NORTHWESTERN
UNIVERSITY

Electrical Engineering and Computer Science Department

Technical Report
NWU-EECS-10-08
September 13, 2010

Symbiotic Virtualization

John R. Lange

Abstract

Virtualization has now become ubiquitous, especially in large-scale data centers. Significant inroads have also be made into high performance computing and adaptive systems, areas where I have focused in the Virtuoso and V3VEE projects. The rapid adoption of virtualization in all of these areas is in no small part due to the ability of full system virtualization to adapt existing OSes to virtual environments with no changes to the OS implementation. On the other hand, paravirtualization, which does require deep OS changes, has been demonstrated to have significant performance and functionality benefits. Beyond paravirtualization, researchers are now discussing other ways to rethink OS design for the virtualization age.

One fundamental problem with existing virtualization architectures is that the interfaces they provide to a guest environment exist at a very low level and do not expose high level semantic information. This has created a situation where underlying VMMs often have very little knowledge about the architecture, behavior, or needs of a guest VM. This situation has come to be described as the *semantic gap*. Furthermore, existing architectures are designed such that obtaining this information is extremely difficult. In order for virtualization to reach its true potential, this problem must be addressed.

This effort was partially supported by the National Science Foundation (NSF) via grants CNS-0709168 and CNS-0707365, and by the Department of Energy (DOE) via Sandia National Laboratories (SNL) and Oak Ridge National Laboratory (ORNL), as well as a Symantec Research Labs Fellowship.

My dissertation focuses on *symbiotic virtualization*, a new approach to designing virtualized systems that are capable of fully bridging the semantic gap. Symbiotic virtualization bridges the semantic gap via a bidirectional set of synchronous and asynchronous communication channels. Unlike existing virtualization interfaces, symbiotic virtualization places an equal emphasis on both semantic richness and legacy compatibility. The goal of symbiotic virtualization is to introduce a virtualization interfaces that provide access to high level semantic information while still retaining the universal compatibility of a virtual hardware interface. Symbiotic virtualization is an approach to designing VMMs and OSes such that both support, but neither requires, the other. A symbiotic OS targets a native hardware interface, but also exposes a software interface, usable by a symbiotic VMM, if present, to optimize performance and increase functionality. Symbiotic virtualization is neither full system virtualization nor paravirtualization, however it can be used with either approach. Symbiotic virtualization introduces OS changes that facilitate rich information gathering by the VMM, and focuses on the VMM's functional interface to the OS and not the inverse.

A symbiotically virtualized architecture supports multiple symbiotic interfaces. Symbiotic interfaces are *optional* for the guest, and a guest which does use it can also run on non-symbiotic VMMs or raw hardware without any changes. A symbiotic OS exposes two types of interfaces. Passive interfaces allow a symbiotic VMM to directly access internal guest state. This information is exposed directly to the VMM, via an asynchronous communication channel. This interface has extremely low overhead, however its asynchronous nature limits the kind of information that can be accessed in this way. Functional interfaces allow a symbiotic VMM to invoke the guest directly in order to request that the guest perform an operation on behalf of the VMM. These interfaces impose a higher overhead than passive interfaces, but allow for synchronous invocation and support more complex state queries. This dissertation will examine symbiotic virtualization and two symbiotic interfaces: *SymSpy* and *SymCall*. I will also describe *SymMod* an interface that allows a VMM to dynamically create additional symbiotic interfaces at runtime. These interfaces allow for both passive, asynchronous and active, synchronous communication between guest and VMM.

I have implemented a symbiotic virtualization framework inside Palacios, an OS independent embeddable virtual machine monitor that I have led the development of. Palacios is a wholly new VMM architecture designed specifically to target areas that have been ill served by existing virtualization tools, namely high performance computing, architecture and education. Palacios supports multiple physical host and virtual guest environments, is compatible with both AMD SVM and Intel VT architectures, and has been evaluated on commodity Ethernet based servers, a high end Infiniband cluster, as well as Red Storm development cages consisting of Cray XT nodes. Palacios also supports the virtualization of a diverse set of guest OS environments, including commodity Linux and other OS distributions, modern Linux kernels, and several lightweight HPC OSes such as CNL, Catamount, and the Kitten Lightweight Kernel.

Keywords: Virtual Machines, Operating Systems, High Performance Computing

NORTHWESTERN UNIVERSITY

Symbiotic Virtualization

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Science

By

John Russell Lange

EVANSTON, ILLINOIS

December 2010

©copyright by John Lange 2010
All Rights Reserved

Abstract

Symbiotic Virtualization

John Lange

Virtualization has now become ubiquitous, especially in large-scale data centers. Significant inroads have also be made into high performance computing and adaptive systems, areas where I have focused in the Virtuoso and V3VEE projects. The rapid adoption of virtualization in all of these areas is in no small part due to the ability of full system virtualization to adapt existing OSes to virtual environments with no changes to the OS implementation. On the other hand, paravirtualization, which does require deep OS changes, has been demonstrated to have significant performance and functionality benefits. Beyond paravirtualization, researchers are now discussing other ways to rethink OS design for the virtualization age.

One fundamental problem with existing virtualization architectures is that the interfaces they provide to a guest environment exist at a very low level and do not expose high level semantic information. This has created a situation where underlying VMMs often have very little knowledge about the architecture, behavior, or needs of a guest VM. This situation has come to be described as the *semantic gap*. Furthermore, existing architectures are designed such that obtaining this information is extremely difficult. In order for virtualization to reach its true potential, this problem must be addressed.

My dissertation focuses on *symbiotic virtualization*, a new approach to designing virtualized systems that are capable of fully bridging the semantic gap. Symbiotic virtualization

bridges the semantic gap via a bidirectional set of synchronous and asynchronous communication channels. Unlike existing virtualization interfaces, symbiotic virtualization places an equal emphasis on both semantic richness and legacy compatibility. The goal of symbiotic virtualization is to introduce a virtualization interfaces that provide access to high level semantic information while still retaining the universal compatibility of a virtual hardware interface. Symbiotic Virtualization is an approach to designing VMMs and OSes such that both support, but neither requires, the other. A symbiotic OS targets a native hardware interface, but also exposes a software interface, usable by a symbiotic VMM, if present, to optimize performance and increase functionality. Symbiotic virtualization is neither full system virtualization nor paravirtualization, however it can be used with either approach. Symbiotic Virtualization introduces OS changes that facilitate rich information gathering by the VMM, and focuses on the VMM's functional interface to the OS and not the inverse.

A symbiotically virtualized architecture supports multiple symbiotic interfaces. Symbiotic interfaces are *optional* for the guest, and a guest which does use it can also run on non-symbiotic VMMs or raw hardware without any changes. A symbiotic OS exposes two types of interfaces. Passive interfaces allow a symbiotic VMM to directly access internal guest state. This information is exposed directly to the VMM, via an asynchronous communication channel. This interface has extremely low overhead, however its asynchronous nature limits the kind of information that can be accessed in this way. Functional interfaces allow a symbiotic VMM to invoke the guest directly in order to request that the guest perform an operation on behalf of the VMM. These interfaces impose a higher overhead than passive interfaces, but allow for synchronous invocation and support more complex state queries. This dissertation will examine symbiotic virtualization and two symbiotic interfaces: *SymSpy* and *SymCall*. I will also describe *SymMod* an interface that allows a VMM to dynamically create additional symbiotic interfaces at runtime. These interfaces allow for both passive, asynchronous and active, synchronous communication between

guest and VMM.

