel-mode), and applications execute in the highest numbered ring (ring 3 on x86). Intermediate rings may be set up by the kernel for special purposes. Such a "ringed organization" is depicted in Figure 2.1.

Privilege changes occur on exceptional events and when the user program invokes the OS with a system call. The elevated privilege that is given to the kernel (and only the kernel), allows it access to various components of the system that are not available to a user program. This includes control over devices, control over virtual memory mappings, access to system tables and registers, control over interrupts, access to I/O instructions, access to model-specific registers (MSRs), and more. More than a third of the Intel processor manual is devoted to instructions that require elevated privilege. The implications of this restriction will be discussed further in Section 2.7.

A processor that supports hardware virtualization augments the ringed structure with another set of privilege levels below ring 0. This results in a "nested" ring organization, where four rings exist for both the host OS and the guest OS.

2.2.2 Architecture Independence

A ROS that targets multiple platforms or architectures will commonly have its code organized modularly so that parts of the kernel can plug in to platform or architecture-

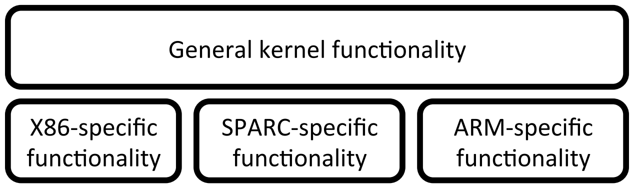

General kernel functionality
X86-specific functionality
SPARC-specific functionality
ARM-specific functionality

Figure 2.2: An architecture-independent kernel organization. Kernel interfaces are designed to plug in components specific to a particular architecture, typically at compile-time.

specific components. These components are decided on when the kernel is compiled. This is common in, for example, low-level boot code, where details of the hardware are important. This is an example of a hardware abstraction layer (HAL). A kernel architecture employing such a HAL is shown in Figure 2.2.

2.2.3 Modules and Drivers

Extensibility is an important aspect of a ROS. It allows the system to adopt new policies and for new devices to be supported without rebuilding the kernel. In addition, kernel developers are relieved the responsibility of keeping up with new devices as they are created. The device vendors can instead write code that controls the devices in the form of a *driver*. Device drivers and other functionality that might not be necessary for all users can be built as *loadable modules*. Users that need that functionality can then insert those modules into the kernel while using the system. The privilege level at which the module code runs depends on the OS, but Linux, for example, executes module code at ring zero. A system with support for such loadable modules is more flexible and customizable. Almost all commodity ROSes support loadable modules, including Linux, MS Windows, and macOS.

Loadable modules, for all their flexibility and convenience, have drawbacks as well. They often contain bugs, and can introduce vulnerabilities into an otherwise secure system.

To address this issue, a cautiously designed module system will limit the damage that a module's code can inflict on the core kernel. Linux accomplishes this by exporting only a subset of the kernel name space to modules. While this limits errant behavior and abuse of kernel interfaces, a module can still, in principle, invoke any part of the kernel by crafting addresses cleverly. This is a fundamental weakness of dynamically loadable kernel code.

2.3 Kernel Abstractions

In Section 2.1, I discussed the role that a ROS plays in creating abstractions that user programs can leverage. In this section, I will go into some detail for some of the more important abstractions, and in Section 2.7, I will give a brief overview of some of their limitations.

2.3.1 Processes

One of the most important abstractions that a ROS provides is one that represents a running program and the state associated with it. This is known as a *process*, which essentially represents an idealized machine. The process abstraction is what allows the ROS to portray to user programs the illusion that there are more CPUs in the system than there actually are. From the user's perspective, the most important operations on a process are starting, suspending, resuming, stopping, and querying its status. Such operations are invoked via an API that the ROS kernel exposes.

A process essentially consists of a collection of information that represents the states of the running program as it executes on the CPU. Important state that the ROS must track include the program's register state, its view of memory, its stack, and its program counter

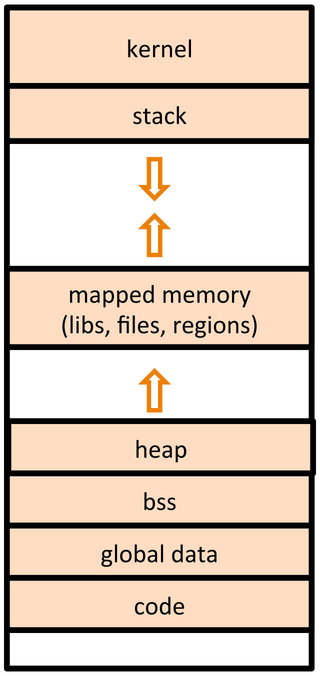


Figure 2.3: An example of a process's address space layout in Linux.

(PC). The PC represents the program's currently executing instruction, and the stack is used to implement scoped variables and activation records.

Processes in a typical system are organized into a tree structure, with a single process acting as the *root* process. This process is invoked by the ROS kernel after the system completes booting, and includes the first user-space instructions executed on the machine. On Linux systems, this process is most often named `init`.

The low-level mechanism that a ROS uses to implement time-sharing of the CPUs is essentially the ability to stash all of a process's associated state in memory. This allows the kernel to *context switch* between different processes by saving one process's state and restoring the state of another.

2.3.2 Address Spaces

One of the most important pieces of state associated with a process is its view of memory. Aside from explicitly shared memory and special chunks of memory exposed by the ROS kernel, the visible memory of each process is distinct. This is accomplished with the abstraction of an *address space*. An address space is just a series of memory addresses that a process can read and/or write. Portions of the address space can have special semantics (e.g. read-only) that are either determined by the ROS kernel or by the process itself. An example of a typical process's address space is depicted in Figure 2.3.

When a new process is created (e.g. with the `execve()` system call in Linux), that process will be provisioned with an entirely new address space. When a new process is created by duplicating an existing process (e.g. with the `fork()` system call), the duplicate process will receive a *copy* of the existing address space. In order to avoid the overhead of unnecessarily copying the underlying memory associated with the original address space, the ROS kernel may mark the memory as *copy-on-write* (COW). Then, only when one of the processes attempts to *modify* the memory is the original memory copied. Thus, any modifications will only be visible in one of the processes.

The underlying mechanism supporting address spaces is the virtual memory system. Common hardware such as x86 chips support virtual memory with *paging*, a system that organizes memory into a hierarchical tree of memory chunks called pages. Pages of memory are allocated from the available physical memory present in the machine by the kernel, and are managed by organizing them into a tree of tables called *page tables*. The tables enable the kernel to track and modify the state of memory pages. Common states include “not present,” “read-only,” and “non-executable.” When an instruction reads from or writes to a page of memory, the hardware automatically “walks” the page table tree to find the page. If the page is not present, the hardware will raise an exception (a page fault), which the ROS kernel must handle. Some systems will initially mark most memory pages as not present, and only allocate backing pages of physical memory when the program attempts to touch those pages. This is called *demand paging*, and it is used in many major ROS kernels, including Linux. There are mechanisms which allow a program to force pages to be allocated at the beginning of its execution (this is called “pre-faulting”), but these mechanisms are only used in special cases. The importance of the effects of demand paging will become evident in Section 2.7.

Because every memory reference must go through the virtual memory system, every load or store will result in a traversal of these page tables by the CPU. This can become expensive, especially for paging systems with more levels of page tables (e.g. on x86_64). To alleviate this cost, many CPU architectures include a small cache for page translations called the Translation Lookaside Buffer (TLB). The TLB is hugely important for performance [20], especially for virtualization [144].

While this section gave a cursory overview of address spaces and page tables, readers interested in more detail are advised to reference manuals from x86 CPU vendors [7, 107].

2.3.3 Threads

Operations on the various structures associated with processes can introduce overheads that make their manipulation rather expensive. In order to alleviate this cost, many ROS kernels provide the notion of *threads*. Threads can be thought of as light-weight processes. They still represent a running program, but the amount of state is reduced. The primary difference between a thread and a process centers around address spaces. Threads are decoupled from the notion of address spaces. Threads may or may not be associated with processes, but when they are, all threads *within* a process share that process's address space. Some OSes lack the notion of processes altogether, and all threads in the system share a single address space. Such OSes are called single address space OSes (SASOSes). The data structures representing an address space are not the only ones that the ROS can omit when structuring the representation of a running program. Other state, such as open files or network connections can also optionally be associated with executing instructions. Thus, there are intermediate representations of running programs that lie between a process and a thread, but such entities defy a simple nomenclature. Linux, for example, allows the programmer to specify the state associated with a newly created process by passing special flags to the `clone()` system call.

2.3.4 Files and Directories

It is often prudent for a programmer to have the ability to manipulate arbitrarily sized chunks of data stored in the system in a logical manner. Most ROSes accomplish this by providing the abstractions of *files* and *directories*. These abstractions allow the programmer (or user of the system) to access data stored in memory or on disk by using a well-defined API. The ROS implements this API with a file system, which allows users to open, close, read, write, and seek files and directories. Different file systems offer different characteristics, such as performance, networked operation, or crash consistency (e.g. with write-ahead logging). File systems can be part of the core ROS kernel or can be dynamically loaded as kernel modules. File systems, the underlying I/O subsystem in the ROS kernel, and the characteristics of I/O devices can become very important in large-scale systems like datacenters and supercomputers, where massive amounts of data are generated and stored in the form of files.

2.3.5 Communication

Most ROSes will provide abstractions that facilitate communication between entities in the system and between different systems. The former fall into the category of inter-process communication (IPC). The latter include networking facilities, which—along with the appropriate hardware—allow systems to communicate even when separated by vast distances. Both types of communication are discussed in the following paragraphs.

IPC Kernel-provided IPC abstractions give a process or thread a way to communicate with others in the system. This may be to notify another process of an asynchronous event, to share data, to send messages, or to create an "assembly line" that passes data between applications, each one completing a particular processing step.

Some common types of IPC facilities include:

- **Signals**: Signals allow one process to asynchronously notify another, often with a particular type being associated with the notification. Signals can vary in their intent, from notifying a process that it should terminate, to informing it of a window focus change or memory protection violation. Signals typically have a small amount of information, and are really only used to notify state changes. That is, there is no "payload" associated with a signal. The kernel can enforce policies which determine which processes can send signals to which recipients.
- **Message queues**: Message queues allow processes to use out-of-band communication between them. They may use direct addressing (sending to a particular process)

or they may encode a type to which multiple receivers can "subscribe." Some message queue implementations (like POSIX messages queues in Linux) support more advanced semantics like priority encoding. Unlike signals (which may be used to deliver these messages), the messages can include data payloads, typically with a maximum limit on their size.

- **Client-server**: Some ROSes support client-server IPC facilities, where one process acts as the server, and other processes send requests to the server to fetch data or to send updates. Examples in Linux include UNIX domain sockets, which allow familiar networking-style communication between processes, and Netlink sockets, which enable client-server interactions across the user/kernel boundary.

- **Pipes**: Pipes allow processes to stream data to one another. A pipe has one side dedicated to reading, and one side dedicated to writing, making it a *half-duplex* communication channel. Pipes can be *named*, allowing explicit communication, or *anonymous*, allowing them to be transparently used as standard I/O streams. Pipes are an essential mechanism for constructing "assembly lines" of programs typified by the UNIX philosophy.

- **Shared memory**: Perhaps one of the most performant IPC mechanisms, communication via shared memory allows a great degree of flexibility for applications using it, as they can determine the sharing semantics. Furthermore, once sharing is initiated, the kernel need not be involved in subsequent communication. Shared memory communication is natural for threads, where address spaces are automatically shared. For processes, however, the kernel must provide a mechanism to explicitly setup shared segments of processes' address spaces.

Networking Most ROSes will provide user programs with mechanisms to facilitate communication between machines. In a modern ROS, this involves implemenation of several protocols required by a layered network architecture. From the programmer's perspective, however, the most important abstraction is the one that represents a communication channel between two processes on different machines. Most ROSes name this type of abstraction a *socket*. It provides a logical representation of a persistent connection between machines, much like plugging a device into a wall socket. It is usually just a handle which represents the connection. Sockets can be backed by different protocols and mechanisms, depending on the user's preference and on the specific use case. Sockets typically support a standard communication API that will support functions such as `connect()`, `send()`, `recv()`, and `close()`.

By using kernel-provided networking abstractions like sockets, applications can build more complicated communication facilities with special-purpose uses. One such facility is the Message Passing Interface (MPI), widely used in parallel applications, which allows logical, high-performance, point-to-point communication between processes on different machines.

2.3.6 Synchronization and Mutual Exclusion Primitives

For a ROS that supports concurrent and parallel programming, care must be taken to coordinate access to shared resources by processes or threads running concurrently. In other words, there must be a way to guarantee the *mutual exclusion* of entities operating on such resources. The piece of code within which mutual exclusion must be guaranteed is called a *critical section*. The ROS will typically provide a set of mechanisms that allow the creation of efficient abstractions for to implement mutual exclusion in various forms, as outlined below.

Mutexes and locks Abstractions that provide mutual exclusion within a critical section are often called *mutexes*. The simplest form of a mutex is a lock. A lock consists of variable that can be in one of two states, locked or unlocked. On a uniprocessor system, a lock does not even require this lock variable. Instead, mutual exclusion within the critical section can be guaranteed by one processor or thread turning off interrupts when it enters, and re-enabling them when it leaves. This solution, however, is inadequate for multiprocessor systems, where different threads on different CPUs could enter the critical section simultaneously. In this case, a lock must have a state variable that can be modified atomically. In modern systems, the atomicity is provided by atomic read-modify-write instructions such as test-and-set or compare-and-swap. More sophisticated locks that require certain properties (such as fairness), can be implemented with algorithms like the bakery algorithm (these are also called ticket locks).

The simplest form of a lock routine will attempt to acquire the lock repeatedly until it becomes available. Such a lock is referred to as a *spin lock*. More advanced (and efficient) locks can be constructed with support from the ROS kernel, where after initially spinning for a predetermined number of times, the thread attempting to acquire the lock can go to sleep on a queue associated with the lock. When another thread releases the lock, the kernel will awaken the waiting thread.

Condition variables A programmer designing concurrent programs will often encounter situations that require that a thread wait until some condition is satisfied. For example, a thread might need to wait until a queue is populated before extracting an element from it (e.g. a producer-consumer queue). Locks alone are not a sufficient abstraction for this situation. Instead, the programmer can turn to *condition variables*. A condition variable's API will often have three primary routines: `wait()`, `signal()`, and `broadcast()`. The `wait()` routine will allow a thread to go to sleep until a condition is met. The `signal()` routine allows another thread to signal that the condition is met, and that the waiting thread can continue running. Finally, `broadcast()` is similar to `signal()`, but notifies several waiting threads instead of just one. The implementation of condition variables will become important in Chapter 6, where condition variables are used by a runtime for asynchronous software events.

Barriers Barriers enable synchronous operation among many threads by providing a mechanism by which threads can block until they have all reached a predetermined point in the code. There is one primary routine associated with a barrier, namely the `arrive()` routine. When all threads arrive at the barrier, they can continue normal operation. Barriers are used, for example, to implement stages of computation followed by stages of communication. This is a common pattern encountered in parallel programs.

Note that the performance of mutual exclusion and synchronization primitives can be very important for parallel and concurrent applications that use them heavily.

2.4 Operation

This section will give a brief overview of the operation of a typical ROS, both during startup and during its normal operation.

For a ROS to boot the system, it is commonly given control of the machine by a piece of code that resides in firmware called the BIOS (Basic Input/Output System). The BIOS, which contains the instructions first executed by the processor on startup, is charged with enumerating the devices attached to the CPU, populating system tables with their information, and finding a bootloader to bring up the operating system. There are many more steps that a commodity BIOS will, carry out, but it is sufficient for the purposes of this section to describe the BIOS as a management layer for platform-specific hardware present in the system. Once the bootloader loads the operating system into memory, it jumps to its entry point, thus relinquishing control of the machine. From that point forward, the BIOS is only invoked for a special class of system routines that deal with the management of platform hardware.

When the ROS starts, it will begin by navigating a series of steps necessary to bring the CPU into its final operating mode (for example, "long mode" on x86_64 chips). These steps include initialization of the memory management unit (e.g. paging), setup of system registers, bootup of secondary CPU cores, and population of system tables. At this point, the ROS can handle external interrupts, manage the memory system, and invoke pieces of code on other CPUs. It will also have mechanisms in place (e.g. a `panic()` routine) to provide meaningful output in the event of an error or failure.

Once the initial setup of the CPU is complete, the ROS must initialize several other subsystems before it can run user-space programs. This includes the setup of an external device infrastructure (e.g. to manage disks, network interfaces, and keyboards) and a device driver subsystem. It also must initialize the scheduling subsystem, which might include threads and processes and their associated address spaces, and the ability to context switch between them. Typically the ROS will initialize a file system layer to allow logical access to stored data.

Finally, the ROS will initialize the necessary infrastructure on the CPU to support a user-space layer with lower privileges, and the associated interface that will allow user-space programs to invoke the kernel (namely, system calls). It can then invoke an initial user-space program (`init` on Linux) that will form the root of a hierarchy of processes and threads. At this point, an interactive shell can run, allowing the user to issue commands in an interpreted language designed for the management of the system. Subsequently, the ROS acts as a passive bystander, running only when invoked by a program or when prompted by external interrupts or exceptional conditions such as faults.

2.5 Lightweight Kernels

A lightweight kernel (LWK) is meant to provide a more high-performance environment, particularly for supercomputing applications, where system interference (discussed in Section 2.7) can become a serious issue. LWKs prevent such interference by eliminating non-deterministic behavior. One of the most common is the elimination of regular timer ticks, which interrupt program execution. In a ROS, these timer ticks serve as mechanism by which the kernel can preempt running tasks regularly to enforce scheduling requirements among different processes and to load balance the system. However, in an HPC setting, there is typically only one important application running on the machine, obviating the necessity of the periodic timer mechanism. An LWK system will eliminate these regular ticks by default[1].

LWKs will also typically target a small subset of the hardware that a typical ROS will. The hardware setup is often known ahead of time, which allows the kernel to be tailored to a particular system. The overheads of abstraction layers necessary to facilitate portable operation are thus avoided, and only a few drivers must be included in the kernel. The

[1]Newer ROSes such as Linux as of kernel version 3.10, the NT kernel in Windows 8, and the XNU kernel in Apple's OSX can also be configured for such "tickless" operation.

implication of this design is that the kernel code is much simpler and comprehensible for a small group of developers. Because of their special-purpose nature and requirements for high performance, LWKs are most often monolithic.

Like ROSes, most LWKs provide an API to user applications that make it easy to program the machine. In some cases, this API is designed to be semi-compatible with ROS APIs. While this API is less comprehensive in an LWK, it enables a target set of applications to gain the benefits of the high-performance kernel without prohibitive porting effort.

The Kitten LWK serves as an instructive example of a lightweight kernel [129]. Kitten supports the Linux ABI and a subset of Linux system calls, namely those necessary to natively support OpenMP and glibc's NPTL threading facilities. As outlined above, Kitten has limited hardware support and operates without a regular timer tick. In addition, Kitten avoids complex and unpredictable performance issues in the memory system by allowing user-space to manage memory. Furthermore, Kitten uses a linear and contiguous virtual-to-physical address mapping scheme, which greatly simplifies address translation and avoids complex interactions caused by demand paging. Kitten's task scheduler employs per-CPU run queues with a simple round-robin scheduling policy. This scheme is appropriate for typical HPC applications that evenly map threads to CPUs.

While LWKs provide predictably high performance for applications, they still have limitations that can hinder runtime performance and functionality. These limitations will be discussed in Section 2.7.

2.6 Virtualization

While operating systems traditionaly provide varying forms of virtualization, some systems go even further, introducing the concept of a *virtual machine* (VM). Much like a process represents an idealized machine for an application program, a VM provides the illusion of an idealized machine to an OS. When virtualization is used, we typically refer to the physical machine as the *host* and the virtual machine as the *guest*. VMs enjoy many benefits, from allowing more than one OS to be used on the same physical machine, to enabling flexible debugging in different OS environments, to consolidating many, relatively small VMs onto a single physical machine. Such functionality is facilitated by a layer of software that sits below the OS, with even higher privilege, called a virtual machine monitor (VMM). A VMM can be thought of as an OS for OSes, but it operates at a much lower level. VMMs retain control of the machine by *trapping* privileged operations that an OS carries out in its regular operation (much like an OS retains control when exceptions and interrupts occur and when system calls are invoked).

Virtual machines can be broadly separated into two categories, *full system virtual machines* and *process virtual machines*. The latter run within the context of a process in a host OS, and only run when that process is scheduled. These type of VMs are many times used to implement *language virtual machines*, which provide an abstract machine that executes instructions in a portable instruction set that can be targeted by high-level languages. This approach is used by, for example, Java and Python. In contrast, full system virtualization uses a VMM to provide a logical view of an entire system to guest OSes running on top of it. Full system virtualization can be implemented with two types of VMMs, namely Type 1 and Type 2. Type 1 VMMs sit directly on top of the hardware, and thus are also called "bare metal" VMMs. Widely used Type 1 VMMs include VMware's ESX hypervisor [192], Windows Hyper-V, Xen [17], and KVM [164]. Type 2 VMMs require a host OS, and they usually fit within the context of a library that the host OS can be linked with or within a dynamically loadable kernel module. Examples of Type 2 VMMs include VMware Fusion, Oracle's Virtual Box, Parallels, and our Palacios VMM [129].

While virtual machines were first used in the IBM System/370 mainframe to multiplex between different OSes (see Section 8.1.1), their performance overheads outweighed their benefits for some time. They experienced a resurgence in the 1990s with the introduction of the Disco system [41], which later led to the creation of VMware. Since then, hardware vendors have introduced many features to support hardware virtualization, which eliminates many prohibitive overheads. There has also been a consistent improvement in VMM software design and VMM features that ameliorate performance issues [15, 87]. Work on our Palacios VMM showed overheads as low as 5% on a large-scale supercomputer [128].

More recent VMM software allows more sophisticated usage models that give users even more fine-grained control over their machines. One examples is the Pisces Co-Kernel system [157], which allows a machine to be physically partitioned among different OSes. Another example is our hybrid virtual machine, which will be introduced in Chapter 5.

While this section gave a cursory introduction to virtualization, readers interested in a more detailed treatment of full system virtualization are directed to more comprehensive texts [179, 147, 3].

2.7 Limitations

While a ROS can provide a great deal of flexibility and convenience, issues can arise in the face of large amounts of parallelism and in environments where deterministic and high performance are paramount concerns for a parallel runtime.

Restricted hardware access The first major issue that a parallel runtime can face when built for a ROS is restricted access to hardware. While this restricted access does increase the security and reliability of the system, it can place limits on the performance that a runtime can achieve.

For example, a runtime implementing a garbage collector might need to keep track of dirty elements (those that have been touched recently). The paging hardware on modern x86 chips provide a mechanism of accomplishing this behavior directly. However, because the ROS does not permit user programs to access this paging hardware, the runtime must use other methods to track dirty pages.

Mismatched abstractions The abstractions that the ROS provides to software at user-level may not be a good fit for a runtime's needs. For example, in a parallel runtime that uses light-weight, short-lived tasks, a heavy-weight process abstraction is unnecessary, and can actually hinder performance (e.g. due to overheads of context switching).

Both restricted access to hardware and mismatched abstractions are two major issues that can limit both the design and the performance of parallel runtimes. Appendix A contains complaints from developers of the Legion runtime specifically related to such issues with ROSes. When runtime developers run into such issues, there are often two effects. The first is that the developers have to make compromises, either by not implementing the desired functionality, or implementing it in an unnatural way that can limit its performance and elegance. The second effect is that the runtime ends up with duplicated functionality. In fact, many runtimes contain components that one can also find within the code base of a ROS. Examples include task execution frameworks, event systems, memory management, and communication facilities. Having such duplicates in the system is unnecessary and counterproductive.

Another drawback of having a rigid set of kernel abstractions and an immutable system call interface is that it can limit runtime developers' imagination during development. A runtime may have high-level information regarding its execution model, its memory access patterns, and the types of applications it supports. Runtime developers could leverage this high-level information to explore tailored solutions to problems they encounter. However, the current structure of a ROS demands that they fit their runtimes into a predetermined template.

Non-determinism Some runtimes need the ability to guarantee deterministic behavior. This can be very important for parallel runtimes in particular, where threads can run in parallel across many thousands of machines. It may be important in such an environment for the parallel tasks to operate in lock step (e.g. in the BSP model of parallel computation). When sources of non-determinism are introduced, the performance effects of a single straggler can reverberate across the entire system. Some common sources of such non-determinism (often called *OS noise* [74, 75, 101]) include external interrupts, page faults, and background processes. It can be difficult to squash OS noise in a ROS completely.

Raw performance ROSes must be fairly general in the services and abstractions they provide so that they can enable adequate performance for a large number of applications.

This generality comes at the cost of performance. Many layers of abstraction and highly modular systems incur overheads through control-flow redirection. Management of structures related to the virtualization of CPUs and memory also incurs some overheads, however highly tuned. When these heavy-weight features are not even necessary, the parallel runtime can pay a dear performance cost for the sake of convenience. These costs can be fairly small at face value, but if a runtime uses enough ROS features that are effected by overheads, the costs can compound.

In subsequent chapters, we will see empirically how all of these issues can conspire to limit both performance and functionality.

Chapter 3

Hybrid Runtime Model

This chapter details the merits of hybrid runtimes (HRTs) and their potential benefits for performance and functionality. Section 3.1 presents this argument. Section 3.2 briefly recounts the high-level goals of the Nautilus Aerokernel. Sections 3.3– 3.5 describe example HRTs that we have created using Nautilus as the underlying framework, along with performance results. Finally, Section 3.6 concludes the chapter.

3.1 Argument

A language's runtime is a system (typically) charged with two major responsibilities. The first is allowing a program written in the language to interact with its environment (at run-time). This includes access to underlying software layers (e.g., the OS) and the machine itself. The runtime abstracts the properties of both and impedance-matches them with the language's model. The challenges of doing so, particularly for the hardware, depend considerably on just how high-level the language is—the larger the gap between the language model and the hardware and OS models, the greater the challenge. At the same time, however, a higher-level language has more freedom in implementing the impedance-matching.

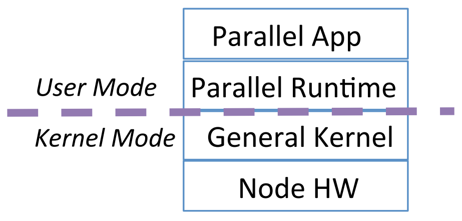
The second major responsibility of the runtime is carrying out tasks that are hidden from the programmer but necessary to program operation. Common examples include garbage collection in managed languages (discussed in the previous chapter), JIT compilation or interpretation for compilers that target an abstract machine, exception management, profiling, instrumentation, task and memory mapping and scheduling, and even management of multiple execution contexts or virtual processors. While some runtimes may offer more or less in the way of features, they all provide the programmer with a much simpler view of the machine than if he or she were to program it directly.

As a runtime gains more responsibilities and features, the lines between the runtime and the OS often become blurred. For example, the Legion runtime, a data-centric parallel runtime aimed at HPC, manages execution contexts (an abstraction of cores or hardware threads), regions (an abstraction of NUMA and other complex memory models), task to execution context mapping, task scheduling with preemption, and events. In the worst case this means that the runtime and the OS are actually trying to provide the *same* functionality. In fact, what we have found is that in some cases this duplication of functionality is brought about by inadequacies of or grievances with the OS and the services it provides. A common refrain of runtime developers (see Appendix A) is that they want the kernel to simply give them a subset of the machine's resources and then leave them alone. They attempt to approximate this as best they can within the confines of user-space and with the available system calls.

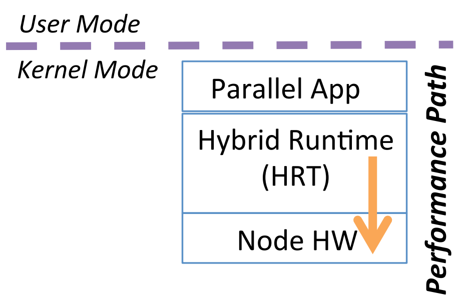
That this problem would arise is not entirely too surprising. After all, the operating system is, *prima facie*, designed to provide adequate performance for a broad range of general-purpose applications. However, when applications demand more control of the machine, the OS can often get in the way, whether due to rigid interfaces or to mismatched priorities in the design of those interfaces. Not only may the kernels abstractions be at odds with the runtime, it may also completely prevent the runtime from using hardware features that might otherwise significantly improve performance or functionality. If it provides access to these features, it does so through a system call, which—even if it has an appropriate interface for the runtime—nonetheless exacts a toll for use, as the system call mechanism itself has a cost. Similarly, even outside system calls, while the kernel might build an abstraction on top of a fast hardware mechanism, an additional toll is taken. For example, signals are simply more expensive than interrupts, even if they are used to abstract an interrupt.

A runtime that *is* a kernel will have none of these issues. It would have full access to all hardware features of the machine, and the ability to create any abstractions that it needs using those features. We want to support the construction of such *hybrid runtimes* (HRTs). To do so, we will provide basic kernel functionality on a take-it-or-leave-it basis to make the process easier. We also want such runtime kernels to have available the full functionality of the ROS for components not central to runtime operation.

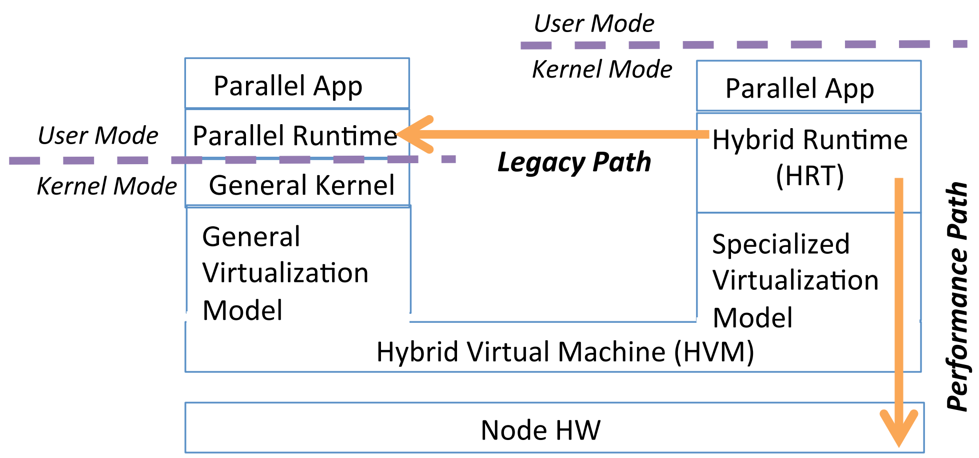
Figure 3.1 illustrates three different models for supporting a parallel runtime. The current model (a) layers the parallel runtime over a general-purpose kernel (a ROS kernel). The parallel runtime runs in user mode without access to privileged hardware features and uses a kernel API designed for general-purpose computation. In the hybrid runtime model (b) the parallel runtime is integrated with a specialized Aerokernel framework such as Nautilus. The resulting HRT runs exclusively in kernel mode with full access to all hardware features and with kernel abstractions designed specifically for it. Notice that



(a) Current Model



(b) Hybrid Runtime Model



(c) Hybrid Runtime Model Within a Hybrid Virtual Machine

Figure 3.1: Overview of hybrid runtime (HRT) approach: (a) current model used by parallel runtimes, (b) proposed HRT model, and (c) proposed HRT model combined with a hybrid virtual machine (HVM).

both the runtime and the parallel application itself are now below the kernel/user line. Figure 3.1(b) is how we run Legion, NESL, and NDPC programs. We refer to this as the *performance path*.

A natural concern with the structure of Figure 3.1(b) is how to support general-purpose OS features. For example, how do we open a file? We do not want to reinvent the wheel within an HRT or a kernel framework such as Nautilus in order to support kernel functionality that is not performance critical. Figure 3.1(c) is our response, the hybrid virtual machine (HVM), which will be discussed in detail in Chapter 5. In an HVM, the virtual machine monitor (VMM) or other software will partition the physical resources provided to a guest, such as cores and memory into two parts. One part will support a general-purpose virtualization model suitable for executing full OS stacks and their applications, while the second part will support a virtualization model specialized to the HRT and will allow it direct hardware access. We will see that more advanced models are also possible, where some resources are partitioned and others (namely, memory) are shared. The specialized virtualization model enables the performance path of the HRT, while the general virtualization model and communication between the two parts of the HVM enable a *legacy path* for the runtime and application that will let it leverage the capabilities of the general-purpose ROS kernel for non-performance critical work.

3.2 Nautilus

Nautilus[1] is a small prototype Aerokernel framework that we built to support the HRT model, and is thus the first of its kind. We designed Nautilus to meet the needs of parallel runtimes that may use it as a starting point for taking full advantage of the machine. We chose to minimize imposition of abstractions to support general-purpose applications in lieu of flexibility and small codebase size. As I will show in Sections 3.3–3.5, this allowed us to port three very different runtimes to Nautilus and the HRT model in a very reasonable amount of time (in addition to one runtime that was automatically ported; see Chapter 7). Nautilus is a Multiboot2-compliant kernel and we have tested it on several Intel and AMD machines, as well as QEMU and our own Palacios VMM.

As Nautilus is a prototype for HRT research, we initially targeted the most popular

[1]Named after the submarine-like, mysterious vessel from Jules Verne's Twenty Thousand Leagues Under the Sea.

Language	**S**LOC
C++	133
C	636

Figure 3.2: Lines of code added to Nautilus to support Legion, NDPC, NESL, and Racket.

architecture for high-performance and parallel computing, x86_64. Nautilus also runs on the Intel Xeon Phi.

The design of Nautilus is, first and foremost, driven by the needs of the parallel runtimes that use it. Nevertheless, it is complete enough to leverage the full capabilities of a modern 64-bit x86 machine to support three runtimes, one of which (Legion) is quite complex and is used in practice today.

More details on the design, implementation, and evaluation of Nautilus will be presented in Chapter 4.

Building a kernel, however, was not our main goal. Our main focus was supporting the porting and construction of runtimes for the HRT model. The Legion runtime, discussed at length in the next section, was the most challenging and complex of the three runtimes to bring up in Nautilus. Legion is about double the size of Nautilus in terms of codebase size, consisting of about 43000 lines of C++. Porting Legion and the other runtimes took a total of about four person-months of effort. Most of this work went into understanding Legion and its needs. The lines of code actually added to Nautilus to support all four runtimes is shown in Figure 3.2. We only needed to add about 800 lines of code. This is tiny considering the size of the Legion runtime.

This suggests that exploring the HRT model for existing or new parallel runtimes, especially with a small kernel like Nautilus designed with this in mind, is a perfectly manageable task for an experienced systems researcher or developer. We hope that these results will encourage others to similarly explore the benefits of HRTs.

3.3 Example: Legion

The Legion runtime system is designed to provide applications with a parallel programming model that maps well to heterogeneous architectures [21, 189]. Whether the application runs on a single node or across nodes—even with GPUs—the Legion runtime can manage the underlying resources so that the application does not have to. There are several reasons why we chose to port Legion to the HRT model. The first is that the primary focus of the Legion developers is on the design of the runtime system. This not only allows us to leverage their experience in designing runtimes, but also gives us access to a system designed with experimentation in mind. Further, the codebase has reached the point where the developers' ability to rapidly prototype new ideas is hindered by abstractions imposed by the OS layer. Another reason we chose Legion is that it is quickly gaining adoption among the HPC community, including within the DoE's Exascale effort. The third reason is that we have corresponded directly with the Legion developers and discussed with them issues with the OS layer that they found when developing their runtime. These issues are outlined in Appendix A.

Under the covers, Legion bears many similarities to an operating system and concerns itself with many issues that an OS must deal with, including task scheduling, isolation, multiplexing of hardware resources, and synchronization. As discussed at the beginning of this chapter, the way that a complex runtime like Legion attempts to manage the machine to suit its own needs can often conflict with the services and abstractions provided by the OS.

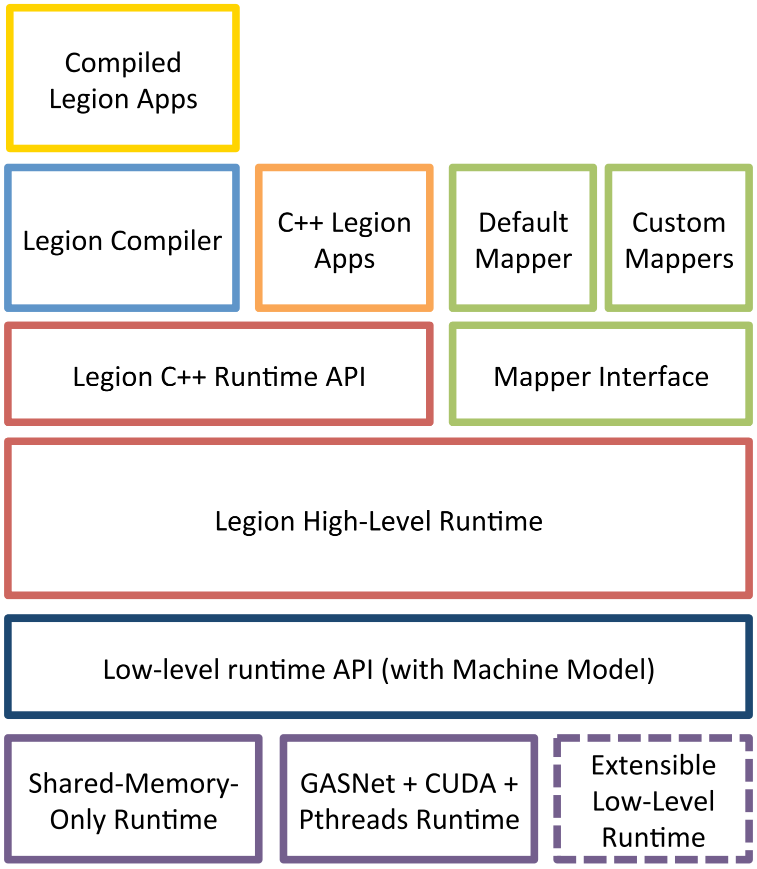

Compiled Legion Apps
Legion Compiler
C++ Legion Apps
Default Mapper
Custom Mappers
Legion C++ Runtime API
Mapper Interface
Legion High-Level Runtime
Low-level runtime API (with Machine Model)
Shared-Memory-Only Runtime
GASNet + CUDA + Pthreads Runtime
Extensible Low-Level Runtime

Figure 3.3: Legion system architecture.

As Legion is designed for heterogeneous hardware, including multi-node clusters and machines with GPUs, it is designed with a multi-layer architecture. It is split up into the *high-level* runtime and the *low-level* runtime. The high-level runtime is portable across machines, and the low-level runtime contains all of the machine specific code. There is a separate low-level implementation called the *shared low-level runtime*. This is the low-level layer implemented for shared memory machines. The overall architecture of the Legion runtime is depicted in Figure 3.3. As we are interested in single-node performance, we naturally focused our efforts on the shared low-level Legion runtime. All of our modifications to Legion when porting it to Nautilus were made to the shared low-level component. Outside of optimizations using hardware access, and understanding the needs of the runtime, the port was fairly straight-forward.

Legion, in its default user-level implementation, uses pthreads as representations of logical processors, so the low-level runtime makes fairly heavy use of the pthreads interface. In order to transform Legion into a kernel-level HRT, we simply had to provide a similar interface in Nautilus. The amount of code added to Nautilus was less than 800 lines, and is described in Figure 3.2. After porting Legion into Nautilus, we then began to explore how Legion could benefit from unrestricted access to the machine.

3.3.1 Evaluation

I now present an evaluation of our transformation of the user-level Legion runtime into a kernel using Nautilus, highlighting the realized and potential benefits of having Legion operate as an HRT. Our port is based on Legion as of October 14, 2014, specifically commit e22962d, which can be found via the Legion project web site.[2]

The Legion distribution includes numerous test codes, as well as an example parallel application that is a circuit simulator. We used the test codes to check the correctness of our work and the circuit simulator as our initial performance benchmark. Legion creates an abstract machine that consists of a set of cooperating threads that execute work when it is ready. These are essentially logical processors. The number of such threads can vary, representing an abstract machine of a different size.

Experimental setup We took all measurements on our lab machine named *leviathan*. We chose this machine for our experiment because it has a large number of cores and an interesting organization, similar to what a supercomputer node might look like. It is

[2] `http://legion.stanford.edu`

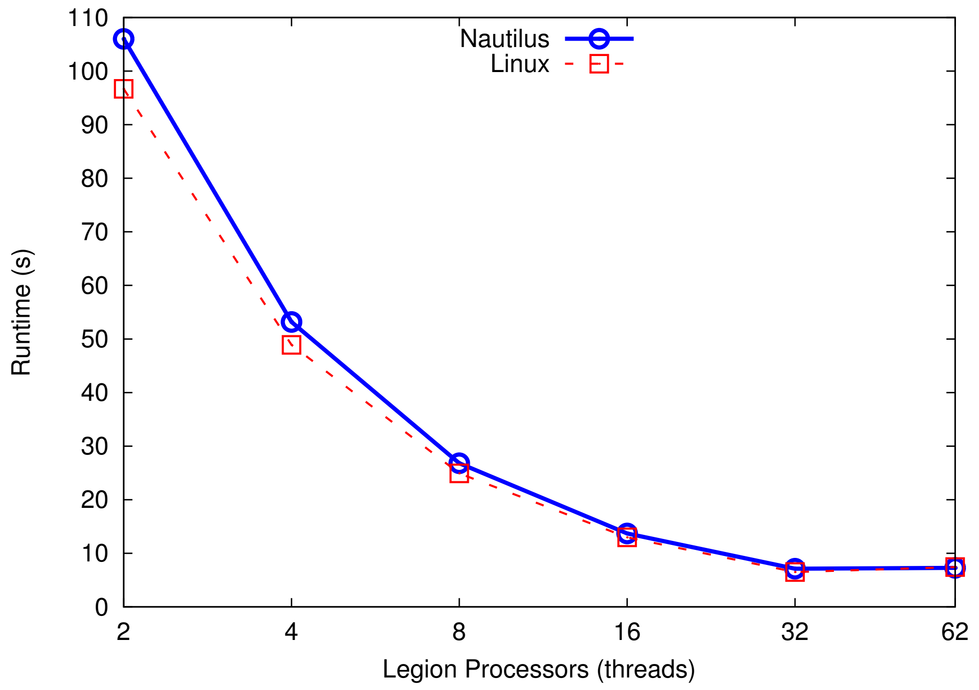


Figure 3.4: Running time of Legion circuit simulator versus core count. The baseline Nautilus version has higher performance at 62 cores than the Linux version.

a 2.1GHz AMD Opteron 6272 (Interlagos) server machine with 64 cores and 128 GB of memory. The cores are spread across 4 sockets, and each socket comprises two NUMA domains. All CPUs within one of these NUMA domains share an L3 cache. Within the domain, CPUs are organized into 4 groups of 2 hardware threads. The hardware threads share an L1 instruction cache and a unified L2 cache. Hardware threads have their own L1 data cache. We configured the BIOS for this machine to "Maximum performance" to eliminate the effects of power management. This machine also has a "freerunning' TSC, which means that the TSC will tick at a constant rate regardless of the operating frequency of the processor core. For Linux tests, it runs Red Hat 6.5 with stock Linux kernel version 2.6.32. For Xeon Phi tests, we use a Xeon Phi 3120A PCI accelerator along with the Intel MPSS 3.4.2 toolchain, which uses a modified 2.6.38 Linux kernel. It is important to point out that this is the current kernel binary shipped by Intel for use with Intel Xeon Phi

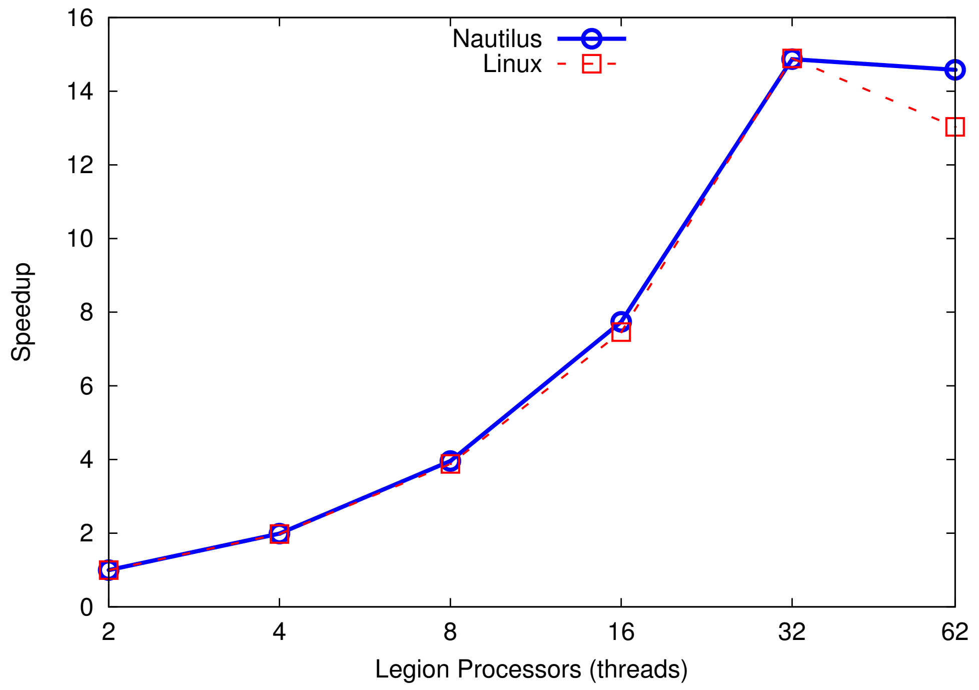


Figure 3.5: Speedup of Legion (normalized to 2 Legion processors) circuit simulator running on Linux and Nautilus as a function of Legion processor (thread) count.

hardware.

3.3.2 Legion Circuit Simulator

We ran the circuit simulator with a medium problem size (100000 steps) and varied the number of cores Legion used to execute Legion tasks. Figure 3.4 shows the results. The x-axis shows the number of threads/logical processors. The thread count only goes up to 62 because the Linux version would hang at higher core counts, we believe due to a livelock situation in Legion's interaction with Linux. Notice how closely, even with no hardware optimizations, Nautilus tracks the performance of Linux. The difference between the two actually increases when scaling the number of threads. They are essentially at parity. Nautilus is slightly faster at 62 cores.

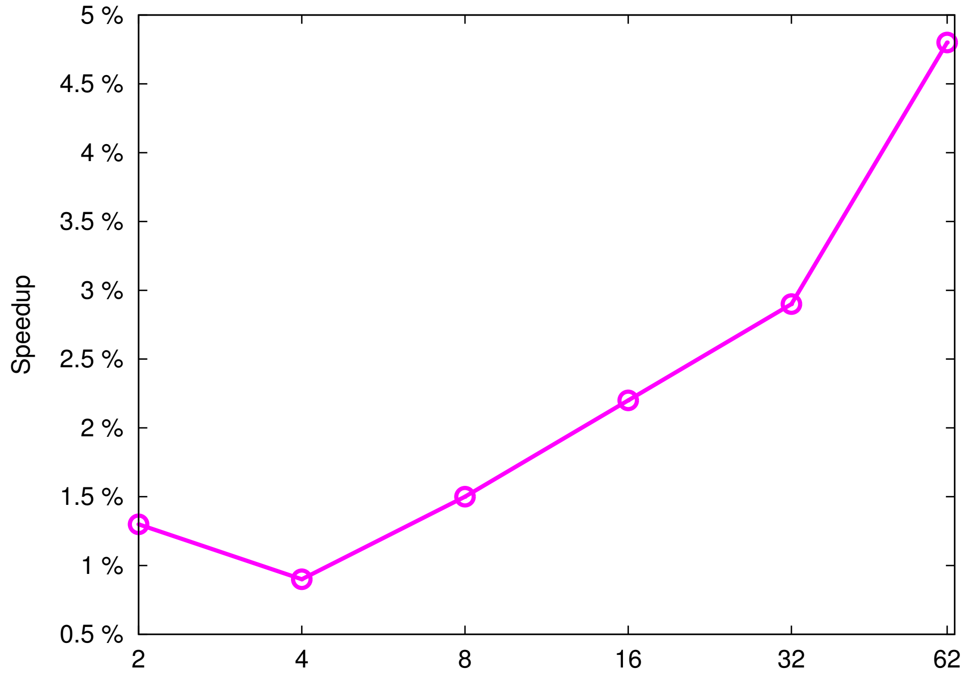


Figure 3.6: Speedup of Legion circuit simulator comparing the baseline Nautilus version and a Nautilus version that executes Legion tasks with interrupts off.

The speedup of the circuit simulator running in Legion as a function of the number of cores is shown in Figure 3.5. Speedups are normalized to Legion running with two threads.

To experiment with hardware functionality in the HRT model, we wanted to take advantage of a capability that normally is unavailable in Linux at user-level. We decided to use the capability to disable interrupts. In the Legion HRT, there are no other threads running besides the threads that Legion creates, and so there is no need for timer interrupts (or device interrupts). Observing that interrupts can cause interference effects at the level of the instruction cache and potentially in task execution latency, we inserted a call to disable interrupts when Legion invokes a task (in this case the task to execute a function in the circuit simulator). Figure 3.6 shows the results, where the speedup is over the baseline case where Legion is running in Nautilus but without any change in the default interrupt policy. While this simple change involved only adding two short lines of code, we can see a measurable benefit that scales with the thread count, up to 5% at 62 cores.

3.3.3 Legion HPCG

To evaluate the performance benefits of applying the HRT model to Legion using a more complex benchmark, we used the HPCG (High Performance Conjugate Gradients) mini-app. HPCG is an application benchmark effort from Sandia National Labs that is designed to help rank top-500 supercomputers for suitability to scalable applications of national interest [67, 97]. Los Alamos National Laboratory has ported HPCG to Legion, and we used this port to further evaluate our HRT variant of Legion. HPCG is a complex benchmark ($\sim$5100 lines of C++) that exercises many Legion features. Recall that Legion itself comprises another $\sim$43,000 lines of C++.

Figure 3.7 shows the speedup of the HPCG/Legion in Nautilus over HPCG/Legion on Linux as a function of the number of Legion processors being used. Each Legion processor is bound to a distinct hardware thread. On the Xeon Phi, Nautilus is able to speed up HPCG by up to 20%. On x64, Nautilus increases its performance by almost 40%. We configured HPCG for a medium problem size which, on a standard Linux setup, runs for roughly 2 seconds. We see similar results for other HPCG configurations.

There are many contributors to the increased performance of HPCG in Nautilus, particularly fast condition variables. An interesting one is simply due to the simplified paging model. On x64 the Linux version exhibited almost 1.6 million TLB misses during execution. In comparison, the Nautilus version exhibited about 100.

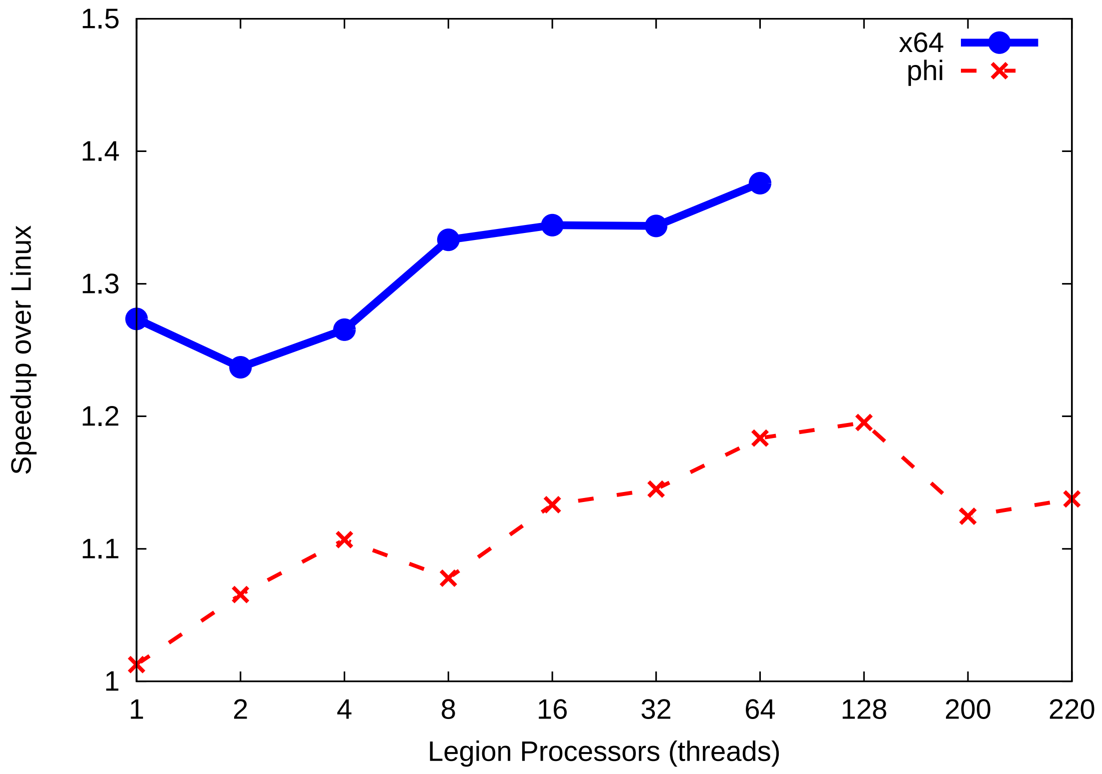


Figure 3.7: Nautilus speedup of Legion HPCG on our x64 machine and the Intel Xeon Phi.

Language	**SLOC**
Compiler	
Lisp	11005
Runtime	
C	8853
lex	230
yacc	461

Figure 3.8: Source lines of code for NESL. The runtime consists of the VCODE interpreter and the CVL implementation we use.

3.4 Example: NESL

NESL [33] is a highly influential implementation of nested data parallelism developed at CMU in the '90s. Very recently, it has influenced the design of parallelism in Manticore [80, 78], Data Parallel Haskell [48, 50], and arguably the nested call extensions to

CUDA [150]. NESL is a functional programming language, using an ML-style syntax that allows the implementation of complex parallel algorithms in a very compact and high-level way, often 100s or 1000s of times more compactly than using a low-level language such as C+OpenMP. NESL programs are compiled into abstract vector operations known as VCODE through a process known as flattening. An abstract machine called a VCODE interpreter then executes these programs on physical hardware. Flattening transformations and their ability to transform nested (recursive) data parallelism into "flat" vector operations while preserving the asymptotic complexity of programs is a key contribution of NESL [34] and very recent work on using NESL-like nested data parallelism for GPUs [29] and multicore [28] has focused on extending flattening approaches to better match such hardware.

As a proof of concept, we have ported NESL's existing VCODE interpreter to Nautilus, allowing us to run any program compiled by the out-of-the-box NESL compiler in kernel mode on x86_64 hardware. We also ported NESL's sequential implementation of the vector operation library CVL, which we have started parallelizing. Currently, point-wise vector operations execute in parallel. The combination of the core VCODE interpreter and a CVL library form the VCODE interpreter for a system in the NESL model.

While this effort is a work in progress, it gives some insights into the challenges of porting this form of language implementation to the kernel level. In summary, such a port is quite tractable. A detailed breakdown of the source code in NESL as we use it is given in Figure 3.8. Compared to the NESL source code release[3], our modifications currently comprise about 100 lines of Makefile changes and 360 lines of C source code changes. About 220 lines of the C changes are in CVL macros that implement the point-wise vector

[3]http://www.cs.cmu.edu/~scandal/nesl/nesl3.1.html

Language	SLOC
Compiler	
Perl	2000
lex	82
yacc	236
Runtime	
C	2900
C++	93
x86 assembly	477

Figure 3.9: Source lines of code for NDPC. The runtime counts include code both for use on Nautilus and for user-level use.

operations we have parallelized. The remainder (100 Makefile lines, 140 C lines) reflect the amount of glue logic that was needed to bring the VCODE interpreter and the serial CVL implementation into Nautilus. The hardest part of implementing this glue logic is assuring that the compilation and linking model match that of Nautilus, which is reflected in the Makefile changes. The effort took roughly ten weeks to complete.

3.5 Example: NDPC

We have created a different implementation of a subset of the NESL language which we refer to as "Nested Data Parallelism in C/C++" (NDPC). This is implemented as a source-to-source translator whose input is the NESL subset and whose output is C++ code (with C bindings) that uses recursive fork/join parallelism instead of NESL's flattened vector parallelism. The C++ code is compiled directly to object code and executes without any interpreter or JIT. Because C/C++ is the target language, the resulting compiled NDPC program can easily be directly linked into and called from C/C++ codebases. NDPC's collection type is defined as an abstract C++ class, which makes it feasible for the generated code to execute over any C/C++ data structure provided it exposes or is wrapped with the suitable interface. We made this design decision to further facilitate "dropping into NDPC" from C/C++ when parallelism is needed. In the context of Figure 3.1(c), we plan that the runtime processing of a call to an NDPC function will include crossing the boundary between the general-purpose and specialized portions of the hybrid virtual machine (HVM).

Figure 3.9 breaks down the NDPC implementation in terms of the languages used and the size of the source code in each. The NDPC compiler is written in Perl and its code breaks down evenly between (a) parsing and name/type unification, and (b) code generation. The code generator can produce sequential C++ or multithreaded C++. The generated code uses a runtime that is written in C, C++, and assembly that provides preemptive threads and simple work stealing.

Code generation is greatly simplified because the runtime supports a `thread_fork()` primitive. The runtime guarantees that a forked thread will terminate at the point it attempts to return from the current function. The NDPC compiler guarantees the code it generates for the current function will only use the current caller and callee stack frames, that it will not place pointers to the stack *on* the stack, and that the parent will join with any forked children before it returns from the current function. The runtime's implementation of `thread fork()` can thus avoid complex stack management. Furthermore, it can potentially provide very fast thread creation, despite the `fork()` semantics, because it can avoid most stack copying as only data on the caller and callee stack frames may be referenced by the child. In some cases, the compiler can determine the maximum stack size (e.g., for a leaf function), and supply this to the runtime, further speeding up thread creation.

We also note that the compiler knows exactly what parts of the stack can be written, and it knows that the lifetime of a child thread nests within the lifetime of its parent. This knowledge could be potentially leveraged at the kernel level by maintaining only a single copy of read-only data on the parent stack and the child stacks.

Two user-level implementations of threading are included in the runtime, one of which is a veneer on top of pthreads, while the other is an implementation of user-level fibers that operates similarly to a kernel threading model. The kernel-level implementation, for Nautilus, consists of only 150 lines of code, as Nautilus supports this threading model internally. It is important to point out that the thread fork capability was quite natural to add to Nautilus, but somewhat painful to graft over pthreads. Even for user-level fibers, the thread fork capability requires somewhat ugly trampoline code which is naturally avoided in the kernel.

As with the NESL VCODE port (Section 3.4) the primary challenges in making NDPC operate at kernel-level within Nautilus have to do with assuring that the compilation and linking models match those of Nautilus. An additional challenge has been dealing with C++ in the kernel, although the C++ feature set used by code generated by NDPC is considerably simpler than that needed by Legion. Currently, we are able to compile simple benchmarks such as nested data parallel quicksort into Nautilus kernels and run them. NDPC is a work in progress, but the effort to bring it up in Nautilus in its present state required about a week.

3.6 Conclusions

In this chapter, I made the case for transforming parallel runtimes into operating system kernels, forming hybrid runtimes (HRTs). The motivations for HRTs include the increasing complexity of hardware, the convergence of parallel runtime concerns and abstractions in managing such hardware, and the limitations of executing the runtime at user-level, both in terms of limited hardware access and limited control over kernel abstractions. For the Legion runtime, we were able to exceed Linux performance with simple techniques that can only be done in the kernel. Building Nautilus was a six person-month effort, while porting the runtimes was a four person-month effort. It is somewhat remarkable that even with a fairly nascent kernel framework, *just* by dropping the runtime down to kernel level and taking advantage of a kernel-only feature in two lines of code, we can exceed performance on Linux, an OS that has undergone far more substantial development and tuning effort.

How this is possible will become more clear in the next chapter, where I discuss the detailed design, implementation, and evaluation of Nautilus.

Chapter 4

Nautilus Aerokernel Framework

This chapter discusses in depth the design, implementation, and evaluation of the Nautilus Aerokernel, an enabling tool for hybrid runtimes. As I outlined briefly in previous chapters, Nautilus is a kernel framework specifically designed to support the creation of HRTs. It provides a basic kernel that can be booted within milliseconds after boot loader execution on a multicore, multisocket machine, accelerator, or virtual machine. Nautilus includes basic building blocks such as simple memory management, threads, synchronization, IPIs and other in-kernel abstractions that a parallel runtime can be ported to or be built on top of to become an HRT. While Nautilus provides functionality, it does not require the HRT to use it, nor does it proscribe the implementation of other functionality. Nautilus was developed for 64-bit x86 machines (x64) and then ported to the Intel Xeon Phi.

I will present detailed microbenchmark evaluations of Nautilus, comparing its functionality and performance to that of analogous facilities available at user-level on Linux that are typically used within parallel runtimes. Nautilus functionality such as thread creation and events operate up to two orders of magnitude faster than Linux due to their implementation and by virtue of the fact that there is no kernel/user boundary to cross.

They also operate with much less variation in performance, an important consideration for many models of parallelism, particularly with scale.

Section 4.1 presents an overview of the Nautilus Aerokernel. Section 4.2 describes the design of Nautilus and its various mechanisms. Section 4.3 describes the Intel Xeon Phi port of Nautilus. Section 4.4 contains a detailed evaluation of the mechanisms and abstractions that Nautilus provides. Finally, Section 4.5 closes this chapter with concluding comments.

4.1 Nautilus

The design and implementation of Nautilus has been driven by studying parallel runtimes, including the three (Legion, NESL, NPDC) described in the previous chapter, the SWARM data flow runtime [131], ParalleX [113], Charm++ [114], the futures and places parallelism extensions to the Racket runtime [185, 187, 184], and nested data parallelism in Manticore [81, 79] and Haskell [48, 49]. We have studied these codebases and in the case of Legion, NPDC, SWARM, and Racket, we also interviewed their developers to understand their views of the limitations of existing kernel support.

Nautilus is *not* a general-purpose kernel. In fact, there is not even a user-space. Instead, its design focuses specifically on helping parallel runtimes to achieve high performance as HRTs. The non-critical path functionality of the runtime is assumed to be delegated, for example to the host in the case of an accelerator or to the ROS portion of an HVM (as described in Chapter 5). The abstractions Nautilus provides are based on our analysis of the needs of the runtimes we examined. Our abstractions are optional. Because an HRT runs entirely at kernel level, the developer can also directly leverage all hardware

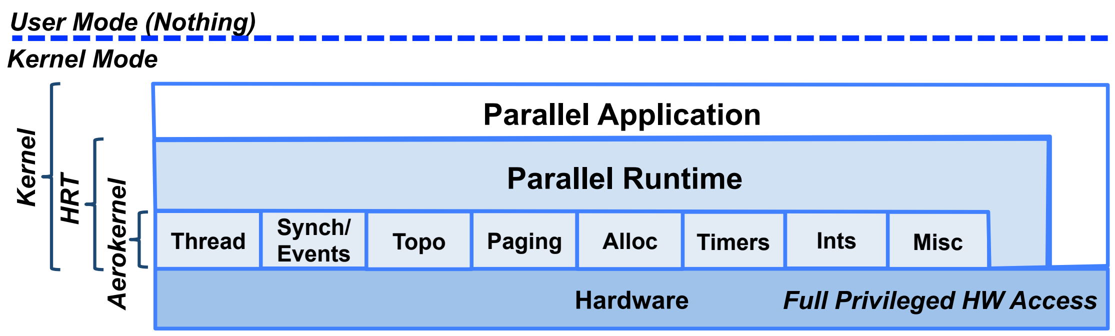


Figure 4.1: Structure of Nautilus.

functionality to create new abstractions.

Our choice of abstractions was driven in part to make it feasible to port existing parallel runtimes to become Aerokernel-based HRTs. A more open-ended motivator was to facilitate the design and implementation of new parallel runtimes that do not have a built-in assumption of being user space processes.

4.2 Design

Nautilus is designed to boot the machine, discover its capabilities, devices, and topology, and immediately hand control over to the runtime. Figure 4.1 shows the basic structure, showing the functionality provided by Nautilus in the context of the runtime and application. Note that Nautilus is a thin layer in the HRT model, and that in this model there is no user space. The runtime and the application have full access to hardware and can pick and choose which Aerokernel functionality to use. The entire assemblage of the figure is compiled into a Multiboot2-compliant kernel.

We focus the following discussion on functionality where Nautilus differs most from other kernels. In general, the defaults for Nautilus functionality strive to be simple and easy to reason about from the HRT developer's viewpoint.

4.2.1 Threads

In designing a threading model for Aerokernel, we considered the experiences of others, including work on high-performance user-level threading techniques like scheduler activations [10] and Qthreads [194]. Ultimately, we designed our threads to be lightweight in order to provide an efficient starting point for HRTs. Nautilus threads are *kernel* threads. A context switch between Nautilus threads *never* involves a change of address spaces. Nautilus threads can be configured to operate either preemptively or cooperatively, the latter allowing for the elimination of timer interrupts and scheduling of threads exactly as determined by the runtime.

The nature of the threads in Nautilus is determined by how the runtime uses them. This means that we can directly map the logical view of the machine from a runtime's point of view to the physical machine. This is not typically possible to do with any kind of guarantees when running in user-space. In fact, this is one of the concerns that the Legion runtime developers expressed with running Legion on Linux (see Appendix A). The default scheduler and mapper binds a thread to a specific hardware thread as selected by thread creator, and schedules round-robin. The runtime developer can easily change these policies.

Another distinctive aspect of Nautilus threads is that a thread fork (and join) mechanism is provided in addition to the common interface of starting a new thread with a clean new stack in a function. A forked thread has a limited lifetime and will terminate when it returns from the current function. It is incumbent upon the runtime to manage the parent and child stacks correctly. This capability is leveraged in our ports of NESL and NDPC.

Thread creation, context switching, and wakeup are designed to be fast and to leverage runtime knowledge. For example, maximum stack sizes and context beyond the GPRs can be selected at creation time. Because interrupt context uses the current thread's stack, it is even possible to create a single large-stacked idle thread per hardware thread and then drive computation entirely by inter-processor interrupts (IPIs), one possible mapping of an event-driven parallel runtime such as SWARM.

4.2.2 Synchronization and Events

Nautilus provides several variants of low-level spinlocks, including MCS locks and bakery locks. These are similar to those available in other kernels, and comparable in performance.

Nautilus focuses to a large extent on asynchronous events, which are a common abstraction that runtime systems often use to distribute work to execution units, or workers. For example, the Legion runtime makes heavy use of them to notify logical processors (Legion threads) when there are Legion tasks that are ready to be executed. Userspace events require costly user/kernel interactions, which we *eliminate* in Nautilus.

Nautilus provides two implementations of condition variables that are compatible with those in pthreads. These implementations are tightly coupled with the scheduler, eliminating unnecessary software interactions. When a condition is signaled, the default Nautilus condition variable implementation will simply put the target thread on its respective hardware thread's ready queue. This, of course, is not possible from a user-mode thread.

When a thread is signaled in Nautilus it will not run until the scheduler starts it. For preemptive threads, this means waiting until the next timer tick, or an explicit yield from the currently running thread. Our second implementation of condition variables mitigates this delay by having the signaling thread "kick" the appropriate core with an IPI after it has woken up the waiting thread. The scheduler recognizes this condition on returning from the interrupt and switches to the awakened thread.

The runtime can also make direct use of IPIs, giving it the ability to force immediate execution of a function of its choosing on a remote destination core. Note that the IPI mechanism is unavailable when running in user-space.

A more detailed treatment of asynchronous events will be given in Chapter 6.

4.2.3 Topology and Memory Allocation

Modern NUMA machines organize memory into separate domains according to physical distance from a physical CPU socket, core, or hardware thread. This results in variable latency when accessing memory in the different domains and also means achieving high memory bandwidth requires leveraging multiple domains simultaneously. Platform firmware typically enumerates these NUMA domains and exposes their sizes and topology to the operating system in a way that supports both modern and legacy OSes.

Nautilus captures this topology information on boot and exposes it to the runtime. The page and heap allocators in Nautilus allow the runtime to select which domains to allocate from, with the default being that allocations are satisfied from the domain closest to the current location of the thread requesting the allocation. All allocations are carried out immediately. This is in contrast to the policy of deferred allocations whose domains are determined on first touch, the typical default policy for general purpose kernels. A consequence is that a runtime that implements a specific execution policy, for example the owner-computes rule (e.g., as in HPF [98]) or inspector-executor [62], can more easily reason about how to efficiently map a parallel operation to the memory hardware.

A thread's stack is allocated using identity-mapped addresses based on the initial binding of the thread to a hardware thread, again to the closest domain. Since threads do not by default migrate, stack accesses are low latency, even across a large stack. If the runtime is designed so that it does not allow or can fix pointers into the stack, even the stack can be moved to the most friendly domain if the runtime decides to move the thread to a different hardware thread.

We saw NUMA effects that would double the execution time of a long-running parallel application on the Legion runtime. While user-space processes do typically have access to NUMA information and policies, runtimes executing in Nautilus have *full* control over the placement of threads and memory and can thus enjoy guarantees about what can affect runtime performance.

4.2.4 Paging

Nautilus has a simple, yet high-performance paging model aimed at high-performance parallel applications. When the machine boots up, each hardware thread identity-maps the entire physical address space using large pages (2MB and 1 GB pages currently, 512 GB pages when available in hardware) to create a single *unified* address space. Optionally, the identity map can be offset into the "higher half" of the x64 address space (one use of this is discussed in Chapter 5). Nautilus can also be linked to load anywhere in the physical address space.

The static identity map eliminates expensive page faults and TLB shootdowns, and reduces TLB misses. These events not only reduce performance, but also introduce unpredictable OS noise [74, 75, 101] from the perspective of the runtime developer. OS noise is well known to introduce timing variance that becomes a serious obstacle in large-scale distributed machines running parallel applications. The same will hold true for single

Environment	**TLB Misses**
Linux user-space	1588314
Nautilus	100

Figure 4.2: Unified TLB misses during a run of HPCG on our x64 machine.

nodes as core counts continue to scale up. The introduction of variance by OS noise (not just by asynchronous paging events) not only limits the performance and predictability of existing runtimes, but also limits the *kinds* of runtimes that can take advantage of the machine. For example, runtimes that need tasks to execute in synchrony (e.g., in order to support a bulk-synchronous parallel [85] application or a runtime that uses an abstract vector model) will experience serious degradation if OS noise comes into play.

To measure the benefits of the Nautilus paging model, we ran HPCG in Legion (discussed in the previous chapter) on top of both Nautilus and Linux user-space and counted the TLB misses during this run. Figure 4.2 shows the results. This includes dTLB load and store misses and iTLB load misses for all page sizes. In Nautilus, we program the performance counters directly, and for Linux we use the `perf stat` utility.

The use of a single unified address space also allows fast communication between threads, and eliminates much of the overhead of context switches. The only context switches are between kernel threads, so no page table switch or kernel-triggered TLB flush ever occurs. This is especially useful when Nautilus runs virtualized, as a large portion of VM exits come from paging related faults and dynamic mappings initiated by the OS, particularly using shadow paging. A shadow-paged Aerokernel exhibits the minimum possible shadow page faults, and shadow paging can be more efficient that nested paging, except when shadow page faults are common [14].

4.2.5 Timers

Nautilus optionally enables a per-hardware thread scheduler tick mechanism based on the Advanced Programmable Interrupt Controller (APIC) timer. This is only needed when preemption is configured.

For high resolution time measurement across hardware threads, Nautilus provides a driver for the high-precision event timer (HPET) available on most modern x64 machines. This is a good mapping for real-time measurement in the runtimes we examined. Within per-hardware thread timing, the cycle counter is typically used.

4.2.6 Interrupts

External interrupts in Nautilus work just like any other operating system, with the exception that by default only the APIC timer interrupt is enabled at bootup (and only when preemption is configured). The runtime has complete control over interrupts, including their mapping, assignment, and priority ordering.

4.2.7 API for Porting

A unique feature of Nautilus is its internal API meant to facilitate porting user-space applications and runtimes to kernels. These APIs will be familiar to developers used to writing parallel code, and they include interfaces for manipulating threads, events, locks, and others. The threading interface is partly compatible with POSIX threads (pthreads). Nautilus also includes many familiar functions from glibc.

4.3 Xeon Phi

We have ported Nautilus to the Intel Xeon Phi. Although the Phi is technically an x64 machine, it has differences that make porting a kernel to it challenging. These include the lack of much PC legacy hardware, a distinctive APIC addressing model, a distinctive frequency/power/thermal model, and a bootstrap and I/O model that is closely tied to Intel's MPSS stack. Our port consists of two elements.

Philix is an open-source set of tools to support booting and communicating with a third-party kernel on the Phi in compliance with Intel's stack, while at the same time not requiring the kernel to itself support the full functionality demanded of MPSS. Philix also includes basic driver support for the Phi that can be incorporated into the third-party kernel. This includes console support on both the host and Phi sides to make debugging a new Phi kernel easier. Philix comprises 1150 lines of C.

Our changes to add Phi support to Nautilus comprised about 1350 lines of C. This required about 1.5 person months of kernel developer effort, mostly spent in ferreting out the idiosyncrasies of the Phi.

4.4 Microbenchmarks

I now present an evaluation of the performance of the basic primitives in Nautilus that are particularly salient to HRT creation, comparing them to Linux user-level and kernel-level primitives. The performance of basic primitives is important because runtimes build on these mechanisms. Although they can use the mechanisms cleverly (Legion's task model is effectively a thread pool model, for example), making the underlying primitives and

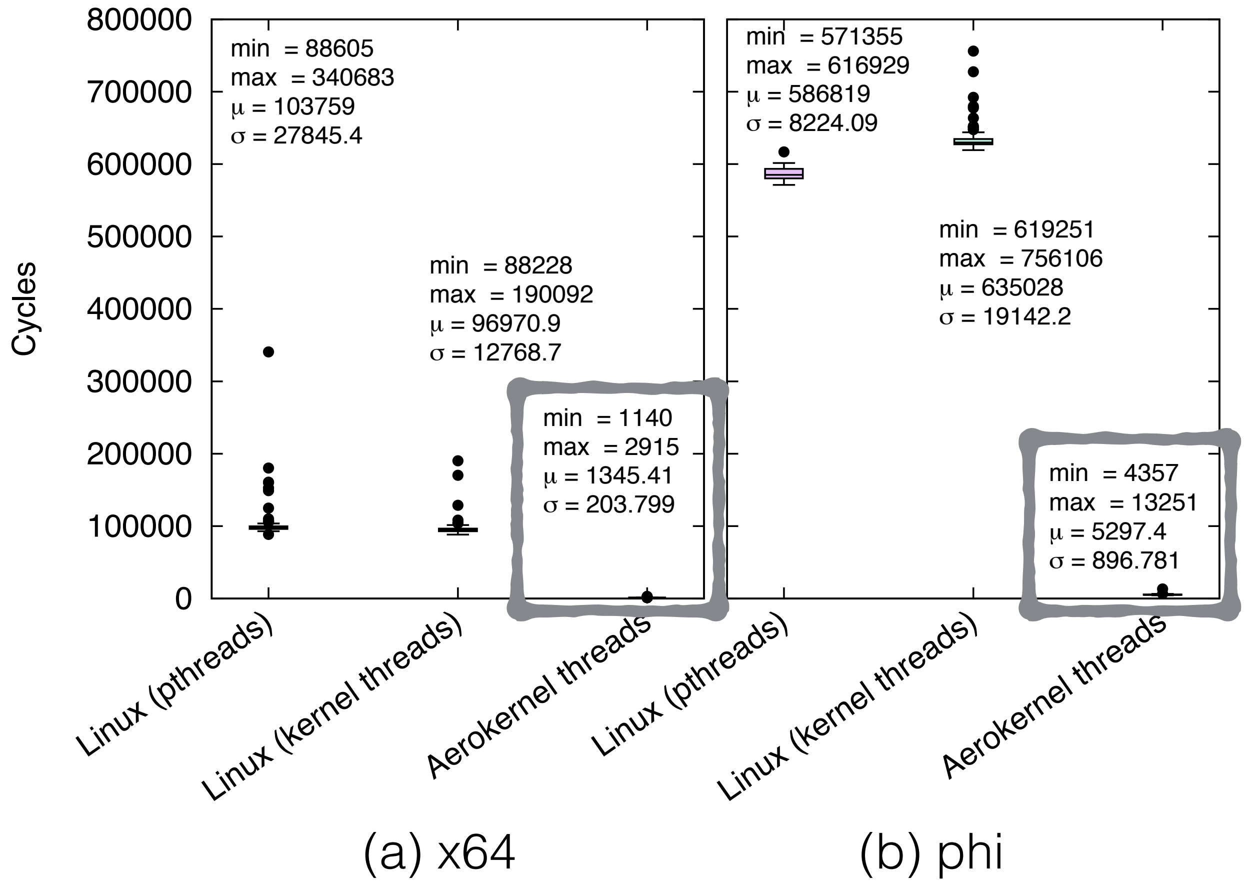


Figure 4.3: Thread creation latency. Nautilus thread creations are on average two orders of magnitude faster than Linux user-space (pthreads) or kernel thread creations and have at least an order of magnitude lower variance.

environment faster can make runtimes faster, as we shall see. The experimental setup in this section is the same as discussed in Section 3.3.1.

4.4.1 Threads

Figure 4.3 compares thread creation latency between Linux user-space (pthreads), Linux kernel threads, and Nautilus threads. We compare with pthreads because runtimes (such as Legion) build on these mechanisms. While thread creation may or may not be on the performance critical path for a particular runtime (Legion creates threads only once at startup time and binds them to physical cores), the comparison demonstrates the lightweight capabilities of Nautilus. The cost measured is the time for the thread creation function to return to the creator thread. For Nautilus, this includes placing the new thread in the run queue of its hardware thread. A thread fork in Nautilus has similar latency since the primary difference compared to ordinary thread creation has to do the content of the initial stack for the new thread. The time for the new thread to begin executing is bounded by the context switch time, which we measure below.

On both platforms, thread creation in Nautilus has two orders of magnitude lower latency on average than both Linux options, and, equally important, the latency has little variance. Thread creation in Nautilus also scales well, as, like the others, it involves constant work. From an HRT developer's point of view, these performance characteristics potentially makes the creation of smaller units of work feasible, allows for tighter synchronization of their execution, and allows for large numbers of threads.

Figure 4.4 illustrates the latencies of context switches between threads on the two platforms, comparing Linux and Nautilus. In both cases, no floating point or vector state is involved—the cost of handling such state is identical across Linux and Nautilus. The average cost of a Nautilus context switch on the x64 is about 10% lower than that of Linux, but Nautilus exhibits a variance that's lower by a factor of two. On the Phi, Nautilus exhibits two orders of magnitude lower variance in latency and more than factor of two lower average latency. The instruction count for a thread context switch in Nautilus is much lower than that for Linux. On the x64, this does not have much effect because the hardware thread is superscalar. On the other hand, the hardware thread on the Phi is not only not superscalar, but four hardware threads round-robin instruction-by-instruction for the execution core. As a consequence, the lower instruction count translates into a much

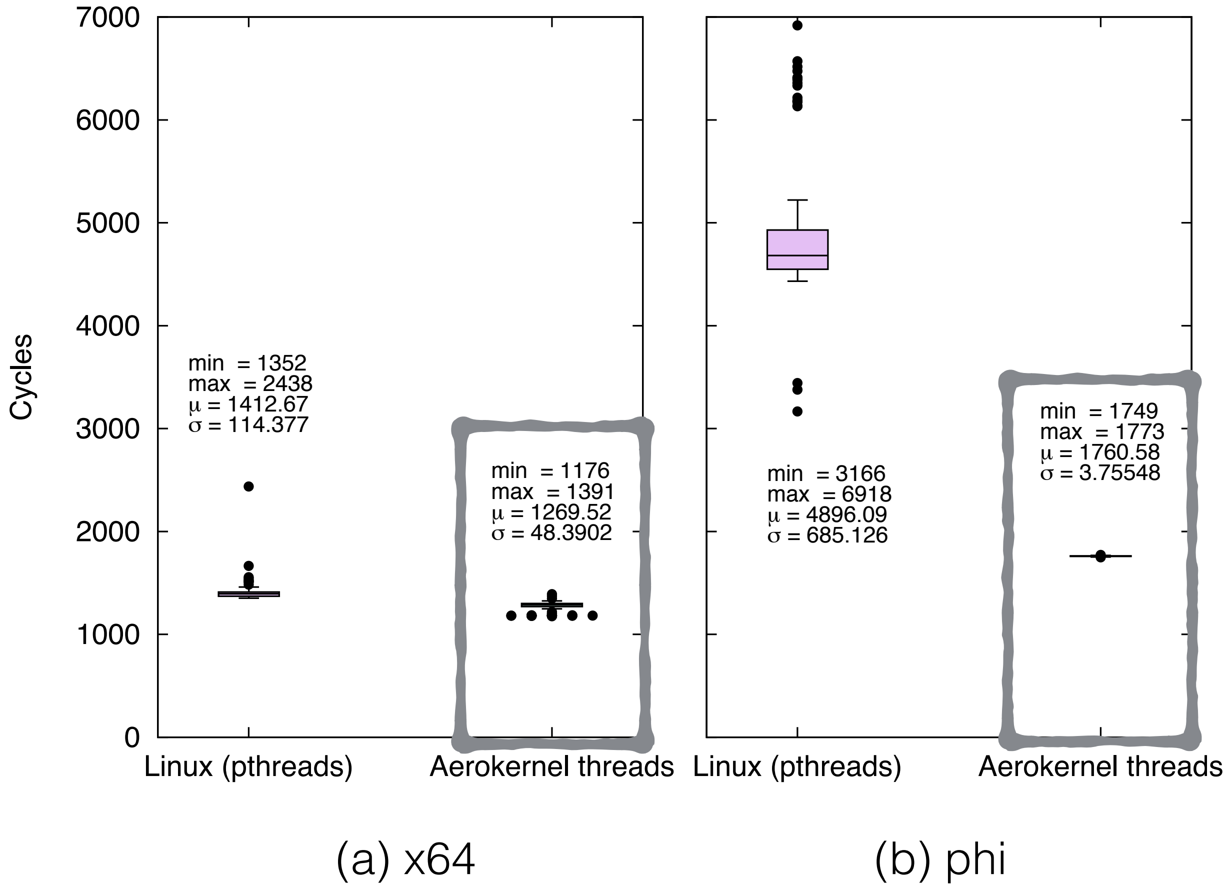
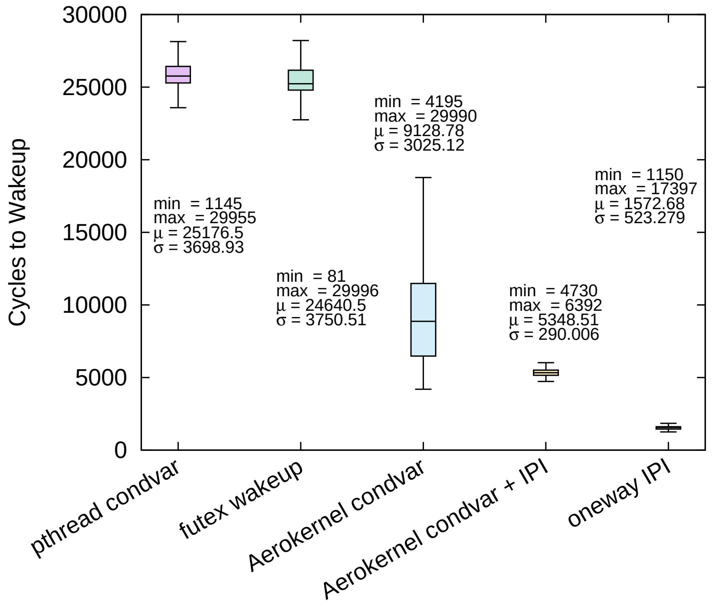


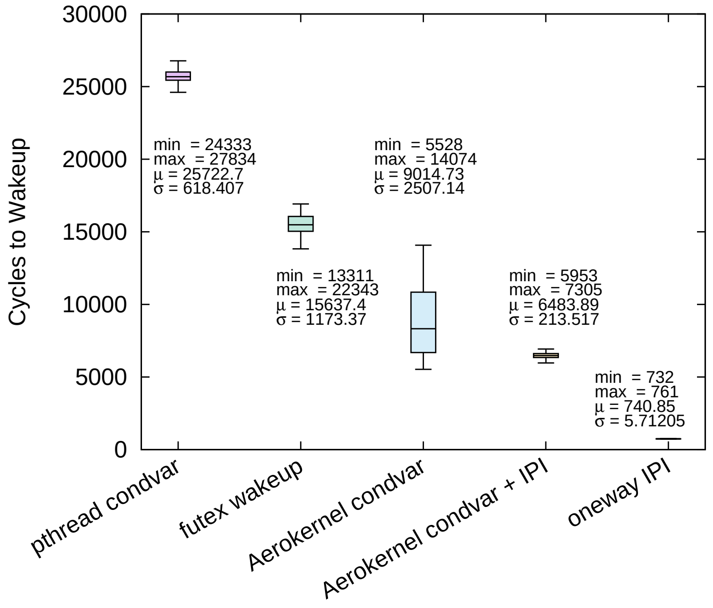
Figure 4.4: Thread context switch latency. Nautilus thread context switches similar in average performance to Linux on x64 and over two times faster on Phi. In both cases, the variance is considerably lower.

lower average latency on the Phi.

The lower average context switch costs on the Phi translate directly into benefits for an HRT developer because it makes it feasible to more finely partition work. On both platforms, the lower variance makes more fine grain cooperation feasible. The default policies described in Section 4.2, combined with the performance characteristics shown here are intended to provide a predictable substrate for HRT development. The HRT developer can also readily override the default scheduling and binding model while still leveraging the fast thread creation/fork and context switch capabilities.



(a) x64



(b) phi

Figure 4.5: Event wakeup latency. Nautilus condition variable wakeup latency is on average five times faster than Linux (pthreads), and has 3–10 times less variation.

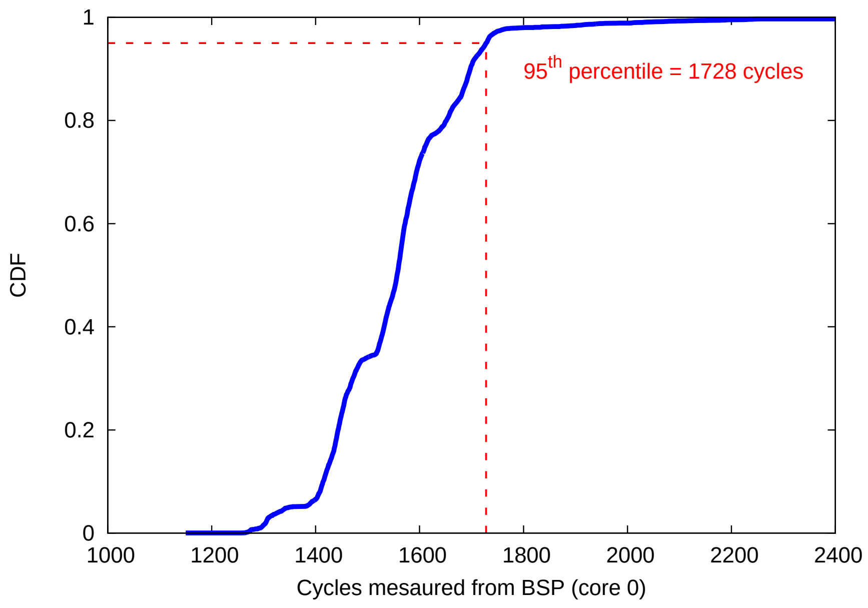
4.4.2 Events

Figure 4.5 compares the event wakeup performance for the mechanisms discussed in Section 4.2 on the two platforms. We measure the latency from when an event is signaled to when the waiting thread executes. We compare the cost of condition variable wakeup in user mode in Linux with our two implementations of them (with and without IPI) in Nautilus. We also show the performance of the Linux fast user space mutex ("futex") primitive, and of a one-way IPI, which is the hardware limit for an event wakeup.