I have implemented a symbiotic virtualization framework inside Palacios, an OS independent embeddable virtual machine monitor that I have led the development of. Palacios is a wholly new VMM architecture designed specifically to target areas that have been ill served by existing virtualization tools, namely high performance computing, architecture and education. Palacios supports multiple physical host and virtual guest environments, is compatible with both AMD SVM and Intel VT architectures, and has been evaluated on commodity Ethernet based servers, a high end Infiniband cluster, as well as Red Storm development cages consisting of Cray XT nodes. Palacios also supports the virtualization of a diverse set of guest OS environments, including commodity Linux and other OS distributions, modern Linux kernels, and several lightweight HPC OSes such as CNL, Catamount, and the Kitten Lightweight Kernel.

Thesis Committee

Peter A. Dinda, Northwestern University, Committee Chair

Fabián Bustamante, Northwestern University, Committee Member

Russ Joseph, Northwestern University, Committee Member

Karsten Schwan, Georgia Institute of Technology, Committee Member

Dedication

To everyone who believed in me

Acknowledgments

I am perpetually amazed at the incredible amount of luck I have been blessed with when it comes to my family, friends, and colleagues. I am acutely aware that nothing I have achieved would have been remotely possible without their friendship, compassion, guidance, and love. I truly could not have asked for a more perfect assortment of people to spend my life with.

First, I must express my sincere and overwhelming gratitude for my advisor Peter Dinda. He has been a consistent source of wisdom, advice, and patience. His faith in me has been constant for almost 10 years, even when I lacked it in myself. As an advisor he encourages rather than pushes, and has always been willing to allow me freedom in my research directions. He is a true role model whose example I can only hope to live up to.

As well I need to thank the members of my thesis committee. Fabián Bustamante has been a consistent friend and mentor throughout my graduate career. His door has always been open to me and he has been a constant source of candid conversations without any hint of pretense. Similarly, Russ Joseph has always been willing to offer help and listen to my complaints. He has always been a willing sounding board for my ideas on research, graduate school, my career, and academia in general. Karsten Schwan has not only advised my thesis work, but also constantly offered assistance in both my career and general research.

Patrick Bridges and Kevin Pedretti have been very close collaborators for the past few years. I constantly look forward to our weekly phone conversations, and working with them has been an incredibly fulfilling experience. This dissertation would not have been possible without the knowledge I gained as a direct result of our interactions.

There have been many contributors to Palacios that deserve recognition. My fellow

graduate students deserve ample praise for all of the work they have contributed to the project. Of note are Lei Xia and Zheng Cui who have been stalwart project members for the last several years. Also Chang Bae, Phil Soltero and Yuan Tang have also contributed much to Palacios and deserve ample recognition. Many undergraduates as well have been involved at various stages. Andy Gocke, Steven Jaconette, Rob Deloatch, Rumou Duan, Peter Kamm, Matt Wojcik, Brad Weinberger, and Madhav Suresh have all played active roles during Palacios' development.

I also want to mention other graduate students who I have worked with over the past 6 years. Ananth Sundararaj was a selfless friend for several years both as an office mate and as a collaborator. He was always willing to offer advice on both research and being a graduate student. Both Ashish Gupta and Bin Lin were also very helpful throughout the first few years of my graduate career. My office mates Scott Miller and Ionut Trestian were always willing to listen to my opinions on research, graduate school and life in general. I have many fond memories of our numerous entertaining conversations.

My family has been particularly supportive throughout this entire process. My parents, John and Vicki Lange, have provided me with a tremendous number of opportunities and have always been supportive no matter what whim possessed me at the time. My two brothers, Jim and Joe, have given me with a very exciting childhood and provide a window into a very different side of life that I would not have the good fortune to experience otherwise. My grandparents have always been proud of me, and I have spent my life hoping to live up to their opinions of me. I want to thank my uncle Cam and his family for their constant support, and my aunt Cindy who, from a very early age, inspired me with a love of scientific pursuits. Finally I want to thank my in-laws who have fully welcomed me into their family and have provided me with a constant source of friendship.

Also I want to acknowledge my friends who have constantly set a high pace of achievement which I have struggled to match. I lack the space to name all of them here, but suffice

it to say that if they are reading this then they probably know who they are.

And finally I must personally thank my wife Laura for her many years of patience and love. She has always been truly supportive of my choice in career paths, even after it became clear that it wasn't going to make us rich. I could not have wished for a better friend and companion throughout this journey.

Contents

List of Figures

Chapter 1

Introduction

Virtualizing large scale systems with minimal overhead requires cooperation between a guest OS and a Virtual Machine Monitor (VMM). This level of cooperation requires both communication and trust across the VMM/guest interface. We might say that the relationship between the VMM and the guest is *symbiotic*. Existing virtualization interfaces focus on achieving wide compatibility at the expense of making semantically rich information unavailable. This limitation makes VMM/guest cooperation very difficult, as the state needed to architect symbiotic behaviors is not readily available. This has resulted in a large *semantic gap* [13] between the OS and VMM, which impedes any form of cooperation across layers. The limitation on a VMMs ability to optimize itself for a guest environment will always exist unless a method can be found to bridge the semantic gap.

Considerable effort has been put into better bridging the of the VMM↔OS interface and leveraging the information that flows across it [42, 43, 49, 96, 51, 75, 31]. However, the information gleaned from such black-box and gray-box approaches is still semantically poor, and thus constrains the decision making that the VMM can do. Further, it goes one way; the OS learns nothing from the VMM. To fully bridge the semantic gap a new set of interfaces, that make internal state information easily accessible to both the VMM and guest, are necessary.

1.1 Symbiotic Virtualization

My dissertation focuses on *symbiotic virtualization*, an approach to designing virtualized architectures such that high level semantic information is available across the virtualization interface. Symbiotic virtualization bridges the semantic gap via a bidirectional set of synchronous and asynchronous communication channels.

To explore symbiotic virtualization I have developed a symbiotically virtualized architecture that provides new virtualization interfaces that are capable of fully bridging the semantic gap. Unlike existing virtualization interfaces, symbiotic interfaces place an equal emphasis on both semantic richness and legacy compatibility. The goal of symbiotic virtualization is to introduce a virtualization interface that provides access to high level semantic information while still retaining the universal compatibility of a virtual hardware interface. Symbiotic Virtualization is an approach to designing VMMs and OSes such that both support, but neither requires, the other. A symbiotic OS targets a native hardware interface, but also exposes a software interface, usable by a symbiotic VMM, if present, to optimize performance and increase functionality. Symbiotic virtualization is neither full system virtualization nor paravirtualization, however it can be used with either approach. Symbiotic Virtualization introduces OS changes that facilitate rich information gathering by the VMM, and focuses on the VMM's functional interface to the OS and not the inverse.