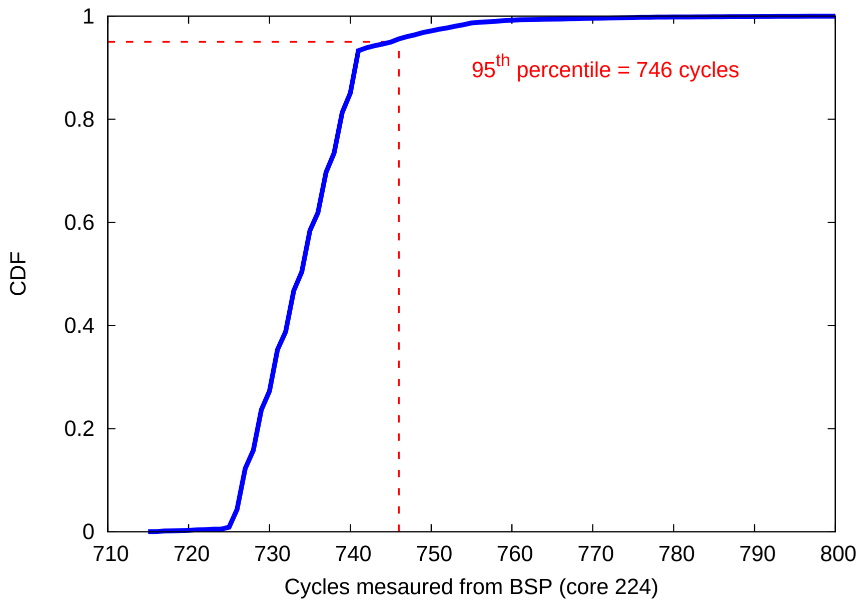
For condition variables, the latency measured is from the call to `pthread_cond_signal` (or equivalent) and the subsequent wakeup from `pthread_cond_wait` (or equivalent). The IPI measurement is the time from when the IPI is initiated until when its interrupt handler on the destination hardware thread has written a memory location being monitored by the source hardware thread.

The average latency for Nautilus's condition variables (with IPI) is five times lower than that of Linux user-level on both platforms. It is also three to five times lower than the futex. Equally important, the variance in this latency is much lower on both platforms, by a factor of three to ten. From an HRT developer's perspective, these performance results mean that much "smaller" events or smaller units of work can feasibly be managed, and that these events and work can be more tightly synchronized in time.

Because they operate in kernel mode, HRTs can make direct use of IPIs and thus operate at the hardware limit of asynchronous event notification, which is one to three thousand cycles on our hardware. Figure 4.6 illustrates the latency of IPIs, as described earlier, on our two platforms. The specific latency depends on which two cores are involved and the machine topology. This is reflected in the notches in the CDF curve. Note however that



(a) x64



(b) phi

Figure 4.6: CDF of IPI one-way latencies, the hardware limit of asynchronous signaling that is available to HRTs.

there is little variation overall—the 5^{th} and 95^{th} percentile are within hundreds of cycles. Asynchronous event notification and IPIs will be explored in detail in Chapter 6.

4.5 Conclusion

In this chapter we saw a detailed description of the Nautilus Aerokernel framework, one of the enabling tools for hybrid runtimes. Nautilus provides a suite of functionality specialized to HRT development that can perform up to two orders of magnitude faster than the general purpose functionality in the Linux kernel while also providing much less variation in performance. Nautilus functionality leads to 20-40% performance gains in an application benchmark for the Legion runtime system on x64 and Xeon Phi, as seen in the previous chapter. In the next chapter, we will see how the hybrid virtual machine, the other main tool for enabling HRTs, eases their deployment.

Chapter 5

Hybrid Virtual Machines

While running an HRT on bare metal is suitable for some contexts (e.g., an accelerator or a node of a supercomputer), we may also want to use an HRT in shared contexts or ease the porting of runtimes that have significant dependencies on an existing kernel. The hybrid virtual machine (HVM), facilitates these use cases. HVM is an extension[1] to the open-source Palacios VMM [129] that makes it possible to create a VM that is internally partitioned between virtual cores that run a ROS and virtual cores that run an HRT. The ROS cores see a subset of the VM's memory and other hardware, while HRT cores see all of it and may be granted specialized physical hardware access by the VMM. The ROS application can invoke functions in the HRT that operate over data in the ROS application. Finally, the HRT cores can be booted independently of the ROS cores using a model that allows an HRT to begin executing in 10s of microseconds, and with which an Aerokernel-based HRT (e.g. an HRT based on Nautilus) can be brought up in 10s of milliseconds. This makes HRT startup in the HVM comparable in cost to `fork()`/`exec()` functionality in the ROS.

[1]Our prototype comprises about 3,500 lines of C and assembly, and we believe it could be readily implemented in other VMMs.

In effect, the HVM allows a portion of an x64 machine to act as an accelerator.

The HVM extensions to Palacios are open-source and publicly available at `http://v3vee.org`.

In this chapter, I first outline the deployment models for HRT and HVM in Section 5.1. Section 5.2 gives the details of the HVM model. Section 5.3 discusses issues related to security and protection in an HRT+HVM environment. Section 5.4 contains a description of the *merged address space* mechanism for leveraging ROS functionality in an HRT. Section 5.5 gives an explanation of communication between a ROS and an HRT in the HVM model. Section 5.6 discusses the boot and reboot process of HRTs within the HVM. Section 5.7 ends the chapter with final thoughts.

5.1 Deployment

We currently envision four deployment models for an HRT:

- *Dedicated*: the machine is dedicated to the HRT, for example on a machine is an accelerator or a node of a supercomputer. This is the model used in the previous chapters.
- *Partitioned*: the machine is a supercomputer node that is physically partitioned [154], with a partition of cores dedicated to the HRT.
- *VM*: The machine's hypervisor creates a VM that runs the HRT.
- *HVM*: The machine's hypervisor creates a VM that is internally partitioned and runs both the HRT and a general purpose OS we call the "regular" OS (ROS).

For the *VM* and *HVM* deployment models, the hypervisor partitions/controls resources, and provides multiprogramming and access control. In the *HVM* model, the HRT can leverage both kernels, as we describe in detail here. Note that in an HVM, the ROS memory is vulnerable to modification by the HRT, but we think of the two as a unit; an unrelated process would be run in a separate VM (or on the host).

The purpose of the hybrid virtual machine (HVM) environment, illustrated in Figure 3.1, is to enable the execution of an HRT in a virtual environment simultaneously with the ROS. That is, a single VM is shared between a ROS and an HRT. The virtual environment exposed to the HRT may be different from and indeed much lower-level than the environment exposed to the ROS. It can also be rebooted independently of the ROS. At the same time, the HRT has access to the memory of the ROS and can interrupt it. In some ways, the HRT can be viewed as providing an "accelerator" for the ROS and its applications, and the HVM provides the functionality of bridging the two. Using the HVM, the performance critical elements of the parallel runtime can be moved into the HRT, while runtime functionality that requires the full stack remains in the ROS.

5.2 Model

The user creates a virtual machine configuration, noting cores, memory, NUMA topology, devices, and their initial mappings to the underlying hardware. An additional configuration block specifies that this VM is to act as an HVM. The configuration contains three elements: (1) a partition of the cores of the VM into two groups: the HRT cores and the ROS cores, which will run their respective kernels; (2) a limit on how much of the VM's physical memory will be visible and accessible from a ROS core; and (3) a Multiboot2-compliant kernel, such as an HRT built on top of Aerokernel. We extend the Multiboot2 specification to support HRTs. The existence of a special header in the Multiboot section of the ELF file indicates that the HRT boot model described here is to be followed instead of the standard Multiboot2 model.

The remainder of the model can be explained by considering the view of a ROS core versus that of an HRT core. The VMM maintains the following invariants for a ROS core:

1. Only the portion of the VM's physical memory designated for ROS use is visible and accessible.
2. Inter-processor interrupts (IPIs) can be sent only to another ROS core.
3. Broadcast, group, lowest-priority, and similar IPI destinations consider only the ROS cores.

An HRT core, on the other hand, has no such constraints. All of the VM's physical memory is visible and accessible. IPIs can be sent to any core, and broadcast, group, lowest-priority and similar IPI destinations can consider either all cores or only HRT cores. This is set by the HRT. Generally speaking, interrupts from interrupt controllers such as IOAPICs and physical interrupts can be delivered to both ROS and HRT cores. The default is that they are delivered only to ROS cores, but an HRT core can request them. Broadcast, group, lowest priority, and similar destinations are computed over the ROS cores and any requesting HRT cores. Collectively, the ROS cores form what appears to the kernel running on them as a smaller machine (in terms of memory and cores) than it actually is.

5.3 Protection

The invariants described above are implemented through two techniques. The memory invariants leverage the fact that each core in the VMM we use, similar to other VMMs, maintains its own paging state, for example shadow or nested page tables. This state is incrementally updated due to exits on the core, most importantly due to page faults or nested page faults. For example, a write to a previously unmapped address causes an exit during which we validate the access and add the relevant page table entries in the shadow or nested age tables if it is an appropriate access. A similar validation occurs for memory-mapped I/O devices. We have extended this model so that validation simply takes into account the type of core (ROS or HRT) on which the exit occurred. This allows us to catch and reject attempts by a ROS core to access illegal guest physical memory addresses.

The interrupt invariants are implemented by extensions to the VMM's APIC device. At the most basic level, IPI delivery from any APIC, IOAPIC, or MSI takes into account the type of core from which the IPI originates (if any) and the type of core that it targets. IPIs from a ROS core to an HRT core are simply dropped. External interrupts targeting an HRT core are delivered only if previously requested. Additionally, the computations involved with broadcast and group destinations for IPIs and external interrupts are modified so that only cores that are prospective targets are included. Similarly, the determination of the appropriate core or cores for a lowest priority destination includes only those cores to which the interrupt could be delivered under the previous restrictions. These mechanisms allow us to reject any attempt to send an HRT core an unwanted interrupt.

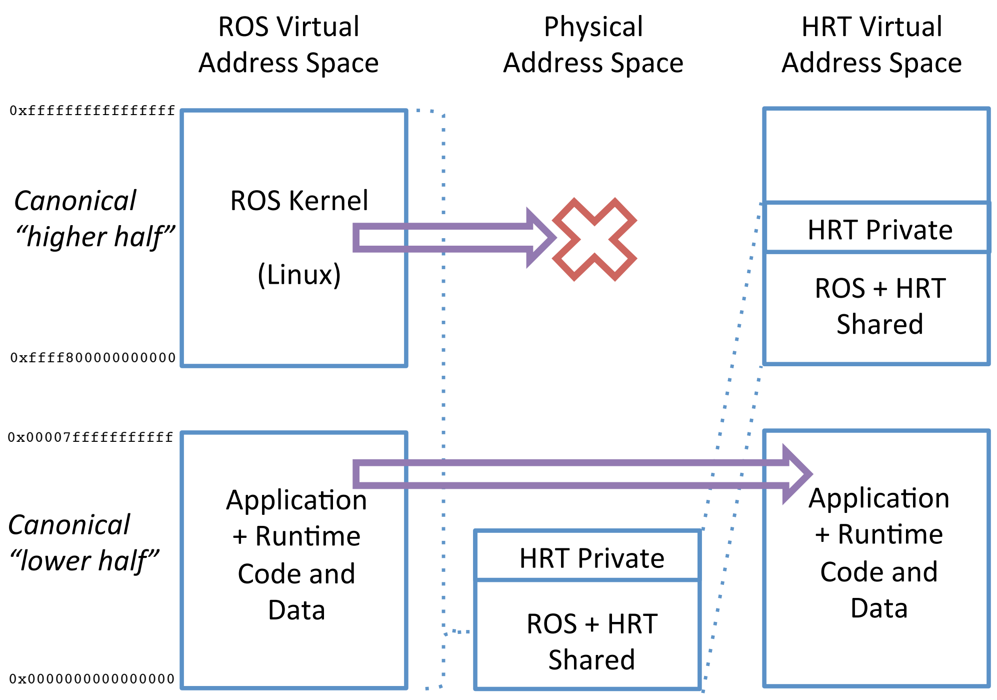


Figure 5.1: Merged address space.

5.4 Merged Address Space

The HRT can access the entire guest physical address space, and thus can operate directly on any data within the ROS. However, to simplify the creation of the legacy path shown in Figure 3.1, we provide the option to merge an address space within the ROS with the address space of the HRT, as is shown in Figure 5.1. When a merged address space is in effect, the HRT can use the same user-mode virtual addresses that are used in the ROS. For example, the parallel runtime in the ROS might load files and construct a pointer-based data structure in memory. It could then invoke a function within its counterpart in the HRT to operate on that data.

To achieve this we leverage the canonical 64-bit address space model of x64 processors, and its wide use within existing ROS kernels, such as Linux. In this model, the virtual address space is split into a "lower half" and a "higher half" with a gap in between, the size of which is implementation dependent. In a typical process model, e.g., Linux, the lower half is used for user addresses and the higher half is used for the kernel.

For an HRT that supports it, the HVM arranges so that the physical address space is identity-mapped into the higher half of the HRT address space. That is, within the HRT, the physical address space mapping (including the portion of the physical address space only the HRT can access) occupies the same portion of the virtual address space that is occupied by the ROS kernel, the higher half. Without a merger, the lower half is unmapped and the HRT runs purely out of the higher half. When a merger is requested, we map the lower half of the ROS's current process's address space into the lower half of the HRT address space. For a Aerokernel-based HRT (like Nautilus), this is done by copying the first 256 entries of the PML4 from the PML4 pointed to by the ROS's CR3 to the HRT's PML4 and then broadcasting a TLB shootdown to all HRT cores.

Because the parallel runtime in the ROS and the HRT are co-developed, the responsibility of assuring that page table mappings exist for lower half addresses used by the HRT in a merged address space is the parallel runtime's. For example, the parallel runtime can pin memory before merging the address spaces, or introduce a protocol to send page faults back to the ROS. The former is not an unreasonable expectation in a high performance environment as we would never expect to be swapping.

5.5 Communication

The HVM model makes it possible for essentially any communication mechanism between the ROS and HRT to be built, and most of these require no specific support in the HVM.

Item	Cycles	Time
Address Space Merger	$\sim$33 K	15 μs
Asynchronous Call	$\sim$25 K	11 μs
Synchronous Call (different socket)	$\sim$1060	482 ns
Synchronous Call (same socket)	$\sim$790	359 ns

Figure 5.2: Round-trip latencies of ROS$\leftrightarrow$HRT interactions (x64).

As a consequence, we minimally defined the *basic* communication between the ROS, HRT, and the VMM using shared physical memory, hypercalls, and interrupts.

The user-level code in the ROS can use hypercalls to sequentially request HRT reboots, address space mergers, and asynchronous sequential or parallel function calls. The VMM handles reboots internally, and forwards the other two requests to the HRT as interrupts. Because additional information may need to be conveyed, a data page is shared between the VMM and the HRT. For a function call request, the page essentially contains a pointer to the function and its arguments at the start and the return code at completion. For an address space merger, the page contains the CR3 of the calling process. The HRT indicates to the VMM when it is finished with the current request via a hypercall.

After an address space merger, the user-level code in the ROS can also use a single hypercall to initiate synchronous operation with the HRT. This hypercall ultimately indicates to the HRT a virtual address which will be used for future synchronization between the HRT and ROS. A simple memory-based protocol can then be used between the two to communicate, for example for the ROS to invoke functions in the HRT, without VMM intervention.

Figure 5.2 shows the measured latency of each of these operations, using Aerokernel as the HRT.

5.6 Boot and Reboot

The ROS cores follow the traditional PC bootstrap model with the exception that the ACPI and MP tables built in memory show only the hardware deemed visible to the ROS by the HVM configuration.

Boot on an HRT core differs from both the ROS boot sequence and from the Multiboot2 specification [155], which we leverage. Multiboot2 for x86 allows for bootstrap of a kernel into 32-bit protected mode on the first core (the BSP) of a machine. Our extension allows for bootstrap of a kernel in full 64-bit mode. There are two elements to HRT boot—memory setup and core bootstrap. These elements combine to allow us to *simultaneously* start all HRT cores immediately at the entry point of the HRT. At the time of this startup, each core is running in long mode (64-bit mode) with paging and interrupt control enabled. The HRT thus does not have much bootstrap to do itself. A special Multiboot tag within the kernel indicates compatibility with this mode of operation and includes requests for how the VMM should set up the kernel environment.

In memory setup, which is done only once in the lifetime of the HRT portion of the VM, we select an HRT-only portion of the guest physical address space and lay out the basic machine data structures needed: an interrupt descriptor table (IDT) along with dummy interrupt and exception handlers, a global descriptor table (GDT), a task state segment (TSS), and a page table hierarchy that identity-maps physical addresses (including the higher-half offset as shown in Figure 5.1, if desired) using the largest feasible page table entries. We also select an initial stack location for each HRT core. A simple ELF loader then copies the HRT ELF into memory at its desired target location. Finally, we build a Multiboot2 information structure in memory. This structure is augmented with headers

Item	Cycles (and exits)	Time
HRT core boot of Aerokernel to `main()`	~135 K (7 exits)	61 μs
Linux `fork()`	~320 K	145 μs
Linux `exec()`	~1 M	476 μs
Linux `fork()` + `exec()`	~1.5 M	714 μs
HRT core boot of Aerokernel to idle thread	~37 M (~2300 exits)	17 ms

Figure 5.3: HRT reboot latencies in context (x64).

that indicate our variant of Multiboot2 is in use, and provide fundamental information about the VM, such as the number of cores, the APIC IDs, interrupt vectoring, and the memory map, including the areas containing the memory setup. Because bootstrap occurs on virtual hardware this information can be much simpler than that supplied via ACPI.

In core bootstrap, which may be done repeatedly over the lifetime of the HRT portion of the HVM, the registers of the core are set. The registers that must be set include the control registers (IDTR, GDTR, LDTR, TR, CR0, CR3, CR4, EFER), the six segment registers including their descriptor components, and the general purpose registers RSP, RBP, RDI, and RAX. The point is that core bootstrap simply involves setting about 20 register values. The instruction pointer (RIP) is set to the entry point of the HRT, while RSP and RBP are set to the initial stack for the core, and RDI points to the multiboot2 header information and RAX contains the multiboot2 cookie.

Unlike a ROS boot, all HRT cores are booted together simultaneously. The HRT is expected to synchronize these internally. In practice this is easy as a core can quickly find its rank by consulting its APIC ID and looking at the APIC ID list given in the extended Multiboot2 information.

Fast HRT reboot Because core bootstrap involves changing a small set of registers and then reentering the guest, the set of HRT cores can be rebooted very quickly. An HRT reboot is also independent of the execution of the ROS, and an HRT can be therefore be rebooted many times over the lifetime of the HVM. We allow an HRT reboot to be initiated from the HRT itself, from a userspace utility running on the host operating system, and via a hypercall from the ROS, as described above.

Figure 5.3 illustrates the costs of rebooting an HRT core, and compares it with the cost of typical process operations on a Linux 2.6.32 kernel running on the same hardware. An HRT core can be booted and execute to the first instruction of Aerokernel's `main()` in ∼50% of the time it takes to do a Linux process `fork()`, ∼13% of the time to do a Linux process `exec()` and ∼8% of the time to do a combined `fork()` and `exec()`. The latter is the closest analog in Linux to what the HRT reboot accomplishes. Note also that timings on Linux were done "hot"—executables were already memory resident.

A complete reboot of Aerokernel on the HRT core to the point where the idle thread is executing takes 17 ms. This time is also blindingly fast compared to the familiar norm of booting a physical or virtual machine. We anticipate that this time will further improve for two reasons. First, we can in principle skip much of the general purpose startup code in Aerokernel, which is currently executed, given that we know exactly what the virtual hardware looks like. Second, by starting the core from a memory and register snapshot, specifically at the point of execution we desire to start from, we should be able to even further short-circuit startup code.

It is important to note that even at 17 ms, a complete Aerokernel reboot is 60 to 300 times faster than a typical 1-5 minute node or server boot time. It should be thought of in those terms, similar to the MicroReboot concept [45] for cheap recovery from software failures. We can use HRT reboots to address many issues and, in the limit, treat them as being on par with process creation in a traditional OS.

5.7 Conclusion

In this chapter I introduced the hybrid virtual machine (HVM). HVM is VMM functionality that allows us to simultaneously run two kernels, an HRT and a traditional kernel, within the same VM, allowing a runtime to benefit from the performance and capabilities provided by the HRT model while not losing the performance non-critical functionality of the traditional kernel. We will see more on HVM in Chapter 7, where it is used to support an *automatically hybridized* runtime. In the next chapter, I will introduce current limits on asynchronous software events and ways to overcome these limits using the HRT model.

Chapter 6

Nemo Event System

Many runtimes leverage event-based primitives as an out-of-band notification mechanism that can signal events ranging from task completions or arrivals to message deliveries or changes in state. They may occur between logical entities like processes or threads, or they may happen at the hardware level. They often provide a foundation for building low-level synchronization primitives like mutexes and wait queues. The correct operation of parallel programs written for the shared-memory model relies crucially on low-latency, microarchitectural event notifications traversing the CPU's cache coherence network. The focus of this chapter is on asynchronous software events, namely events that a sending thread or handler can trigger without blocking or polling, and for which the receiving thread or handler can wait without polling.

Ultimately these events are just an instance of unidirectional, asynchronous communication, so one might expect little room for performance improvement. We find, however, that the opposite is true. While a cache line invalidation can occur in a handful of CPU cycles and an inter-processor interrupt (IPI) can reach the opposite edge of a many-core chip in less than one thousand cycles, commonly used event signaling mechanisms like user-level condition variables fail to come within even three orders of magnitude of that mark.

The crux of the work in this chapter is to determine the hardware limits for asynchronous event notification on today's hardware, particularly on x64 NUMA machines and the Intel Xeon Phi, and then to approach those limits with software abstractions implemented in an environment uninhibited by an underlying kernel, namely an HRT environment.

In the limit, an asynchronous event notification is bounded from below by the signaling latency on a hardware line. We measure and analyze inter-processor interrupts (IPIs) on our hardware, arguing that they serve as a first approximation for this lower bound. We consider both unicast and broadcast event notifications, which are used in extant runtimes, and have IPI equivalents. In this chapter, I will describe the design and implementation of *Nemo*[1], a system for asynchronous event notifications in HRTs that builds on IPIs. Nemo presents abstractions to the runtime developer that are identical to the pthreads condition variable unicast and broadcast mechanisms and thus are friendly to use, but are much faster. Unlike IPIs, where a thread invokes an interrupt handler on a remote core, these abstractions allow a thread to wake another thread on a remote core. In addition, Nemo provides an interface for unconventional event notification mechanisms that operate near the IPI limit.

As I will show through a range of microbenchmarking on both platforms, Nemo can approach the hardware limit imposed by IPIs for the average latency and variance of latency for asynchronous event notifications. Unicast notifications in Nemo enjoy up to five times lower average latency than the user-level pthreads constructs and the Linux

[1]Nemo is available as an open-source extension of the Nautilus Aerokernel framework, which was discussed in detail in Chapter 4.

futex construct, while broadcast notifications have up to 4,000 times lower average latency. Furthermore, the variance seen in Nemo is up to an order of magnitude lower for unicast, and many orders of magnitude lower for broadcast. Finally, Nemo can deliver broadcast events with much higher synchrony, exhibiting nearly identical latency to all destinations.

I will then discuss a small hardware change that would reduce the hardware limit (and Nemo's latency). Our measurements suggest that a large portion of IPI cost stems from the interrupt dispatch mechanism. The `syscall`/`sysret` instructions avoid similar costs for hardware thread-local system calls by avoiding this dispatch overhead. I will discuss our proposal that `syscall` be included as an IPI type. When receiving a "remote `syscall`" IPI, the faster dispatch mechanism would be used, reducing IPI costs for specific asynchronous events such as those in Nemo.

Section 6.1 gives an overview of asynchronous software events, their usage, and their current limitations. Section 6.2 introduces the Nemo event system. Section 6.3 discusses ways in which we can improve the hardware limit for asynchronous events. Finally, Section 6.4 concludes this chapter.

6.1 Limits of Event Notifications

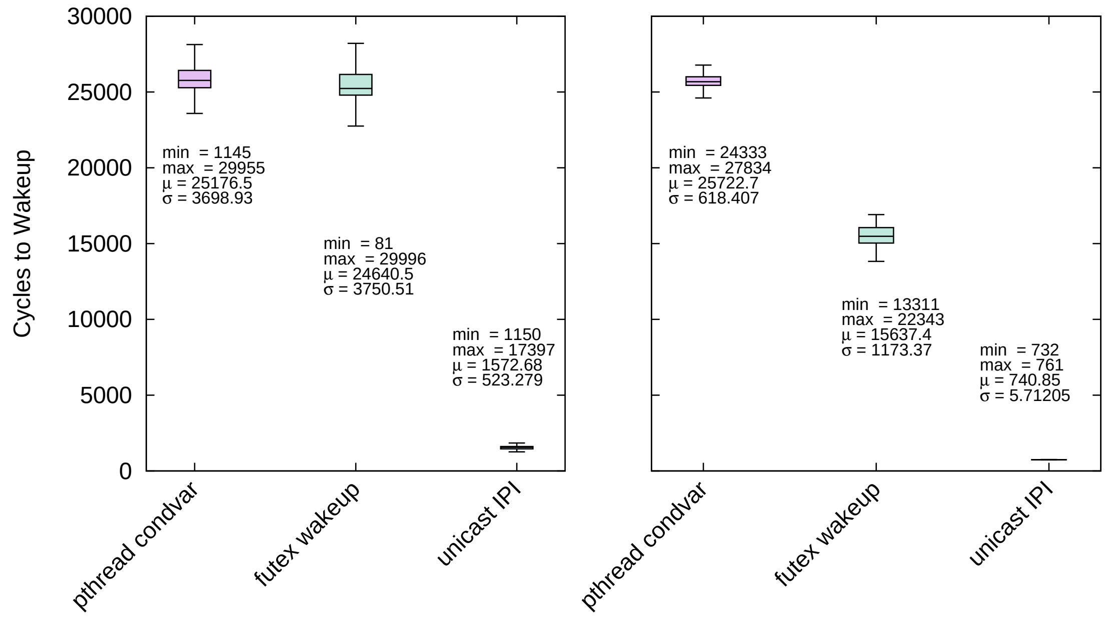
Our motivation in exploring asynchronous event notifications in the HRT model stems from the observation that many parallel runtimes use expensive, user-level software events even though modern hardware already includes mechanisms for low-latency event communication. However, these hardware capabilities are traditionally reserved for kernel-only use. I discuss common uses of event notification mechanisms, particularly for task invocations in parallel runtimes, then present measurements on modern multicore machines for common event-based primitives, demonstrating potential benefits of a kernel-mode environment for low-latency events. The core question for this section is just how fast *could* asynchronous event notification in current x64 and Phi hardware go. We expect that our findings could also apply to asynchronous events in other runtime environments that expose privileged hardware features or forego traditional privilege separation such as Dune [25], IX [26], and Arrakis [160].

6.1.1 Runtime Events

In one common usage pattern of asynchronous event notifications, a signaling thread notifies one or more waiting (and not polling) threads that they should continue. The signaling thread continues executing regardless of the status of waiting threads.

In examining the usage of event notifications, we worked with Charm++[114], SWARM [131], and Legion [21, 189], all examples of modern parallel runtimes. They all use asynchronous events in some way, whether explicitly through an event programming interface or implicitly by runtime design. In many cases, these runtimes use events as vehicles to notify remote workers of available work or tasks that should execute.

Legion provides a good example. It uses an execution model in which a thread (e.g., a pthread) implements a logical processor. Each logical processor sequentially operates over tasks. In order to notify remote logical processors of tasks ready to execute, the signaling processor broadcasts on a condition variable (e.g., a `pthread_cond_t`) that wakes up any idle logical processors, all of which race to claim the task for execution. This process bears some similarity to the `schedule()` interrupts used in Linux at the kernel level. Since `pthread_cond_broadcast()` must involve the kernel scheduler (via a system call), it is fairly expensive, as we will show in Section 6.1.2. Linux's futex abstraction attempts to



(a) x64 (b) phi

Figure 6.1: Comparing existing unicast event wakeups in user-mode (pthreads and futexes on Linux) with IPIs on *x64* and *phi*. Unicast IPIs are at least an order of magnitude faster and exhibit much less variation in performance on both platforms.

ameliorate this cost with mixed success.

6.1.2 Microbenchmarks

Section 4.4 gave a cursory examination of the performance of asynchronous software events. This section will present a more detailed exploration of event performance. The experimental setup is the same as that discussed in Section 3.3.1. Time measurement is with the cycle counter and measurements are taken over at least 1000 runs (unless otherwise noted) with results shown as box plots or CDFs and summary statistics overlaid in some cases.

Figure 6.1 shows the latency for event wakeups on *x64* and *phi*. In each of these experiments, we create a thread on a remote core. This thread goes to sleep until it receives an event notification. We measure the time from the instant before the signalling thread sends its notification to when the remote thread wakes up. We map threads to distinct cores. The numbers represent statistics computed over 100 trials for each remote core (6,300 trials on *x64*, 22,700 on *phi*).

I include a comparison of three mechanisms. The first two are the most commonly used asynchronous event mechanisms in user-space: condition variables and futexes. The pthreads implementation of condition variables depicted builds on top of futexes. The overhead of condition variables compared to futexes may be significant, but it is platform dependent or implementation dependent—the average event wakeup latency of condition variables is nearly double that of futexes on *phi*, but only a small increment more on *x64*.

The third mechanism, denoted with “unicast IPI” on the figure, shows the unicast latency of an inter-processor interrupt (IPI). On *x64* and *phi*, each hardware thread has an associated interrupt controller (an APIC). The APIC commonly *receives* external interrupts and initiates their delivery to the hardware thread, but it is also exposed as a memory-mapped I/O device to the hardware thread. From this interface, the hardware thread can program the APIC to *send* an interrupt (an IPI) to one or more APICs in the system. APICs are privileged devices and are typically used only by the kernel.

We also considered events triggered using the MONITOR/MWAIT pair of instructions present on modern AMD and Intel chips. These instructions allow a hardware thread to wait on a write to a particular range of memory, potentially entering a low-energy sleep state while waiting. Because the entire hardware thread is essentially blocked when executing the MWAIT instruction, I did not include a comparison with this technique.

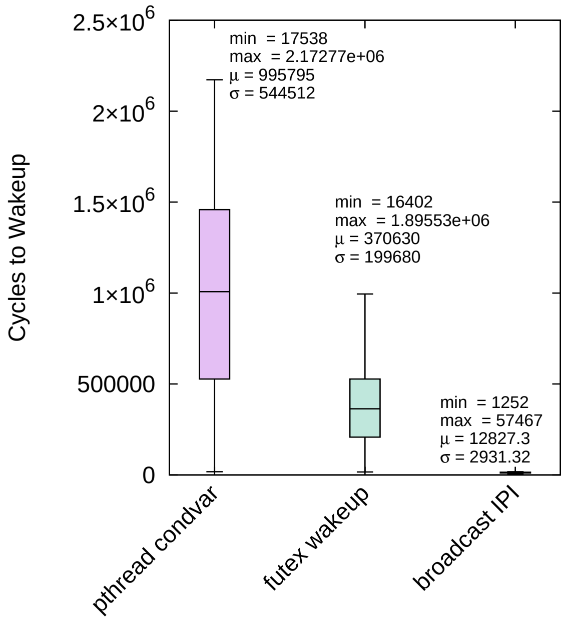
Figure 6.1 shows that IPIs are, on average, much faster than either condition variables or futexes. On *x64*, they have roughly 16 times lower latency than either, while on *phi*, they have roughly 32 times lower latency than condition variables and 16 times lower latency than futexes. On *phi*, the average IPI latency is only 700 cycles. The wall-clock time on the two machines is similar, as *x64* has roughly twice the clock rate.

IPIs are *not* doing the same thing as a condition variable or a futex. For IPIs, we measure the time from the instant before the signaling thread sends its notification to when the *interrupt handler* begins executing, not the waiting thread. We measure the IPI time because this latency represents a lower bound for a wakeup mechanism using existing hardware functionality on commodity machines. There is significant room for an improvement of more than an order of magnitude ($\sim$20x). We attempt to achieve this improvement in Section 6.2 by moving towards a *purely* asynchronous mechanism enabled by the HRT environment.

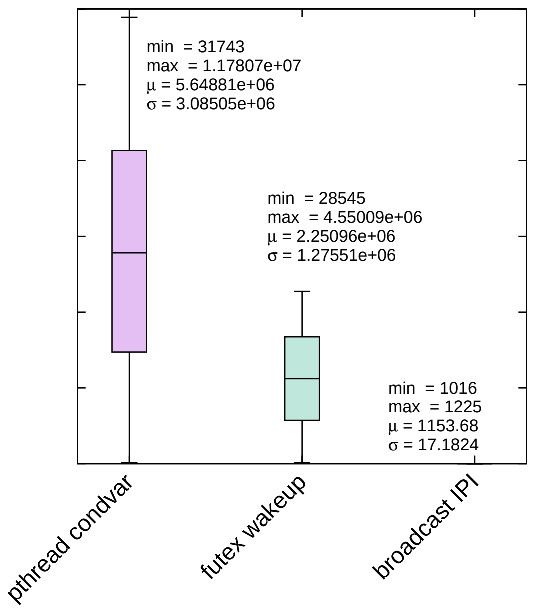
Not only is the average time much lower for an IPI, but its variance is also diminished considerably. As I noted in the introduction, variance in performance limits parallel runtime performance and scalability. This is an OS noise problem. The hardware has fewer barriers to predictable performance.

Broadcast events are of significant interest in parallel runtimes, for example in Legion as described above. Figure 6.2 shows the wakeup latency for a broadcast event on *x64* and *phi*, again comparing a condition variable based approach in pthreads, Linux futex, and IPIs. Measurements here operate as with the unicast events, but we additionally keep track of time of delivery on every destination so we can assess the synchronicity of the broadcasts.

The relative latency improvements for broadcasts with condition variables and futexes are similar to the unicast case, but the gain from using broadcast IPIs is much larger. On



(a) x64



(b) phi

Figure 6.2: Comparing existing user-space event broadcasts vs. IPIs on *x64* and *phi*. Broadcast IPIs have over 4000 times lower latency than condition variables and almost 2000 times lower latency than futexes on *phi*. *x64* shows 78 times lower latency than condition variables and 30 times lower latency than futexes. Variance in latency is similarly reduced.

phi, the average latency of a broadcast IPI received by all targets is over 4,000 times lower than for a mechanism based on condition variables. The gain in variance is similarly startling. On *x64*, this gain is 78 times. While broadcast IPIs exploit the hardware's own parallelism, the implementations of all the other techniques are essentially serialized in a loop that wakes up waiting threads sequentially. In part this difference between *phi* and *x64* is simply that the Phi has almost four times as many cores. While one could argue that a programmer should choose a barrier over a condition variable to signal a wakeup on multiple cores, barriers lack the asynchrony needed for these kinds of event notifications.

We should also hope that a broadcast event causes wakeups to occur across cores with

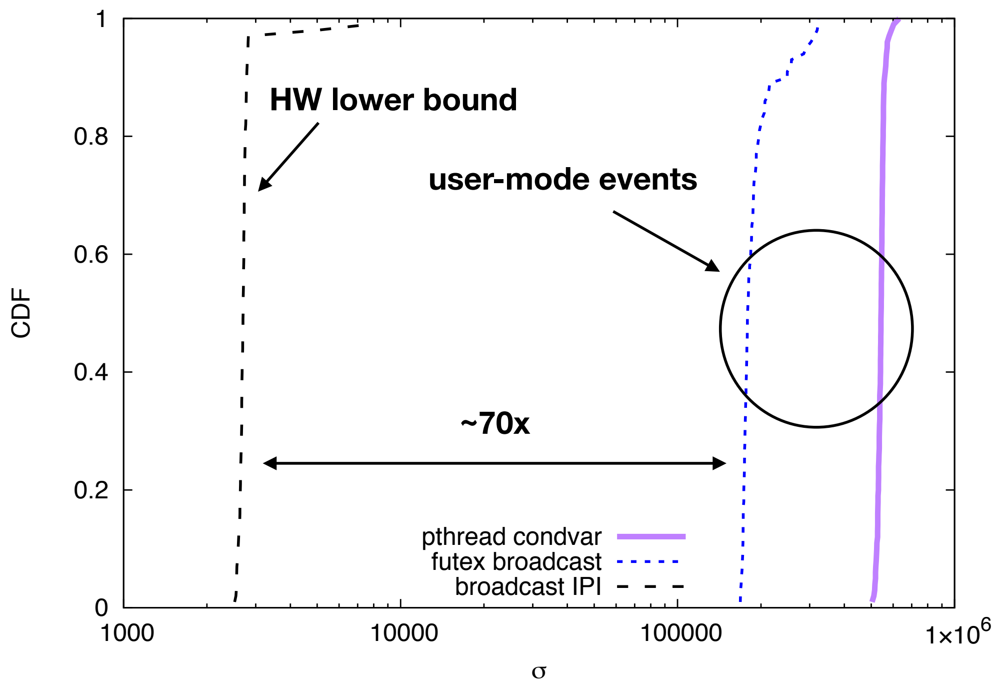


Figure 6.3: CDF comparing the σs for various broadcast (one-to-all) wakeup mechanisms in user-space vs. IPIs on *x64*. Broadcast IPIs achieve a synchrony that is three orders of magnitude better than that achieved by pthread condition variables or Linux futexes.

synchrony; when a broadcast event is signaled, we would like all recipients to awaken as close to simultaneously as possible. However, Figures 6.3 and 6.4 show that this is clearly not the case for the condition variable or futex-based broadcasts. Recall that we measure the time of the wakeup on each destination. For one broadcast wakeup, we thus have as many measurements as there are cores, and we can compute the standard deviation (σ) among them. In these figures, we repeat this many times and plot the CDFs of these σ estimates. Note that in the figures the x-axes are on a log scale. On these platforms, there are orders of magnitude difference in the degree of synchrony in wakeups achievable on the hardware and what is actually achieved by the user-space mechanisms.

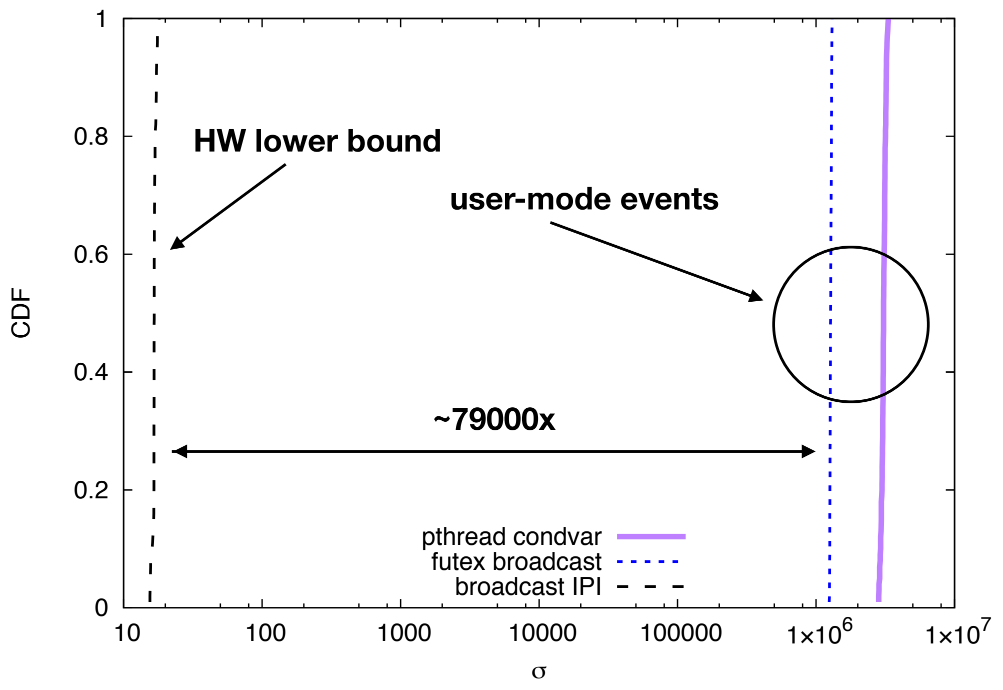


Figure 6.4: CDF comparing the σs for various broadcast (one-to-all) wakeup mechanisms in user-space vs. IPIs on *phi*. Broadcast IPIs achieve a synchrony that is five orders of magnitude better than that achieved by pthread condition variables or Linux futexes.

6.1.3 Discussion

The large gap between the performance of asynchronous software events in user-mode and the hardware capabilities should cause concern for runtime developers. Not only do these latencies indicate that software wakeups may happen roughly on the millisecond time-scale of a slow network packet delivery, but also that the programmer can do little to ensure that these wakeups occur with *predictable* performance. The problem is worse for broadcast events, and *the problem* appears to scale with increasing core count.

Recall again that we claim IPIs are a hardware limit to asynchronous event notifications, and that it is important to understand that an IPI is not an event notification by itself. The goal of Nemo is to achieve event notifications compatible with those expected by parallel

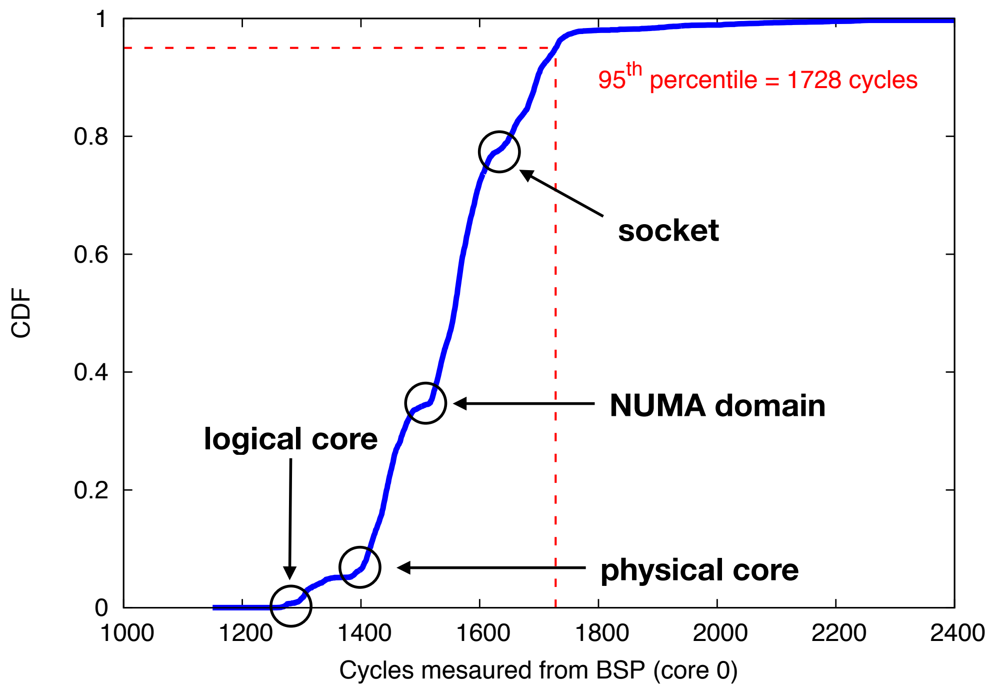


Figure 6.5: CDF of unicast IPI latency from the Bootstrap Processor (BSP) to all other cores on *x64*. Approaching this profile is the goal of Nemo.

runtimes with performance that approaches that of IPIs, as well as to offer unconventional mechanisms that trade off ease of use for performance near the IPI limit.

6.2 Nemo Events

Nemo is an asynchronous event notification system for HRTs built within the Nautilus Aerokernel framework. Nemo addresses the performance issues of asynchronous user-space notifications by leveraging hardware features not commonly available to runtime or application developers. That is, they are enabled by the fact that the entire HRT runs in kernel mode.

The goal of Nemo is to approach the hardware IPI latency profile. Figure 6.5 represents

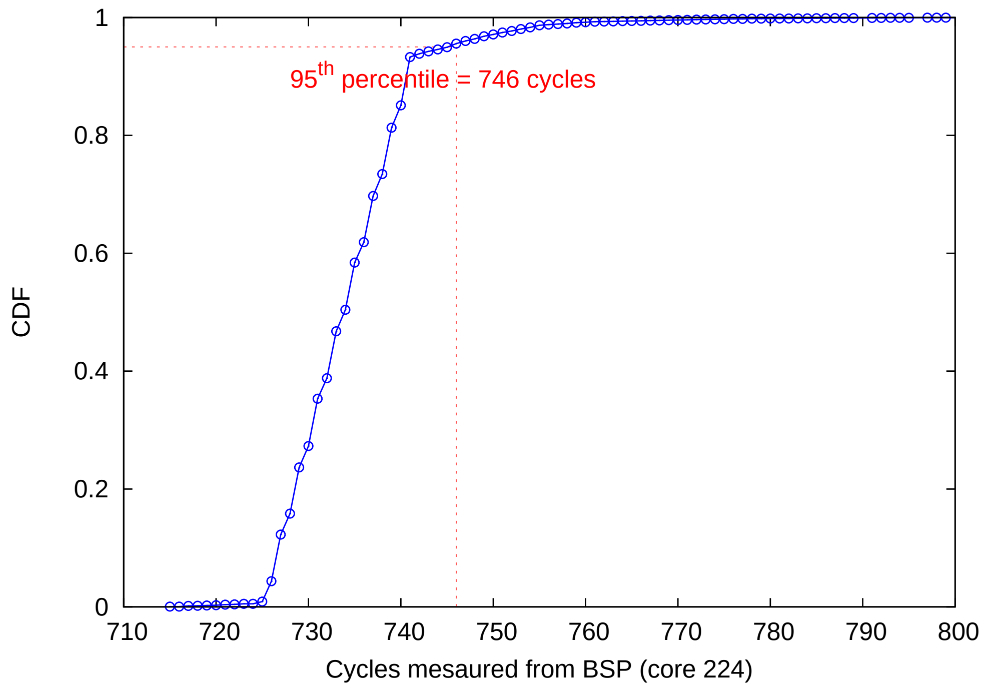


Figure 6.6: CDF of unicast IPI latency from the Bootstrap Processor (BSP) to all other cores on *phi*.

in detail the kind of profile we would like to achieve. We expect that these numbers, which were measured on *x64*, will tell us something about the machine, given its complex organization. The knees in the curve (marked with black circles) indicate boundaries in the IPI network. While we could not find reliable documentation from AMD or other parties on the topology of the IPI network on this machine, we are confident that these inflection points correspond to distances within the chip hierarchy as indicated in the captions. As Nemo begins to exhibit similar behavior, we will know we are near the limits of available hardware.

Unicast IPI latencies on our *phi* card, shown in Figure 6.6, are smaller and show less pronounced inflection points. We suspect this stems from its use of a single-chip processor with a balanced interconnect joining the cores.

6.2.1 Kernel-mode Condition Variables

Existing runtimes, such as Legion, use pthreads features in their user-space incarnations. Nautilus tries to simplify the porting of such runtimes to become HRTs. To support thread creation, binding, context switching, and similar elements, Nautilus provides a pthreads-like interface for its kernel threads. Default thread scheduling (round-robin with or without preemption—preemption is not used here) and mapping policies (initial placement by creator, no migration) are intended to be simple to reason about. Similarly, memory allocation is NUMA-aware and based on the calling thread's location, not by first touch.

For Nemo, the relevant event mechanism in pthreads is the condition variable, implemented in the `pthread_cond_X()` family of functions. Nemo implements a compatible set of these functions within Aerokernel. There are two implementations. In the first, there is no special interaction with the scheduler. When a waiting thread goes to sleep on a condition variable, it puts itself on the condition variable's queue and deschedules itself. When a signaling thread invokes `condvar_signal()`, this function will put the waiting thread back on the appropriate processor's ready queue. The now signaled thread will not run until the processor's background thread `yield()`s. We would expect this implementation to increase performance simply by eliminating user/kernel transitions from system calls, e.g. the `futex()` system call.

The second implementation uses a more sophisticated interaction with the scheduler in order to better support common uses in runtimes like Legion and SWARM. In these, the threads that are sleeping on condition variables are essentially logical processors. Ideally each one would map to a single physical CPU and would not compete for resources on that

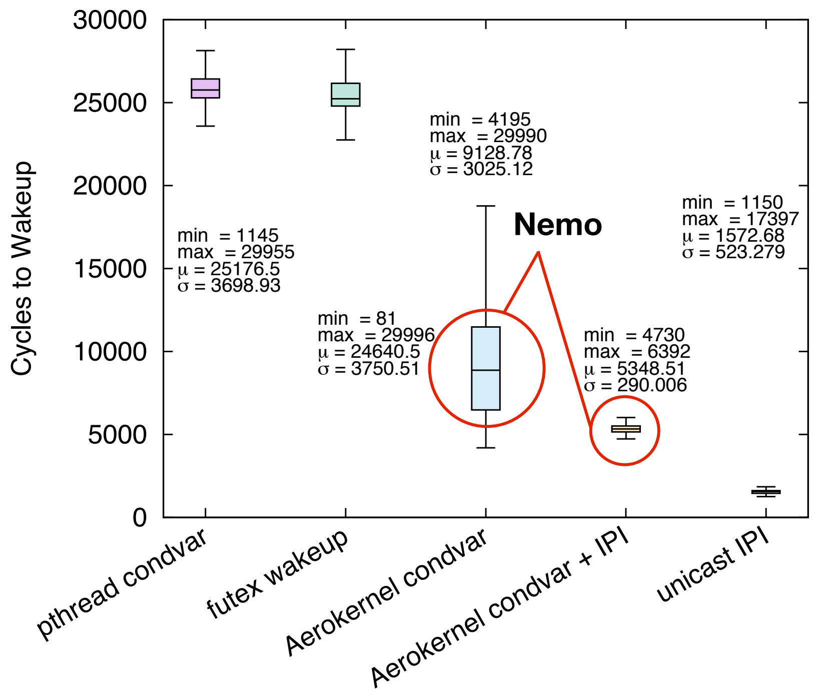


Figure 6.7: Nemo kernel-mode event mechanisms for single wakeups on *x64*. Average latency is reduced by over a factor of four, and variation is considerably reduced.

CPU. Scheduling of tasks (Legion tasks or SWARM tasks, not kernel threads) are handled by the runtime, so kernel-level scheduling is superfluous. The condition variable in such systems is used essentially to awaken logical processors.

On a `condvar_signal()` our second implementation sends an IPI to "kick" the physical processor of the newly runnable thread. The scheduler on the physical processor can then immediately switch to it. The kick serves to synchronize the scheduling of the sleeping thread, reducing the effects of background threads that may be running.

The two implementations comprise about 200 lines of code within the Nautilus Aerokernel framework.

Figures 6.7 and 6.8 show the performance of these two implementations compared to the existing user-space techniques and to the unicast IPI. Our first implementation

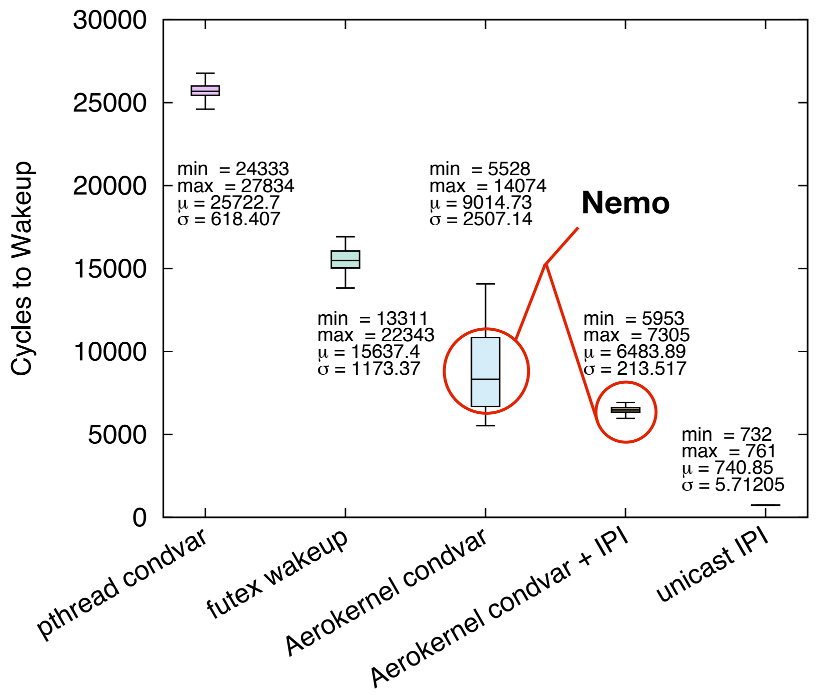


Figure 6.8: Nemo kernel-mode event mechanisms for single wakeups on *phi*. Average latency is reduced by over a factor of four and variation is considerably reduced.

("Aerokernel condvar") roughly halves the median latency of user-mode event wakeups on both *x64* and *phi*. This latency improvement represents a rough estimate of the speedup achieved solely by moving the application/runtime into kernel-mode, thus avoiding kernel/user transition overheads. The implementation does, however, exhibit considerable variance in wakeup latency. This is because the wakeup time depends on how long it takes for the CPU's background thread to `yield()` again.

Our second implementation ("Aerokernel condvar + IPI") ameliorates this variation, and further reduces average and median latency. The use of the IPI kick collapses the median latency of the wakeup down to the minimum latency of the standard kernel-mode condition variable. The variance in this case is much lower than all of the other wakeup mechanisms.

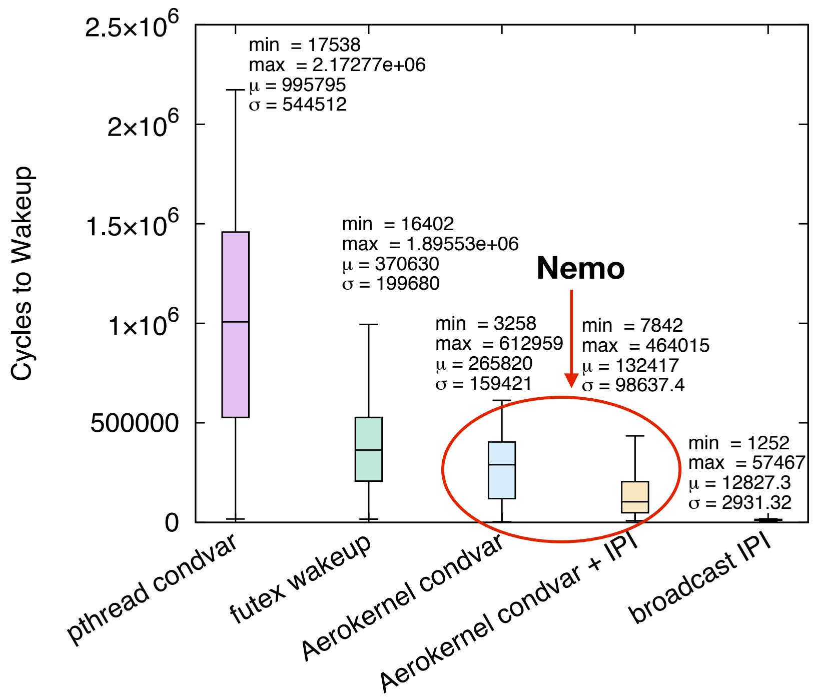


Figure 6.9: Nemo kernel-mode event mechanisms for broadcast wakeups on *x64*. Average latency is reduced by a factor of 10 and variation is considerably reduced.

Figures 6.9 and 6.10 show the performance of broadcast events, where the gain is larger (a factor of 10–16). Figures 6.11 and 6.12 show the improvement of the synchrony of broadcast event wakeups. This is improved by a factor of 10 on both platforms. Section 6.1.2 gives a description of the format of the latter two figures and a discussion of broadcast IPIs.

In the current Nemo implementations for broadcast events, the signaling thread moves each waiting thread to its processor's run queue and then (in the second implementation) kicks that processor with an IPI. Although there is considerable overlap between context switches, in-flight IPIs, and moving the next thread to its run queue, we expect that this sequential behavior of the signaling thread is a current limit on the broadcast event mechanism both in terms of average/median latency and in terms of synchrony of the

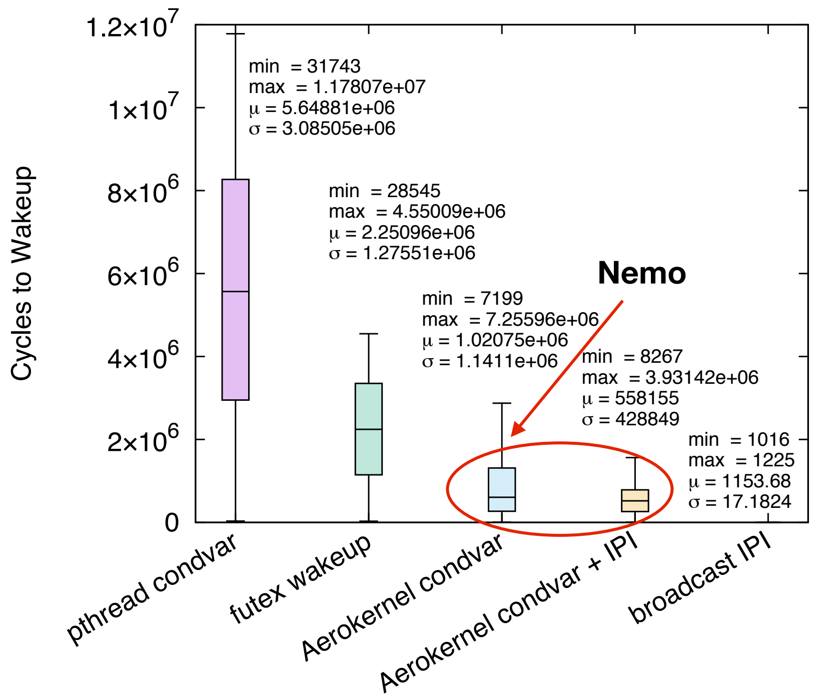


Figure 6.10: Nemo kernel-mode event mechanisms for broadcast wakeups on *phi*. Average latency is reduced by an factor of 16 and variation is considerably reduced.

awakened threads. This is in contrast to the IPI broadcast in hardware, which is inherently parallel and exhibits significant synchrony in arrivals, as indicated in the figures and previous discussions.

6.2.2 IPIs and Active Messages

In the previous section, I introduced Nemo events which were built to conform to the pthreads programming interface, particularly condition variables. With the inherent limitations of this interface and the privileged hardware available to us in an HRT in mind, we can now explore a new event mechanism with a different interface built directly on top of IPIs. The mechanism is also informed by how pthreads condition variables are actually used in Legion and SWARM, namely to *indirectly* implement behavior via

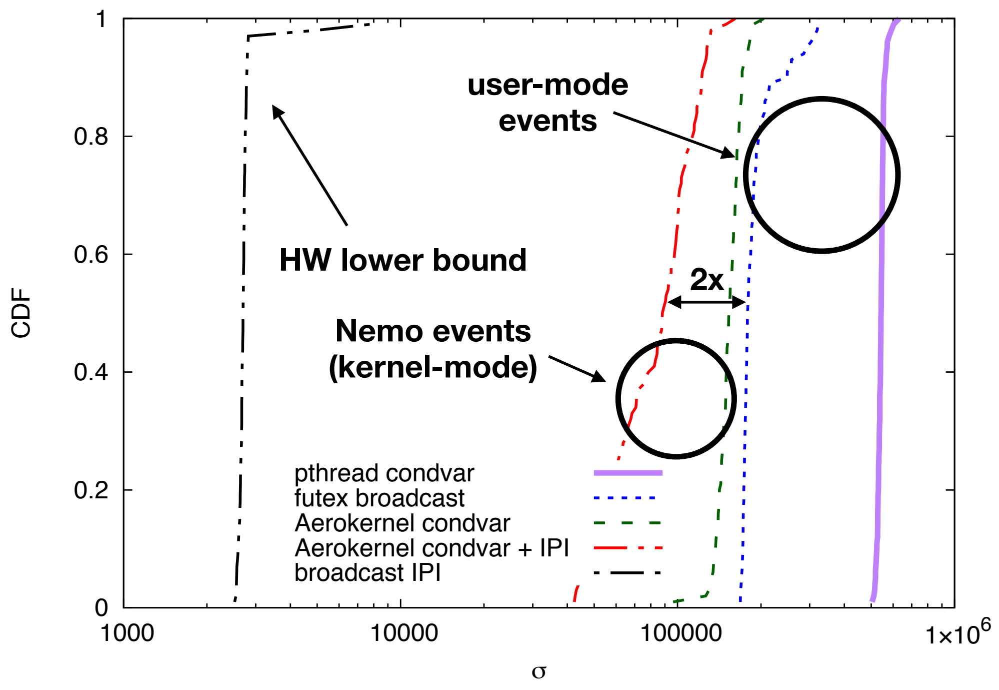


Figure 6.11: CDF showing the Nemo kernel-mode event mechanisms for broadcast wake-ups on *x64*. Nemo achieves an order of magnitude better synchrony in thread wakeups.

user-level mechanisms that can be *directly* implemented in the kernel context.

We claim that Active Messages [191] would better match the functional behavior that event-based runtimes need. Active Messages enable low-latency message handling for distributed memory supercomputers with high-performance interconnects. Since a message delivery is ultimately just one kind of asynchronous event, we looked to Active Messages for inspiration on how to approach the hardware limit for asynchronous software events. In short, we use the IPI as the basis for an Active Message model within the shared memory node.

In an Active Message system, the message payload includes a reference (a pointer) to a piece of code on the destination that should handle the message receipt. One advantage of this model is that it reduces the load on the kernel and results in a faster delivery

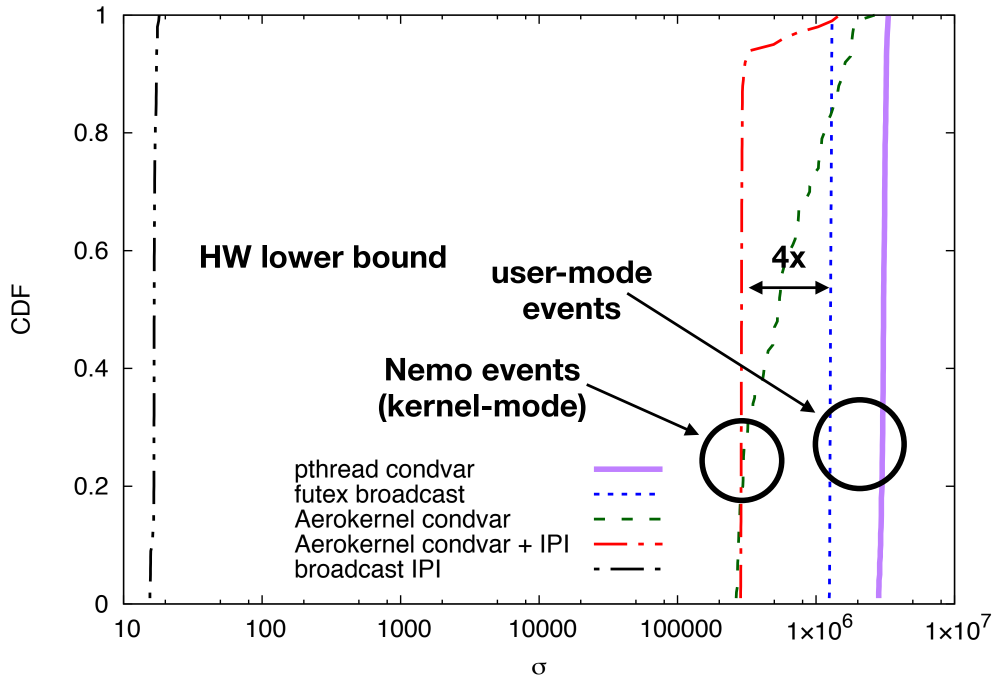


Figure 6.12: CDF showing the Nemo kernel-mode event mechanisms for broadcast wake-ups on *phi*.

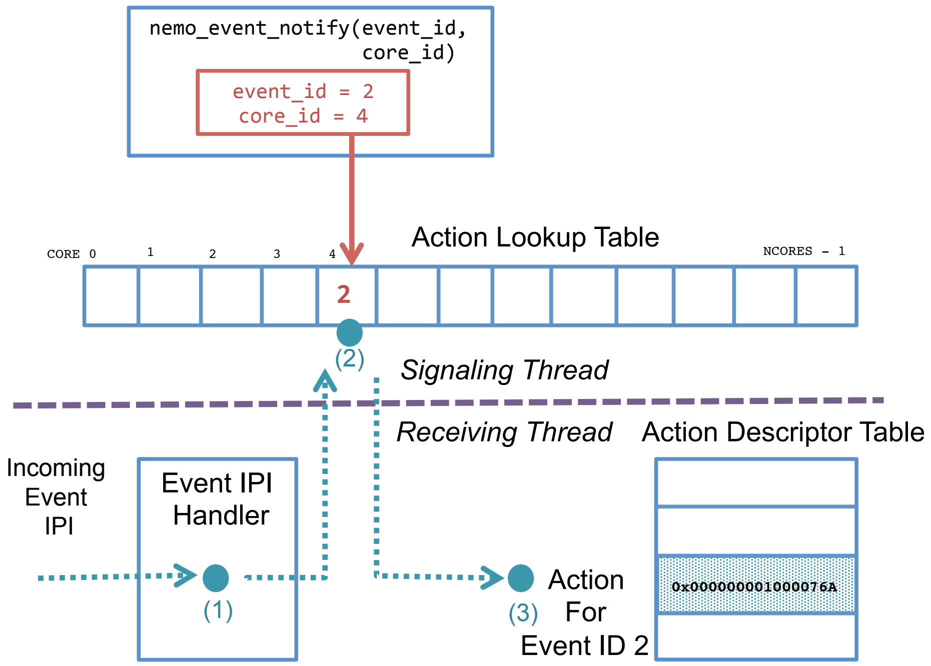


Figure 6.13: Nemo Active Message-inspired event wakeups implemented using IPIs.

to the user-space application. Since the HRT *is* the kernel, we do not need to avoid transferring control to it on an event notification. Furthermore, since the HRT is not a multi-programmed environment, we can be sure that the receiving thread is a participant in the parallel runtime/application, and thus has the high-level information necessary to process the event. We can eliminate handling overhead by leveraging existing logic in the hardware already meant for handling asynchronous events—in this case, IPIs. IPIs by themselves, however, cannot implement a complete Active Message substrate, as there is no payload other than the interrupt vector and state pushed on the stack by hardware.

Figure 6.13 shows the design and control flow of our Active Message-inspired event mechanism. We reserve a slot in the IDT for a special Nemo event interrupt, which will vector to a common handler (1). If only one type of event is necessary, this handler will be the final handler and thus no more overhead is incurred. However, it is likely that a runtime developer will need to use more than one event. In this case, the common handler will lookup an event *action* (a second-level handler) in an *Action Lookup Table* (ALT), which is indexed by its core ID (2). From this table, we find an *action ID*, which serves as an index into a second table called the *Action Descriptor Table* (ADT). The ADT holds actions that correspond to events. After the top-level handler indexes this table, it then executes the final handler (3). The IPI is used to deliver the active message, while the Action Table effectively contains its content.

The mechanism described here comprises about 160 lines of code in Nautilus.

Figures 6.14 and 6.15 show CDFs of the latency of Nemo's Active Message-inspired events compared to the unicast IPI. Notice that in all cases Nemo events are only roughly 40 cycles slower on *phi* and 100 cycles slower on *x64*. We are now truly close to the capabilities of the hardware as evidenced by the performance and by the observed sensitivity to

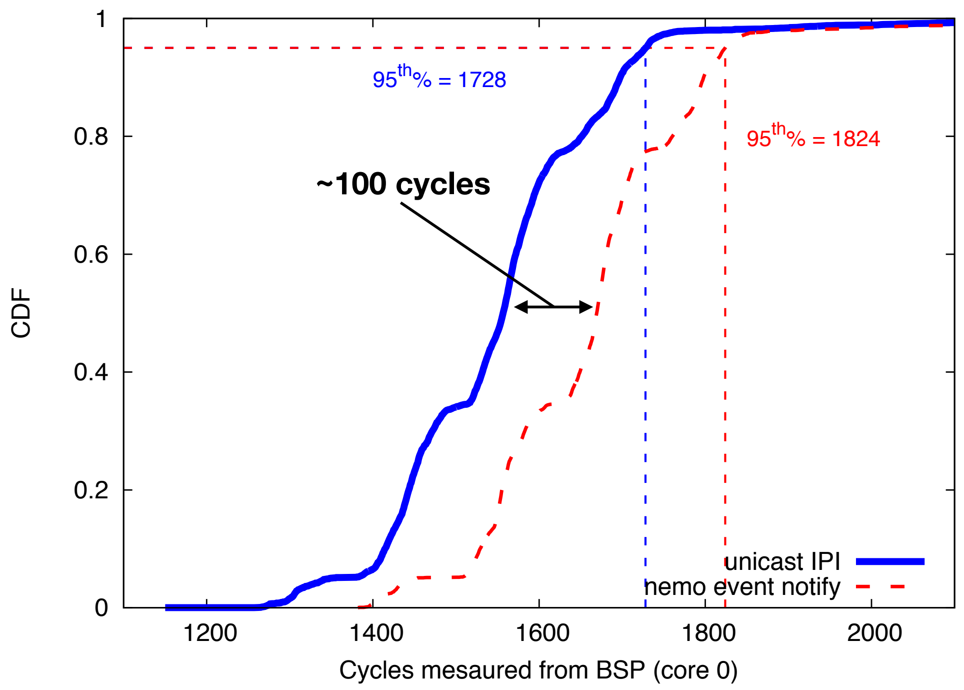


Figure 6.14: CDF showing unicast latency of Nemo's Active Message-inspired events compared to unicast IPIs on *x64*. Nemo lies within ∼5% of IPI performance.

the hardware topology, which is implied by the knees in the IPI latency profile (e.g. in Figure 6.5).

Figure 6.16 shows the latency of Active Message-inspired Nemo events compared to broadcast IPIs. The performance of the Nemo events are within tens of cycles of broadcast IPIs, as we would expect. Figures 6.17 and 6.18 show the amount of synchrony present in the Nemo events. I give an explanation of this type of figure in Section 6.1.2.

6.3 Towards Improving the Hardware Limit

Although I have shown how we achieve a marked improvement for asynchronous software events using hardware features (IPIs in particular) that are not commonly available to

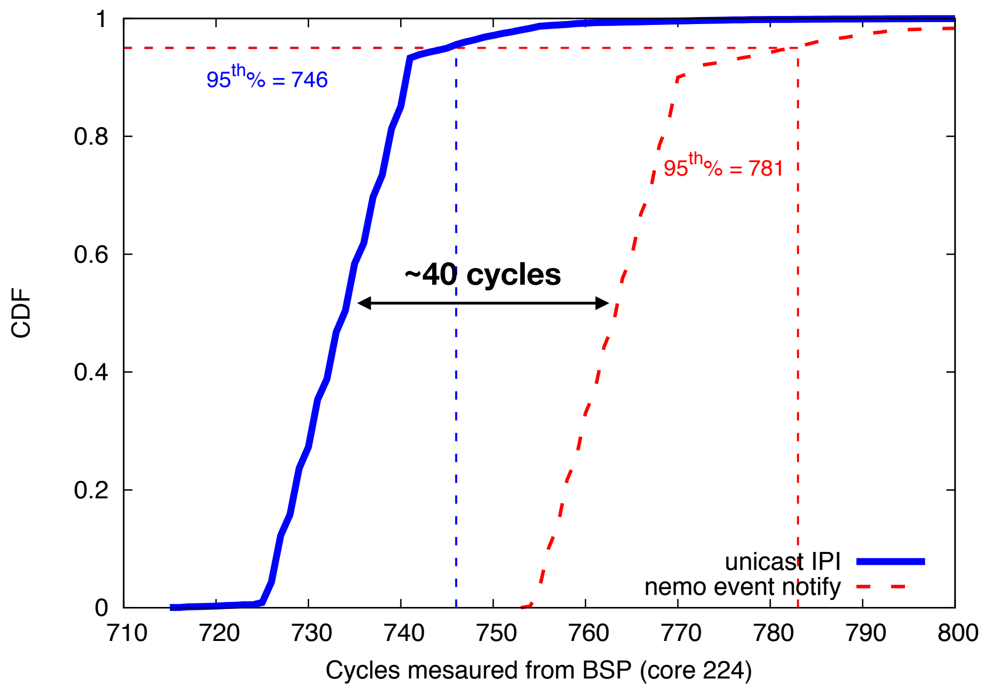


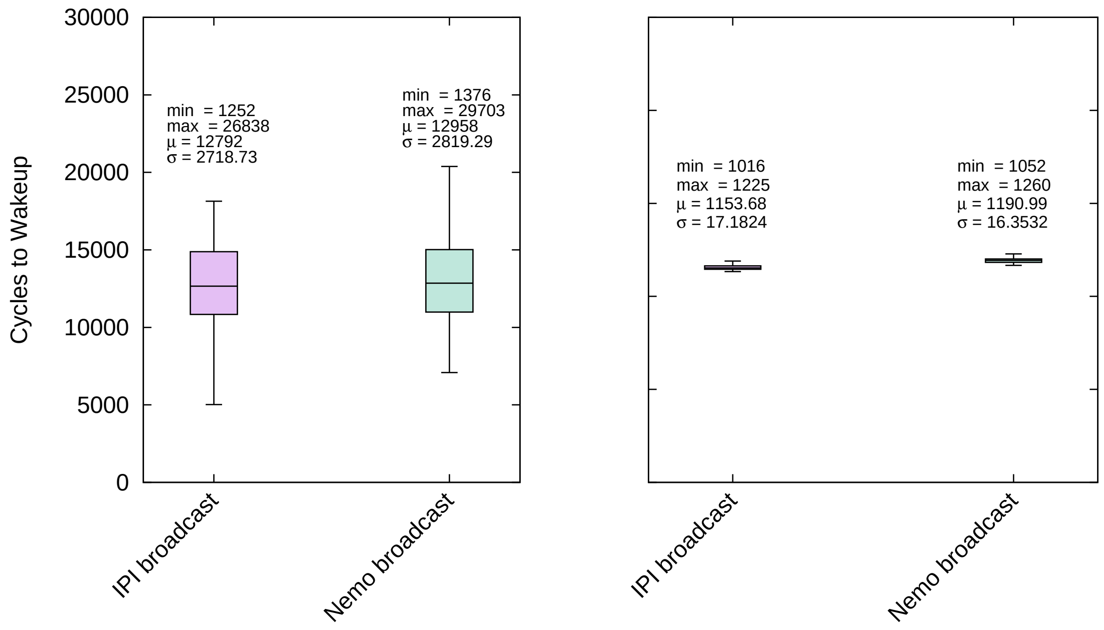
Figure 6.15: CDF showing unicast latency of Nemo's Active Message-inspired events compared to unicast IPIs on *phi*. Nemo lies within ∼5% of IPI performance.

user-space programs, it is important to calibrate this performance to another hardware capability that is critical to the performance of multicore machines, namely the cache coherence network. The question here is can we improve the hardware limit?

Mogul et al. lamented this issue [145] while advocating for lightweight, inter-core notifications: "Unfortunately, today IPIs are the only option."

The coherence network in a modern CPU propagates its own form of events between chips, namely messages that implement the protocol that maintains the coherence model. Not only do we expect the coherence network connecting the chips and the associated logic to have low latency but also predictable performance.

How fast is this network from the perspective of event notification in general? We implemented a small *synchronous* event mechanism using memory polling to assess this.



(a) x64 (b) phi

Figure 6.16: Broadcast latency of Nemo's Active Message-inspired events compared to broadcast IPIs on *x64* and *phi*. Performance is nearly identical in both cases.

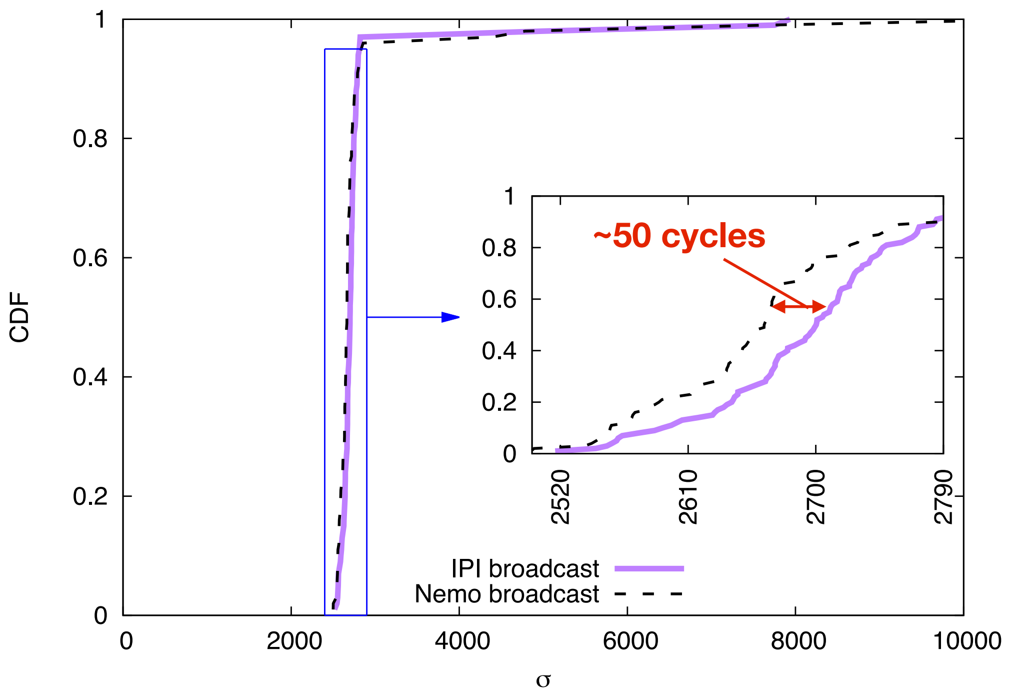


Figure 6.17: CDF comparing Nemo's Active Message-inspired broadcast events to broadcast IPIs on x64. Synchrony is nearly identical.

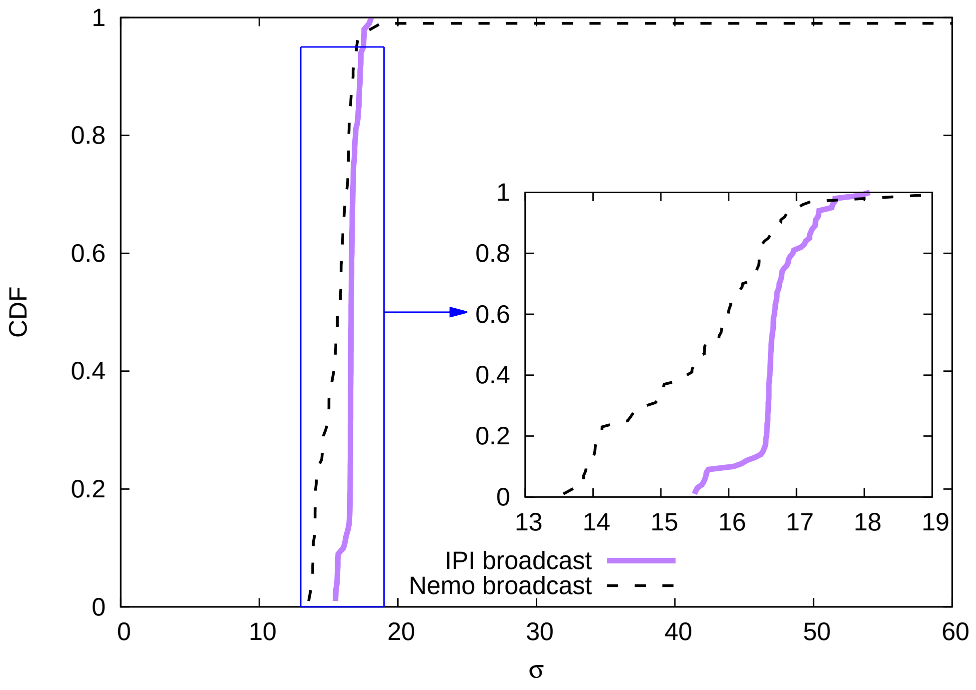


Figure 6.18: CDF comparing Nemo's Active Message-inspired broadcast events to broadcast IPIs on *phi*. Synchrony is nearly identical.

In this mechanism, much like in a barrier or a spinlock, the waiting thread simply spins on a memory location waiting for its value to change. When a signaling thread changes this value, its core's cache controller will send a coherence message to the waiting thread's core, ultimately prompting a cache fill with the newly written value, and an exit from the spin. Figure 6.19 shows the performance of this synchronous mechanism compared to the asynchronous mechanism of unicast IPIs on our x64 hardware. IPIs are roughly 1000 cycles more expensive until the notifications (or invalidations) have to travel further through the chip hierarchy and off chip. The stepwise nature of the "coherence network" curve confirms our prediction of predictable, low-latency performance.

These results prompted us to ask a new question: what prevents the IPI network from achieving performance comparable to the coherence network? To address this question, we performed an analysis of IPIs from the kernel programmer's perspective,

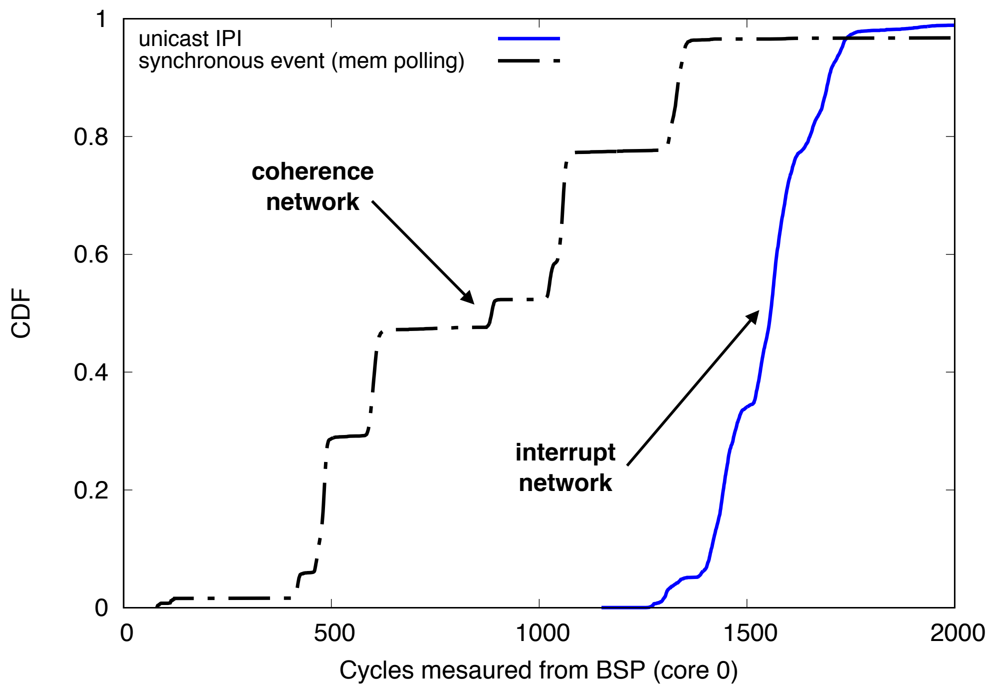


Figure 6.19: CDF comparing latency of asynchronous unicast IPIs compared to a simple synchronous notification scheme using memory polling on x64. This represents the basic cost difference between a synchronous and asynchronous event imposed by the hardware.

Software event	**min. cycles**
Source APIC write	43
Destination handling	729
Communication delay	378
Total for unicast IPI	**1150**
Total for `syscall`	**232**

Figure 6.20: Estimated IPI cost breakdown in cycles on *x64*.

gathering measurements for the hardware and software events necessary for their delivery. Figure 6.20 shows the results.

The latency of a unicast IPI involves three components. The first, “Source APIC write”, is the time to initiate the IPI by writing the APIC registers appropriately at the source. In the figure, we record the minimum time we observed. The second component, denoted “Destination handling,” is the time required at the destination to handle the interrupt, going from message delivery to the time of the first interrupt handler instruction. To estimate this number, we measured the minimum latency from initiating a software interrupt (via an `int` instruction) to the entry of its handler on the same core. We expect that this number is actually an underestimate since it does not include any latency that might be introduced by processing in the destination APIC. The “Communication delay” is simply these two numbers subtracted from the total unicast IPI cost shown in Section 6.1.2. It is likely to be an overestimate.

Integrating the observations of Figures 6.19 and 6.20 suggests that the reason why an asynchronous IPI has so much higher latency than a synchronous coherence event is likely to be due, in large part, to the destination handling costs of an IPI. For asynchronous event notification in an HRT, much of this handling is probably not needed—we would like to simply invoke a remote function, much like a Startup IPI (SIPI) available on modern x86 machines. In particular, the privilege checks, potential stack switches, and stack pushes involved in an IPI are unnecessary.

A similar issue was addressed a decade ago when it was shown how much overhead was involved in processing of the `int` instruction used in system calls, especially as clock speeds grew disproportionally to interrupt handling logic. Designers at Intel and AMD introduced the `syscall` and `sysret` instructions to reduce this overhead considerably.

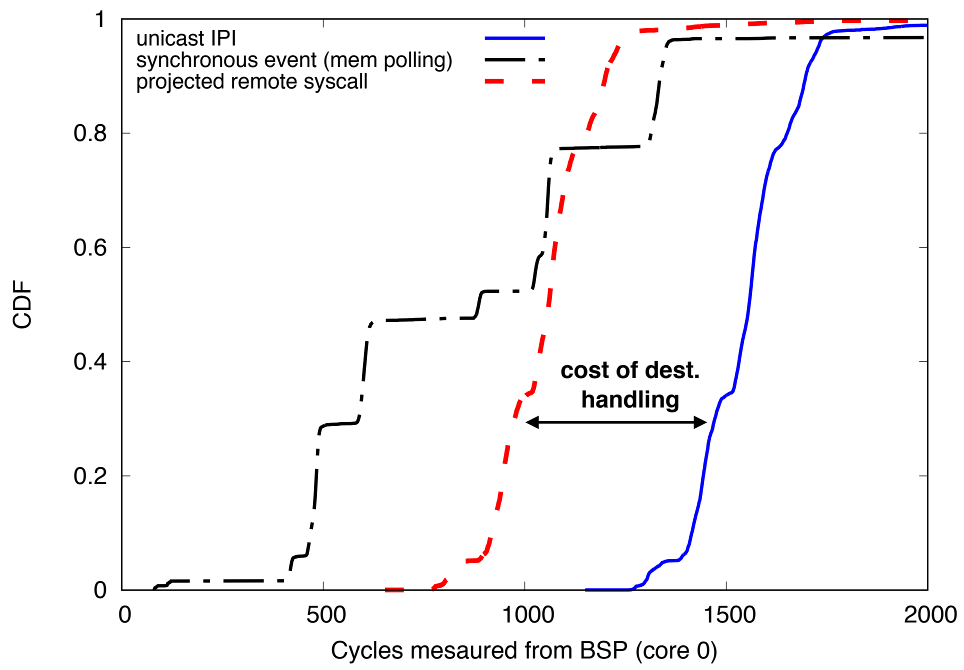


Figure 6.21: CDF of the projected latency of the proposed “remote syscall” mechanism. This comparison is made relative to the measured latencies of Figure 6.19.

Figure 6.20 notes the cost of a `syscall` on our x64 hardware, which is less than 1/3 of our estimated destination handling costs for an IPI.

We believe a similar modification to the architecture could produce comparable benefits for low-latency event delivery and handling. The essential concept is to introduce a new class of IPIs, the “remote `syscall`”. This would combine the IPI generation and transmission logic with the `syscall` logic on the destination core. That is, this form of IPI would act like a `syscall` to the remote core, avoiding privilege checks, stack switches, or any stack accesses. To estimate the gains from this model, we made a projection of IPI performance if one could reduce destination handling to the cost of a `syscall` instruction. Figure 6.21 shows the projected improvements. There is now considerable overlap in the performance of synchronous events based on the coherence network and asynchronous events based on the new "remote `syscall`".

Current Intel APICs use an Interrupt Command Register (ICR), to initiate IPIs. The delivery mode field, which is 3 bits long, indicates what kind of interrupt to deliver. Mode `011` is currently reserved, so this is a possible candidate for a *remote* `syscall` mode. There are, of course numerous varieties of the APIC model between Intel and AMD, but the ICR is a 64 bit register with numerous reserved bits in all of them. Any of these bits could be used to encode a request for a "remote `syscall`". As another example, the 2 bit wide delivery shorthand field could be extended into the adjacent reserved field by one bit to accommodate indicating whether delivery should happen by the traditional interrupt mechanism or via `syscall`-like handling. In these delivery modes or shorthands, the vector might provide a hint to the event handling dispatch software. We expect that these changes would be minimal, although we do not know what effort would be needed to integrate this new functionality with instruction fetch logic. The fact that a SIPI can already vector the core to a specific instruction suggests to us that it might not introduce much new logic. Indeed, another possible approach might be to allow SIPIs when the core is outside of its INIT state.