1.2 Symbiotic interfaces

The symbiotic architecture I developed supports multiple symbiotic interfaces. Symbiotic interfaces are *optional* for the guest, and a guest which implements them can also run on non-symbiotic VMMs or raw hardware without any changes. A symbiotic OS exposes two types of interfaces. Passive interfaces allow a symbiotic VMM to directly access internal guest state. This information is exposed directly to the VMM, via an asynchronous com-

munication channel. This interface has extremely low overhead, however its asynchronous nature limits the kind of information that can be accessed in this way. Functional interfaces allow a symbiotic VMM to invoke the guest directly in order to request that the guest perform an operation on behalf of the VMM. These interfaces impose a higher overhead than passive interfaces, but allow for synchronous invocation and support more complex state queries. This dissertation will examine symbiotic virtualization as well as two symbiotic interfaces: *SymSpy* and *SymCall*,. These interfaces allow for both passive, asynchronous and active, synchronous communication between guest and VMM. I will also examine *SymMod*, a mechanism that allows a VMM to dynamically create new symbiotic interfaces inside a guest.

1.2.1 SymSpy

The SymSpy interface provides a mechanism for the sharing of structured information between the VMM and the guest OS. SymSpy builds on the widely used technique of a shared memory region that is accessible by both the VMM and guest. This shared memory is used by both the VMM and guest to expose semantically rich state information to each other, as well as to provide asynchronous communication channels. The data contained in the memory region is well structured and semantically rich, allowing it to be used for most general purpose cross layer communication. The precise semantics and layout of the data on the shared memory region depends on the symbiotic services that are discovered to be jointly available in the guest and the VMM. The structured data types and layout are enumerated during discovery. During normal operation, the guest can read and write this shared memory without causing an exit. The VMM can also directly access the page during its execution.

1.2.2 SymCall

While SymSpy provides a mechanism for easily exposing state information, it is not ideally suited to handling information that is either very large or very complex. For state that exhibits increased complexity a new approach to collecting this information is required, one that does not require a guest environment to preemptively expose an overwhelming amount of data. Instead of relying on the guest OS to provide the data in an easily accessible manner, it is possible for it to support a functional interface that allows a VMM to run queries against it. This would allow the guest OS to organize its internal state however it wanted, and still provide a mechanism whereby a VMM could easily access it. I have implemented such an interface, SymCall, which exposes a symbiotic interface that provides a VMM with functional access to a guest's internal context. SymCall is a mechanism by which the VMM can execute code *synchronously* in the *guest context* while the VMM is in the process of handling an exit. That is, it provides synchronous upcalls into the guest at any time.

1.3 SymMod

Both SymSpy and SymCall implement new virtual interfaces that expose much more semantic information than is available with current approaches. However, both of these interfaces require the guest environment to implement some level of support for each interface that is to be used. This can lead to a situation where a VMM needs an interface that is not available. To address this issue, a mechanism is needed whereby a symbiotic VMM can dynamically extend a guest environment to enable new interfaces not supported by the guest OS. I have designed and implemented symbiotic modules (SymMod) as an answer to this problem.

SymMod is a mechanism that allows a VMM to run arbitrary blocks of code inside the

guest context. These code blocks are injected into the guest context as modules and are capable of implementing special device drivers, security scanners, performance monitors, new symbiotic interfaces, or any other functionality that is needed by the VMM. In essence these modules can be thought of as a special kind of loadable kernel module in Linux or a driver in Windows. My dissertation will explore three different types of symbiotic modules that interface with the running guest context in different ways. The important thing to note is that symbiotic modules are able to vastly minimize the semantic gap because they actually operate inside the guest context instead of through a generic external interface. Operating inside the VM also allows a guest to directly access the modules functionality with negligible overhead, since the VM does not need to trap into the VMM whenever a module is accessed.

1.4 Virtual Machines

Scalability is one of the greatest challenges currently facing the computing industry. As the demand for computation has continued to increase dramatically throughout the world, the scale of computational resources has increased as well. Furthermore, it has now become well established that the future of computing will be one of ever larger numbers of slightly faster resources instead of exponentially faster ones. While it is no longer possible to buy a computer every year that is twice as fast as the previous one, new CPU architectures are using higher transistor densities to increase the number of available resources inside a machine. This shift of growth towards scale instead of speed has introduced an explosion in the complexity of running and managing computer systems.

The effects of this explosion of scale is already well understood in the context of enterprise data centers as well as the high performance computing centers. Today's large scale computing resources are performance limited primarily by power, cooling, and failure rate

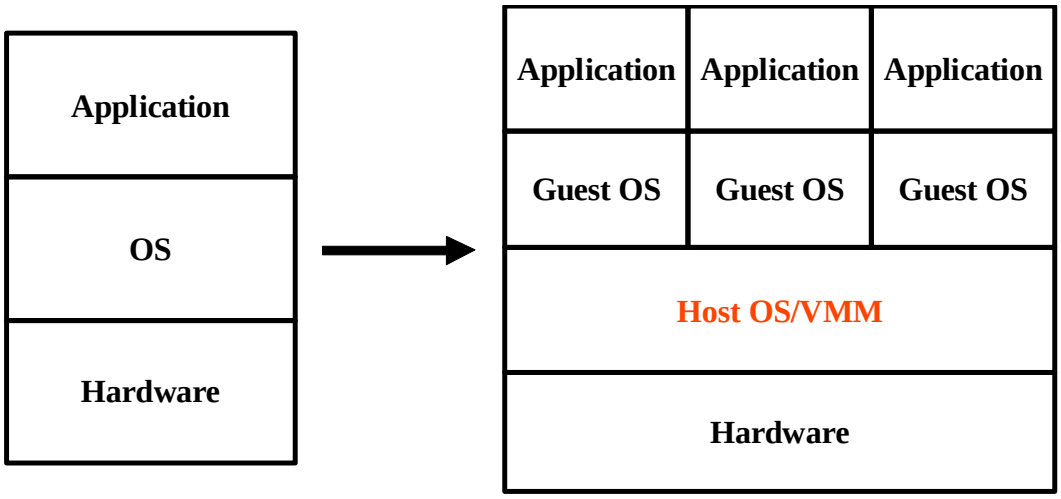


Figure 1.1: Architectural hierarchy of a traditional OS environment vs. a virtualized environment. Traditionally OSes run directly on hardware and multiplexes resources between multiple running applications. In a virtualized environment a VMM runs manages hardware directly multiplexes resources between multiple running guest OSes. This allows VMM to manage OS environments as an OS manages an application.

constraints rather than by the raw performance of the hardware. Addressing these issues requires a great deal more flexibility than is present in traditional data center architectures. A number of solutions have been developed to address the challenges of these large scale systems, not least of which is the reintroduction of Virtual Machines (VMs).

As shown in Figure 1.1, virtual machines provide a layer of abstraction between a computer's physical hardware and an operating system (OS) running on top of it. This abstraction layer allows a Virtual Machine Monitor (VMM) to encapsulate an entire Operating System environment and manage it much in the same way an OS manages applications. A VMM is able to execute, pause, and switch between multiple operating systems without having to reboot an entire machine. Furthermore, because an OS is fully encapsulated by the VMM, it is no longer tied to any dedicated physical hardware, in fact it

can be copied and/or moved to new hardware without requiring any reconfiguration or re-installation. In specific environments this movement can even be accomplished in a matter of milliseconds, without interrupting the execution of the OS or any applications [15]. The encapsulated OS is thus referred to as a "Virtual Machine" or a "Guest".

Virtual Machines have a long history in the field, with their basic architecture having been well established since the early 1970's [73] with the IBM VM/370. However, virtualization quickly fell from prominence due to the limitations of the x86 Instruction Set Architecture (ISA), the defacto standard in use today. These well documented limitations [81] severely limited the ability of x86 platforms to support virtual machines until the late 1990's, when several new software based approaches were developed to bypass the problems with the x86 ISA. This led to a renaissance of virtualization technology that continues through today [23].

1.5 Current VMM architectures

Virtualization is an architectural concept in which a very low level abstraction layer is provided to allow entire computing environments to run in isolated software containers. This abstraction layer is supported by a virtual hardware interface that performs, in software, the same operations traditionally implemented in hardware. Virtualization has been understood for many years and formalized based on a procedure called *Trap and Emulate* [73]. Trap and emulate denotes the two fundamental operations required of a virtual machine monitor (VMM): trap special instructions and events when they occur in a guest environment, and emulate them as necessary inside the VMM. These operations are required because of how virtualized environments actually execute code in a guest context.

Based on my description thus far, it would seem that virtualization is no different from emulation. At a high level, both VMMs and full system emulators provide the same func-

tionality. That is, both enable the execution of a full software system stack in an encapsulated context controlled by another software layer. Both allow for starting and stopping guest images, as well as easily moving entire OS environments between physical hardware as you would application data. However there is one major difference between the two approaches, and that is how the instructions in a guest environment are actually executed on the hardware. In a full system emulator, each individual instruction is decoded and emulated by software. In other words none of the hardware instructions that make up a guest environment are ever actually executed on real hardware. While this allows for a great deal of compatibility, such as running software written for completely different hardware architectures, it imposes a significant performance penalty as a result of instruction translation.

Virtualization on the other hand executes a guest's hardware instructions directly on the hardware itself. Thus, a VM executes directly on hardware except when the guest OS needs to perform a sensitive low level operation, which the hardware is configured to trap and deliver to a VMM. These sensitive operations are typically uncommon and result from such low level OS behavior as switching between processes and interacting with hardware devices. This means that most of the time there is no performance penalty for code executed in a VM versus on real hardware. Furthermore, this is almost always true for code executing in user mode, which never needs to execute any privileged instructions at all.

A more detailed taxonomy of the wide range of virtualization architectures is described by Smith and Nair [88]. The current collection of virtual machine technologies in use today are designed around the classic system VMM architecture they describe. These technologies are based on either a full system virtualization approach (e.g., VMWare [108]), a paravirtualization approach (e.g., Xen [6]), or a combination of the two (e.g., KVM [76]). The former approach provides the guest OS with a hardware interface abstraction, while

the latter provides a more abstract software interface. The former requires no OS changes, while the latter requires that the OS be ported to the abstract interface, but can offer better performance. Increasingly, these categories are turning into a continuum: hardware virtualization features can be used to bridge a commodity OS to a paravirtualization-based VMM, and full system virtualization-based VMMs can also support paravirtualized OSes.

Figure 1.2 shows a high level overview of a virtualized architecture. As I just described, VMs execute directly on hardware but are prevented from arbitrarily reconfiguring it. This is necessary to protect the host OS and VMM. Because a guest OS expects to have full control over the hardware, the VMM must present that illusion to the guest via emulated hardware state. Therefore, a guest OS will execute in a virtual context and perform operations on various pieces of hardware state and devices as if it had full control over them. However, underneath the VM these operations are actually being trapped by the VMM and then emulated in a way that updates this illusory hardware state visible to the guest. Meanwhile the VMM controls the actual hardware directly, and ensures that it is configured in a way that maintains the isolation of the virtual environments. This behavior affords VMM architectures that are designed as large event-dispatch loops. The virtualization extensions implemented in modern x86 processors enable exactly this behavior.

As one might expect, there is additional overhead added to a virtualized system by the fact that operations that used to be executed directly by hardware are now trapped and implemented instead in software. The hardware traps in particular generate full hardware world switches with considerable CPU cycle latencies. This means that guest environments that perform a large number of sensitive operations will experience correspondingly large performance penalties due to the hardware cost of the world switches.