6.4 Conclusions

In this chapter I have shown how the performance of asynchronous software events suffers when the application/runtime is restricted to user-space mechanisms and mismatched event programming interfaces. The performance of these mechanisms comes nowhere near the hardware limits of IPIs, much less cache coherence messages. By leveraging the HRT model, wherein the runtime and application can execute with fully privileged hardware access, we increased the performance of these event mechanisms considerably. We did so by designing, implementing, and evaluating the Nemo asynchronous event system within the Nautilus Aerokernel framework for building HRTs on x64 and Xeon Phi. HRTs built using Nemo primitives can enjoy event wakeup latencies that are as much as 4,000 times lower than the event mechanisms typically used in user-space. Furthermore, the variation in wakeup latencies in Nemo is much lower, allowing a greater degree of synchrony between broadcasts to multiple cores. In addition to Nemo, we also considered the design of IPIs themselves and proposed a small hardware addition that could potentially reduce their cost considerably for constrained use cases, such as asynchronous event notification. I showed that such additions can push the performance of asynchronous software events closer to that of the hardware cache coherence network.

Chapter 7

Multiverse Toolchain

While porting a parallel runtime to the HRT model can produce the highest performance gains, it requires an intimate familiarity with the runtime system's functional requirements, which may not be obvious. These requirements must then be implemented in the Aerokernel layer and the Aerokernel and runtime combined. This requires a deep understanding of kernel development. This manual process is also iterative: the developer adds Aerokernel functionality until the runtime works correctly. The end result might be that the Aerokernel interfaces support a small subset of POSIX, or that the runtime developer replaces such functionality with custom interfaces.

While such a development model *is* tractable, and we have transformed three runtimes to HRTs using it (see Chapter 3), it represents a substantial barrier to entry to creating HRTs, which we seek here to lower. The manual porting method is *additive* in its nature. We must add functionality until we arrive at a working system. A more expedient method would allow us to *start* with a working HRT produced by an automatic process, and then incrementally extend it and specialize it to enhance its performance.

The Multiverse system described in this chapter supports just such a method using a technique called *automatic hybridization* to create a working HRT from an existing, unmodified runtime and application. With Multiverse, runtime developers can take an incremental path towards adapting their systems to run in the HRT model. From the user's perspective, a hybridized runtime and application behaves the same as the original. It can be run from a Linux command line and interact with the user just like any other executable. But internally, it executes in kernel mode as an HRT.

Multiverse bridges a specialized HRT with a legacy environment by borrowing functionality from a legacy OS, such as Linux. Functions not provided by the existing Aerokernel are forwarded to another core that is running the legacy OS, which handles them and returns their results. The runtime developer can then identify hot spots in the legacy interface and move their implementations (possibly even changing their interfaces) into the Aerokernel. The porting process with Multiverse is *subtractive* in that a developer iteratively removes dependencies on the legacy OS. At the same time, the developer can take advantage of the kernel-level environment of the HRT.

To demonstrate the capabilities of Multiverse, we automatically hybridize the Racket runtime system. Racket has a complex, JIT-based runtime system with garbage collection and makes extensive use of the Linux system call interface, memory protection mechanisms, and external libraries. Hybridized Racket executes in kernel mode as an HRT, and yet the user sees precisely the same interface (an interactive REPL environment, for example) as out-of-the-box Racket.

This chapter will first introduce the Multiverse toolchain in Section 7.1. Section 7.2 details the implementation of Multiverse, while Section 7.3 presents its evaluation. Section 7.4 concludes the chapter.

7.1 Multiverse

We designed the Multiverse system to support automatic hybridization of existing runtimes and applications that run in user-level on Linux platforms.

7.1.1 Perspectives

The goal of Multiverse is to ease the path for developers of transforming a runtime into an HRT. We seek to make the system look like a compilation toolchain option from the developer's perspective. That is, to the greatest extent possible, the HRT is a compilation target. Compiling to an HRT simply results in an executable that is a "fat binary" containing additional code and data that enables kernel-mode execution in an environment that supports it. An HVM-enabled virtual machine on Palacios is the first such environment. The developer can then extend this incrementally—Multiverse facilitates a path for runtime and application developers to explore how to specialize their HRT to the full hardware feature set and the extensible kernel environment of the Aerokernel.

From the user's perspective, the executable behaves just as if it were compiled for a standard user-level Linux environment. The user sees no difference between HRT execution and user-level execution.

7.1.2 Techniques

The Multiverse system relies on three key techniques: state superpositions, split execution, and event channels. We now describe each of these.

Split execution In Multiverse, a runtime and its application begin their execution in the ROS. Through a well-defined interface discussed in Section 7.1.3, the runtime on the ROS side can spawn an execution context in the HRT. At this point, Multiverse splits its execution into two components, each running in a different context; one executes in the ROS and the other in the HRT. The semantics of these execution contexts differ from traditional threads depending on their characteristics. I discuss these differences in Section 7.2. In the current implementation, the context on the ROS side comprises a Linux thread, the context on the HRT side comprises an Aerokernel thread, and I will refer to them collectively as an *execution group*. While execution groups in our current system consist of threads in different OSes, this need not be true in general. The context on the HRT side executes until it triggers a fault, a system call, or other event. The execution group then converges on this event, with each side participating in a protocol for requesting events and receiving results. This protocol exchange occurs in the context of HVM event channels, which I discuss below.

Figure 7.1 illustrates the split execution of Multiverse for a ROS/HRT execution group. At this point, the ROS has already made a request to create a new context in the HRT, e.g. through an asynchronous function invocation. When the HRT thread begins executing in the HRT side, exceptional events, such as page faults, system calls, and other exceptions vector to stub handlers in the Aerokernel, in this case Nautilus (1). The Aerokernel then redirects these events through an event channel (2) to request handling in the ROS. The VMM then injects these into the originating ROS thread, which can take action on them directly (3). For example, in the case of a page fault that occurs in the ROS portion of the virtual address space, the HVM library simply replicates the access, which will cause the same exception to occur on the ROS core. The ROS will then handle it as it would normally.

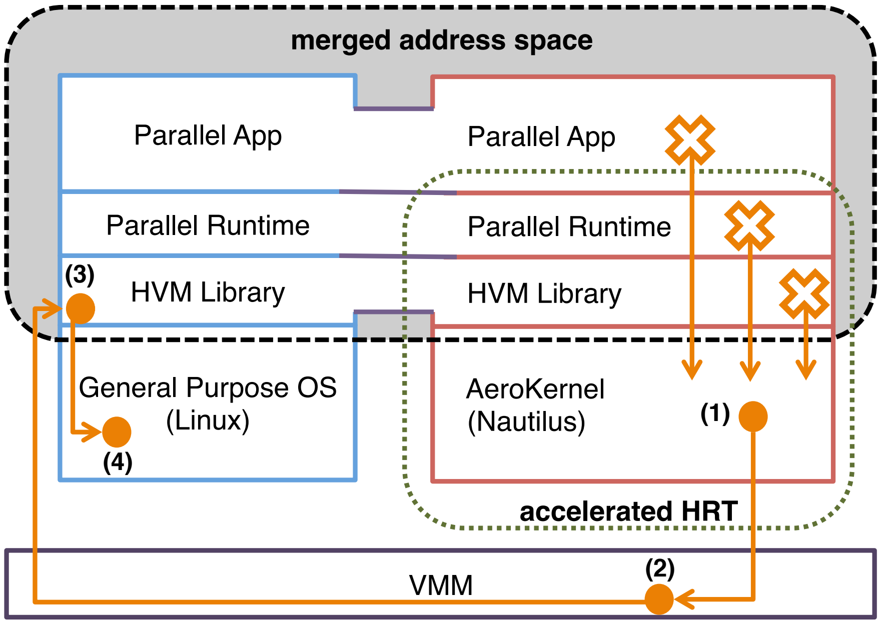


Figure 7.1: Split execution in Multiverse.

Item	Cycles	Time
Address Space Merger	∼33 K	1.5 μs
Asynchronous Call	∼25 K	1.1 μs
Synchronous Call (different socket)	∼1060	48 ns
Synchronous Call (same socket)	∼790	36 ns

Figure 7.2: Round-trip latencies of ROS↔HRT interactions.

In the case of events that need direct handling by the ROS kernel, such as system calls, the HVM library can simply forward them (4).

Event channels When the HRT needs functionality that the ROS implements, access to that functionality occurs over *event channels*, event-based, VMM-controlled communication channels between the two contexts. The VMM only expects that the execution group adheres to a strict protocol for event requests and completion.

Figure 7.2 shows the measured latency of event channels with the Nautilus Aerokernel performing the role of HRT. Note that these calls are bounded from below by the latency of hypercalls to the VMM.

State superpositions In order to forego the addition of burdensome complexity to the Aerokernel environment, it helps to leverage functions in the ROS other than those that lie at a system call boundary. This includes functionality implemented in libraries and more opaque functionality like optimized system calls in the `vdso` and the `vsyscall` page. In order to use this functionality, Multiverse can set up the HRT and ROS to share portions of their address space, in this case the user-space portion. Aside from the address space merger itself, Multiverse leverages other state superpositions to support a shared address space, including superpositions of the ROS GDT and thread-local storage state.

In principle, we could superimpose any piece of state visible to the VMM. The ROS or the runtime need not be aware of this state, but the state is nonetheless necessary for facilitating a simple and approachable usage model.

The superposition we leverage most in Multiverse is a merged address space between the ROS and the HRT, depicted in Figure 5.1. The merged address space allows execution in the HRT without a need for implementing ROS-compatible functionality. When a merged address space takes effect, the HRT can use the same user-mode virtual addresses present in the ROS. For example, the parallel runtime in the ROS might load files and construct a complex pointer-based data structure in memory. It can then invoke a function within its counterpart in the HRT to compute over that data.

7.1.3 Usage models

The Multiverse system is designed to give maximum flexibility to application and runtime developers in order to encourage exploration of the HRT model. While the degree to which

```
static void*
routine (void * in) {
    void * ret = aerokernel_func();
    printf("Result = %d\n", ret);
}

int main (int argc, char ** argv) {
    hrt_invoke_func(routine);
    return 0;
}
```

Figure 7.3: Example of user code adhering to the accelerator model.

a developer leverages Multiverse can vary, we can broadly categorize the usage model into three categories, discussed below.

Native In the native model, the application/runtime is ported to operate fully within the HRT/Aerokernel setting. That is, it does not use any functionality not exported by the Aerokernel, such as `glibc` functionality or system calls like `mmap()`. This category allows maximum performance, but requires more effort, especially in the compilation process. The ROS side is essentially unnecessary for this usage model, but may be used to simplify the initiation of HRT execution (e.g. requesting an HRT boot). The native model is also native in another sense: it can execute on bare metal without any virtualization support. This is the model used in Chapters 3–6.

Accelerator In this category, the app/runtime developer leverages both legacy (e.g. Linux) functionality and Aerokernel functionality. This requires less effort, but allows the developer to explore some of the benefits of running their code in an HRT. Linux functionality is enabled by the merged address space discussed previously, but the developer can also leverage Aerokernel functions.

Figure 7.3 shows a small example of code that will create a new HRT thread and use event channels and state superposition to execute to completion. Runtime initialization is opaque to the user, much like C runtime initialization code. When the program invokes the `hrt_invoke_func()` call, the Multiverse runtime will make a request to the HVM to run `routine()` in a new thread on the HRT core. Notice how this new thread can call an Aerokernel function directly, and then use the standard `printf()` routine to print its result. This `printf` call relies both on a state superposition (merged address space) for the function call linkage to be valid, and on event channels, which will be used when the C library code invokes a system call (e.g. `write()`).

Incremental The application/runtime executes in the HRT context, but does not leverage Aerokernel functionality. Benefits are limited to aspects of the HRT *environment*. However, the developer need only recompile their application to explore this model. Instead of raising an explicit HRT thread creation request, Multiverse will create a new thread in the HRT corresponding to the program's `main()` routine. The Incremental model also allows parallelism, as legacy threading functionality automatically maps to the corresponding Aerokernel functionality with semantics matching those used in pthreads. The developer can then incrementally expand their usage of hardware- and Aerokernel-specific features.

While the accelerator and incremental usage models rely on the HVM virtualized environment of Palacios, it is important to note that they could also be built on physical partitioning [154] as well. At its core, HVM provides to Multiverse a resource partitioning, the ability to boot multiple kernels simultaneously on distinct partitions, and the ability for these kernels to share memory and communicate.

```
static void*
routine (void * in) {
    void * ret = aerokernel_func();
    printf("Result = %d\n", ret);
}

int main (int argc, char ** argv) {
    pthread_t t;
    pthread_create(&t, NULL, routine, NULL);
    pthread_join(t, NULL);
    return 0;
}
```

Figure 7.4: Example of user code adhering to the accelerator model with overrides.

7.1.4 Aerokernel Overrides

One way a developer can enhance a generated HRT is through *function overrides*. The Aerokernel can implement functionality that conforms to the interface of, for example, a standard library function, but that may be more efficient or better suited to the HRT environment. This technique allows users to get some of the benefits of the accelerator model without any explicit porting effort. However, it is up to the Aerokernel developer to ensure that the interface semantics and any usage of global data make sense when using these function overrides. Function overrides are specified in a simple configuration file that is discussed in Section 7.2.

Figure 7.4 shows the same code from Figure 7.3 but using function overrides. Here the Aerokernel developer has overridden the standard pthreads routines so that `pthread_create()` will create a new HRT thread in the same way that `hrt_invoke_func()` did in the previous example.

7.1.5 Toolchain

The Multiverse toolchain consists of two main components, the runtime system code and the build setup. The build setup consists of build tools, configuration files, and an Aerokernel binary provided by the Aerokernel developer. To leverage Multiverse, a user must simply integrate their application or runtime with the provided Makefile and rebuild it. This will compile the Aerokernel components necessary for HRT operation and the Multiverse runtime system, which includes function overrides, Aerokernel binary parsing routines, exit and signal handlers, and initialization code, into the user program.

7.2 Implementation

I now discuss implementation details for the runtime components of the Multiverse system. This includes the portion of Multiverse that is automatically compiled and linked into the application's address space at build time and the parts of Nautilus and the HVM that support event channels and state superpositions. Unless otherwise stated, I assume the incremental usage model discussed in Section 7.1.3.

7.2.1 Multiverse Runtime Initialization

As mentioned in Section 7.1, a new HRT thread must be created from the ROS side (the originating ROS thread). This, however, requires that an Aerokernel be present on the requested core to create that thread. The runtime component (which includes the user-level HVM library) is in charge of booting an Aerokernel on all required HRT cores during program startup. They can either be booted on demand or at application startup. We use

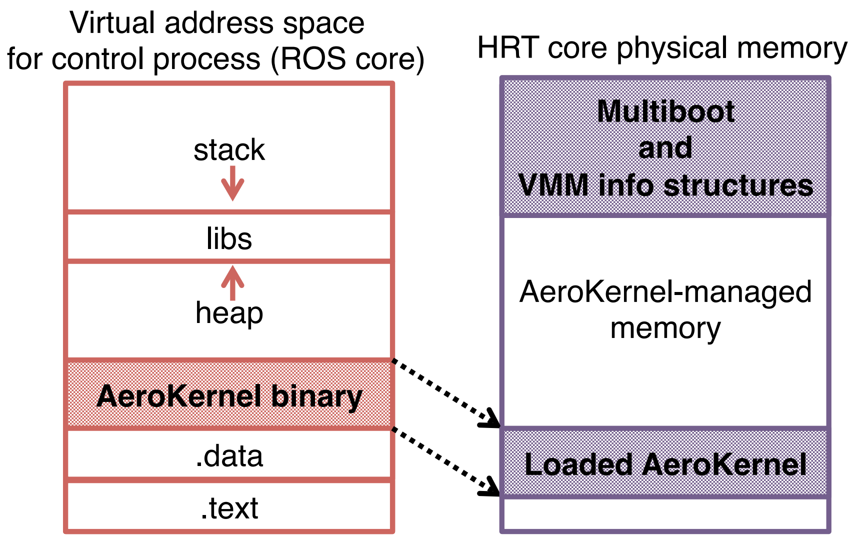


Figure 7.5: Nautilus boot process.

the latter in our current setup.

Our toolchain inserts program initialization hooks before the program's `main()` function, which carry out runtime initialization.

Initialization tasks include the following:

- Registering ROS signal handlers
- Hooking process exit for HRT shutdown
- Aerokernel function linkage
- Aerokernel image installation in the HRT
- Aerokernel boot
- Merging ROS and HRT address spaces

Aerokernel Boot Our toolchain embeds an Aerokernel binary into the ROS program's ELF binary. This is the image to be installed in the HRT. At program startup, the Multiverse runtime component parses this embedded Aerokernel binary and sends a request to the HVM asking that it be installed in physical memory, as shown in Figure 7.5. Multiverse then requests the Aerokernel be booted on that core. The boot process, which I described in detail in Chapter 4, brings the Aerokernel up into an event loop that waits for HRT thread creation requests.

The above initialization tasks are opaque to the user, who needs (in the Accelerator usage model) only understand the interfaces to create execution contexts within the HRT.

7.2.2 Execution Model

To implement split execution, we rely on HVM's ability to forward requests from the ROS core to the HRT, along with event channels and merged address spaces.

The runtime developer can leverage two mechanisms to create HRT threads, as discussed in Section 7.1.3. Furthermore, two types of threads are possible on the HRT side: top-level threads and nested threads. Top-level threads are threads that the ROS explicitly creates. A top-level HRT thread can create its own child threads as well; we classify these as nested threads. The semantics of the two thread types differ slightly in their operation. Nested threads resemble pure Aerokernel threads, but their execution can proceed in the context of the ROS user address space. Top-level threads require extra semantics in the HRT and in the Multiverse component linked with the ROS application.

Threads: Multiverse pairs each top-level HRT thread with a *partner* thread that executes in the ROS. The purpose of this thread is two-fold. First, it allows us to preserve join semantics. Second, it gives us the proper thread context in the ROS to initiate a state superposition for the HRT. Figure 7.6 depicts the creation of HRT threads and their interaction with the ROS. First, in (1) the main thread is created in the ROS. This thread sets

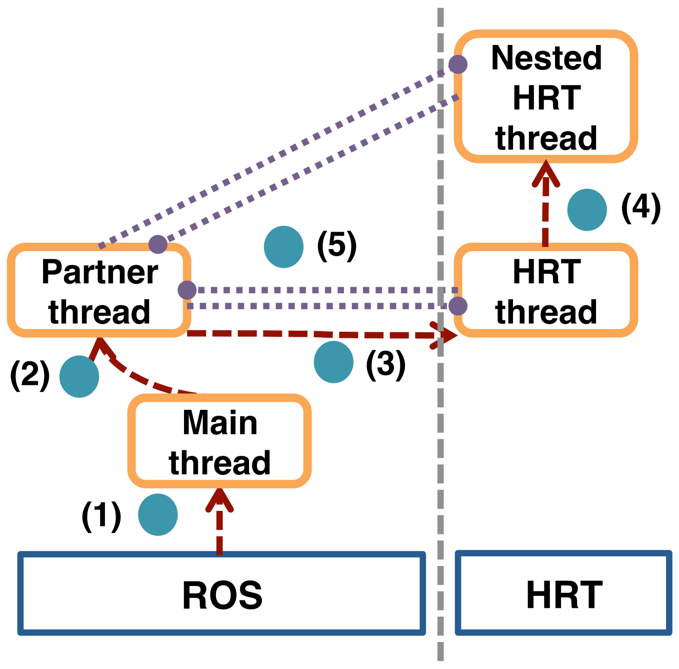


Figure 7.6: Interactions within an execution group.

up the runtime environment for Multiverse. When the runtime system creates a thread, e.g. with `pthread_create()` or with `hrt_invoke_func()`, Multiverse creates a corresponding partner thread that executes in the ROS (2). It is the duty of the partner thread to allocate a ROS-side stack for a new HRT thread then invoke the HVM to request a thread creation in the HRT using that stack (3). When the partner creates the HRT thread, it also sends over information to initiate a state superposition that mirrors the ROS-side GDT and ROS-side architectural state corresponding to thread-local storage (primarily the `%fs` register). The HRT thread can then create as many nested HRT threads as it desires (4). Both top-level HRT threads and nested HRT threads raise events to the ROS through event channels with the top-level HRT thread's corresponding partner acting as the communication end-point (5).

As is typical in threading models, the main thread can wait for HRT threads to finish by using `join()` semantics, where the joining thread blocks until the child exits. While in theory we could implement the ability to join an HRT thread directly, it would add complexity to both the HRT and the ROS component of Multiverse. Instead, we chose to allow the main thread to join a partner thread directly and provide the guarantee that a partner thread will not exit until its corresponding HRT thread exits on the remote core. When an HRT thread exits, it signals the ROS of the exit event. When Multiverse creates an HRT thread, it keeps track of the Nautilus thread data (sent from the remote core after creation succeeds), which it uses to build a mapping from HRT threads to partner threads. The thread exit signal handler in the ROS flips a bit in the appropriate partner thread's data structure notifying it of the HRT thread completion. The partner can then initiate its cleanup routines and exit, at which point the main thread will be unblocked from its initial `join()`.

Disallowed functionality: Because of the limitations of our current Aerokernel implementation, we must prohibit the ROS code executing in HRT context from leveraging certain functionality. This includes calls that create new execution contexts or rely on the Linux execution model such as `execve`, `clone`, and `futex`. This functionality could, of course, be provided in the Aerokernel, but we have not implemented it at the time of this writing.

Function overrides: In Section 7.1.3 I described how a developer can use function overrides to select Aerokernel functionality over default ROS functionality. The Multiverse runtime component enforces default overrides that interpose on `pthread` function calls. All function overrides operate using function wrappers. For simple function wrappers, the Aerokernel developer can simply make an addition to a configuration file included in the Multiverse toolchain that specifies the function's attributes and argument mappings between the legacy function and the Aerokernel variant. This configuration file then allows

Multiverse to automatically generates function wrappers at build time.

When an overridden function is invoked, the wrapper runs instead, consults a stored mapping to find the symbol name for the Aerokernel variant, and does a symbol lookup to find its HRT virtual address. This symbol lookup currently occurs on every function invocation, so incurs a non-trivial overhead. A symbol cache, much like that used in the ELF standard, could easily be added to improve lookup times. When the address of the Aerokernel override is resolved, the wrapper then invokes the function directly (since it is already executing in the HRT context where it has appropriate page table mappings for Aerokernel addresses).

7.2.3 Event Channels

The HVM model enables the building of essentially any communication mechanism between two contexts (in our case, the ROS and HRT), and most of these require no specific support in the HVM. As a consequence, we minimally define the *basic* communication between the ROS, HRT, and the VMM using shared physical memory, hypercalls, and interrupts.

The user-level code in the ROS can use hypercalls to sequentially request HRT reboots, address space mergers (state superpositions), and asynchronous sequential or parallel function calls. The VMM handles reboots internally, and forwards the other two requests to the HRT as special exceptions or interrupts. Because the VMM and HRT may need to share additional information, they share a data page in memory. For a function call request, the page contains a pointer to the function and its arguments at the start and the return code at completion. For an address space merger, the page contains the CR3 of the calling process. The HRT indicates to the VMM when it is finished with the current request via a hypercall.

After an address space merger, the user-level code in the ROS can also use a single hypercall to initiate synchronous operation with the HRT. This hypercall ultimately indicates to the HRT a virtual address which will be used for future synchronization between the HRT and ROS. They can then use a simple memory-based protocol to communicate, for example to allow the ROS to invoke functions in the HRT without VMM intervention.

7.2.4 Merged Address Spaces

To achieve a merged address space, we leverage the canonical 64-bit address space model of x64 processors, and its wide use within existing kernels, such as Linux. In this model, the virtual address space is split into a "lower half" and a "higher half" with a gap in between, the size of which is implementation dependent. In a typical process model, e.g., Linux, the lower half is used for user addresses and the higher half is used for the kernel.

For an HRT that supports it, the HVM arranges that the physical address space is identity-mapped into the higher half of the HRT address space. That is, within the HRT, the physical address space mapping (including the portion of the physical address space only the HRT can access) occupies the same portion of the virtual address space that the ROS kernel occupies, namely the higher half. Without a merger, the lower half is unmapped and the HRT runs purely out of the higher half. When the ROS side requests a merger, we map the lower half of the ROS's current process address space into the lower half of the HRT address space. For an Aerokernel-based HRT, we achieve this by copying the first 256 entries of the PML4 pointed to by the ROS's CR3 to the HRT's PML4 and then broadcasting a TLB shootdown to all HRT cores.

Because the runtime in the ROS and the HRT are co-developed, the responsibility of assuring that page table mappings exist for lower half addresses used by the HRT in a merged address space is the runtime's. For example, the runtime can pin memory before merging the address spaces or introduce a protocol to send page faults back to the ROS. The former is not an unreasonable expectation in a high performance environment as we would never expect a significant amount of swapping.

7.2.5 Nautilus Additions

In order to support Multiverse in the Nautilus Aerokernel, we needed to make several additions to the codebase. Most of these focus on runtime initialization and correct operation of event channels. When the runtime and application are executing in the HRT, page faults in the ROS portion of the virtual address space must be forwarded. We added a check in the page fault handler to look for ROS virtual addresses and forward them appropriately over an event channel.

One issue with our current method of copying a portion of the PML4 on an address space merger is that we need to keep the PML4 synchronized. We must account for situations in which the ROS changes top-level page table mappings, even though these changes are rare. We currently handle this situation by detecting repeat page faults. Nautilus keeps a per-core variable keeping track of recent page faults, and matches duplicates. If a duplicate is found, Nautilus will re-merge the PML4 automatically. More clever schemes to detect this condition are possible, but unnecessary since it does not lie on the critical path.

For correct operation, Multiverse requires that we catch *all* page faults and forward them to the ROS. That is, if we collect a trace of page faults in the application running native and under Multiverse, the traces should look identical. However, because the HRT runs in kernel mode, some paging semantics (specifically with copy-on-write) change. In default operation, an x86 CPU will only raise a page fault when writing a read-only page in user-mode. Writes to pages with the read-only bit while running in ring 0 are allowed to proceed. This issue manifests itself in the form of mysterious memory corruption, e.g. by writing to the zero page. Luckily, there is a bit to enforce write faults in ring 0 in the `cr0` control register.

Before we built Multiverse, Nautilus lacked support for system calls, as the HRT operates entirely in kernel mode. However, a legacy application will leverage a wide range of system calls. To support them, we added a small system call stub handler in Nautilus that immediately forwards the system call to the ROS over an event channel. There is an added subtlety with system calls in HRT mode, as they are now initiating a trap from ring 0 to ring 0. This conflicts with the hardware API for the `SYSCALL/SYSRET` pair of instructions. We found it interesting that `SYSCALL` has no problem making this idempotent ring transition, but `SYSRET` will not allow it. The return to ring 3 is unconditional for `SYSRET`. To work around this issue, we must emulate `SYSRET` and execute a direct `jmp` to the saved `rip` stashed during the `SYSCALL`.

While we can build a particular runtime system with the Multiverse toolchain using custom compilation options, this is not possible for the legacy libraries they rely on. We are then forced into supporting the compilation model that the libraries were initially compiled with. While arbitrary compilation does not typically present issues for user-space programs, complications arise when executing in kernel mode. One such complication is AMD's *red zone*, which newer versions of GCC use liberally. The red zone sits *below* the current stack pointer on entry to leaf functions, allowing them to elide the standard function prologue for stack variable allocation. The red zone causes trouble when interrupts and

Component	*SLOC*			
	C	ASM	Perl	Total
Multiverse runtime	2232	65	0	2297
Multiverse toolchain	0	0	130	130
Nautilus additions	1670	0	0	1670
HVM additions	600	38	0	638
Total	**4502**	**103**	**130**	**4735**

Figure 7.7: Source Lines of Code for Multiverse.

exceptions operate on the same stack, as the push of the interrupt stack frame by the hardware can destroy the contents of the red zone. To avoid this, kernels are typically compiled to disable the red zone. However, since we are executing code in the ROS address space with predetermined compilation, we must use other methods.

In Nautilus, we address the red zone by ensuring that interrupts and exceptions operate on a well known interrupt stack, not on the user stack. We do this by leveraging the x86 Interrupt Stack Table (IST) mechanism, which allows the kernel to assign specific stacks to particular exceptions and interrupts by writing a field in the interrupt descriptor table. `SYSCALL` cannot initiate a hardware stack switch in the same way, so on entry to the Nautilus system call stub, we pull down the stack pointer to avoid destroying any red zone contents.

7.2.6 Complexity

Multiverse development took roughly 5 person months of effort. Figure 7.7 shows the amount of code needed to support Multiverse. The entire system is compact and compartmentalized so that users can experiment with other Aerokernels or runtime systems with relative ease. While the codebase is small, much of the time went into careful design of the execution model and working out idiosyncrasies in the hybridization, specifically those

Benchmark	System Calls	Time (User/Sys) (s)	Max Resident Set (Kb)	Page Faults	Context Switches	Forwarded Events
spectral-norm	23800	39.39/0.24	182300	51452	1695	75252
n-body	18763	41.15/0.19	152300	45064	1430	63827
fasta-3	35115	31.28/0.17	80492	25418	1075	60533
fasta	29989	12.23/0.10	43568	14956	627	44945
binary-tree-2	1260	31.98/0.10	82072	31082	491	32342
mandelbrot-2	3667	7.76/0.05	43600	14250	291	17917
fannkuch-redux	1279	2.73/0.01	21284	5358	33	6637

Figure 7.8: System utilization for Racket benchmarks. A high-level language has many low-level interactions with the OS.

dealing with operation in kernel mode.

7.3 Evaluation

This section presents an evaluation of Multiverse using microbenchmarks and a hybridized Racket runtime system running a set of benchmarks from The Language Benchmark Game. We ran all experiments on a Dell PowerEdge 415 with 8GB of RAM and an 8 Core 64-bit x86_64 AMD Opteron 4122 clock clocked at 2.2GHz. Each CPU core has a single thread with four cores per socket. The host machine has stock Fedora Linux 2.6.38.6-26.rc1.fc15.x86_64 installed. Benchmark results are reported as averages of 10 runs.

Experiments in a VM were run on a guest setup which consists of a simple BusyBox distribution running an unmodified Linux 2.6.38-rc5+ image with two cores (one core for the HVM and one core for the ROS) and 1 GB of RAM.

7.3.1 Racket

Racket [77, 73] is the most widely used Scheme implementation and has been under continuous development for over 20 years. It is an open source codebase that is downloaded over 300 times per day.[1] Recently, support has been added to Racket for parallelism via

[1]`http://racket-lang.org`

futures [182] and places [187].

The Racket runtime is a good candidate to test Multiverse, particularly its most complex usage model, the incremental model, because Racket includes many of the challenging features emblematic of modern dynamic programming languages that make extensive use of the Linux ABI, including system calls, memory mapping, processes, threads, and signals. These features include complex package management via the file system, shared library-based support for native code, JIT compilation, tail-call elimination, live variable analysis (using memory protection), and garbage collection.

Our port of Racket to the HRT model takes the form of an instance of the Racket engine embedded into a simple C program. Racket already provides support for embedding an instance of Racket into C, so it was straightforward to produce a Racket port under the Multiverse framework. This port uses a conservative garbage collector, the SenoraGC, which is more portable and less performant than the default, precise garbage collector. The port was compiled with GCC 4.6.3. The C program launches a pthread that in turn starts the engine. Combined with the incremental usage model of Multiverse, the result is that the Racket engine executes in the HRT.

When compiled and linked for regular Linux, our port provides either a REPL interactive interface through which the user can type Scheme, or a command-line batch interface through which the user can execute a Scheme file (which can include other files). When compiled and linked for HRT use, our port behaves identically.

To evaluate the correctness and performance of our port, we tested it on a series of benchmarks submitted to The Computer Language Benchmarks Game [1]. We tested on seven different benchmarks: a garbage collection benchmark (binary-tree-2), a permutation benchmark (fannkuch), two implementations of a random DNA sequence generator (fasta

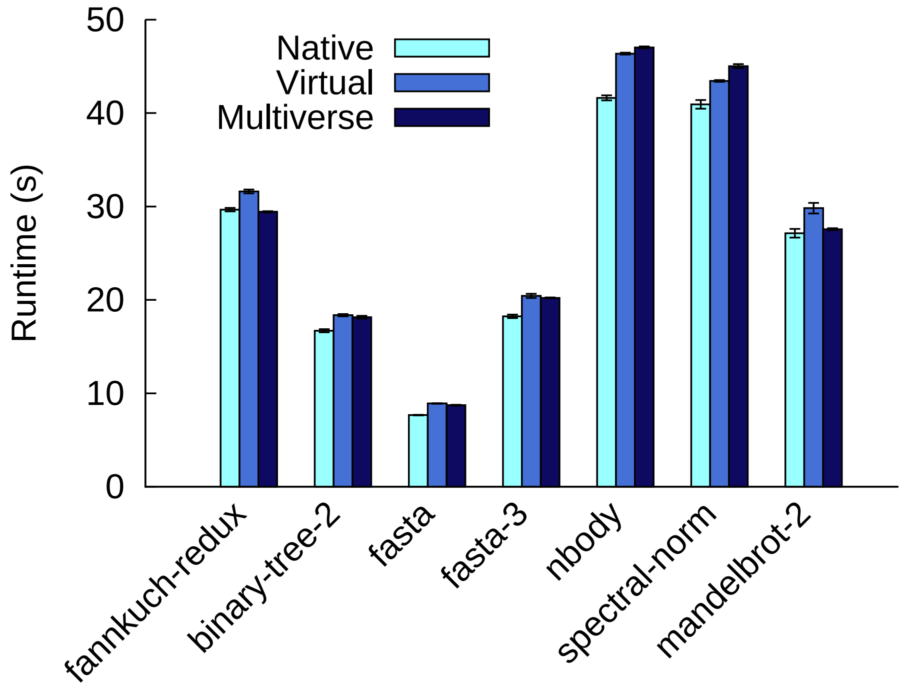


Figure 7.9: Performance of Racket benchmarks running Native, Virtual, and in Multiverse. **With Multiverse, the existing, unmodified Racket implementation has been automatically transformed to run entirely in kernel mode, as an HRT, with little to no overhead.**

and fasta-3), a generation of the mandelbrot set (mandelbrot-2), an n-body simulation (n-body), and a spectral norm algorithm. Figure 7.8 characterizes these benchmarks from the low-level perspective. Note that while this is an implementation of a high-level language, the actual execution of Racket programs involves many interactions with the operating system. These exercise Multiverse's system call and fault forwarding mechanisms. The total number of forwarded events is in the last column.

Figure 7.9 compares the performance of the Racket benchmarks run natively on our hardware, under virtualization, and as an HRT that was created with Multiverse. Error bars are included, but are barely visible because these workloads run in a predictable way. The key takeaway is that Multiverse performance is on par with native and virtualized performance—Multiverse let us move, with little to no effort, the existing, unmodified Racket runtime into kernel mode and run it as an HRT with little to no overhead.

The small overhead of the Multiverse case compared to the virtualized and native cases is due to the frequent interactions, such as those described above, with the Linux ABI. However, in all but two cases, the hybridized benchmarks actually outperform the equivalent versions running without Multiverse. This is due to the nature of the accelerated environment that the HRT provides, which ameliorates the event channel overheads. As we expect, the two benchmarks that perform worse under Multiverse (nbody and spectral-norm) have the most overhead incurred from event channel interactions. The most frequent interactions in both cases are due to page faults.

While Figure 7.8 tells us the number of interactions that occur, we now consider the overhead of each one using microbenchmarks. This estimate will also apply to page faults, since the mechanism by which they are forwarded is identical to system calls.

The most frequent system calls used in the Racket runtime (independent of any benchmark) are `mmap()` and `munmap()`, and so we focus on these two. Figure 7.10 shows microbenchmark results (averages over 100 runs) for these two, comparing virtualized execution and Multiverse. Note that neither system calls nor page faults involve the VMM in the virtualized case. For both system calls, the Multiverse event forwarding facility adds roughly 1500 cycles of overhead. If we multiply this number by the number of forwarded events for these benchmarks listed in Figure 7.8, we expect that Multiverse will add about 112 million cycles (51 ms) of overhead for spectral-norm, and 96 million cycles (43 ms) for nbody. This is roughly in line with Figure 7.9.

Note that event forwarding will decrease as a runtime is incrementally extended to take advantage of the HRT model. As an illustration of this, we modified Racket's garbage

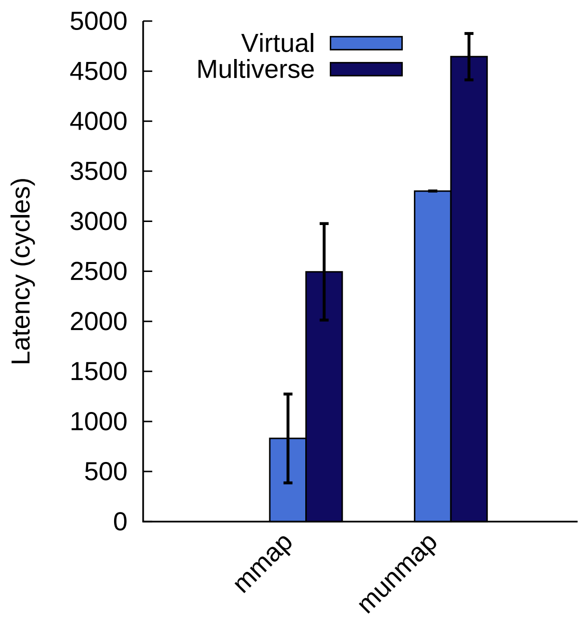


Figure 7.10: Multiverse event forwarding overheads demonstrated with `mmap()` and `munmap()` system calls.

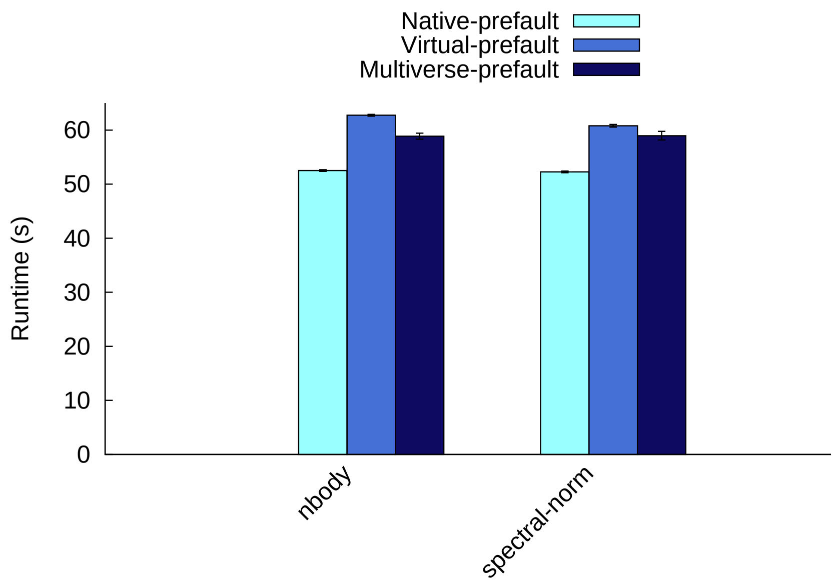


Figure 7.11: Performance of nbody and spectral-norm benchmarks with and without a modification to the runtime system that prefaults in all mapped pages, reducing the number of forwarded events.

collector to prefault in pages any time that it allocates a range of memory for internal use. We accomplished this with a simple change that added the `MAP_POPULATE` flag to Racket's internal `mmap()` invocations. This reduces the number of page faults incurred during the execution of a benchmark, therefore reducing the number of forwarded events. The results are shown in Figure 7.11. While this change increases the running time for the benchmark overall—indicating that this is not a change that one would introduce in practice—it gives us an indication of what the relative performance would be with fewer forwarded events. Indeed, with fewer events, the hybridized versions (Multiverse-prefault) of these benchmarks running in a VM now outperform their counterparts (Virtual-prefault).

It is worth reflecting on what exactly has happened here: we have taken a complex runtime system off-the-shelf, run it through Multiverse without changes, and as a result have a version of the runtime system that correctly runs in kernel mode as an HRT and behaves identically with virtually identical performance. To be clear, **all of the Racket runtime except Linux kernel ABI interactions is seamlessly running as a kernel.** While this codebase is the endpoint for user-level development, it represents a *starting point* for HRT development in the incremental model.

7.4 Conclusions

In this chapter, I introduced Multiverse, a system that implements *automatic hybridization* of runtime systems in order to transform them into hybrid runtimes (HRTs). I illustrated the design and implementation of Multiverse and described how runtime developers can use it for incremental porting of runtimes and applications from a legacy OS to a specialized

Aerokernel.

To demonstrate its power, we used Multiverse to automatically hybridize the Racket runtime system, a complex, widely-used, JIT-based runtime. With automatic hybridization, we can take an existing Linux version of a runtime or application and automatically transform it into a package that looks to the user like it runs just like any other program, but actually executes on a remote core in kernel-mode, in the context of an HRT, and with full access to the underlying hardware. I presented an evaluation focused on the performance overheads of an unoptimized Multiverse hybridization of Racket and showed that performance varies with the usage of legacy functionality. Runtime developers can leverage Multiverse to start with a working system and incrementally transition heavily utilized legacy functions to custom components within an Aerokernel.

Chapter 8

Related Work

This chapter is dedicated to a discussion of work related to this dissertation, first work that is broadly relevant to HRTs in Section 8.1, then specific work relevant to Chapters 5–7.

8.1 Hybrid Runtimes

In this section, I will begin by recounting a short history of kernel design in Section 8.1.1. I will then discuss more recent developments that are important for my work on HRTs (Section 8.1.2). Next, I will discuss OS designs specifically targeted for high-performance, large-scale machines (Section 8.1.3). Finally, I will outline recent research in parallel languages and runtimes (Section 8.1.4).

8.1.1 Kernel Design

Microkernels Starting with some of the ideas laid out in the *Nucleus* of the RC 4000 system [93], researchers began to question some of the design decisions commonly associated with monolithic operating systems. This effort eventually led to the development of microkernels, where only minimal functionality (e.g. threads, IPC, and memory management) merits inclusion in the kernel, and less critical functionality resides in userspace. The following excerpt from Liedtke [136] aptly summarizes the main requirement of a microkernel:

> A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e., permitting competing implementations, would prevent the implementation of the system's required functionality.

Notable examples of first generation microkernels include Mach [31], L3 [135], and Chorus [172]. Mach, which still has influence today in Apple's Darwin OS and GNU Hurd [43], was arguably the most successful of these, but the performance of microkernels, particularly for IPC, proved rather inadequate.

2^{nd} generation microkernels Second generation microkernels mostly aimed to address the performance issues that afflicted the previous generation. The L3 team produced a complete rewrite with L4 [134, 136, 95] and eschewed portability for the sake of performance. Notable features in L4 included recursive construction of address spaces, short messages passed in registers, and lazy scheduling. SPIN [30] was based on the latest version of Mach at the time, and allowed extensions written in a type-safe language (Modula-3) to be dynamically loaded into the kernel, which exported minimal functionality on which these extensions could build.

Early Microcomputer/PC OSes Early microcomputer and PC operating systems such as CP/M and MS-DOS, like Nautilus, had a nebulous separation between the kernel and the programs running on top of it. For example, CP/M had a *logical* separation from the kernel and application programs—a set number of *transient processes* [168] were pre-loaded into memory in the same address space as the kernel—but no strict privilege separation was enforced. Similarly, MS-DOS had applications run in supervisor mode along with the kernel. Such early OSes were intended to support general-purpose processing rather than high-performance and parallel computing.

Embedded and Real-Time OSes While the goals of embedded and real-time operating systems (RTOSes) differ at a high-level from lightweight kernels and HRTs, there are some similarities. One major similarity is the aim for deterministic performance. An RTOS must, for example, provide strict bounds on interrupt handling latency. Some RTOSes only execute a predetermined set of programs which execute with the same privilege of the kernel. For example, in μC/OS-II [125], application programs ("tasks") are allowed to enable and disable external interrupts, and memory protection is not enabled by default. In contrast to the monolithic architecture of μC/OS-II, the designers of the QNX RTOS chose a lightweight microkernel architecture [99]. Other notable RTOSes include VxWorks and Windows CE.

Exokernels In contrast to shifting components into and out of the privileged kernel, Exokernels [70, 71, 69, 112] promote the idea that the kernel should avoid providing *any* high-level abstractions for the hardware. For example, an exokernel, rather than abstracting the logical structure of a disk into files and file systems, will instead expose the *disk's* notion of blocks and sectors directly. The responsibility of providing higher-level abstractions falls on untrusted, user-mode *library OSes*. Nautilus is heavily influenced by Exokernel, but my work on HRTs is more focused on providing the right mechanisms and abstractions to parallel runtimes in particular.

Several exokernel derivatives exist that tout minimal size—even more so than microkernels—as their primary concern. Typically referred to as *picokernels* or *nanokernels*, they comprise a very thin kernel layer that can sometimes support other operating systems running on top of them. Important examples include KeyKOS (a highly influential, persistent, capability-based OS) [36], ADEOS [197], and the Stanford Cache Kernel [54]. These designs quickly become blurred with virtualization, and in many cases they overlap [180, 41].

Virtualization Virtualization has proven itself as a powerful, driving force in the design of operating systems since the early days of the IBM System/370 [89, 162] and more recently with the resurgence of virtualization in the commercial and scientific domains (e.g. VMware [42], Xen [18], KVM [121], and Palacios [130]). It has even been noted that VMMs embody many of the ideas championed by proponents of microkernels [92]. While a serious effort is underway to make virtualization viable for the HPC community [129, 124], it largely eschews both concerns about OS design specifically for parallel runtimes and questions of *which* hardware features should be exposed directly to the runtime. The runtime system is, first and foremost, limited by the choices of the *guest* OS. My previous work on GEARS [91] investigates the ability of the virtualization layer to address the shortcomings of this existing structure.

3^{rd} **generation microkernels** The third, more recent, generation of microkernels focuses on security [180] and the formal properties of the kernel API. The most notable example is EROS [175], which is a "clean-room" implementation of KeyKOS. seL4 [68], a successor of EROS and L4, is a recently developed microkernel that has a formally verified specification. Other projects exist that focus on correctness, resilience, and stability, including type-safe OSes (e.g. Singularity [103]). The concept of modularity plays a key role in the construction of operating systems, particularly form a software-engineering standpoint. The Flux OSKit [82] stands out as a noteworthy effort lowering the barriers of kernel design and development by providing components common to many kernel designs. These concerns, however, are largely orthogonal to the work on HRTs in this dissertation.

Single address space OS Nautilus currently runs with all threads sharing a single, global address space. This is, of course, not a new idea. Single Address Space Operating Systems (SASOSes) provide efficient and logically simple sharing between processes, as the meaning of a virtual address in one process has the same meaning in another. Protection is provided through hardware support (e.g. paging). More recent monolithic OSes such as Linux provide a limited form of global address spaces by virtue of shared page mappings. Notable examples of influential SASOSes include Opal (based on Mach 3.0) [53], Singularity [103], Scout [146], Nemesis[1] [171], Mungi [96, 64], and AmigaOS. Nautilus currently acts as a SASOS mostly for simplicity. If it turns out that a parallel runtime needs better logical separation between address spaces, the simplicity in the design will allow us to quickly modify it.

OSes for multiprocessors and multicores Operating systems have long touted support for multiprocessors [57]. Computer historians cite ILLIAC IV [19] as the first example of a large multiprocessing system[2]. I will discuss systems for large-scale supercomputers in more detail in Section 8.1.3. While much of the parallelism available in the large, expensive machines of these days came in the form of SIMD, the idea of using several units in parallel was highly influential. However, widespread use of this kind of par-

[1]Nemesis also supported *self-paging*, in which an application makes its own paging decisions.
[2]Thinking Machines' Connection Machine [100] is often cited as the first *massively* parallel machine.

allelism did not flourish until much later, with the end of Dennard scaling [65] and the increasing concern over power. These challenges eventually led to the idea of the chip multiprocessor (CMP) [156]. CMPs brought many new challenges, which the operating systems community faced head on. Scalability to many cores became a primary concern for OSes that had previously targeted general purpose machines[3] [38, 123]. Corey [37] addressed this issue by minimizing sharing among kernel data structures. Multikernels such as Barrelfish [174, 22] and later Barrelfish/DC [199] took a different approach and treated a chip multiprocessor as a distributed system of communicating cores. Tesselation OS [138, 55] splits processors into QoS domains called *Cells*, which are gang scheduled [72] on a group of hardware threads. Tesselation does this in order to enforce hard isolation between—and multiplex resources among—applications with disparate scheduling and resource requirements. Tesselation embodies some microkernel/nanokernel ideas in the way it defers fine-grained scheduling decisions to the cells themselves[4]. In contrast to this work, HRTs concentrate not on resource management for a broad range of applications, but rather for a specific class of parallel runtimes.

The above designs still impose abstractions on the applications (and runtimes) running on top of them. With HRTs, we can throw out these abstractions so that the runtime has more control, and can dictate the appropriate abstractions. Of course, these abstractions may share enough commonality to warrant inclusion in a kernel intended to support many parallel runtimes.

Rhoden and others at the UC Berkeley AMPLab are currently developing an OS called *Akaros* [198, 169], which shares many design goals with Nautilus, and thus merits a deeper discussion. They specifically claim that OS abstractions for parallel datacenter applications

[3]Incidentally, a similar renaissance occurred in the databases community [110, 111].
[4] This is logically similar to scheduler activations [10].

(at the node level) need rethinking. I agree. They plan to expose cores directly to the application, much like Tesselation. They also employ gang scheduling of tasks on cores, and allow applications to have more power over memory management, scheduling, and transparent access to block sizes for memory, disk, and network.

There are a few important differences that distinguish HRTs from the Akaros team's goals. First, we concentrate on parallel *runtimes*, and have a unique position to co-design the runtime *with* the kernel. Akaros still runs its many-core processes (MCPs) in user-mode. I believe that the right way to create good designs for parallel runtimes and the kernels that support them is to allow the runtime to have *full* control over the hardware[5].

Threading The scheduling of light-weight tasks goes hand in hand with multicore programming, and much research has focused on developing useful and efficient *threading packages*. POSIX threads (pthreads) [44] have long stood as the *de facto* standard for shared-memory parallel programming. OpenMP [59] builds on pthreads to by using compiler directives instead of library calls, making the process of porting sequential code significantly easier. OpenTM [16] extends OpenMP even further to support *transactional memory*, an alternative to synchronization between threads, one of the most difficult aspects of parallel programming.

Pthreads are an example of *kernel threads*, threads that must be created by issuing a request to the kernel. User-level thread packages advance a—typically faster—alternative model wherein the application or runtime maintains thread state and manages context switches between threads. User-level threads typically enjoy performance benefits but they suffer from a lack of control, especially in the face of blocking system calls. Many popular,

[5]In other words, the runtime is built with the *assumption* that it has full hardware control. Abstractions can come later, once the needs and priorities of the runtime layer are identified.

managed languages employ user-threads. Scheduler activations [10] address some of the shortcomings of user-level threads by allowing the OS to propagate events and information to the user-level scheduler. Qthreads [194] enhance user-level threads with *full/empty-bit* semantics, in which threads can wait on a location in memory to become either completely full or empty[6]. Lithe [158] advocates a different approach that has the OS expose hardware threads directly in lieu of a general-purpose scheduler[7]; this approach allows different parallel libraries, including pthreads, OpenMP, and Cilk to compose cleanly. Callisto [94] achieves a similar goal, but runs on top of a stock Linux kernel.

8.1.2 Revival of Exokernel Ideas

More recently, the ideas laid out in the exokernel work have enjoyed a resurgence, particularly as they relate to hardware virtualization. OSv [122] is essentially a small exokernel designed to run a single application in the cloud. Unikernels [140] also leverage the ubiquity and ease of use of virtual machines in order to compile application-specific libOSes that run in a dedicated VM. Drawbridge [163] and Bascule [23] similarly leverage the libOS concept specifically for Microsoft Windows libOSes, allowing them to run unmodified Windows applications. They also employ the notion of a *picoprocess*, which interacts with the underlying *security monitor* via a small set of stateless calls[8]. Dune uses hardware virtualization support to provide direct access to hardware features for an untrusted application running in a Linux host [25, 26]. Arrakis leverages virtualized I/O devices in a similar vain in order to allow applications direct access to hardware [159]. I, however,

[6]These semantics bear resemblance to *data-triggered threads* [190].

[7]Tesselation borrows heavily from Lithe in this regard.

[8]This form of *lightweight* virtualization has recently become popular with Linux Containers [142]. Of course, LXC was not the first of its kind. The idea is, at least in part, an obeisance to Solaris Containers [126], FreeBSD jails [116], chroot jails [83], Linux-VServer [66], and AIX Workload Partitions [4].

am not concerned *only* with hardware features that rely on the availability of hardware virtualization. I am also interested in exploring unconventional uses of other existing hardware features, especially as they relate to parallel runtimes. As far as I am aware, there is no work, recent or otherwise, that looks at hardware features and abstractions specifically for parallel runtimes.

8.1.3 Operating Systems Targeting HPC

The high-performance computing (HPC) community has been studying the effects of the OS on application performance for some time now. This work produced a general consensus that, especially in large-scale supercomputers, the OS should not “get in the way,” a sentiment that led the way for *light-weight kernels* (LWKs) such as Sandia’s Kitten [129], Catamount [118], IBM’s Compute Node Kernel (CNK) [86], and Cray’s Compute Node Linux (CNL). These LWKs are intended to provide minimal abstractions (only those required by the supported applications). By design, they eliminate non-deterministic behavior that can interfere with application performance. [74, 101, 75]. Current efforts exist to bridge LWKs with existing general-purpose OSes like Linux within the same system. Examples of these include the Hobbes project (in which we are involved) [39], mOS [195], Argo [24], and McKernel [177].

8.1.4 Parallel Languages and Runtimes

In this section, I will describe some of the influential research in parallel languages and runtime systems. I have split this discussion into parts, delineated by the domain in which each language/runtime is relevant. There is, of course, some overlap. First, I will discuss some of the more general-purpose languages and runtimes targeted at parallel systems.

Racket [77] is the most widely used Scheme dialect, partly developed here at Northwestern, that has recently integrated support for parallelism in the form of *futures* and *places* into its runtime [185, 184, 183]. The Manticore project seeks to bring the advantages of Concurrent ML to parallel systems without the performance impact of the synchronization mechanism in the existing implementations of CML [167, 80, 78]. The Manticore runtime has supported CML style constructs using, e.g. threads and continuations. Manticore is primarily an answer to multicore systems from the language developer's perspective [13, 27].

High performance computing The cornerstone of the high-performance community since the advent of large-scale multiprocessing has been the MPI library combined with C, C++, or Fortran. However, there has been significant effort from the language community in developing a better way to program large-scale supercomputers.

Beginning in the early 2000s, DARPA headed the High Productivity Computing Systems (HPCS) project, aimed at creating multi-petaflop systems. Several language innovations came out of this project, including Fortress, Chapel, and X10. Fortress (which is now effectively deprecated) [5], aimed to include the power of mathematical expressions in its syntax, and included support for implicit parallelism with work stealing. Chapel [51] supports task-parallelism, data-parallelism, and nested parallelism, and is based on High-performance Fortran (HPF). X10 is an object-oriented language that uses the concepts of *places* and *asynchronous tasks*, and supports message-passing style, fork-join parallelism, and bulk-synchronous parallel operation [52, 119]. The above examples all made use of a partitioned global address space (PGAS), which aim to bring the convenience and logical simplicity of shared memory programming to distributed-memory systems. Unified Parallel C (UPC) [56, 46] and Co-array Fortran [149] are other widely used examples of PGAS languages. The CHAOS/PARTI runtime libraries also introduced several runtime optimizations to support efficient, distributed shared memory for unstructured applications in the context of computational chemistry and aerodynamics [61, 181].

GPGPUs have brought SIMD machines to the mainstream, and the HPC community has acted as a frontrunner of their use. NVIDIA has developed a runtime library named CUDA [151, 153, 152] that allows programmers to vectorize their code for parallel operation on their GPUs. Intel has followed suit with the Xeon Phi series of accelerators [105, 108, 106] to target the same market. Here the innovations are more focused in the hardware domain. The programming model is more similar to using a library than using a special-purpose language. OpenACC [88] is an effort to integrate these accelerator libraries into a common API.

Languages based on message passing Message passing languages signal events and operations by using messages between objects or entities. Charm++ [115, 114] is one of the most popular languages in the HPC community that uses this model. Many large-scale scientific codes are built on top of Charm++, including NAMD, a popular molecular dynamics suite. More recently, the Charm group has investigated runtime optimizations specifically for multi-core clusters [141]. Erlang is an older language developed by Ericsson, specifically developed for communication systems. Its concurrency model naturally extends to parallel systems, and it has gained traction in recently in both commercial and scientific communities [11, 109, 47].

Data-parallel languages Data parallelism, closely related to the concept of SIMD, involves applying a single operation over a collection of data elements. High-performance Fortran [143, 120] was a popular data-parallel language. Haskell also has rich support for data parallelism [161, 50]. However, one of the main drawbacks of data parallelism is that it does not support *unstructured* parallelism. For example, using a traditional data parallel approach, it is not possible to write a parallel implementation of a recursive operation on a tree (e.g. quicksort). Blelloch addressed this challenge with *nested data-parallelism* [32] and the NESL language, which introduced the notion of *flattening* to a collection, allowing "siblings" in the recursion tree to be coalesced into a single collection. This allows vectors to be operated on in the traditional data-parallel manner. With SIMD machines gaining popularity again in the form of GPGPUs, there has been a revival of research in nested data-parallel languages [176, 28, 29]. Our implementation of NDPC (discussed in Chapter 3) is heavily influenced by this work.

Parallel runtimes This section focuses on systems in which the runtime, not the language itself, is the primary area of concern. SWARM is a new dataflow runtime built on top of pthreads aimed at low-latency task-parallelism [131]. HPX is a runtime system that supports the *ParalleX* execution model, in which, similar to dataflow, modifications to data can trigger events, such as thread creation, in the system [113]. As it uses communication driven by a global address space, ParalleX bears many similarities to the execution models of PGAS languages. Legion [21, 189] is a recently developed runtime aimed at heterogeneous architectures and scalability to many nodes. I discussed Legion in more detail in Chapter 3.

Cilk [35] is another highly influential alternative to traditional threads, which is currently available in a C++ flavor in the form of CilkPlus. Cilk only adds a few keywords to C (e.g. `spawn`); some of its most noteworthy aspects include its ability to extract parallelism from programmer-provided hints and its work-stealing scheduler. Jade [127, 170] is another parallel language that uses high-level information from the programmer. It is intended to make it easy for programmers to transform an existing sequential application for *coarse-grained*[9] concurrency using hints about task and data organization. The runtime can extract this information dynamically and apply machine specific optimizations transparent to the application[10].

While there is a huge swath of research aimed at parallel programming, none of these systems, *prima facie*, address runtime concerns at the kernel level; the focus on these concerns is a primary contribution of the work in this dissertation.

8.2 Nemo Event System

The real-time OS community has studied asynchronous events in depth, focusing on events in contexts such as predictable interrupt handling [166], priority inversion [173], and fast vectoring to user-level code [84]. However, this work does not consider the design of asynchronous events in a context where the application/runtime has kernel-mode privileges and full access to hardware, as in an HRT.

Thekkath and Levy introduced a mechanism [188] to implement *user-level* exception handlers—instances of synchronous events in our terminology—to mitigate the costs of the privilege transitions discussed at the beginning of Chapter 6. The motivation for this technique mirrors motivations for RDMA-based techniques we see used in practice today. In contrast, Nemo's design focuses on asynchronous events.

[9]That is, concurrency that involves tasks that will execute for a long enough time such that running the tasks concurrently does not introduce significant overhead.

[10]It should be clear now that Legion shares *many* ideas with the Jade system.

Horowitz introduced a programmable hardware technique for *Informing Memory Operations*, which vector to a user-level handler with minimal latency on cache miss events [102]. These events bear more similarities to exceptions than to asynchronous events, especially those originating at a remote core.

Keckler et al. introduced *concurrent event handling*, in which a multithreaded processor reserves slots for event handling in order to reduce overheads incurred from thread context switching [117]. Chaterjee discusses further details of this technique, particularly as it applies to MIT's J-Machine [148] and M-Machine [76]. A modern example of this technique can be found in the MONITOR and MWAIT instruction pair. I discuss how this type of technique differs from our goals in Section 6.1.2.

The Message Driven Processor (MDP), from which the J-Machine was built, had hardware contexts specifically devoted to handling message arrivals [60]. Furthermore, this processor had an instruction (`EXECUTE`) that could explicitly invoke an action on a remote node. This action could be a memory dereference, a call to a function, or a read/write to memory. This is essentially the capability that in Section 6.3 we suggested could be implemented in the context of x64 hardware. It is unfortunate that useful explicit messaging facilities like those used in the MDP—save some emerging and experimental hardware from Tilera (née RAW [132]) and the RAMP project [193]—have not made their way into commodity processors used in today's supercomputers, servers, and accelerators.

8.3 HVM and Multiverse

Other approaches to realizing the split-machine model for the HVM shown in Figure 3.1 exist. Dune, described in Section 8.1.2, provides one alternative. Guarded modules [90] could be used to give portions of a general-purpose virtualization model selective privileged access to hardware, including I/O devices. Pisces [154] would enable an approach that could eschew virtualization altogether by partitioning the hardware and booting multiple kernels simultaneously without virtualization.

Libra [8] bears similarities to our Multiverse system in its overall architecture. A Java Virtual Machine (JVM) runs on top of the Libra libOS, which in turn executes under virtualization. A general-purpose OS runs in a *controller partition* and accepts requests for legacy functionality from the JVM/Libra partition. This system involved a manual port, much like the work described in Chapter 3. However, the HVM gives us a more powerful mechanism for sharing between the ROS and HRT as they share a large portion of the address space. This allows us to leverage complex functionality in the ROS like shared libraries and symbol resolution. Furthermore, the Libra system does not provide a way to automatically create these specialized JVMs from their legacy counterparts.

The Blue Gene/L series of supercomputer nodes run with a Lightweight Kernel (LWK) called the Blue Gene/L Run Time Supervisor (BLRTS) [6] that shares an address space with applications and forwards system calls to a specialized I/O node. While the bridging mechanism between the nodes is similar, there is no mechanism for porting a legacy application to BLRTS. Others in the HPC community have proposed similar solutions that bridge a *full-weight* kernel with an LWK in a hybrid model. Examples of this approach include mOS [195], ARGO [24], and IHK/McKernel [177]. The Pisces Co-Kernel [157] treats performance isolation as its primary goal and can partition physical hardware between *enclaves*, or isolated OS/Rs that can involve different specialized OS kernels.

In contrast to the above systems, our HRT model is the only one that allows a runtime to act *as* a kernel, enjoying full privileged access to the underlying hardware. Furthermore, as far as we are aware, none of these systems provide an automated mechanism for producing an initial port to the specialized OS/R environment.

Chapter 9

Conclusion

I have presented the concept of the hybrid runtime (HRT), where a parallel runtime and a light-weight kernel framework are combined, and where the runtime has full access to privileged mode hardware and has full control over abstractions. This model addresses issues that runtimes face when running in user-space in the context of a general-purpose OS. I discussed the potential benefits of the HRT model applied to parallel runtimes. I also described the design of HRTs, as well as their implementation and evaluation. I showed how some of the benefits of HRTs can be realized simply by porting a parallel runtime do the HRT model, which for the case of the HPCG benchmark in Legion produced up to 40% speedup over its Linux counterpart.

At the outset of this work, the hypothesis was that building parallel runtimes as privileged kernel-mode entities can be valuable for performance and functionality. I found this to be supported by evidence, and I made several discoveries with respect to the HRT model along the way. Several of these discoveries led to systems that will continue to serve as foundations for further research in high-performance operating systems.

The key enabler of HRTs is a light-weight kernel framework called an Aerokernel.

We designed and implemented the first such Aerokernel, named *Nautilus*, of which I am the primary developer. Nautilus provides a starting point for parallel runtime developers by exposing several convenient and fast mechanisms familiar to user-space runtime developers, such as threads, memory management, events, and synchronization facilities. I demonstrated how these facilities perform much faster than those provided by a general-purpose OS like Linux, in some cases by several orders of magnitude. Nautilus currently boots on both commodity x86_64 hardware and on the Intel Xeon Phi. It is a Multiboot2-compliant kernel, and will boot on supported systems that have GRUB2 installed. It has been tested on AMD machines, Intel machines (including Xeon Phi cards), and on virtualization under QEMU, KVM, and our Palacios VMM.

Nautilus can also run in an environment *bridged* with a regular operating system (ROS), such as Linux in what is sometimes called a *multi-OS* environment. This is made possible by the hybrid virtual machine (HVM). In the HVM model, an HRT (based on Nautilus) can be created on demand from the ROS. The HRT controls a subset of the cores on the machine, and can be booted on the order of milliseconds (roughly the timescale of a process creation in Linux). The HVM allows the HRT to borrow legacy functionality from a ROS by forwarding events such as page faults and system calls to the ROS. The HVM is primarily aimed at easing the deployment of HRTs, allowing them to be used as software accelerators when high performance is paramount.

During my investigation of parallel runtimes (especially those that leverage software events), I discovered that the underlying mechanisms that Linux provides for such events are significantly slower than the capabilities of the hardware. To address this shortcoming, I implemented versions of familiar event facilities (such as condition variables) in Nautilus that perform orders of magnitude better than those on Linux. I also implemented event facilities with new semantics that perform at the hardware limit of IPIs, which is not possible in a ROS, where the parallel runtime is relegated to user-space operation. I collectively refer to these event facilities in Nautilus as *Nemo*.

Finally, I built the Multiverse toolchain, which allows runtime developers to more quickly adopt the HRT model. This was motivated by another discovery with regard to HRTs, namely that building them can be quite difficult. To ease the development process, Multiverse carries out a process called *automatic hybridization*, wherein a Linux user-space runtime is transformed into a HRT by rebuilding the code with our toolchain. Multiverse allows a user to execute a user-level runtime as normal at the command-line while automatically splitting the execution in an HVM between the ROS and the HRT. The runtime then executes *as a kernel*, enjoying privilege mode access. Multiverse introduces little to no overhead in the runtime's operation, while allowing the developer to begin experimenting with kernel-mode features. This allows the runtime developer to *start* with a working system rather than undertaking the arduous process of building an Aerokernel from scratch.

9.1 Summary of Contributions

Development of the HRT model I have developed the hybrid runtime (HRT) model for parallel runtimes, which allows such runtimes to operate with fully privileged access to the hardware and with full control over the abstractions to the machine. A HRT consists of a parallel runtime and a light-weight Aerokernel framework combined into one entity, and it can enable higher performance and more flexibility for the runtime.

Nautilus I am the primary designer and developer of the Nautilus Aerokernel framework. Nautilus is an open source project, freely available online under the MIT license.

Philix I designed and implemented Philix, a toolkit that allows OS developers to boot their custom OS on Intel's Xeon Phi. Philix provides an interactive console on the host machine so that the OS developer can easily see output of their OS running on the Phi. Philix is built on top of Intel's MPSS toolchain, and is freely available online.

Example HRTs The following runtimes have been ported to the HRT model, demonstrating its benefits:

- **Legion**: The Legion runtime ported to the HRT model outperforms its Linux counterpart by up to 40% on the HPCG mini-app. We also saw speedups on the example circuit simulator benchmark included in the Legion distribution.
- **NESL**: The NESL runtime ported to the HRT model showed that we could take an existing, complete parallel language from a user-space runtime to a kernel with reasonable effort, and with only a few hundred lines of code.
- **NDPC**: We co-designed the NDPC language (based on a subset of NESL) to explore using non-traditional kernel features within a runtime context, such as thread fork.
- **Hybridized Racket**: With the Multiverse toolchain, I automatically transformed the Racket runtime into an HRT. Racket has a complex runtime system which includes JIT compilation and garbage collection. This work showed that not only is automatically transforming such a complex runtime feasible, but also that its operation as a kernel is both feasible and beneficial.

Evaluation of Nautilus I carried out a detailed evaluation of the various mechanisms that Nautilus provides with extensive microbenchmarking. In most cases Nautilus outperforms Linux by significant margins.

Evaluation of Legion I evaluated our HRT port of Legion using the HPCG mini-app and Legion's circuit simulator example application. Just by porting, we saw considerable speedups for these Legion benchmarks. A simple change allowing control over interrupts (kernel-mode only access) also allowed non-trivial performance improvement.

Nemo I designed and implemented the Nemo event system within Nautilus. Nemo provides facilities that enjoy orders of magnitude lower event notification latency than what is possible in user-space Linux.

HVM I aided in the design and development of the hybrid virtual machine, which allows an HRT to be bridged with a ROS with low overheads.

Multiverse I designed and implemented the Multiverse toolchain, which enables the *automatic hybridization* of Linux user-space runtimes. I demonstrated the feasibility of automatic hybridization using the Racket runtime as an example.

Palacios I have contributed to various efforts to the open-source Palacios VMM, listed below:

- **Virtualized DVFS**: I aided Shiva Rao in his masters thesis work on virtualizing dynamic voltage and frequency scaling (DVFS) using Palacios. This system allows fine-grained control of the Dynamic Voltage and Frequency Scaling (DVFS) hardware during VM exits, leveraging inferred information about guests to make informed power management decisions.

- **Virtual HPET**: I implemented a virtual implementation of the High-Precision Event Timer, a fine-grained platform timer present on most contemporary high-performance hardware. I added support for the HPET to allow us to run experimental systems like OSv on Palacios.

- **QEMU backend**: QEMU provides a rich diversity of virtual devices. This is one contributor to the simplicity of, e.g. the KVM codebase, as it leverages these device implementations. I wanted to similarly be able to leverage these devices for Palacios. This system implements that functionality with a software bridge between Palacios and QEMU.

- **VMM-emulated RTM**: This was the first VMM-emulated implementation of the Restricted Transactional Memory (RTM) component of the Intel Transactional Synchronization Extensions (TSX). Its performance is roughly 60x relative to Intel's emulator. I implemented this system with Maciej Swiech.

- **Palacios on the Cray XK6**: I ported the Palacios VMM to run on the Cray XK6 series of supercomputer nodes. This comprised several bug fixes and enhancements to the Palacios codebase.

- **Other contributions**: I have also participated in regular development and maintenance of the Palacios codebase. This includes bug fixes, enhancements to the extension architecture, guest configuration and loading, software interrupt and system call interception, and others.

GEARS Guest Examination and Revision Services (GEARS) is a set of tools that allows developers to create guest-context virtual services, VMM-based services that extend into the guest. GEARS is implemented within Palacios, and is described in detail in Appendix B.

Guarded modules Guarded modules extend the concept of guest-context virtual services by granting them privileged access to hardware and VMM state. Guarded modules protect this privilege from the rest of the guest by maintaining a software border with compile-time and run-time techniques. More detail on guarded modules is presented in Appendix C.

9.2 Future Work

While the work in this dissertation has shown considerable promise for the HRT model, there remain many avenues of research yet to be explored. I outline some potential directions below.

9.2.1 Hybrid Runtimes

We have shown that the hybrid runtime model can increase performance in a parallel runtime without major changes to the runtime's operation. These increases can be attributed to the simplification of the underlying mechanisms in the Aerokernel and the elimination of user/kernel transitions. However, there is still untapped potential for HRTs that I have yet to explore.

New execution models By operating in kernel-mode, a parallel runtime can take full advantage of its control over the machine. This allows the runtime to implement new execution models from the ground up, rather than on top of abstractions enforced by a

ROS. I would like to explore new execution models that would not be possible in a ROS. One example is a data-flow runtime system that eschews the notion of threads entirely, operating on the basis of asynchronous data-flow events.

Unconventional hardware uses While I have explored leveraging interrupt control in the context of Legion and IPIs for events, there are many other hardware features that could be used in unconventional ways. These include system facilities like page tables, IOMMUs, floating point structures, debugging mechanisms, and many more.

Comparative study with Unikernels Unikernels, which are single-application, specialized kernels based on Exokernels, have risen to prominence in the research community in parallel with my work on hybrid runtimes. While Unikernels do not focus on parallelism, hybrid runtimes may benefit from some of the lessons learned as Unikernels reach maturity.

Debugging One of the primary drawbacks of hybrid runtimes, and indeed executing code with elevated privilege, is the difficulty that arises in debugging. I do not believe this is a *fundamental* weakness of HRTs. My work on Multiverse indicates that with the proper mechanisms, the convenience of a conventional OS environment can be combined with the high performance of an HRT. I would like to extend the bridging work of HVMs and Multiverse to the area of debugging.

9.2.2 Nautilus

At this point, Nautilus is still very much a prototype codebase. In addition to stability improvements, there are several additions to the codebase that I believe will enable further interesting research.

Multinode While my work on HRTs until now has primarily focused on node-level parallelism (which is becoming more important), the operation of a parallel runtime across many nodes in a large-scale machine introduces new challenges. I would like to explore ways to exploit Nautilus and the HRT model to improve the operation of parallel runtimes at massive scales.

File systems Currently, Nautilus does not have support for file systems. File systems, however, can play a major role in data-intensive parallel runtimes. While Nautilus can leverage a ROS filesystem via the HVM, I am interested in exploring how the HRT model can influence file system design. A parallel runtime's usage of the file system may be very different from applications in general, which may result in a radically different file system design.

Porting Currently, Nautilus runs on x86_64 and the Xeon Phi. There are several architectures that, if supported, would provide foundations for further research with Nautilus. Targets that I am particularly interested in include IBM POWER (e.g. for BlueGene/Q nodes) and Intel's newer Knights Landing (KNL) chips.

9.2.3 OS Architectures

There are several aspects of OS architecture that I believe warrant further research. I outline them below.

HRT-aware ROS Our current implementations of HVM and Multiverse allow bridged operation between an HRT and ROS. Currently, the ROS needs no modifications for this setup to work correctly. However, much like a paravirtualized guest in a virtual environment, it may be advantageous for the ROS to contain HRT-aware components in order to increase bridged performance or functionality.

Data-parallel kernels Much work has focused on designing kernels to expose task-level parallelism, mainly in the form of threads. While OS kernels typically provide support for data-parallel (SIMD) instruction set extensions, such as AVX, there are, as far as I am aware, no kernels designed specifically for data-parallelism, for example to support a data-parallel runtimes such as NESL or data-parallel Haskell. It is an open question whether or not an OS kernel itself can and should be designed as a data-parallel system.

9.2.4 Languages

Finally, I believe there is considerable room for research in both systems languages and parallel languages, especially for a model like HRT, where the runtime enjoys full access to hardware. For example, system software has largely been written in C for the past several decades. While C is a suitable tool for writing programs that require modeling of the underlying machine, a significant portion of OS code relies on C's ability to include inline assembly, especially for privileged mode components and setup of the machine. This presents an opportunity for improvement in language design, especially for languages used by OS and runtime developers.

References

[1] The computer language benchmarks game. http://benchmarksgame.alioth.debian.org/.

[2] OSU micro-benchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/.

[3] Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Wiegert. Intel virtualization technology for directed I/O. *Intel Technology Journal*, 10(3), August 2006.

[4] Jack Alford. AIX 6.1 workload partitions. http://www.ibm.com/developerworks/aix/library/au-workload, November 2007.

[5] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan Willem Maessen, Sukyoung Ryu, Guy L. Steele, and Sam Tobin-Hochstadt. The Fortress language specification. Technical Report Version 1.0, Sun Microsystems, March 2008.

[6] George Almási, Ralph Bellofatto, José Brunheroto, Călin Caşcaval, José Castaños, Luis Ceze, Paul Crumley, C. Christopher Erway, Joseph Gagliano, Derek Lieber, Xavier Martorell, José E. Moreira, Alda Sanomiya, and Karin Strauss. An overview of the BlueGene/L system software organization. In *Proceedings of the Euro-Par Conference on Parallel and Distributed Computing (EuroPar 2003)*, August 2003.

[7] AMD Corporation. *AMD64 Architecture Programmer's Manual Volume 2: Systems Programming*, May 2013.

[8] Glenn Ammons, Jonathan Appavoo, Maria Butrico, Dilma Da Silva, David Grove, Kiyokuni Kawachiya, Orran Krieger, Bryan Rosenburg, Eric Van Hensbergen, and Robert W. Wisniewski. Libra: A library operating system for a JVM in a virtualized execution environment. In *Proceedings of the 3^{rd} International Conference on Virtual Execution Environments (VEE 2007)*, pages 44–54, June 2007.

[9] Thomas Anderson and Michael Dahlin. *Operating Systems: Principles and Practice*. Recursive Books, second edition, 2012.

[10] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the* 13th *ACM Symposium on Operating Systems Principles (SOSP 1991)*, pages 95–109, October 1991.

[11] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice Hall, second edition, January 1996.

[12] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.91 edition, May 2015.

[13] Sven Auhagen, Lars Bergstrom, Matthew Fluet, and John Reppy. Garbage collection for multicore NUMA machines. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC 2011)*, pages 51–57, June 2011.

[14] Chang Bae, John Lange, and Peter Dinda. Enhancing virtualized application performance through dynamic adaptive paging mode selection. In *Proceedings of the* 8th *International Conference on Autonomic Computing (ICAC 2011)*, June 2011.

[15] Chang Seok Bae, John R. Lange, and Peter A. Dinda. Enhancing virtualized application performance through dynamic adaptive paging mode selection. In *Proceedings of the* 8th *International Conference on Autonomic Computing (ICAC 2011)*, pages 255–264, June 2011.

[16] Woongki Baek, Chi Cao Minh, Martin Trautmann, Christos Kozyrakis, and Kunle Olukotun. The OpenTM transactional application programming interface. In *Proceedings of the* 16th *International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 376–387, September 2007.

[17] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, 2003.

[18] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the* 19th *ACM Symposium on Operating Systems Principles (SOSP 2003)*, pages 164–177, October 2003.

[19] George H. Barnes, Richard M. Brown, Maso Kato, David J. Kuck, Daniel L. Slotnick, and Richard A. Stokes. The ILLIAC IV computer. *IEEE Transactions on Computers*, C-17(8):746–757, August 1968.

[20] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation caching: Skip, don't walk (the page table). In *Proceedings of the* 37th *Annual International Symposium on Computer Architecture (ISCA 2010)*, pages 48–59, June 2010.

[21] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of Supercomputing (SC 2012)*, November 2012.

[22] Andrew Baumann, Paul Barham, Pierre Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the* 22nd *ACM Symposium on Operating Systems Principles (SOSP 2009)*, pages 29–44, October 2009.

[23] Andrew Baumann, Dongyoon Lee, Pedro Fonseca, Lisa Glendenning, Jacob R. Lorch, Barry Bond, Reuben Olinsky, and Galen C. Hunt. Composing OS extensions safely and efficiently with Bascule. In *Proceedings of the* 8th *ACM European Conference on Computer Systems (EuroSys 2013)*, pages 239–252, April 2013.

[24] Pete Beckman. Argo: An exascale operating system. `http://www.mcs.anl.gov/project/argo-exascale-operating-system`.

[25] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *Proceedings of the* 10th *USENIX Conference on Operating Systems Design and Implementation (OSDI 2012)*, pages 335–348, October 2012.

[26] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the* 11th *USENIX Symposium on Operating Systems Design and Implementation (OSDI 2014)*, pages 49–65, October 2014.

[27] Lars Bergstrom. *Parallel Functional Programming with Mutable State*. PhD thesis, The University of Chicago, Chicago, IL, June 2013.

[28] Lars Bergstrom, Matthew Fluet, Mike Rainey, John Reppy, Stephen Rosen, and Adam Shaw. Data-only flattening for nested data parallelism. In *Proceedings of the* 18th *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2013)*, pages 81–92, February 2013.

[29] Lars Bergstrom and John Reppy. Nested data-parallelism on the GPU. In *Proceedings of the* 17th *ACM SIGPLAN International Conference on Functional Programming (ICFP 2012)*, pages 247–258, September 2012.

[30] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the* 15th *ACM Symposium on Operating Systems Principles (SOSP 1995)*, pages 267–283, December 1995.

[31] David L Black, David B Golub, Daniel P Julin, Richard F Rashid, Richard P Draves, Randall W Dean, Alessandro Forin, Joseph Barrera, Hideyuki Tokuda, Gerald Malan, et al. Microkernel operating system architecture and Mach. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 11–30, April 1992.

[32] Guy E. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-92-103, School of Computer Science, Carnegie Mellon University, January 1992.

[33] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

[34] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *Proceedings of the International Conference on Function Programming (ICFP)*, May 1996.

[35] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[36] Allen C. Bomberger, William S. Frantz, Ann C. Hardy, Norman Hardy, Charles R. Landau, and Jonathan S. Shapiro. The KeyKOS nanokernel architecture. In *Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, pages 95–112, April 1992.

[37] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI 2008)*, pages 43–57, December 2008.

[38] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of Linux scalability to many cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI 2010)*, October 2010.

[39] Ron Brightwell, Ron Oldfield, Arthur B. Maccabe, and David E. Bernholdt. Hobbes: Composition and virtualization as the foundations of an extreme-scale OS/R. In *Proceedings of the International Workshop on Runtime and Operating Systems for Supercomputers (ROSS 2013)*, pages 2:1–2:8, June 2013.

[40] Ron Brightwell, Kevin Pedretti, and Keith D. Underwood. Initial performance evaluation of the Cray SeaStar interconnect. In *Proceedings of the 13th Symposium on High Performance Interconnects (HOTI 2005)*, pages 51–57, August 2005.

[41] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. In *Proceedings of the* 16th *Symposium on Operating Systems Principles (SOSP 1997)*, pages 143–156, October 1997.

[42] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y. Wang. Bringing virtualization to the x86 architecture with the original VMware workstation. *ACM Transactions on Computer Systems*, 30(4):12:1–12:51, November 2012.

[43] Thomas Bushnell. Towards a new strategy of OS design. *GNU's Bulletin*, 1(16), January 1994.

[44] Dick Buttlar and Jacqueline Farrell. *Pthreads programming: A POSIX standard for better multiprocessing*. O'Reilly Media, Inc., 1996.

[45] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot: A technique for cheap recovery. In *Proceedings of the* 6th *USENIX Symposium on Operating Systems Design and Implementation (OSDI 2004)*, pages 31–44, December 2004.

[46] William W. Carlson, Jesse M. Draper, David E. Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, May 1999.

[47] Richard Carlsson. An introduction to core Erlang. In *Proceedings of the PLI 2001 Workshop on Erlang*, September 2001.

[48] Manuel Chakravarty, Gabriele Keller, Roman Leshchinskiy, and Wolf Pfannenstiel. Nepal—nested data-parallelism in haskell. In *Proceedings of the* 7th *International Euro-Par Conference (EUROPAR)*, August 2001.

[49] Manuel Chakravarty, Roman Leshchinskiy, Simon Peyon Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: A status report. In *Proceedings of the Workshop on Declarative Aspects of Multicore Programming*, January 2007.

[50] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel Haskell: A status report. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming (DAMP 2007)*, pages 10–18, January 2007.

[51] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, August 2007.

[52] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioğlu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the* 20th

ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005), pages 519–538, October 2005.

[53] Jeffrey S. Chase, Jeffrey S. Chase, Henry M. Levy, Henry M. Levy, Michael J. Feeley, Michael J. Feeley, Edward D. Lazowska, and Edward D. Lazowska. Sharing and protection in a single address space operating system. *ACM Transactions on Computer Systems*, 12(4):271–307, November 1994.

[54] David R. Cheriton and Kenneth J. Duda. A caching model of operating system kernel functionality. In *Proceedings of the* 1st *USENIX Symposium on Operating Systems Design and Implementation (OSDI 2004)*, pages 14:1–14:15, November 1994.

[55] Juan A. Colmenares, Gage Eads, Steven Hofmeyr, Sarah Bird, Miguel Moretó, David Chou, Brian Gluzman, Eric Roman, Davide B. Bartolini, Nitesh Mor, Krste Asanović, and John D. Kubiatowicz. Tessellation: Refactoring the OS around explicit resource containers with continuous adaptation. In *Proceedings of the* 50th *ACM/IEEE Design Automation Conference (DAC 2013)*, pages 76:1–76:10, May/June 2013.

[56] UPC Consortium. UPC language and library specifications, v1.3. Technical Report LBNL-6623E, Lawrence Berkeley National Lab, November 2013.

[57] F. J. Corbató and V. A. Vyssotsky. Introduction and overview of the Multics system. In *Proceedings of the 1965 Fall Joint Computer Conference, Part I (AFIPS 1965)*, pages 185–196, November 1965.

[58] Zheng Cui, Lei Xia, Patrick G. Bridges, Peter A. Dinda, and John R. Lange. Optimistic overlay-based virtual networking through optimistic interrupts and cut-through forwarding. In *Proceedings of Supercomputing (SC 2012)*, November 2012.

[59] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, 1998.

[60] William J. Dally, Roy Davison, J.A. Stuart Fiske, Greg Fyler, John S. Keen, Richard A. Lethin, Michael Noakes, and Peter R. Nuth. The message-driven processor: A multicomputer processing node with efficient mechanisms. *IEEE Micro*, 12(2):23–39, April 1992.

[61] Raja Das, Joel Saltz, and Alan Sussman. Applying the CHAOS/PARTI library to irregular problems in computational chemistry and computational aerodynamics. In *Proceedings of the 1993 Scalable Parallel Libraries Conference*, pages 45–56, October 1993.

[62] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–478, September 1994.

[63] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the* 6*th* *Symposium on Operating Systems Design and Implementation (OSDI 2004)*, December 2004.

[64] Luke Deller and Gernot Heiser. Linking programs in a single address space. In *Proceedings of the 1999 USENIX Annual Technical Conference (USENIX ATC 1999)*, pages 283–294, June 1999.

[65] Robert H. Dennard, Fritz H. Gaensslen, Hwa-Nien Yu, V. Leo Rideout, Ernest Bassous, and Andre R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, October 1974.

[66] Benoit des Ligneris. Virtualization of Linux based computers: the Linux-VServer project. In *Proceedings of the* 19*th* *International Symposium on High Performance Computing Systems and Applications (HPCS 2005)*, pages 340–346, May 2005.

[67] Jack Dongarra and Michael A. Heroux. Toward a new metric for ranking high performance computing systems. Technical Report SAND2013-4744, University of Tennessee and Sandia National Laboratories, June 2013.

[68] Kevin Elphinstone and Gernot Heiser. From L3 to seL4—what have we learnt in 20 years of L4 microkernels? In *Proceedings of the* 24*th* *ACM Symposium on Operating Systems Principles (SOSP 2013)*, pages 133–150, November 2013.

[69] Dawson R. Engler. *The Exokernel Operating System Architecture*. PhD thesis, Massachusetts Institute of Technology, October 1998.

[70] Dawson R. Engler and M. Frans Kaashoek. Exterminate all operating system abstractions. In *Proceedings of the* 5*th* *Workshop on Hot Topics in Operating Systems (HotOS 1995)*, pages 78–83, May 1995.

[71] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the* 15*th* *ACM Symposium on Operating Systems Principles (SOSP 1995)*, pages 251–266, December 1995.

[72] Dror G. Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16(4):306–318, December 1992.

[73] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. The Racket Manifesto. In 1*st* *Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 113–128, 2015.

[74] Kurt B. Ferreira, Patrick Bridges, and Ron Brightwell. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *Proceedings of Supercomputing (SC 2008)*, November 2008.

[75] Kurt B. Ferreira, Patrick G. Bridges, Ron Brightwell, and Kevin T. Pedretti. Impact of system design parameters on application noise sensitivity. *Journal of Cluster Computing*, 16(1), March 2013.

[76] Marco Fillo, Stephen W. Keckler, William J. Dally, Carter Nicholas P., Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The M-Machine multicomputer. In *Proceedings of the 29th Annual International Symposium on Microarchitecture (MICRO 29)*, pages 146–156, November 1995.

[77] Matthew Flatt and PLT. The Racket reference—version 6.1. Technical Report PLT-TR-2010-1, August 2014.

[78] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly-threaded parallelism in Manticore. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP 2008)*, pages 119–130, September 2008.

[79] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly threaded parallelism in manticore. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, September 2008.

[80] Matthew Fluet, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. Manticore: A heterogeneous parallel language. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming (DAMP 2007)*, pages 37–44, January 2007.