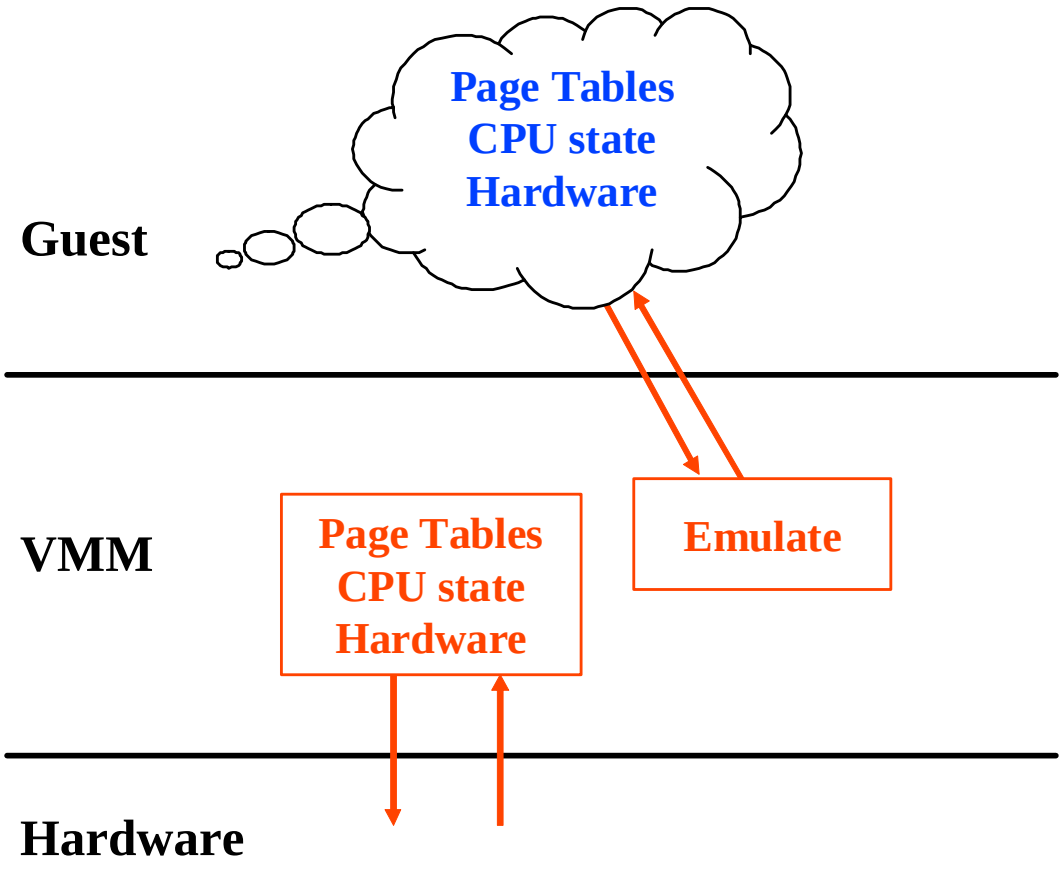


Figure 1.2: High level architecture of a VMM. A guest environment controls what it believes to be real hardware, but is instead emulated inside the VMM. The VMM meanwhile configures the actual hardware appropriately to maintain the illusion and provide isolation to the guest environment.

1.6 Enterprise and data center environments

Virtualization has become exceedingly popular with enterprise and large scale data center users and administrators. This adoption and acceptance of Virtual Machines has increased in step and in parallel with the dramatic increase in scale. The reason behind this is fairly obvious when one examines both the problems resulting from scale and the benefits provided by virtual machines.

1.6.1 Server consolidation

Service isolation has long been an established mantra for managing large scale enterprise environments. The principle reason behind this practice is that it is important to ensure that two services do not interfere with each other in unpredictable ways. This interference is capable of manifesting itself in any number of ways. For instance if one application is suddenly placed under heavy load it could cause a denial of service against another application running on the same host. Also many applications require highly specialized OS configurations that many times are incompatible with other applications, thus requiring separately configured environments for each. For these reasons, and many more, the long standing practice in data centers was to have every application run on its own dedicated hardware. As the number of applications increased both by adding new services, and replicating old ones to handle additional loads, the number of physical machines increased in turn. As the number of machines increased, the cost of powering them and managing them increased as well, eventually to unsustainable levels.

With the introduction of virtual machines this upward trend was abated considerably. Because virtualization ensures the isolation of each OS environment running concurrently on the same hardware, multiple applications can be consolidated on the same physical machine. This allows data centers to reduce the number of physical machines on hand

substantially, often by 12-15X [28].

1.6.2 Fault tolerance

It is inevitable that as the number of components in a system increases the failure rate of the system increases as well. This is an acute problem for any large scale system, especially one that is as highly complex and interdependent as a modern data center environment. Any service interruption or downtime of a large scale commercial system is highly undesirable, with the principle mission of every system manager being to avoid such a scenario. Traditional approaches to this problem have focused on reducing the time to recovery of the system, often resulting in replicating entire enterprise environments to serve as a hot spare should a failure occur.

Virtualization again alleviates these issues by separating the software services from the hardware substrate. This allows any physical machine to run any software service without reconfiguring or reinstalling. This allows hot spare machines to be dramatically scaled back, as there is no longer a need to keep a 1:1 ratio between live machines and preconfigured replicas.

1.7 High Performance and Supercomputing

High performance computing (HPC) is another area that faces many problems resulting from ever increasing scale. Just as with enterprise environments, virtualization has the potential to dramatically increase the usability and reliability of HPC systems [37, 63, 27, 68]. HPC is generally characterized as an area of computing that runs highly tuned applications at extremely large scales. Furthermore, HPC systems generally place an overriding focus on performance. This is because even small performance losses can have dramatic multiplying effects on large scale tightly integrated systems. Virtualization cannot succeed in

HPC systems unless the performance overheads are truly minimal and do not compound as the system and its applications scale up.

This challenge is amplified on high-end machines for several reasons. First, these machines typically run a carefully crafted custom HPC OS that itself already minimizes overheads and OS noise [21, 71], as well as makes the capabilities of the raw hardware readily available to the application developer. Second, the applications on these machines are intended to run at extremely large scales, involving thousands of nodes. Finally, the applications are typically tightly coupled and communication intensive, making them very sensitive to performance overheads, particularly unpredictable overheads. For this reason, they often rely on the deterministic behavior of the HPC OSes on which they run.

Virtualization offers the option to enhance the underlying machine with new capabilities or better functionality. Virtualized lightweight kernels can be extended at runtime with specific features that would otherwise be too costly to implement. Legacy applications and OSes would be able to use features such as migration that they would otherwise be unable to support.

1.7.1 Usability

Full system virtualization provides full compatibility at the hardware level, allowing existing unmodified applications and OSes to run. The machine is thus immediately available to be used by any legacy applications, increasing system utilization when ported application jobs are not available. This allows application developers to target the OS best suited to their application's needs while not precluding them from running on any specific HPC machines. The virtual hardware behavior can also be modified to expose a hardware environment more suitable for a particular guest OS or application.

1.7.2 System Management

Full system virtualization would allow a site to dynamically configure nodes to run a full OS or a lightweight OS without requiring rebooting the whole machine on a per-job basis. Management based on virtualization would also make it possible to backfill work on the machine using loosely-coupled programming jobs or other low priority work. A batch-submission or grid computing system could be run on a collection of nodes where a new OS stack could be dynamically launched; this system could also be brought up and torn down as needed.

1.7.3 Fault Tolerance

Virtualization also provides new opportunities for fault tolerance, a critical area that is receiving more attention as the mean time between system failures continues to decrease. Virtual machines can be used to implement full system checkpoint procedures, where an entire guest environment is automatically checkpointed at given intervals. This would allow the centralized implementation of a feature that is currently the responsibility of each individual application developer. Migration is another feature that can be leveraged to increase HPC system resiliency. If hardware failures can be detected and predicted, the software running on the failing node could be preemptively migrated to a more stable node.

The challenge is not, however, limited to high-end machines. The primary driver behind the challenge is the scaling of the machines and the applications, and this scaling is ubiquitous. Today's high-end machines are tomorrow's typical machines. Even at the extremes, such as server or desktop environments, scalability is a key driver with the advent of multicore processors, which are rapidly starting to look like message-passing parallel supercomputers [36].

1.8 Palacios

Modern VMMs have been designed with a focus on targeting enterprise data center environments. While this is a sensible approach, it is unclear whether the architectures developed for that environment accurately map to HPC or other specialized environments. While steps are being taken to offer specialized VMM architectures for desktop [106, 70] machines, there has been no specialized architecture designed for HPC [102]. To date, HPC virtualization technologies have consisted of adaptations of techniques developed for enterprise data centers [98].