[81] Matthew Fluet, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. Manticore: A heterogeneous parallel language. In *Proceedings of the Workshop on Declarative Aspects of Multicore Programming*, January 2007.

[82] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP 1997)*, pages 14–19, October 1997.

[83] Steve Friedl. Go directly to jail: Secure untrusted applications with chroot. *Linux Magazine*, December 2002.

[84] Gerald Fry and Richard West. On the integration of real-time asynchronous event handling mechanisms with existing operating system services. In *Proceedings of the 2007 International Conference on Embedded Systems and Applications (ESA 2007)*, June 2007.

[85] Alexandros V. Gerbessiotis and Leslie G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22(2):251–267, August 1994.

[86] Mark Giampapa, Thomas Gooding, Todd Inglett, and Robert W. Wisniewski. Experiences with a lightweight supercomputer kernel: Lessons learned from Blue Gene's CNK. In *Proceedings of Supercomputing (SC 2010)*, November 2010.

[87] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. ELI: Bare-metal performance for I/O virtualization. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)*, pages 411–422, March 2012.

[88] OpenACC Working Group et al. The OpenACC application programming interface, version 1.0. `http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf`, November 2011.

[89] P. H. Gum. System/370 extended architecture: Facilities for virtual machines. *IBM Journal of Research and Development*, 27(6):530–544, November 1983.

[90] Kyle C. Hale and Peter A. Dinda. Guarded modules: Adaptively extending the VMM's privilege into the guest. In *Proceedings of the 11th International Conference on Autonomic Computing (ICAC 2014)*, pages 85–96, June 2014.

[91] Kyle C. Hale, Lei Xia, and Peter A. Dinda. Shifting GEARS to enable guest-context virtual services. In *Proceedings of the 9th International Conference on Autonomic Computing (ICAC 2012)*, pages 23–32, September 2012.

[92] Steven Hand, Andrew Warfield, Keir Fraser, Evangelos Kostovinos, and Dan Magenheimer. Are virtual machine monitors microkernels done right? In *Proceedings of the 10th Workshop on Hot Topics on Operating Systems (HotOS 2005)*, June 2005.

[93] Per Brinch Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4):238–241, April 1970.

[94] Tim Harris, Martin Maas, and Virendra J. Marathe. Callisto: Co-scheduling parallel runtime systems. In *Proceedings of the 9th ACM European Conference on Computer Systems (EuroSys 2014)*, pages 24:1–24:14, April 2014.

[95] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Jean Wolter, and Sebastian Schönberg. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP 1997)*, pages 66–77, October 1997.

[96] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, and Stephen Russell. The Mungi single-address-space operating system. *Software: Practice and Experience*, 28(9):901–928, July 1998.

[97] Michael A. Heroux, Jack Dongarra, and Piotr Luszczek. HPCG technical specification. Technical Report SAND2013-8752, Sandia National Laboratories, October 2013.

[98] High Performance Fortran Forum. High Performance Fortran language specification, version 2.0. Technical report, Center for Research on Parallel Computation, Rice University, January 1996.

[99] Dan Hildebrand. An architectural overview of QNX. In *Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, April 1992.

[100] William Daniel Hills. *The Connection Machine*. PhD thesis, Massachusetts Institute of Technology, June 1985.

[101] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. Characterizing the influence of system noise on large-scale applications by simulation. In *Proceedings of Supercomputing (SC 2010)*, November 2010.

[102] Mark Horowitz, Margaret Martonosi, Todd C. Mowry, and Michael D. Smith. Informing memory operations: Providing memory performance feedback in modern processors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA 1996)*, pages 260–270, May 1996.

[103] Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. *SIGOPS Operating Systems Review*, 41(2):37–49, April 2007.

[104] InfiniBand Trade Association. *InfiniBand Architecture Specification: Release 1.0*. InfiniBand Trade Association, 2000.

[105] Intel Corporation. *Intel® Xeon Phi™ Coprocessor Developer's Quick Start Guide Version 1.7*.

[106] Intel Corporation. *Intel® Coprocessor Instruction Set Architecture Reference Manual*, September 2012.

[107] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B & 3C): System Programming Guide*, February 2014.

[108] Intel Corporation. *Intel® Xeon Phi™ Coprocessor System Software Developer's Guide*, March 2014.

[109] Erik Johansson, Mikael Pettersson, and Konstantinos Sagonas. A high performance Erlang system. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 2000)*, pages 32–43, September 2000.

[110] Ryan Johnson, Nikos Hardavellas, Ippokratis Pandis, Naju Mancheril, Stavros Harizopoulos, Kivanc Sabirli, Anastassia Ailamaki, and Babak Falsafi. To share or not to share? In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB 2007)*, pages 351–362, September 2007.

[111] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-MT: A scalable storage manager for the multicore era. In *Proceedings of the* 12th *International Conference on Extending Database Technology (EDBT 2009)*, pages 24–35, March 2009.

[112] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on Exokernel systems. In *Proceedings of the* 16th *ACM Symposium on Operating Systems Principles (SOSP 1997)*, pages 52–65, October 1997.

[113] Harmut Kaiser, Maciej Brodowicz, and Thomas Sterling. ParalleX: An advanced parallel execution model for scaling-impaired applications. In *Proceedings of the* 38th *International Conference on Parallel Processing Workshops (ICPPW 2009)*, pages 394–401, September 2009.

[114] Laxmikant V. Kalé, Balkrishna Ramkumar, Amitabh Sinha, and Attila Gursoy. The Charm parallel programming language and system: Part II–the runtime system. Technical Report 95-03, Parallel Programming Laboratory, University of Illinois at Urbana-Champaign, 1994.

[115] Laxmikant V. Kalé, Balkrishna Ramkumar, Amitabh Sinha, and Attila Gursoy. The Charm parallel programming language and system: Part I–description of language features. Technical Report 95-02, Parallel Programming Laboratory, University of Illinois at Urbana-Champaign, 1995.

[116] Poul-Henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *Proceedings of the* 2nd *International System Administration and Networking Conference (SANE 2000)*, May 2000.

[117] Stephen W. Keckler, Andrew Chang, Whay S. Lee, Sandeep Chatterjee, and William J. Dally. Concurrent event handling through multithreading. *IEEE Transactions on Computers*, 48(9):903–916, September 1999.

[118] Suzanne M. Kelly and Ron Brightwell. Software architecture of the light weight kernel, Catamount. In *Proceedings of the 2005 Cray User Group Meeting (CUG 2005)*, May 2005.

[119] Ebcioğlu Kemal, Vijay Saraswat, and Vivek Sarkar. X10: Programming for hierarchical parallelism and non-uniform data access. In *Proceedings of the* 3rd *International Workshop on Language Runtimes*, October 2004.

[120] Ken Kennedy, Charles Koelbel, and Hans Zima. The rise and fall of High Performance Fortran: An historical object lesson. In *Proceedings of the* 3rd *ACM SIGPLAN Conference on History of Programming Languages (HOPL 3)*, pages 7:1–7:22, June 2007.

[121] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium*, pages 225–230, June 2007.

[122] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. OSv—optimizing the operating system for virtual machines. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC 2014)*, June 2014.

[123] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. K42: Building a complete operating system. In *Proceedings of the 1st ACM European Conference on Computer Systems (EuroSys 2006)*, pages 133–145, April 2006.

[124] Alexander Kudryavtsev, Vladimir Koshelev, Boris Pavlovic, and Arutyun Avetisyan. Modern HPC cluster virtualization using KVM and Palacios. In *Proceedings of the Workshop on Cloud Services, Federation, and the 8th Open Cirrus Summit (FederatedClouds 2012)*, pages 1–9, September 2012.

[125] Jean J. Labrosse. μC/OS-II the real-time kernel: User's manual. `https://doc.micrium.com/display/osiidoc?preview=/10753158/20644170/100-uC-OS-II-003.pdf`, 2015. Accessed: 2016-09-07.

[126] Menno Lageman. Solaris containers—what they are and how to use them. *Sun BluePrints OnLine*, May 2005.

[127] Monica S. Lam and Martin C. Rinard. Coarse-grain parallel programming in Jade. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 1991)*, pages 94–105, April 1991.

[128] John Lange, Kevin Pedretti, Peter Dinda, Patrick Bridges, Chang Bae, Philip Soltero, and Alexander Merritt. Minimal overhead virtualization of a large scale supercomputer. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2011)*, pages 169–180, March 2011.

[129] John Lange, Kevin Pedretti, Trammell Hudson, Peter Dinda, Zheng Cui, Lei Xia, Patrick Bridges, Andy Gocke, Steven Jaconette, Mike Levenhagen, and Ron Brightwell. Palacios and Kitten: New high performance operating systems for scalable virtualized and native supercomputing. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010)*, April 2010.

[130] John R. Lange, Peter Dinda, Kyle C. Hale, and Lei Xia. An introduction to the Palacios virtual machine monitor—version 1.3. Technical Report NWU-EECS-11-10, Department of Electrical Engineering and Computer Science, Northwestern University, November 2011.

[131] Christopher Lauderdale and Rishi Khan. Towards a codelet-based runtime for exascale computing. In *Proceedings of the* 2nd *International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era (EXADAPT 2012)*, pages 21–26, March 2012.

[132] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-time scheduling of instruction-level parallelism on a raw machine. In *Proceedings of the* 8th *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 1998)*, pages 46–57, October 1998.

[133] Bo Li, Jianxin Li, Tianyu Wo, Chunming Hu, and Liang Zhong. A VMM-based system call interposition framework for program monitoring. In *Proceedings of the* 16t^{th} *IEEE International Conference on Parallel and Distributed Systems (ICPADS 2010)*, pages 706–711, December 2010.

[134] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the* 14th *ACM Symposium on Operating Systems Principles (SOSP 1993)*, pages 175–188, December 1993.

[135] Jochen Liedtke. A persistent system in real use—experiences of the first 13 years. In *Proceedings of the* 3rd *International Workshop on Object-Orientation in Operating Systems*, pages 2–11, December 1993.

[136] Jochen Liedtke. On micro-kernel construction. In *Proceedings of the* 15th *ACM Symposium on Operating Systems Principles (SOSP 1995)*, pages 237–250, December 1995.

[137] Jiuxing Liu, Wei Huang, Bulent Abali, and Dhabaleswar Panda. High performance VMM-bypass I/O in virtual machines. In *Proceedings of the USENIX Annual Technical Conference*, May 2006.

[138] Rose Liu, Kevin Klues, Sarah Bird, Steven Hofmeyr, Krste Asanović, and John Kubiatowicz. Tessellation: Space-time partitioning in a manycore client OS. In *Proceedings of the* 1st *USENIX Conference on Hot Topics in Parallelism (HotPar 2009)*, pages 10:1–10:6, March 2009.

[139] Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[140] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the* 18th *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2013)*, pages 461–472, March 2013.

[141] Chao Mei, Gengbin Zheng, Filippo Gioachin, and Laxmikant V. Kalé. Optimizing a parallel runtime system for multicore clusters: A case study. In *Proceedings of the 2010 TeraGrid Conference (TG 2010)*, pages 12:1–12:8, August 2010.

[142] Paul B Menage. Adding generic process containers to the Linux kernel. In *Proceedings of the Linux Symposium*, pages 45–58, June 2007.

[143] John Merlin and Anthony Hey. An introduction to High Performance Fortran. *Scientific Programming*, 4(2):87–113, April 1995.

[144] Timothy Merrifield and H. Reza Taheri. Performance implications of extended page tables on virtualized x86 processors. In *Proceedings of the* 12th *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2016)*, pages 25–35, April 2016.

[145] Jeffrey C Mogul, Andrew Baumann, Timothy Roscoe, and Livio Soares. Mind the gap: reconnecting architecture and os research. In *Proceedings of the* 13th *Workshop on Hot Topics in Operating Systems (HotOS 2011)*, May 2011.

[146] Allen B. Montz, David Mosberger, Sean W. O'Malley, Larry L. Peterson, and Todd A. Proebsting. Scout: A communications-oriented operating system. In *Proceedings of the* 5th *Workshop on Hot Topics in Operating Systems (HotOS 1995)*, pages 58–61, May 1995.

[147] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3), August 2006.

[148] Michael D. Noakes, Deborah A. Wallach, and William J. Dally. The j-machine multicomputer: An architectural evaluation. In *Proceedings of the* 20th *Annual International Symposium on Computer Architecture (ISCA 1993)*, pages 224–235, May 1993.

[149] Robert W. Numrich and John Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.

[150] NVIDIA Corporation. Dynamic parallelism in CUDA, December 2012.

[151] NVIDIA Corporation. *CUDA C Programming Guide—Version 6.0*, February 2014.

[152] NVIDIA Corporation. *CUDA Driver API—Version 6.0*, February 2014.

[153] NVIDIA Corporation. *CUDA Runtime API—Version 6.0*, February 2014.

[154] Jiannan Oayang, Brian Kocoloski, John Lange, and Kevin Pedretti. Achieving performance isolation with lightweight co-kernels. In *Proceedings of the* 24th *International ACM Symposium on High Performance Parallel and Distributed Computing, (HPDC 2015)*, June 2015.

[155] Yoshinori K. Okuji, Bryan Ford, Erich Stefan Boleyn, and Kunihiro Ishiguro. The multiboot specification—version 1.6. Technical report, Free Software Foundation, Inc., 2010.

[156] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *Proceedings of the* 7th *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 7)*, pages 2–11, October 1996.

[157] Jiannan Ouyang, Brian Kocoloski, John R. Lange, and Kevin Pedretti. Achieving performance isolation with lightweight co-kernels. In *Proceedings of the* 24th *International Symposium on High-Performance Parallel and Distributed Computing (HPDC 2015)*, pages 149–160, June 2015.

[158] Heidi Pan, Benjamin Hindman, and Krste Asanović. Composing parallel software efficiently with Lithe. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2010)*, pages 376–387, June 2010.

[159] Simon Peter and Thomas Anderson. Arrakis: A case for the end of the empire. In *Proceedsings of the* 14th *Workshop on Hot Topics in Operating Systems (HotOS 2013)*, pages 26:1–26:7, May 2013.

[160] Simon Peter, Jialin Li, Irene Zhang, Dan R.K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *Proceedings of the* 11th *USENIX Symposium on Operating Systems Design and Implementation (OSDI 2014)*, pages 1–16, October 2014.

[161] Simon Peyton Jones. Harnessing the multicores: Nested data parallelism in Haskell. In *Proceedings of the* 6th *Asian Symposium on Programming Languages and Systems (APLAS 2008)*, December 2008.

[162] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, July 1974.

[163] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the library OS from the top down. In *Proceedings of the* 16th *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2011)*, pages 291–304, March 2011.

[164] Qumranet Corporation. Kvm - kernel-based virtual machine. Technical report, 2006. KVM has been incorporated into the mainline Linux kernel codebase.

[165] Himanshu Raj and Karsten Schwan. High performance and scalable I/O virtualization via self-virtualized devices. In *Proceedings of the* 16th *IEEE International Symposium on High Performance Distributed Computing (HPDC 2007)*, July 2007.

[166] Paul Regnier, George Lima, and Luciano Barreto. Evaluation of interrupt handling timeliness in real-time linux operating systems. *ACM SIGOPS Operating Systems Review*, 42(6):52–63, October 2008.

[167] John Reppy, Claudio V. Russo, and Yingqi Xiao. Parallel concurrent ML. In *Proceedings of the* 14th *ACM SIGPLAN International Conference on Functional Programming (ICFP 2009)*, pages 257–268, August 2009.

[168] Digital Research. Concurrent CP/M operating system: System guide. `http://www.cpm.z80.de/manuals/ccpmsg.pdf`, 1984. Accessed: 2016-09-07.

[169] Barret Rhoden, Kevin Klues, David Zhu, and Eric Brewer. Improving per-node efficiency in the datacenter with new OS abstractions. In *Proceedings of the* 2nd *ACM Symposium on Cloud Computing (SoCC 2011)*, pages 25:1–25:8, October 2011.

[170] Martin C. Rinard, Daniel J. Scales, and Monica S. Lam. Jade: A high-level, machine-independent language for parallel programming. *IEEE Computer*, 26(6):28–38, June 1993.

[171] Timothy Roscoe. Linkage in the Nemesis single address space operating system. *ACM SIGOPS Operating Systems Review*, 28(4):48–55, October 1994.

[172] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. CHORUS distributed operating systems. *Computing Systems*, 1(4):305–370, Fall 1988.

[173] Fabian Scheler, Wanja Hofer, Benjamin Oechslein, Rudy Pfister, Wolfgang Shröder-Preikschat, and Daniel Lohmann. Parallel, hardware-supported interrupt handling in an event-triggered real-time operating system. In *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2009)*, pages 167–174, December 2009.

[174] Adrian Schüpbach, Simon Peter, Andrew Baumann, and Timothy Roscoe. Embracing diversity in the Barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-core Systems (MMCS 2008)*, June 2008.

[175] Jonathan S. Shapiro and Norman Hardy. EROS: A principle-driven operating system from the ground up. *IEEE Software*, 19(1):26–33, January 2002.

[176] Adam Michael Shaw. *Implementation Techniques for Nested-Data-Parallel Languages*. PhD thesis, The University of Chicago, Chicago, IL, August 2011.

[177] Taku Shimosawa, Balazs Gerofi, Masamichi Takagi, Gou Nakamura, Tomoki Shirasawa, Yuji Saeki, Masaaki Shimizu, Atsushi Hori, and Yutaka Ishikawa. Interface for heterogeneous kernels: A framework to enable hybrid os designs targeting high performance computing on manycore architectures. In *Proceedings of the IEEE International Conference on High Performance Computing (HiPC 2014)*, December 2014.

[178] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley, ninth edition, 2012.

[179] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, first edition, 2005.

[180] Udo Steinberg and Bernhard Kauer. NOVA: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th ACM European Conference on Computer Systems (EuroSys 2010)*, pages 209–222, April 2010.

[181] Alan Sussman and Joel Saltz. A manual for the multiblock PARTI runtime primitives, revision 4. Technical Report UMIACS-TR-93-36, University of Maryland Institute for Advanced Computer Studies, College Park, Maryland, 1993.

[182] J. Swaine, K. Tew, P. Dinda, R. Findler, and M. Flatt. Back to the futures: Incremental parallelization of existing sequential runtime systems. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2010)*, October 2010.

[183] James Swaine. *Incremental Parallelization of Existing Sequential Runtime Systems*. PhD thesis, Northwestern University, Evanston, Illinois, June 2014.

[184] James Swaine, Burke Fetscher, Vincent St-Amour, Robert Bruce Findler, and Matthew Flatt. Seeing the futures: Profiling shared-memory parallel Racket. In *Proceedings of the 1st ACM SIGPLAN Workshop on Functional High-performance Computing (FHPC 2012)*, September 2012.

[185] James Swaine, Kevin Tew, Peter Dinda, Robert Bruce Findler, and Matthew Flatt. Back to the futures: Incremental parallelization of existing sequential runtime systems. In *Proceedings of the 2nd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2010)*, pages 583–597, October 2010.

[186] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Pearson, fourth edition, 2014.

[187] Kevin Tew, James Swaine, Matthew Flatt, Robert Findler, and Peter Dinda. Places: Adding message passing parallelism to racket. In *Proceedings of the 2011 Dynamic Languages Symposium (DLS 2011)*, October 2011.

[188] Chandramohan A. Thekkath and Henry M. Levy. Hardware and software support for efficient exception handling. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 1994)*, pages 110–119, October 1994.

[189] Sean Treichler, Michael Bauer, and Alex Aiken. Language support for dynamic, hierarchical data partitioning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2013)*, pages 495–514, October 2013.

[190] Hung-Wei Tseng and Dean M. Tullsen. Data-triggered threads: Eliminating redundant computation. In *Proceedings of the* 17^{th} *International Symposium on High Performance Computer Architecture (HPCA 2011)*, pages 181–192, February 2011.

[191] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrating communication and computation. In *Proceedings of the* 25^{th} *Annual International Symposium on Computer Architecture (ISCA 1998)*, pages 430–440, July 1998.

[192] Carl Waldsburger. Memory resource management in vmware esx server. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[193] John Wawrzynek, David Patterson, Mark Oskin, Shih-Lien Lu, Christoforos Kozyrakis, James C. Hoe, Derek Chiou, and Krste Asanović. Ramp: Research accelerator for multiple processors. *IEEE Micro*, 27(2):46–57, March 2007.

[194] Kyle B Wheeler, Richard C Murphy, and Douglas Thain. Qthreads: An API for programming with millions of lightweight threads. In *Proceedings of the* 22^{nd} *International Symposium on Parallel and Distributed Processing (IPDPS 2008)*, April 2008.

[195] Robert W. Wisniewski, Todd Inglett, Pardo Keppel, Ravi Murty, and Rolf Riesen. mOS: An architecture for extreme-scale operating systems. In *Proceedings of the* 4^{th} *International Workshop on Runtime and Operating Systems for Supercomputers (ROSS 2014)*, pages 2:1–2:8, June 2014.

[196] Lei Xia, Zheng Cui, John Lange, Yuan Tang, Peter Dinda, and Patrick Bridges. VNET/P: Bridging the cloud and high performance computing through fast overlay networking. In *Proceedings of* 21^{st} *International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC 2012)*, pages 259–270, June 2012.

[197] Karim Yaghmour. Adaptive domain environment for operating systems. `http://www.opersys.com/ftp/pub/Adeos/adeos.pdf`.

[198] Matei Zaharia, Benjamin Hindman, Andy Konwinski, Ali Ghodsi, Anthony D. Joesph, Randy Katz, Scott Shenker, and Ion Stoica. The datacenter needs an operating system. In *Proceedings of the* 3^{rd} *USENIX Conference on Hot Topics in Cloud Computing (HotCloud 2011)*, pages 17:1–17:5, August 2011.

[199] Gerd Zellweger, Simon Gerber, Kornilios Kourtis, and Timothy Roscoe. Decoupling cores, kernels, and operating systems. In *Proceedings of the* 11^{th} *USENIX Symposium*

on Operating Systems Design and Implementation (OSDI 2014), pages 17–31, October 2014.

Appendix A

OS Issues as Stated by Legion Runtime Developers

Below is a list of features and requests for OSes received from the Legion runtime developers. They give a sense for some of the problems with building a runtime atop a generalized OS interface. Each block delineates comments from a different developer.

Developer 1

Some quick thoughts:

1. All operations should come in non-blocking form (i.e. in the traditional "non-blocking" definition used by OSes - return `E_AGAIN` rather than blocking), ideally with an optional semaphore/futex/whatever that the OS can use to signal when an operation has a non-zero chance of success (i.e. when it's worth re-attempting the request).

2. Mechanisms for pinning memory that can be used by multiple hardware devices need to be improved. Some of this is on the GPU/NIC/etc. drivers but anything the OS can do to help would be nice.

3. The overhead of `SIGSEGV` handling is preventing us from trying any `mmap()` shenanigans to implement lazy load and/or copy-on-write semantics for instances. A lower-overhead way to trap those accesses and monkey with page tables would be great.

4. An OS-level API for letting runtimes acquire hardware resources (e.g. CPU cores, pinnable memory) would be the way we'd want to interact with other runtimes. When we "own" the resource, we have the whole thing, but we can scale our usage up/down in a fine-grained way (i.e. better than cpuset wrappers on the command line and `OMP_NUM_THREADS` environment variables). There's a Berkeley project that worked on this about 5 years ago that I'm blanking on the name of[1].

5. More detailed affinity information. Dynamic power information on cores/memories. Neither of these need to be perfect—higher fidelity is obviously better.

6. For deep memory hierarchies, we just need the low-level control of allocation and data movement. For resilience, an API that: a. allows the HW/OS to let the runtime/app know about faults (with as much information as is available) b. allows the runtime/app to tell the HW/OS if/how it was handled, and whether or not the OS (or the HW) should pull the fire alarm.

[1]He is almost certainly referring to Akaros [198, 169], which I discussed in Section 8.1.1.

7. Just low-level control of all the physical resources in the system (processors, memories, caches, etc.). We'd rather do the virtualization ourselves (modulo the answer to #2 above).

Developer 2

1. From a runtime developer's perspective, the way that current operating systems manage resources is fundamentally broken. For example, pinned memory right now is handled by the OS in Linux, which means that it is very hard for us in Legion to create a pinned allocation of memory that is visible to both the CUDA and Network drivers (either IB or Cray) because they both want to make the system call that does the pinning. Instead I think a better model is to move resource allocation up into user space so the runtime can make it visible with a nice abstraction (Legion mapper) using an approach similar to how exokernels work (`http://pdos.csail.mit.edu/exo/`). The OS should provide safety for resources, but management and policy should be available to both the runtime and ultimately to the application instead of being baked into the OS implementation the way it is today.

2. Similar in spirit to what I mentioned above, control data structures like OS page tables should be available to the runtime system. On the long-term list of interesting things to do with Legion is to use something like Dune (`http://dune.scs.stanford.edu/`) to manipulate the virtual to physical mapping of pages to support things like copy-on-write instances of physical regions. It would also allow us to do things like de-duplicate copies of

data (especially for ghost cell regions), as long as we knew that regions were being accessed with read-only privileges. Again this fits into the idea that like the runtime system, the OS should provide mechanism and not policy like it does today.

3. Error handling and fault detection in operating systems today are fundamentally broken for distributed systems. If there is data corruption detected on a checksum for a packet sent over the interconnect, which node handles that, source or destination? Furthermore, assuming error detection gets significantly better, using signals to interrupt processes every time a fault is detected just seems broken. Also, the way signal handlers run today aren't really designed for multi-threaded systems and so they employ a stop-the-world approach that halts all threads within a process. In a world where we think of memory as one big heap that is probably the only way to do things. However, in a world where the runtime knows about which tasks are accessing which regions, the runtime will only want to restart tasks accessing corrupted regions without ever halting other tasks. In this way, something like Legion could be better at hiding the latency of handling faults as well, assuming the OS doesn't stop the whole world.

4. Consistent with the ideas above, when it comes to scheduling the OS should provide mechanism and not policy. Threads should be scheduled directly onto hardware, with maybe only very coarse grained thread scheduling to guarantee forward progress. The application should be able to easily specify things like priorities and hints about where to place

threads onto hardware cores. The application should also have the ability to pin a thread to a core and never give it up. At most, the OS should assume that it can own one core in order to handle system calls. The application should always be able to claim and own the rest of the cores for performance reasons.

In summary, one of the main ideas of Legion is to provide mechanism and not policy, instead allowing application and architecture specific mappers to make these kinds of decisions. If the OS has any policy decisions baked into it, then Legion can't make the full range of mapping decisions available to the mapper and that will limit performance. Just like the runtime system, it is the OS's job to provide mechanism and stay out of the way of the mappers.

While some of these requests could likely be fulfilled by modifying or tweaking an existing OS like Linux, the broad range of complaints hints at an underlying issue. Designers of a runtime with such specific constraints on performance and functionality need more control of the machine. Ease of use is not an issue for them; they simply need to extract everything from the hardware that they can. If the existing interfaces do not provide this, something needs to change. This change is what I propose to enact within Nautilus.

Appendix B

GEARS

This chapter describes the Guest Examination and Revision Services (GEARS) framework that we developed for the Palacios VMM. This work was published in the proceedings of the 9th International Conference on Autonomic Computing in 2012 [91]. In particular we will look at the design, implementation, and evaluation of GEARS. I will begin by describing the notion of *guest-context virtual services* below. Then I will describe the design, implementation, and evaluation of GEARS (Section B.2). I will then outline two services we built using GEARS, an accelerated overlay networking component (Section B.3), and an MPI accelerator (Section B.4). Finally, I will draw conclusions and discuss how GEARS fits into the broader picture of my thesis work (Section B.5).

The GEARS system is a prototype to further the argument that the implementation of VMM-based virtual services for a guest OS should extend into the guest itself, even without its cooperation. Placing service components directly into the guest OS or application can reduce implementation complexity and increase performance. VMM code running *directly within the guest OS or application without its cooperation* can considerably simplify the design and implementation of services because the services can then directly manipulate aspects of the guest from within the guest itself. Further, these kinds of services can eliminate many overheads associated with costly exits to the VMM, improving their performance. Finally, extending a service into a guest enables new types of services not previously possible or that are prohibitively difficult to implement solely in the context of the VMM. I refer to virtual services that can span the VMM, the guest kernel, and the guest application, as *guest-context virtual services*.

When a VMM utilizes its higher privilege level to enable or enhance functionality, optimize performance, or otherwise modify the behavior of the guest in a favorable manner, it is said to provide a service to the guest. VMM-based services are usually provided transparently to the guest without its knowledge (e.g. via a virtual device). We now consider several example services that did profit from GEARS.

Overlay networking acceleration An important service for many virtualized computing environments is an overlay networking system that provides fast, efficient network connectivity among a group of VMs and the outside world, regardless of where the VMs are currently located. Such an overlay can also form the basis of an adaptive/autonomic environment. A prominent challenge for overlay networks in this context is achieving low latency and high throughput, even in high performance settings, such as supercomputers and next-generation data centers. I will show in Section B.3 how GEARS can enhance an existing overlay networking system with a guest-context component.

MPI acceleration MPI is the most widely used communication interface for distributed memory parallel computing. In an adaptive virtualized environment, two VMs running an application communicating using MPI may be co-located on a single host. Because the MPI library has no way of knowing this, it will use a sub-optimal communication path between them. Our MPI acceleration service can detect such cases and automatically convert message passing into memory copies and/or memory ownership transfers. I will discuss the design of this service in Section B.4.

Procrustean services While administrators can install services or programs on guests already, this task must be repeated many times. Furthermore, because the administrators of guests and those of provider hosts may not be the same people, providers may execute guests that are not secure. GEARS functionality permits the creation of services that would automatically deploy security patches and software updates on a provider's guests.

B.1 Guest-context Virtual Services

Services that reside within the core of the VMM have the disadvantage of relying on the mechanism by which control is transferred to the VMM. A VMM typically does not run until an exceptional situation arises, such as the execution of a privileged instruction (a direct call to the VMM in the case of paravirtualization) or the triggering of external or software interrupts. Much like in an operating system, the transition to the higher privilege level, called an *exit*, introduces substantial overhead. Costly exits remain one of the most prohibitive obstacles to achieving high-performance virtualization[1].

Eliminating these exits can, thus, improve performance considerably. The motivation is similar to minimizing costly system calls to OS code in user-space processes. Modern Linux implementations, for example, provide a mechanism called *virtual system calls*, in which the OS maps a read-only page into every process's address space on start-up. This

[1]Gordon et al., with the ELI system [87], ameliorate this problem on x86 by employing a *shadow* interrupt descriptor table (IDT) that vectors assigned interrupts directly to the guest.

page contains code that implements commonly used services and obviates the need to switch into kernel space. If the implementation of a VMM service is pushed up into the guest in a similar manner, more time is spent in direct execution of guest code rather than costly invocations of the VMM. This is precisely what GEARS accomplishes.

Moving components of a service implementation into the guest can not only improve performance, but also enable services that would otherwise not be feasible. In particular, guest-context services have a comprehensive view of the state of the guest kernel and application. These services can make more informed decisions than those implemented in a VMM core, which must make many indirect inferences about guest state. The VMM must reconstruct high-level operations based on the limited information that the guest exposes architecturally. Moreover, in order to manipulate the state of a guest kernel or application, the VMM must make many transformations from the low-level operations that the guest exposes to high-level operations that affect guest execution. While services certainly exist that can accomplish this transformation, their implementation takes on a more elegant design by operating at the same semantic level as the guest components they intend to support.

GEARS employs a tiered approach, which involves both the host and the VMM, to inject and run services in guest context. The process is outlined in Figure B.1. Users (service developers) provide standard C code for the VMM without needing extensive knowledge of VMM internals. This makes the procedure of implementing a service straight-forward, enabling rapid development. The code provided is a service implementation, split into two clearly distinguishable parts. We refer to these as the *top-half* and the *bottom-half*. The top-half is the portion of the service that runs in the guest-context. The bottom-half, which may not always be present, resides within a host kernel module readily accessible to the

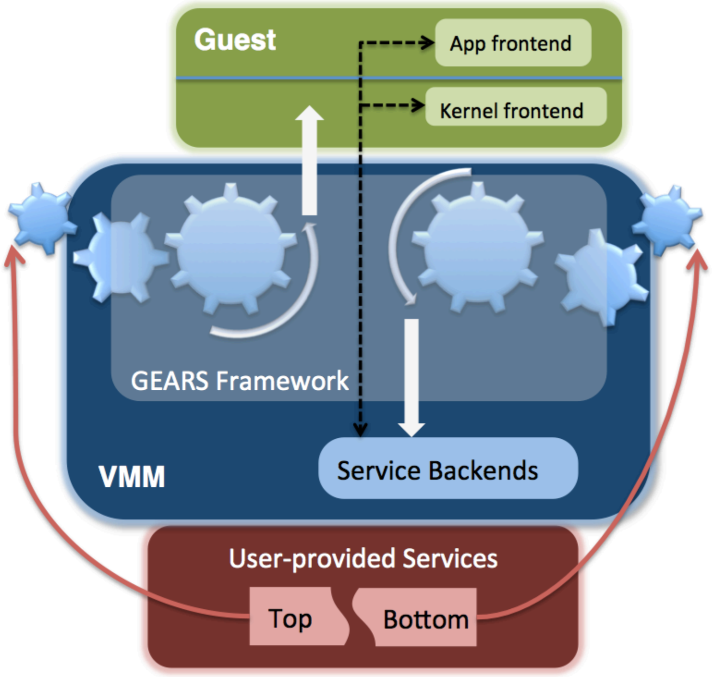


Figure B.1: GEARS services are broken into two parts; one exists in the guest and the other in the VMM. GEARS takes both parts provided as source code and uses several utilities to register and manage execution of the service.

VMM. The top-half calls into its respective bottom-half if it requires its functionality. The bottom-half can similarly invoke the top-half, allowing for a two-way interaction between the service components.

The code for the top-half must adhere to a guest-specific format. However, GEARS provides host-resident utilities that transform the code appropriately. Hence, from the user's perspective, writing the top-half of a guest-context service implies little more requisite knowledge than the ability to write a normal program or kernel module for the guest in question. GEARS simply provides the transformation utilities appropriate for that guest. If the service requires access to, or assistance from the VMM core, the developer can design a bottom half by writing a host kernel module that implements the relevant interface. Detailed discussion of bottom halves can be found in [91].

The notion of leaving service implementations entirely up to the user allows a clean separation between the framework and the services that it enables. GEARS provides the necessary tools to create cross-layer services, and users are responsible for using this platform to design innovative guest-VMM interactions.

B.2 GEARS Design and Evaluation

I will now outline the design and evaluation of the GEARS framework, independent of any services built on top of it. Readers interested in implementation details are directed to [91].

To enable guest-context services, the VMM must provide a mechanism that can place components directly within the guest. GEARS implements this mechanism with the code injection tool. Further, the VMM must have a way to select an appropriate time at which to perform this placement. The GEARS system call interception utility provides one way of accomplishing this. Finally, in order for the VMM to control guest execution paths at a higher level, e.g. for library calls, it must have the ability to modify the environment passed to the process. GEARS achieves this with process environment modification. Because these three conditions alone can facilitate the creation of guest-context services, we claim that GEARS provides the necessary and sufficient tools to accomplish this task. After describing our testbed for this work, I will describe in more detail how we satisfy these three conditions.

Experimental setup Before proceeding, it is prudent to describe the hardware and software setup for the experiments described in the following sections. We performed all experiments on AMD 64-bit hardware. We primarily used two physical machines for our testbed.

- 2GHz quad-core AMD Opteron 2350 with 2GB memory, 256KB L1, 2MB L2, and 2MB L3 caches. We refer to this machine as *vtest*.
- 2.3GHz 2-socket, quad-core (8 cores total) AMD Opteron 2376 with 32GB of RAM, 512KB L1, 2MB L2, and 6MB L3 caches, called *lewinsky*.

Both of these machines had Fedora 15 installed, with Linux kernel versions 2.6.40 and 2.6.42, respectively. The guests we used in our testbed ran Linux kernel versions 2.6.38. All experiments were run using the Palacios VMM configured for nested paging.

System call interception System call interception allows a VMM to monitor the activity of the guest at a fine granularity. Normally, system calls are not exceptional events from the standpoint of the VMM. However, system calls are commonly triggered using software interrupts, which modern hardware allows VMM interception of. GEARS can use more advanced techniques to intercept system call execution with the newer `SYSCALL` instructions. Once the VMM can track the execution of system calls, it can provide a wide range of services to the guest, such as sanity checking arguments to sensitive kernel code or matching system call patterns as in [133]. Figure B.2 shows the overhead introduced by *selective* system call exiting (the technique we use to intercept the `SYSCALL` instruction) for the `getpid` system call. This particular call is one of the simpler system calls in Linux, so these numbers represent the fixed cost introduced by system call interception. The row labeled *guest* represents a standard guest with no GEARS extensions. The *guest+intercept*

Strategy	**Latency (μs)**
guest	4.26
guest+intercept	4.51

Figure B.2: Average system call latency for getpid system call using selective exiting.

row indicates a GEARS-enabled guest using system call exiting. In this case, no system calls are selected to exit, so this overhead is the overhead paid by all system calls. Notice that the overhead introduced is only a small fraction of a microsecond. More details on the selective system call exiting technique can be found in the GEARS paper.

Process environment modification System call interception enables the VMM to essentially see the creation of every process in the guest through calls to `execve`, for example. The interception is done before the system call even starts, so the VMM has the option to modify the guest's memory at this point. One useful thing it can do is modify the environment variables that the parent process passes on to its child.

There are certain environment variables that are particularly useful. One is the `LD_PRELOAD` variable, which indicates that a custom shared library should be given precedence over the one originally indicated. This variable gives the VMM an opportunity to directly modify the control flow of a guest application. Other interesting environment variables affecting control flow include `LD_BIND_NOW` and `LD_LIBRARY_PATH`. The very fact that the Linux kernel itself uses the environment to pass information to the process (e.g. with the `AT_SYSINFO` variable) opens up a broad range of interesting possibilities.

Environment variables can not only be modified, but also, with careful treatment of memory, added or removed. This allows the VMM to provide information directly to the guest application without the need for paravirtualization. The guest OS requires no cognizance of the underlying VMM. Instead, VMM-awareness can vary on an application

Component	Lines of Code
System Call Interception	833
Environment Modification	683
Code Injection	915
Total	2431

Figure B.3: Implementation complexity for GEARS and its constituent components.

by application basis. This allows developers to make rapid optimizations to employ VMM services. As will become clear, these developers can even implement their own VMM service with minimal effort. Notice that the marked divergence from the usual reliance of user space applications on the operating system's ABI.

Code injection Code injection is perhaps the most unique mechanism in the GEARS framework. Because it allows the VMM to run arbitrary code in the context of the guest without any cooperation or knowledge on the part of the guest OS or application, it is the core tool enabling guest-context services.

We employ two types of code injection—user-space and kernel-space. User-space injection allows the VMM to map a piece of trusted code into the address space of a user-space process. On exits, the VMM can invoke this code manually or redirect user-space function calls to it by patching the process binary image.

GEARS can also inject code into a guest that will dynamically link with the libraries mapped into the applications' address spaces. Our current example services do not utilize this GEARS feature.

The other type of injection is kernel-space code injection, which relies on the ability to inject code into a user-space process. Injected kernel code must currently be implemented in a kernel module compiled for the guest. We used user-space code injection to write the module into the guest file system and subsequently insert it into the guest kernel.

While the existing GEARS implementation is targeted at Linux guests, it consists of relatively few components, each of which rely on features provided almost universally by modern OSes and architectures. This means that porting GEARS for other kinds of guests entails no great effort. Figure B.3 shows the size of the GEARS codebase. Each component is relatively compact. GEARS is currently implemented as a set of extensions to the Palacios VMM, and would likely become even more compact if integrated into the hypervisor core.

B.3 VNET/P Accelerator

VNET/P is an overlay networking system with a layer 2 abstraction implemented inside the Palacios VMM [196]. It currently achieves near-native performance in both the 1 Gbps and 10 Gbps switched networks common in clusters today and even more advanced interconnects [58] like Infiniband [104] and Cray's SeaStar [40]. Speeds on even faster networks, such as Infiniband, in the future. At the time we wrote the GEARS paper in 2012, VNET/P could achieve, with a fully encapsulated data path, 75% of the native throughput with 3-5x the native latency between directly connected 10 Gbps machines[2]. We used GEARS tools to further improve this performance.

The throughput and latency overheads of VNET/P are mostly due to guest/VMM context switches, and data copies or data ownership transfers. We can reduce the number of context switches, and the volume of copies or transfers, by shifting more of the VNET/P data path into the guest itself. In the limit, the entire VNET/P data path can execute in the guest with guarded privileged access to the underlying hardware. In this work [91], we

[2]More recent numbers can be found in the VNET/P SC paper [58]

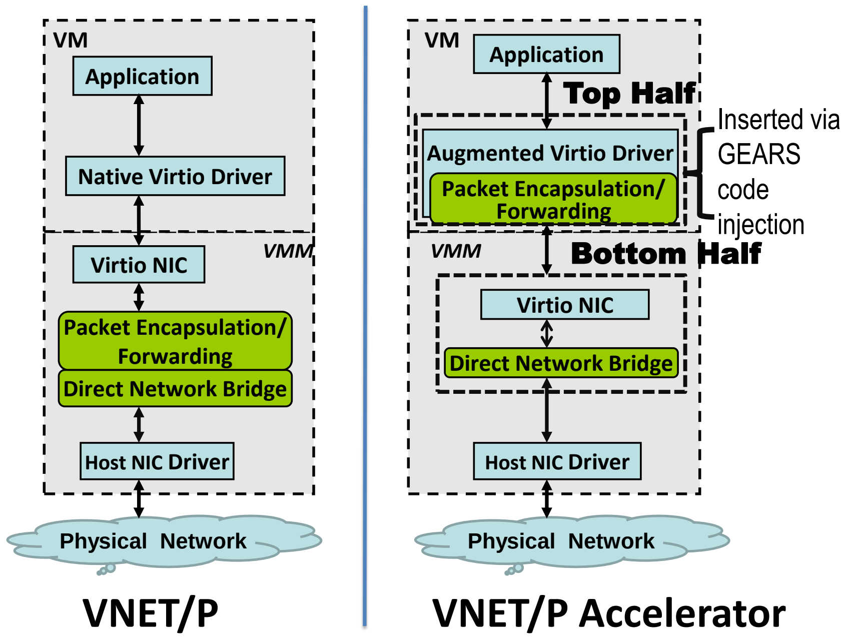


Figure B.4: Implementation of the prototype VNET/P Accelerator.

explored an initial step towards this goal that does not involve privileged access and aims at reducing the latency overhead as a proof-of-concept.

Figure B.4 illustrates this initial proof-of-concept implementation of the VNET/P Accelerator Service. In the baseline VNET/P data path, shown on the left, raw Ethernet packets sent from a VM through a virtual NIC are encapsulated and forwarded within the VMM and sent via a physical NIC. In the VNET/P Accelerator data path, shown on the right, the encapsulation and forwarding functionality of VNET/P resides within the guest as part of the guest's device driver for the virtual NIC. This augmented device driver kernel module is uncooperatively inserted into the guest kernel using the GEARS code injection tool. The augmented driver then delivers Ethernet frames containing the encapsulated packets to the virtual NIC. In our implementation, the driver that has been augmented is the Linux virtio NIC driver. The backend virtio NIC implementation in Palacios has no changes; it is simply bridged to the physical NIC.

Component	**Lines of Code**
vnet-virtio kernel module (Top Half)	329
vnet bridge (Bottom Half)	150
Total	479

Figure B.5: Implementation complexity of prototype VNET/P Accelerator Service. The complexity given is the total number of lines of code that were changed. The numbers indicate that few changes are necessary to port VNET/P functionality into a Linux virtio driver module.

Benchmark	Native	VNET/P	VNET/P Accel
Latency			
min	0.082 ms	0.255 ms	0.205 ms
avg	0.204 ms	0.475 ms	0.459 ms
max	0.403 ms	2.787 ms	2.571 ms
Throughput			
UDP	922 Mbps	901 Mbps	905 Mbps
TCP	920 Mbps	890 Mbps	898 Mbps

Figure B.6: VNET/P Accelerator results.

The implementation complexity of the proof-of-concept VNET/P accelerator is shown in Figure B.5, which illustrates that few changes are needed to split VNET/P functionality into a top half and a bottom half. The control plane of VNET/P remains in the bottom half in the VMM; only the encapsulation and forwarding elements move into the top half that GEARS injects into the guest.