To address the lack of research into the area of HPC virtualization architectures (as well as the areas of architecture and education), we have developed the Palacios[1] Virtual Machine Monitor. Palacios is an OS independent embeddable VMM specifically designed for HPC environments as part of the the V3VEE project (`http://v3vee.org`). Currently, Palacios targets the x86 and x86_64 architectures (hosts and guests) and is compatible with both the AMD SVM [2] and Intel VT [38] extensions. Palacios supports both 32 and 64 bit host OSes as well as 32 and 64 bit guest OSes. Palacios supports virtual memory using either shadow or nested paging. Palacios implements full hardware virtualization while providing targeted paravirtualized extensions.

As of the writing of this dissertation, Palacios has been evaluated on commodity Ethernet based servers, a high end Infiniband cluster, as well as Red Storm development cages consisting of Cray XT nodes. Palacios also supports the virtualization of a diverse set of guest OS environments, including commodity Linux and other OS distributions, modern Linux kernels, and several lightweight HPC OSes such as CNL [45], Catamount [46], and the Kitten Lightweight Kernel.

[1] Palacios, TX is the "Shrimp Capital of Texas."

1.9 Symbiotic Virtualization in Palacios

Palacios was originally designed as a full system VMM, and so presented a traditional virtual hardware abstraction to the HPC guest environments. While early experiments showed that this was a viable approach for scalable virtualization, there was however a non-negligible degree of overhead added to applications. This overhead was the result of both added hardware performance penalties as well as the necessity of emulating hardware operations inside Palacios. In order to alleviate these overheads it became clear that the VMM layer needed additional state information from the guest OS in order to operate in an optimal manner.

Based on this observation, we applied and evaluated symbiotic techniques as a solution to the issues identified with virtualizing an HPC platform. In later chapters I will describe the symbiotic techniques as they apply to HPC systems, as well as describe how symbiotic virtualization can still be applied in the future. I will also show how Symbiotic Virtualization is capable of delivering scalable virtualization with $\leq$5% overhead in a high-end message-passing parallel supercomputer. The exemplar of such a machine is Sandia National Lab's Red Storm machine [83], a Cray XT [11] machine.

1.10 Palacios in High Performance and Supercomputing

As stated earlier, Palacios is an OS independent VMM that requires a host OS in order to run. In the HPC context, Palacios specifically targets the Kitten Lightweight Kernel from Sandia National Laboratories. Kitten is a publicly available, GPL-licensed, open source OS designed specifically for high performance computing. It employs a "lightweight" philosophy [79] to achieve superior scalability on massively parallel supercomputers. The general philosophy being used to develop Kitten is to borrow heavily from the Linux kernel when doing so does not compromise scalability or performance (e.g., adapting the Linux

bootstrap code for Kitten). Performance critical subsystems, such as memory management and task scheduling, are replaced with code written from scratch for Kitten.

Kitten's focus on HPC scalability makes it an ideal host OS for Palacios on HPC systems, and Palacios' design made it easy to embed it into Kitten. In particular, Kitten/ Palacios integration was accomplished with a single interface file of less than 300 lines of code. The integration includes no internal changes in either Kitten or Palacios, and the interface code is encapsulated with the Palacios library in an optional compile time module for Kitten.

When Palacios is linked with the Kitten Lightweight Kernel, it is capable of virtualizing large scale supercomputing environments with minimal overhead. Palacios is the first VMM specifically designed to target these environments, and is able to deliver scalable virtualized performance within 5% of native.

1.11 Outline

The remainder of my dissertation will be organized around both symbiotic virtualization and the Palacios virtual machine monitor.

Chapter 2 describes the architecture of Palacios. Palacios is a full featured OS independent embeddable VMM that I primarily designed and developed. Palacios is designed to be highly configurable and easily portable to diverse host OS architectures. This chapter will provide an in depth description of the organization and design choices of Palacios.

Chapter 3 provides an evaluation of Palacios in an HPC environment. Palacios is the first VMM designed specifically for HPC environments and through a collaboration with Sandia National Laboratories we were able to analyze its effectiveness. Our results show that Palacios is capable of virtualizing an HPC system and deliver performance within 5% of native. This evaluation was performed on part of the RedStorm Cray XT supercomputer

at Sandia.

Chapter 4 describes Symbiotic Virtualization. In this chapter I will describe the symbiotic virtualization approach and philosophy as well as introduce SymSpy, a basic symbiotic interface using shared memory as an asynchronous communication channel.

Chapter 5 explores the integration of symbiotic techniques with HPC environments. Having developed a new interface that allows a VMM and guest to optimize themselves based on the other's architecture and behavior, we proceeded to evaluate how well these techniques apply to an HPC system. This chapter includes the largest scale performance study of virtualization performed as of the writing of this dissertation. Of particular note is the passthrough device architecture built on top of SymSpy that allows a guest to access physical devices with zero overhead.

Chapter 6 moves beyond the realm of HPC and introduces the architecture of the SymCall symbiotic interface. SymCall is a functional interface that allows a VMM to make synchronous upcalls into a guest environment. This chapter will describe the SymCall architecture both from the VMM's and guest's perspective. Furthermore, this chapter will describe and evaluate SwapBypass, a VMM service that is designed to optimize a guest's swapping performance.

Chapter 7 describes the design and implementation of symbiotic modules. While symbiotic upcalls do allow a VMM to perform complex queries of internal guest state, this interface is restricted to explicit queries implemented by the guest OS. In this chapter I will describe the symbiotic module framework that allows a VMM to load arbitrary code blocks into a running guest. These code blocks execute inside the guest's context and have full access to it's internal API.

Chapter 8 elaborates on related work in the areas of virtualization, HPC, and VMM/guest interaction.

Chapter 9 concludes with a summary of contributions and future research directions.

Chapter 2

Palacios

My dissertation is based on the design and use of the Palacios Virtual Machine Monitor. Palacios is an OS independent VMM developed from scratch at Northwestern University and the University of New Mexico as part of the V3VEE project. The V3VEE project began as a result of the dearth of VMM architectures for particular environments such as HPC, architecture research, and education. Palacios is designed to be a highly configurable and portable VMM architecture that can be deployed in a number of specialized and constrained environments. To date, Palacios has successfully virtualized commodity desktops and servers, high end Infiniband clusters, and supercomputers such as a Cray XT.

2.1 Introduction

Virtualization has emerged as a critical enabler of new systems research and has simultaneously lowered the barriers to deployment faced by such research. However, despite this broad impact there are only a small number of existing Virtual Machine Monitor (VMM) architectures [6, 76, 107]. As a side affect of this small architectural foundation, virtualization technologies have not been able to penetrate into many specialized areas to the degree that is possible. Of particular interest to us are the areas of high performance computing (HPC), architecture research and education.

Existing VMMs have been developed with a business centric purpose and specifically target environments that are capable of generating the largest revenue. As a result, existing VMM architectures are primarily designed for both enterprise and large data center environments. While these environments represent the largest market segment, and have the greatest need for virtualization, they do not represent the full virtualization user base, which has specific needs and priorities which in turn translate into highly specialized architectures that are optimized for specific use cases. Concentrating on enterprise environments has also resulted in a very tight integration of the VMs with existing commodity OSes, to the exclusion of others. As a result of this concentration on large scale enterprise users, existing architectures are no longer suitable for use in HPC and other specialized environments. An explanation of the shortcomings of existing VMMs in HPC environments will be presented in the next chapter.

As a response to these trends we have developed the Palacios Virtual Machine Monitor as part of the V3VEE Project, a collaborative community resource development project involving Northwestern University and the University of New Mexico for which I am the primary designer and developer. Palacios is designed to provide a flexible VMM that can be used in many diverse environments, while providing specific support for HPC. Palacios is also designed to be OS agnostic, with the goal of providing virtualization functionality to any OS that wishes to include it.

At a high level Palacios is designed to be an OS independent, embeddable VMM that is widely compatible with existing OS architectures. In other words, Palacios is not an operating system, nor does it depend on any one specific OS. This OS agnostic approach allows Palacios to be embedded into a wide range of different OS architectures, each of which can target their own specific environment (for instance 32 or 64 bit operating modes). Palacios is intentionally designed to maintain the separation between the VMM and OS. In accordance with this, Palacios relies on the hosting OS for such things as scheduling and

Palacios lines of code		
Component	sloccount .	wc *.c *.h *.s
Palacios Core (C+Assembly)	15,084	24,710
Palacios Virtual Devices (C)	8,708	13,406
XED Interface (C+Assembly)	4,320	7,712
Total	28,112	45,828

Figure 2.1: Lines of code in Palacios as measured with the SLOCCount tool and with the `wc` tool.

process/thread management, memory management, and physical device drivers. This allows OS designers to control and use Palacios in whatever ways are most suitable to their architecture. Palacios is also designed to be as compact as possible, with a simple and clear code base that is easy to understand, modify, and extend.

The scale of Palacios' code base is shown in Figure 2.1, as measured by two different source code analysis tools. Note that the Palacios core is quite small. The entire VMM, including the default set of virtual devices is on the order of 28–45 thousand lines of C and assembly. In comparison, Xen 3.0.3 consists of almost 580 thousand lines of which the hypervisor core is 50–80 thousand lines, as measured by the `wc` tool.

As mentioned previously, Palacios supports a wide range of host OS architectures and is fully capable of running in whatever context the host OS implements. At one extreme, Palacios can be linked with a simple bootstrap OS that handles the initial machine configuration before relinquishing full control to Palacios. Conversely, if Palacios is linked with a full featured multitasking OS, Palacios can run in a kernel or user thread context subject to the host OS' scheduling policies.

To better understand Palacios' embeddability, consider a few examples. The first example would be an embedded minimalist OS that only wants to run a single VM container with everything executing inside of it. Such an OS could be a monitoring or instrumentation layer that only intends to collect performance data for some given OS/application

stack. This OS would provide only rudimentary and basic functionality such as memory management. Palacios would fully and straightforwardly support this environment. It would exist as the primary thread of execution, with occasional interruptions for measurement tasks, while also exposing the full set of hardware devices directly to the guest OS.

As a second example, consider a lightweight OS designed for HPC environments, such as Catamount [46]. These OSes would run HPC applications while retaining a virtualization layer to support legacy applications or to provide better management functionality. This OS would implement many of the standard OS functions with a few notable exceptions such as a file system. In this case as well, Palacios would be able to provide a full featured virtualization layer that operated in a more traditional sense. VMs could be loaded via the job dispatch framework, and then be instantiated as kernel threads with voluntary preemption while having direct access to a small set of high performance devices. Finally, Palacios could be embedded into a full featured OS designed for general use. In this environment, VMs could be started as user space processes and be able to load disk images directly from the host file system. These examples span the full range of potential OSes and operating environments, and each is fully supported by Palacios.

Providing support for such a wide range of OS architectures and environments required a careful design process for Palacios. As a result Palacios exposes a minimal and partitioned set of OS requirements, that take the form of function hooks that are implemented in the OS. These hooks are used by Palacios to access internal OS functionality such as memory allocation and deallocation. Palacios is also designed to be highly configurable, both at compile and run time. The configurability is supported by means of modularization of the Palacios code base, that allows components to be selectively linked into the final Palacios executable. VMs themselves are also configurable at run time via a configuration file that is loaded with each VM image. This allows Palacios to tailor the virtual environment for

each guest it executes.

The Palacios implementation relies entirely on the virtualization extensions deployed in current generation x86 processors, specifically AMD's SVM [2] and Intel's VT [38, 100]. A result of this is that Palacios only supports both host and guest environments that target the x86 hardware platform. However, while the low level implementation is constrained, the high level architecture is not, and can be easily adapted to other architectures with or with out hardware virtualization support. Specifically Palacios supports both 32 and 64 bit host and guest environments, both shadow and nested paging models, and a significant set of devices that comprise the PC platform. Work is also underway to support future I/O architectures such as IOMMUs [8]. In addition to supporting full-system virtualized environments, Palacios provides support for the implementation of paravirtual interfaces. Due to the ubiquity of the x86 architecture Palacios is capable of operating across many classes of machines. To date, Palacios has successfully virtualized commodity desktops and servers, high end Infiniband clusters, and supercomputers such as a Cray XT.

2.2 Host OS interfaces

Palacios is designed to be embedded into a wide range of host OSes, encompassing both minimalistic as well as more full featured implementations. Palacios requires only a very minimal set of functionality.

From the host OS perspective, Palacios is just another host OS service However, there are important differences:

- Palacios independently manages paging for guests.
- Palacios can provide guests with direct access to physical resources, such as memory-mapped and I/O-space–mapped devices.

- In certain cases Palacios requires the host OS to provide lowlevel notifications of hardware events, such as keyboard scan codes as well as mouse events.

Palacios expects to be able to request particular services from the OS in which it is embedded. Function pointers to these services are supplied in a `v3_os_hooks` structure:

```
struct v3_os_hooks {
  void (*print)(const char * format, ...);

  void *(*allocate_pages)(int numPages);
  void (*free_page)(void * page);

  void *(*malloc)(unsigned int size);
  void (*free)(void * addr);

  void *(*paddr_to_vaddr)(void *addr);
  void *(*vaddr_to_paddr)(void *addr);

  int (*hook_interrupt)(struct guest_info * vm,
                        unsigned int irq);

  int (*ack_irq)(int irq);

  unsigned int (*get_cpu_khz)(void);

  void (*yield_cpu)(void);
```

```
  void *(*mutex_alloc)(void);
  void (*mutex_free)(void * mutex);
  void (*mutex_lock)(void * mutex, int must_spin);
  void (*mutex_unlock)(void * mutex);

  unsigned int (*get_cpu)(void);
  void (*interrupt_cpu)(struct v3_vm_info * vm,
                        int logical_cpu, int vector);
  void (*call_on_cpu)(int logical_cpu,
                      void (*fn)(void * arg),
                      void * arg);
  void (*start_thread_on_cpu)(int logical_cpu,
                              int (*fn)(void * arg),
                              void * arg,
                              char * thread_name);


};
```

The `print` function is expected to take standard `printf` argument lists and is used to print debugging or performance related messages.