Figure B.6 depicts the performance of the initial, proof-of-concept VNET/P Accelerator. Here, the round-trip latency and throughput was measured between a VM running on the *vtest* machine and a VM running on an adjacent machine that does not use the accelerator. We measured latency using ping with 1000 round-trips. The throughputs were measured using ttcp, where both TCP and UDP throughput are reported. We ran ttcp with a 6400 byte buffer, 10000 packets sent, and a standard 1500 byte MTU. We compared accelerated VNET/P with standard VNET/P and native performance between the two host machines without virtualization or overlays.

The VNET/P Accelerator achieves the same bandwidth as VNET/P, and both are as close to native as possible given that encapsulation is used. The VNET/P Accelerator achieves a modest improvement in latency compared to VNET/P (20% minimum, 3% average, 8% maximum).

B.4 MPI Accelerator

Consider an MPI application executing within a collection of VMs that may migrate due to decisions made by an administrator, an adaptive computing system, or for other reasons. The result of such migrations, or even initial allocation, may be that two VMs are co-located on the same host machine. However, the MPI application and the MPI implementation itself are oblivious to this, and will thus employ regular network communication primitives when an MPI process located in one VM communicates with an MPI process in the other. VNET/P will happily carry this communication, but performance will be sub-optimal.

Fundamentally, the communication performance in such cases is limited to the main memory copy bandwidth. Ideally, matching MPI send and receive calls on the two VMs would operate at this bandwidth. We assume here that the receiver touches all of the data. If that is not the case, the performance limit could be even higher because copy-on-write techniques might apply. Our MPI Accelerator service performs precisely this transformation of MPI sends and receives between co-located VMs into memory copy operations.

Building such an MPI Accelerator purely within the VMM would be extremely challenging because MPI send and receive calls are *library routines* that indirectly generate system calls and ultimately cause guest device driver interactions with the virtual hardware the VMM provides. It is these virtual hardware interactions that the VMM sees. In order to implement an MPI Accelerator service, it would be necessary to reconstruct the lost semantics of MPI operation. The ability to discern the MPI semantics *from the guest application* is the key enabler of our MPI Accelerator implementation.

GEARS provides two essential tools that the MPI Accelerator service leverages: (a) user space code injection, and (b) process environment modification (Section B.2). At any point during VM execution, the service uses (a) to inject and run a program that creates a file in the VM. The file contains a shared library that is an `LD_PRELOAD` wrapper for MPI. The system then uses (b) to force `exec()`s of processes to use the `LD_PRELOAD` wrapper. This can be limited to specific executables by name if desired. The wrapper installs itself between the processes and the MPI shared library such that MPI calls bind to the wrapper. The wrapper, which constitutes the top half of the service can then decide how to process each MPI call in coordination with the bottom half of the service that resides in the VMM. The top and bottom halves communicate using service-specific hypercalls.

In our prototype implementation, illustrated in Figure B.7, we focused on the blocking MPI_Send and MPI_Recv calls. The top half intercepts the appropriate MPI calls as follows:

- MPI_Init() : After normal initialization processing in MPI, this call also notifies the bottom half of this MPI process, including its name, arguments, and other parameters. It registers the process for consideration with the service.

- MPI_Finalize(): Before normal de-initialization in MPI, this call notifies the bottom half that the process can be unregistered.

- MPI_Comm_rank(): After a normal ranking in MPI, this call notifies the bottom half

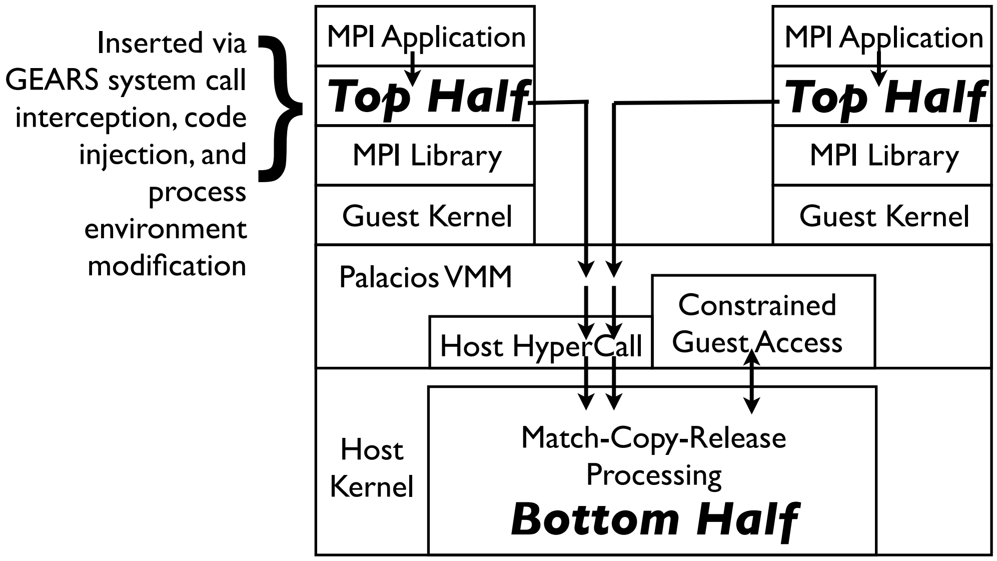


Figure B.7: Implementation of the MPI Accelerator service for co-located VMs. This illustrates the fast path between an MPI_Send and its matching MPI_Recv.

of the process's rank.

- MPI_Send(): The wrapper checks to see if this is an MPI_Send() that the bottom half can implement. If it is not, it hands it to the MPI library. If it is, it touches each page of the data to assure it is faulted in, and then hands the send request to the bottom half and waits for it to complete the work. If the bottom half asserts that it cannot, the wrapper defaults to the MPI library call.

- MPI_Recv(): This is symmetric to MPI_Send().

Our implementation is intended as a proof of concept demonstrating the utility of GEARS tools and advancing the overall argument of this work. Nonetheless, it also performs quite well. We have run the OSU MPI Latency benchmark [2] (osu_latency) between two co-located VMs using VNET/P, VNET/P with the GEARS tools enabled in Palacios, and with the MPI Accelerator active. We used the MPICH2 MPI library [139] for

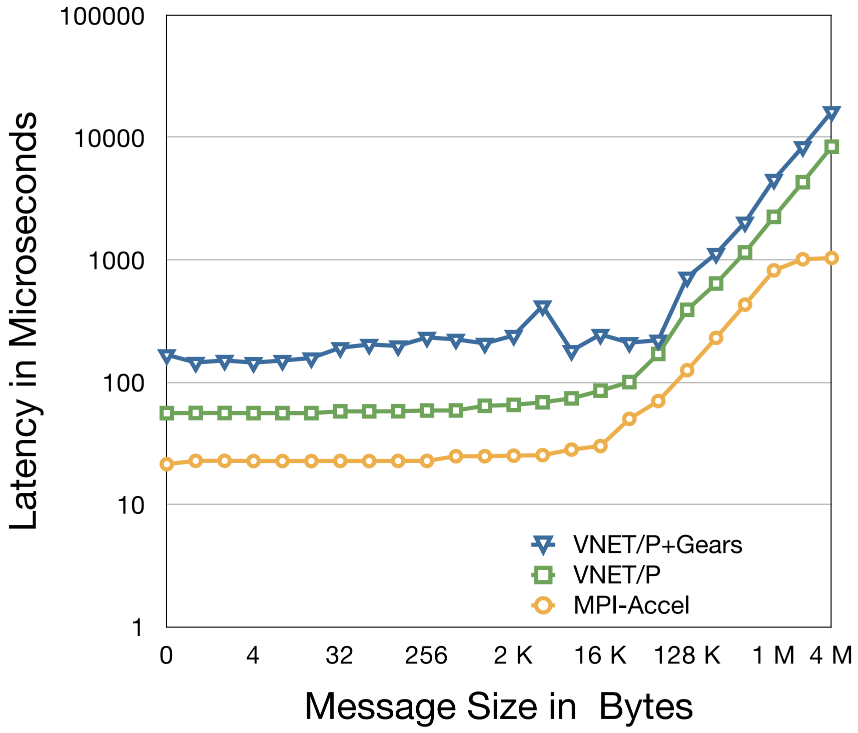


Figure B.8: Performance of MPI Accelerator Service on OSU MPI Latency Benchmark running in two co-located VMs on *lewinsky* test machine. For small messages, it achieves a 22 Âţs latency, limited by system call interception and hypercall overheads. For large messages, the MPI accelerator approaches the maximum possible performance given the memory copy bandwidth of the machine (4.6 GB/s).

our measurements. Our performance measurements were taken on the *lewinsky* machine, previously described. The results are shown in Figure B.8. It is important to note that for larger message sizes, the message transfer time is dominated by the machine's memory bandwidth. According to the STREAM benchmark, the machine has a main memory copy bandwidth of 4.6 GB/s. Our results suggest that we approach this—specifically, the MPI latency for 4 MB messages implies a bandwidth of 4 GB/s has been achieved. For small messages the MPI latency is approximately 22 μs (about 50,000 cycles). The small message latency is limited by system call interception, exit/entry, and hypercall overheads. Here, GEARS selective system call interception was not enabled. Using it further reduces the

Component	**Lines of Code**
Preload Wrapper (Top Half)	345
Kernel Module (Bottom Half)	676
Total	1021

Figure B.9: Implementation complexity of MPI Accelerator.

overhead for small messages.

The figure also shows the performance of using VNET/P for this co-located VM scenario. The "VNET/P" curve illustrates the performance of VNET/P without any GEARS features enabled. Without system call interception overheads, we see that VNET/P achieves a 56 μs latency for small messages and the large message latency is limited due to a transfer bandwidth of a respectable 500 MB/s. The "VNET/P+Gears" curve depicts VNET/P with the GEARS features enabled and illustrates the costs of non-selective system call interception. The small message latency grows to 150 μs, while the large message latency is limited due to a transfer bandwidth of 250 MB/s. In contrast to these, the MPI Accelerator Service, based on GEARS, achieves 1/3 the latency and 8 times the bandwidth, approaching the latency limits expected due to the hypercall processing and the bandwidth limits expected due to the system's memory copy bandwidth. Note that the impact of GEARS system call interception on the MPI Accelerator's small message latency is much smaller than its impact on VNET/P. This is not a discrepancy. With the MPI Accelerator, far fewer system calls are made per byte transferred because the injected top-half intercepts each MPI library call before it can turn into multiple system calls.

Figure B.9 illustrates that the service implementation is quite compact. The GEARS tools are the primary reason for the service's feasibility and compactness.

B.5 Conclusions

GEARS is a set of tools that enable the creation of *guest-context virtual services* which span the VMM, the guest kernel and the guest application. I showed with an implementation within the Palacios VMM that the complexity of these tools is tractable, suggesting that they could be implemented in other VMMs without great effort. GEARS in Palacios allows developers to write VMM services with relatively little knowledge of VMM internals. Further, I showed that the implementations of the services themselves can remain relatively compact while still delivering substantial performance or functionality improvements.

GEARS is an example of changing well-established system software interfaces for performance, functionality, complexity, or otherwise useful reasons. It is a testament to the power of specialized runtime systems with direct support from the underlying system software layer. In this case the support resided between the hypervisor layer and the application/guest kernel layer. The design and implementation of the GEARS system informed the development of my ideas regarding the drawbacks of generalized (and restricted) operating system interfaces.

One interesting aspect of guest-context virtual services is their boundaries with the rest of the guest. How difficult are these boundaries to identify? Can we secure them? The next chapter presents the results of my investigation of these questions in my work on *guarded modules*—the follow-up work to GEARS.

Appendix C

Guarded Modules

In this chapter, I will first give a high-level overview of guarded modules and their motivation. This work was published in the proceedings of the 11th International Conference on Autonomic Computing in 2014 (co-located with USENIX ATC) [90]. I will then describe the trust model we assume (Section C.1) and the design and implementation of the guarded module system (Section C.2). I will then present an evaluation of guarded modules (Section C.3). Next, I will discuss the design and evaluation of two examples we built using the guarded module system, a selectively privileged PCI passthrough NIC (Section C.4), and selectively privileged access to `mwait` (Section C.5). Finally, I will draw conclusions connections to my thesis work (Section C.6).

By design, a VMM does not trust the guest OS and thus does not allow it access to privileged hardware or VMM state. However, such access can allow new or better services for the guest, such as the following examples.

- Direct guest access to I/O devices can allow existing guest drivers to be used, avoid the need for virtual devices, and accelerate access when the device could be dedicated to the guest. In existing systems, the VMM limits the damage that a rogue guest could inflict by only using self-virtualizing devices [137, 165] or by operating in contexts such as HPC environments, where the guest is trusted and often runs alone [128].

- Direct guest access to the Model-Specific Registers (MSRs) that control dynamic voltage and frequency scaling (DVFS) would allow the guest's adaptive control of these features to be used instead of the VMM's whenever possible. Because applications running on the guest enjoy access to more rich information than the VMM does, there is reason to believe that guest-based control would perform better.
- Direct guest access to instructions that can halt the processor, such as `monitor` and `mwait`, can allow more efficient idle loops and spinlocks when the VMM determines that such halts can be permitted given the current configuration.

Since we cannot trust the guest OS, to create such services we must be able to place a component *into* the guest that is both tightly coupled with the guest and yet protected from it. We leverage GEARS, discussed in the previous section, to do exactly this. Here, we extend GEARS to allow for injected code to be endowed with privileged access to hardware and the VMM that the VMM selects, but only under specific conditions that preclude the rest of the guest from taking advantage of the privilege. We refer to this privileged, injected code as a *guarded module*, which is effectively a protected *guest-context virtual service* (described in Section B.1).

Our technique leverages compile-time and link-time processing which identifies valid entry and exit points in the module code, including function pointers. These points are in turn "wrapped" with automatically generated stub functions that communicate with the VMM. Our current implementation of this technique applies to Linux kernel modules. The unmodified source code of the module is the input to the implementation, while the output is a kernel object file that includes the original functionality of the module and the wrappers. Conceptually, a guarded module has a *border*, and the wrapper stubs (and their locations) identify the valid *border crossings* between the guarded module, which is trusted, and the rest of the kernel, which is not.

A wrapped module can then be injected into the guest using GEARS or added to the guest voluntarily. The wrapper stubs and other events detected by the VMM drive the second component of our technique, a state machine that executes in the VMM. An initialization phase determines whether the wrapped module has been corrupted and where it has been loaded, and then protects it from further change. Attempted border crossings, either via the wrapper functions or due to interrupt/exception injection, are caught by the VMM and validated. Privilege is granted or revoked on a per-virtual core basis. Components of the VMM that implement privilege changes are called back through a standard interface, allowing the mechanism for privilege granting/revoking to be decoupled from the mechanism for determining when privilege should change. The privilege policy is under the ultimate control of the administrator, who can determine the binding of specific guarded modules with specific privilege mechanisms.

C.1 Trust and Threat Models; Invariants

We assume a completely untrusted guest kernel. A developer will add to the VMM selective privilege mechanisms that are endowed with the same level of trust as the rest of the core VMM codebase. A module developer will assume that the relevant mechanism exists. The determination of whether a particular module is allowed access to a particular selective privilege mechanism is made at run-time by an administrator. The central relationship we are concerned with is between the untrusted guest kernel and the module. A compilation process transforms the module into a guarded module. This then interacts with run-time components to maintain specific invariants in the face of threats from the guest kernel. These invariants are described below.

Control-flow integrity The key invariant we provide is that the privilege on a given virtual core will be enabled if and only if that virtual core is executing within the code of the guarded module and the guarded module was entered via one of a set of specific, agreed-upon entry points. The privilege is disabled whenever control flow leaves the module, including for interrupts and exceptions.

The guarded module boasts the ability to interact freely with the rest of the guest kernel. In particular, it can call other functions and access other data within the guest. A given call stack might intertwine guarded module and kernel functions, but the system guards against attacks on the stack as part of maintaining the invariant.

A valid entry into the guarded module is not checked further. Our system does not guard against an attack based on function arguments or return values, namely Iago attacks. The module author needs to validate these himself. Note, however, that the potential damage of performing this validation incorrectly is limited to the specific privilege the module has.

Code integrity Disguising the module's code is not a goal of our system. The guest kernel can read and even write the code of the guarded module. However, any modifications of the code by any virtual core will be caught and the privilege will be disabled for the remainder of the module's lifetime in the kernel. The identity of the module is determined by its content, and module insertion is initiated external to the guest with a second identifying factor, guarding against the kernel attempting to spoof or replay a module insertion.

Data integrity Data integrity, beyond the registers and the stack, is managed explicitly by the module. The module can request private memory as a privilege. On a valid entry, the memory is mapped and is usable , while on departing the module, the memory is unmapped and rendered invisible and inaccessible to the rest of the kernel.

C.2 Design and Implementation

The specific implementation of guarded modules I describe here applies to Linux kernel modules. Our implementation fits within the context of Palacios and takes advantage of code generation and linking features of the GCC and GNU binutils toolchains. The VMM-based elements leverage functionality commonplace in modern VMMs, and thus could be readily ported to other VMMs. The code generation and linking aspects of our implementation seem to us to be feasible in any C toolchain that supports ELF or a similar format. The technique could be applicable to other guest kernels, although we do assume that the guest kernel provides runtime extensibility via some form of load-time linking.

In our implementation, a guarded Linux kernel module can either be voluntarily inserted by the guest or involuntarily injected into the guest kernel using GEARS. The developer of the module needs to target the specific kernel he wants to deploy on, exactly as in creating a Linux kernel module in general.

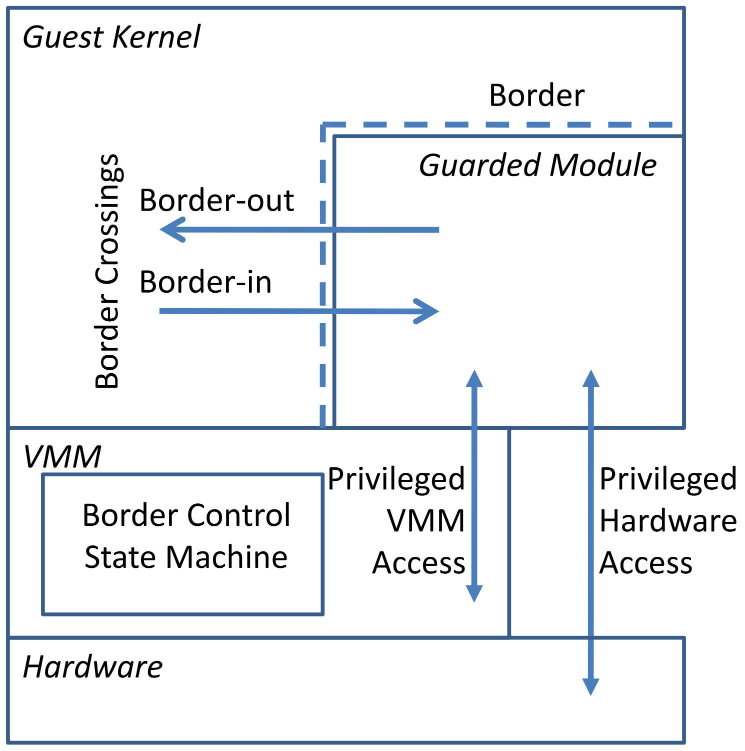


Figure C.1: Guarded module big picture.

Figure C.1 illustrates the run-time structure of our guarded module implementation, and documents some of the terminology we use. The guarded module is a kernel module within the guest Linux kernel that is allowed privileged access to the physical hardware or to the VMM itself. The nature of this privilege, which I will describe later, depends on the specifics of the module. I refer to the code boundary between the guarded module and the rest of the guest kernel as the *border*.

Border crossings consist of control flow paths that traverse the border. A *border-out* is a traversal from the module to the rest of the kernel, of which there are three kinds. The first, a *border-out call* occurs when a kernel function is called by the guarded module, while the second, a *border-out ret*, occurs when we return back to the rest of the kernel. The third, a *border-out interrupt* occurs when an interrupt or exception is dispatched. A *border-in* is a traversal from the rest of the kernel to the guarded module. There are similarly three forms here. The first, a *border-in call* consists of a function call from the kernel to a function within the guarded module, while the second, a *border-in ret* consists of a return from a *border-out call*, and the third, a *border-in rti* consists of a return from a border-out interrupt. Valid border-ins should raise privilege, while border-outs should lower privilege. Additionally, any attempt to modify the module should lower privilege.

The VMM contains a new component, the *border control state machine*, that determines whether the guest has privileged access at any point in time. The state machine also implements a registration process in which the injected guarded module identifies itself to the VMM and is matched against validation information and desired privileges. This allows the administrator to decide which modules, by content, are allowed which privileges. After registration, the border control state machine is driven by hypercalls from the guarded module, exceptions that occur during the execution of the module, and by interrupt or exception injections that the VMM is about to perform on the guest.

The VMM detects attempted border crossings jointly through its interrupt/exception mechanisms and through hypercalls in special code added to the guarded module as part of our compilation process. Figure C.2 illustrates how the two interact.

Compile-time Our compilation process, *Christoization*[1], automatically wraps an existing kernel module with new code needed to work with the rest of the system. Our toolchain that implements this is essentially a *guarded module compiler*. Two kinds of wrappers are generated. *Exit wrappers* are functions that interpose on the calls from the guarded module to the rest of the kernel. Figure C.3 shows an example entry wrapper. The important thing

[1]Named after the famed conceptual artist, Christo, who was known for wrapping large objects such as buildings and islands in fabric.

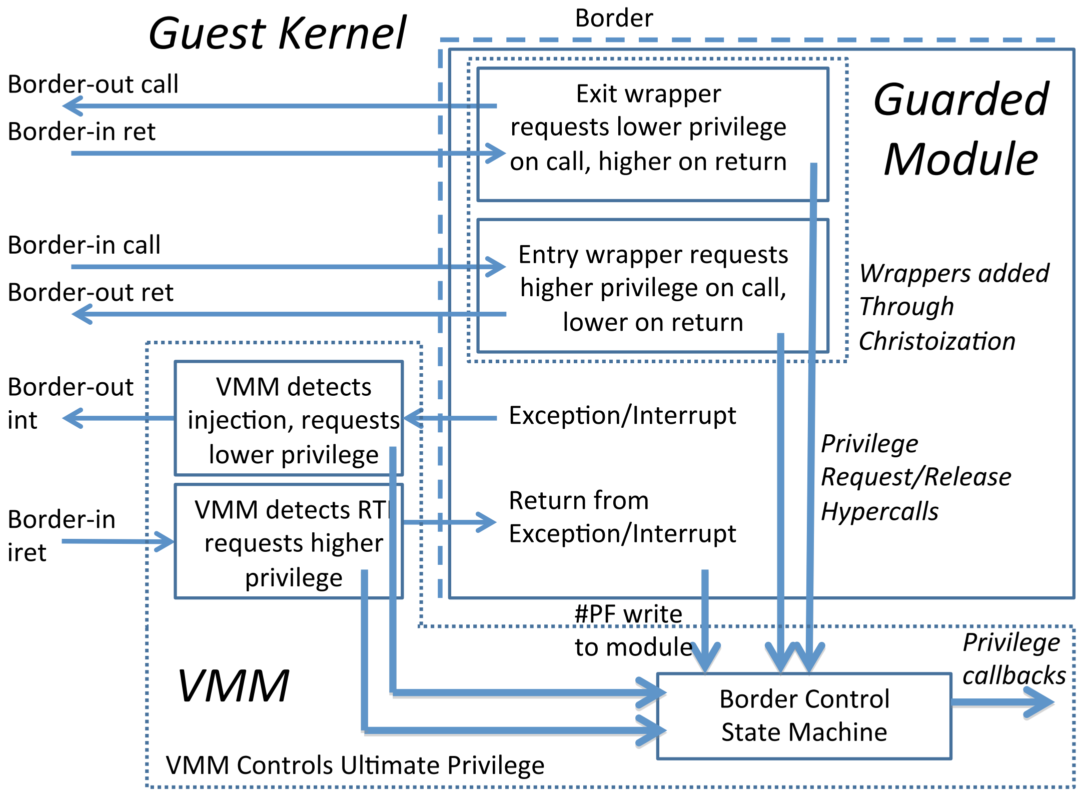

Guest Kernel
Border
Guarded Module
Border-out call
Border-in ret
Exit wrapper requests lower privilege on call, higher on return
Border-in call
Border-out ret
Entry wrapper requests higher privilege on call, lower on return
Wrappers added Through Christoization
Border-out int
VMM detects injection, requests lower privilege
Exception/Interrupt
Privilege Request/Release Hypercalls
Border-in iret
VMM detects RTI requests higher privilege
Return from Exception/Interrupt
#PF write to module
Privilege callbacks
VMM
Border Control State Machine
VMM Controls Ultimate Privilege

Figure C.2: Guarded modules, showing operation of wrappers and interaction of state machine on border crossings.

to note about our compilation process is that the wrappers serve to drive the guarded module runtime component, described next. More details on compilation can be found in the original paper [90].

Run-time The run-time element of our system is based around the border control state machine. As Figure C.2 illustrates, the state machine is driven by hypercalls originating from the wrappers in the guarded module, and by events that are raised elsewhere in the VMM. As a side-effect of the state machine's execution, it generates callbacks to other

```
entry_wrapped:
    popq  %r11
    pushq %rax
    movq $border_in_call, %rax (a)
    vmmcall
    popq  %rax
    callq entry
    pushq %rax
    movq $border_out_ret, %rax (b)
    vmmcall
    popq  %rax
    pushq %r11
    ret   (to rest of kernel)
```

Figure C.3: An entry wrapper for a valid entry point. Exit wrappers are similar, except they invoke border out on a call, and border in after returning. In this case, (a) invokes the VMM to perform validation and raise privilege, here (b) invokes the VMM to save state for later checks, lower privilege, and return to the guest kernel

components of the VMM that implement specific privilege changes, notifying them when valid privilege changes occur. The state machine also handles the initialization of a guarded module and its binding with these other parts of the VMM. We refer to the collective effects of these hypercalls and subsequent callback invocations as *border crossings*; they are a transition from a privileged to an unprivileged state. As depicted in Figure C.4, there are several types of border crossings. I will not discuss their details here, but interested readers can refer to the guarded module paper [90].

Each border crossing *into* the module has more overhead than its associated border-out. This is because we must perform validations of the stack to ensure control flow integrity (discussed in Section C.1). The performance implications of this validation are discussed

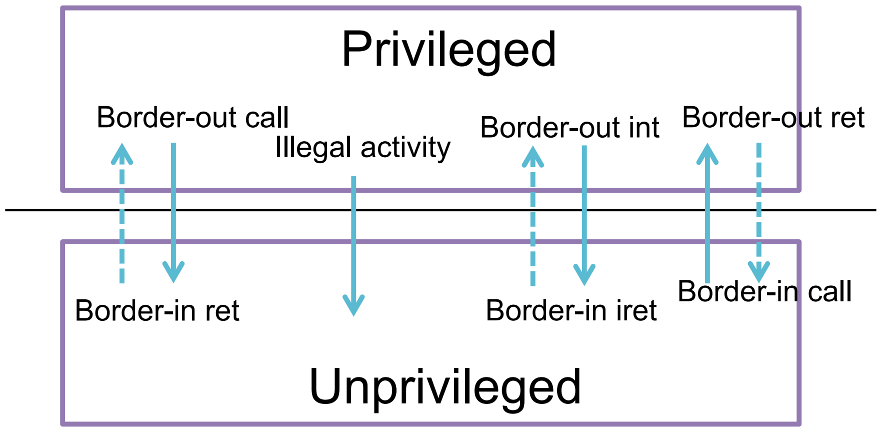


Figure C.4: Border Control.

in the next section.

It is worth noting that the entry wrapper shown in Figure C.3 and the exit wrappers are linked such that they are only invoked on border crossings. Calls internal to the guarded module do not have any additional overhead. The same applies for calls internal to the kernel.

The state machine detects suspicious activity by noting privilege changing hypercalls at invalid locations, shadow or nested page faults indicating attempts to write the module code, and stack validation failures. Our default behavior is simply to lower privilege when these occur, and continue execution. Other reactions are, of course, possible.

C.3 Evaluation

We now consider the costs of the guarded module system, independent of any specific guarded module that might drive it, and any selective privilege-enabled VMM component it might drive. We focus on the costs of border crossings and their breakdown. The most important contributors to the costs are VM exit/entry handling and the stack validation

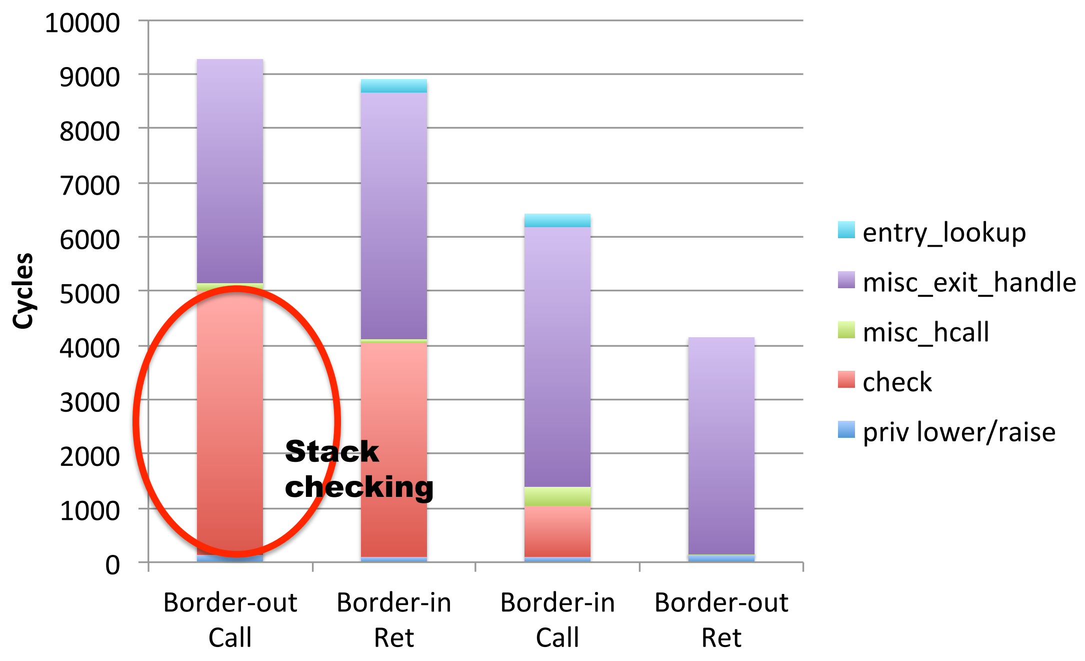


Figure C.5: Privilege change cost with stack integrity checks.

mechanism.

All measurements were conducted on a Dell PowerEdge R415. This is a dual-socket machine, each socket comprising a quad-core, 2.2 GHz AMD Opteron 4122, giving a total of 8 physical cores. The machine has 16 GB of memory. It runs Fedora 15 with a stock Fedora 2.6.38 kernel. Our guest environment uses a single virtual core that runs a BusyBox environment based on Linux kernel 2.6.38. The guest runs with nested paging, using 2 MB page mappings, with DVFS control disabled.

Figure C.5 illustrates the overheads in cycles incurred at runtime. All cycle counts were averaged over 1000 samples. There are five major components to the overhead. The first is the cost of initiating a callback to lower or raise privilege. This cost is very small at around 100 cycles. The second cost, labeled "hypercall handling", denotes the cycles spent inside the hypercall handler itself, not including entry validations, privilege changes, or other processing involved with a VM exit. This cost is also quite small, and also typically under 100 cycles. “Entry point lookup” represents the cost of a hash table lookup, which is invoked on border-ins when the instruction pointer is checked against the valid entry points that have been registered during guarded module initialization. The cost for this lookup is roughly 240 cycles. “Exit handling” is the time spent in the VMM handling the exit outside of guarded module runtime processing. This is essentially the common overhead incurred by any VM exit. Finally, “stack checking” denotes the time spent ensuring control-flow integrity by validating the stack. This component raises the cost of a border crossing by 5000 cycles, mostly due to stack address translations and hash computations. Border-in calls are less affected due to the initial translation and recording of the entry stack pointer, while border-out rets are unaffected. Reducing the cost of this validation is the subject of on-going work.

I now consider two examples of using the guarded module functionality, drawn from the list in the introduction. In the first example, selectively-privileged PCI passthrough, the guarded module, and only the guarded module, is given direct access to a specific PCI device. I illustrate the use of this capability via a guarded version of a NIC driver. In the second example, selectively-privileged `mwait`, the guarded module, and only the guarded module, is allowed to use the `mwait` instruction. I illustrate the use of this capability via guarded module that adaptively replaces the kernel idle loop with a more efficient `mwait` loop when it is safe to do so.

All measurements in this section are with the configuration described above.

C.4 Selectively Privileged PCI Passthrough

Like most VMMs, Palacios has hardware passthrough capabilities. Here, we use its ability to make a hardware PCI device directly accessible to the guest. This consists of a generic PCI front-end virtual device ("host PCI device") , an interface it can use to acquire and release the underlying hardware PCI device on a given host OS ("host PCI interface"), and an implementation of that interface for a Linux host.

A Palacios guest's physical address space is contiguously allocated in the host physical address space. Because PCI device DMA operations use host physical addresses, and because the guest programs the DMA engine using guest physical addresses it believes start at zero, the DMA addresses the device will actually use must be offset appropriately. In the Linux implementation of our host PCI interface, this is accomplished using an IOMMU: acquiring the device creates an IOMMU page table that introduces the offset. As a consequence, any DMA transfer initiated on the device by the guest will be constrained to that guest's memory. A DMA can then only be initiated by programming the device, which is restricted to the guarded module. This restriction also prevents DMA attacks on the module that might originate from the guest kernel.

A PCI device is programmed via control/status registers that are mapped into the physical memory and I/O port address spaces through standardized registers called BARs. Each BAR contains a type, a base address, and a size. Palacios's host PCI device virtualizes the BARs (and other parts of the standardized PCI device configuration space). This lets the guest map the device as it pleases. For a group of registers mapped by a BAR into the physical memory address space, the mapping is implemented using the shadow or nested page tables to redirect memory reads and writes. For a group of registers mapped

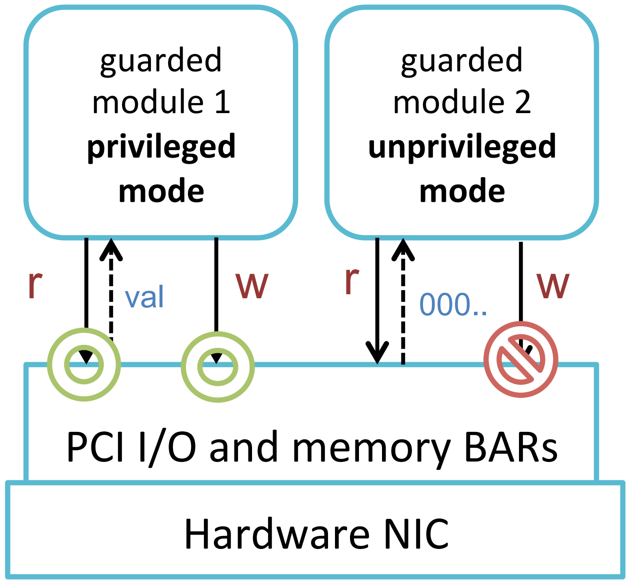

guarded module 1 **privileged mode**
guarded module 2 **unprivileged mode**
r
val
w
r
000..
w
PCI I/O and memory BARs
Hardware NIC

Figure C.6: High-level view of the operation of selective privilege for a PCI device.

into the I/O port space, there is no equivalent to these page tables, and thus the mappings are implemented by I/O port read/write hooks. When the guest executes an IN or OUT instruction, an exit occurs, the hook is run, and the handler simply executes an IN or OUT to the corresponding physical I/O port. If the host and guest mappings are identical, the ports are not intercepted, allowing the guest to read/write them directly.

We extended our host PCI device to support selective privilege. In this mode of operation, virtualization of the generic PCI configuration space of the device proceeds as normal. However, at startup, BAR virtualization ensures that the address space regions of memory and I/O BARs are initially hooked to stub handlers. The stub handlers simply ignore writes and supply zeros for reads. This is the *unprivileged mode*. In this mode, the guest sees the device on its PCI bus, and can even remap its BARs as desired, but any attempt to program it will simply fail because the registers are inaccessible. In selectively privileged operation, the host PCI device also responds to callbacks for raising and lowering privilege. Raising privilege switches the device to *privileged mode*, which is implemented by remapping the registers in the manner described earlier, resulting in successful accesses to the registers. Lowering privilege switches back to unprivileged mode, and remaps the registers back to the stubs. Privilege changes happen on a per-core basis. This process is depicted in Figure C.6.

While the above description is complex, it is important to note that only about 60 lines of code were needed to add selectively privileged operation to our existing PCI passthrough functionality. Combined with the rest of the guarded module system, the selectively privileged host PCI device permits fully privileged access to the underlying device within a guarded module, but disallows it otherwise.

Making a NIC driver into a guarded module As an example, we used the guarded module system to generate a guarded version of an existing NIC device driver within the Linux tree, specifically the Broadcom BCM5716 Gigabit NIC. No source code modifications were made to the driver or the guest kernel. We Christoized this driver, creating a kernel module that later is injected into the untrusted guest. The border control state machine in Palacios pairs this driver with the selectively privileged PCI passthrough capability.

Overheads Compared to simply allowing privilege for the entire guest, a system that leverages guarded modules incurs additional overheads. Some of these overheads are system-independent, and were covered in Section C.3. The most consequential component of these overheads is the cost of executing a border-in or border-out, each of which consists of a hypercall or exception interception (requiring a VM exit) or interrupt/exception injection detection (done in the context of an in-progress VM exit), a lookup of the hypercall's address, a stack check or record, conducting a lookup to find the relevant privilege callback function, and then the cost of invoking that callback.

Packet Sends	
Border-in	1.06
Border-out	1.06
Border Crossings / Packet Send	**2**.12
Packet Receives	
Border-in	4.64
Border-out	4.64
Border Crossings / Packet Receive	**9**.28

Figure C.7: Border crossings per packet send and receive for the NIC example.

We now consider the system-dependent overhead for the NIC. There are two elements to this overhead: the cost of changing privilege and the number of times we need to change privilege for each unit of work (packet sent or received) that the module finishes. The cost of raising privilege for the NIC is 4800 cycles (2.2 μs), while lowering it is 4307 cycles (2.0 μs).

Combining the system-independent and system-dependent costs, we expect that a typical border crossing overhead, assuming no stack checking will consist of about 3000 cycles for VM exit/entry, 4000 cycles to execute the border control state machine, and about 4500 cycles to enable/disable access to the NIC. These 11500 cycles comprise 5.2 μs on this machine. Stack checking would add an average of about 4500 cycles, leading to 16000 cycles (7.3 μs).

To determine the number of these border crossings per packet send or receive, we counted them while running the guarded module with a controlled traffic source (ttcp) that allows us to also count packet sends and/or receives. Dividing the counts gives us the average. There is variance because the NIC does interrupt coalescing.

Figure C.7 shows the results of this analysis for the NIC. Sending requires on the order of 2 border crossings (privilege changes) per packet, while receiving requires on the order of 9 border crossings per packet. Note that many of the functions that constitute border crossings are actually leaf functions defined in the kernel. This indicates that we could further reduce the overall number of border crossings per packet by pulling the implementations of these functions into the module itself.

C.5 Selectively Privileged MWAIT

Recent x86 machines include a pair of instructions, `monitor` and `mwait`, that can be used for efficient synchronization among processor cores. The `monitor` instruction indicates an address range that should be watched. A subsequent `mwait` instruction then places the core into a suspended sleep state, similar to a `hlt`. The core resumes executing when an interrupt is delivered to it (like a `hlt`), or when another core writes into the watched address range (unlike a `hlt`). The latter allows a remote core to wake up the local core without the cost of an inter-processor interrupt (IPI). One example of such use is in the Linux kernel's idle loop.

In Palacios, and other VMMs, we cannot allow an untrusted guest to execute `hlt` or `mwait` because the guest runs with physical interrupts disabled. A physical interrupt is intended to cause a VM exit followed by subsequent dispatch of the interrupt in the VMM. If an `mwait` instruction were executed in the guest under uncontrolled conditions, it could halt the core indefinitely. This precludes the guest using the extremely fast inter-core wakeup capability that `mwait` offers.

Under controlled conditions, however, letting the guest run `mwait` may be permissible. When no other virtual core is mapped to the physical core (so we can tolerate a long wait) and we have a watchdog that will eventually write the memory, the guest might safely run an `mwait`. To achieve these controlled conditions requires that we limit the execution

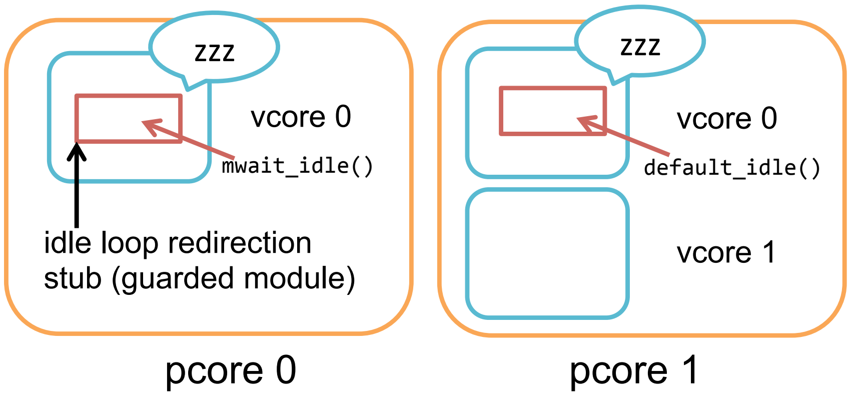


Figure C.8: Direct access to the MWAIT instruction provided by a guarded module.

of these instructions to code that the VMM can trust and that this code only execute `mwait` when the VMM deems it safe to do so. A malicious guest could use an unrestricted ability to execute `mwait` to launch a denial-of-service attack on other VMs and the VMM. We enforce this protection and adaptive execution by encapsulating the `mwait` functionality within the safety of a guarded module.

Adding selectively-privileged access to `mwait` to Palacios was straightforward, involving only a few lines of code. We then implemented a tiny kernel module that interposes on Linux's default idle loop, specifically modifying `pm_idle`, a pointer to the function that points to the idle implementation. Our module points this to a function internal to itself that dispatches either to an `mwait`-based idle implementation within the module or to the original idle implementation, based on a flag in protected memory that is shared with Palacios. Palacios sets this flag when it is safe for the module to use `mwait`. In these situations, the guest kernel enjoys much faster wake-ups of the idling core.

To assure that only our module can execute `mwait` we transform it into a guarded module using the techniques outlined earlier. A border-in to our module occurs when Linux calls its idle loop. If the border-in succeeds, Palacios stops intercepting the use of `mwait`. When control leaves the module, a border-out occurs, and Palacios resumes intercepting `mwait`. If code elsewhere in the guest attempts to execute these instructions, they will trap to the VMM and result in an undefined opcode exception being injected into the guest. Figure C.8 shows this process at a high level. Each of the larger boxes represents a physical CPU core (pcore). The smaller boxes within represent virtual cores (vcores). In the case on the right-hand side, there are two vcores assigned to the same pcore, so the VMM cannot allow either one to use the `mwait` version of the idle loop. On the left-hand side, however, there is only one vcore, so it is free to use the optimized `mwait` version of idle.

This proof-of-concept illustrates how the VMM can use guarded modules to safely adapt the execution environment of a VM to changing conditions.

C.6 Conclusions

I presented the design, implementation, and evaluation of a system for guarded modules. The system allows the VMM to add modules to a guest kernel that have higher privileged access to physical hardware and the VMM while protecting these guarded modules and access to their privileges from the rest of the guest kernel. This system is based on joint compile-time and run-time techniques that bestow privilege only when control flow enters the guarded module at verified locations. I demonstrated two example uses of the guarded module system. The first is passthrough access to a PCI device, for example a NIC, that is limited to a designated guarded module (a device driver). The guest kernel can use this guarded module just like any other device driver. I further demonstrated selectively privileged use of the `monitor` and `mwait` instructions in the guest, which could wreak havoc if their use was not constrained to a guarded module that cooperates with the VMM.