`allocate_pages()` is expected to allocate contiguous physical memory, specifically `numPages` 4 KB pages, and return the physical address of the memory. `free_page()` deallocates a physical page at a time. `malloc()` and `free()` allocate kernel memory and return virtual addresses suitable for use in kernel mode.

The `paddr_to_vaddr()` and `vaddr_to_paddr()` functions should translate

from host physical addresses to host virtual addresses and from host virtual addresses to host physical addresses, respectively.

The `hook_interrupt()` function is how Palacios requests that a particularly interrupt should be vectored to itself. Palacios will acknowledge the interrupt by calling back via `ack_irq()`.

`get_cpu_khz()` and `start_kernel_thread()` are self-explanatory. The Palacios guest execution thread will call `yield_cpu()` when the guest does not currently require the CPU. The host OS can, of course, also preempt it, as needed.

The host OS allocates, configures, initializes, and starts a guest VM using an external API that is exported from the Palacios library.

2.2.1 Optional host interfaces

As can be seen, the host OS interface includes only essential functionality necessary to run and manage a VM environment. These interfaces are required for each OS being targeted by Palacios. While this might seem somewhat limiting, Palacios also supports a number of optional host features that can provide extended functionality. These interfaces are fully optional and Palacios will retain its core functionality even in their absence. Making these interfaces optional is a crucial design decision that allows Palacios to be compatible with a wide range of host operating systems, especially HPC OSes that have a minimal feature set.

Sockets: A host OS with networking support can provide a socket based interface to allow internal Palacios components to communicate over a network. This interface is loosely based on Berkeley Sockets, and provides both TCP and UDP based client and server connections.

Console: A host OS that wants to provide console access to a running VM can implement a special interface that allows Palacios to notify the host of console events. The host

OS receives these notifications and displays them to the user.

2.2.2 Guest configuration

Guest environments are launched at runtime from externally loaded VM images. A VM image consists of a guest configuration as well as packed copies of any binary data (such as disk images) needed by the guest environment. Palacios includes a special user utility that generates these guest images from an XML based configuration file. Once these images have been generated they can be loaded from the host OS and launched via the Palacios control API. The host OS is responsible for actually loading the guest image into memory and passing its address to Palacios. This approach taken by the host OS depends on the host OS architecture as well as the environment it is running in.

The guest configuration itself is expressed through a slightly modified XML syntax. The syntax itself is not type checked, treats attributes and sub-tags equally, is parsed in order, and does not support forward dependencies when referencing other configuration options. The configuration syntax has a set of standard top level tags that are recognized as core configuration options. These options include such things as the amount of memory granted to the VM, the number of cores, the virtualized paging method, and the list of virtual devices assigned to the VM. The configuration file also specifies the paths of any binary data files needed by the VM. The configuration syntax allows arbitrary syntaxes for any sub tag options. Thus a virtual device can have an arbitrary set of configuration parameters that are passed directly to the device implementation. This means that the core configuration implementation does no syntax checking other than to ensure that the configuration is specified with a valid XML format.

2.2.3 Execution process

The host OS is responsible for booting the physical machine, establishing basic drivers and other simple kernel functionality, and creating a kernel thread to run each Palacios core. Palacios reads a description of the desired guest configuration, and calls back into the host OS to allocate physical resources. Having the resources, Palacios establishes the initial contents of the VM control structures available on the hardware (a VMCB for SVM, and a VMCS for VT), an initial set of intercepts, and an initial memory map. It maps into the guest's physical address space a copy of the BOCHS ROM and VGA BIOSes [52]. It then uses the SVM or VT hardware features to launch the guest, starting the guest in legacy real mode identical to a real hardware environment. The guest context begins executing at what looks to it like a processor reset, which results in an entry into the BIOS image mapped into the guest's memory space. The BIOS handles the first stage of the guest OS boot process: initializing hardware devices, loading a boot sector from some storage medium, and jumping to it.

2.3 Core architecture

Palacios is an OS independent VMM, and as such is designed to be easily portable to diverse host operating systems. Palacios integrates with a host OS through a minimal and explicitly defined functional interface that the host OS is responsible for supporting. Furthermore, the interface is modularized so that a host environment can decide its own level of support and integration. Palacios is designed to be internally modular and extensible and provides common interfaces for registering event handlers for common operations. Figure 2.2 illustrates the Palacios architecture.

Palacios fully supports concurrent operation, both in the form of multicore guests as well as multiplexing guests on the same core or across multiple cores. Concurrency is

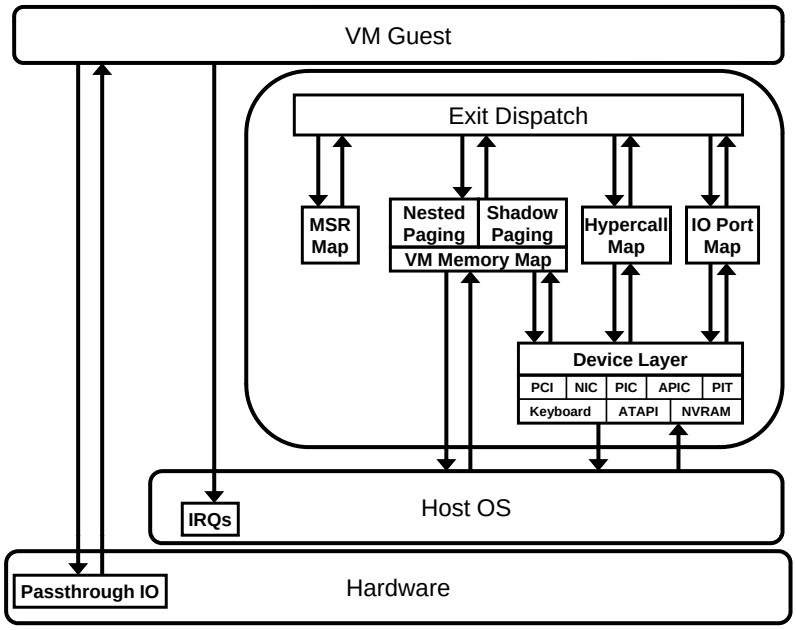


Figure 2.2: High level architecture of Palacios. Palacios consists of configurable components that can be selectively enabled. The architecture is designed to allow extensibility via event routing mechanisms.

achieved with a combination of local interrupt masking and host OS provided synchronization primitives. The behavior of the synchronization primitives is dependent on the host OS, for instance waiting on a lock can be done via either busy waiting or sleeping. The primitives are available to Palacios via a set of OS interface hooks that export a mutex style interface.

2.3.1 VM exits and entries

As explained in Section 2.1 VMMs are primarily designed as event dispatch loops that trap exceptional events that occur while a VM is executing. This means that a guest executes just as it would natively on real hardware until an exceptional condition occurs. The occurrence of an exceptional condition causes a *VM exit*. On a VM exit, the context of the guest OS is saved, the context of the host OS kernel (where Palacios is running) is restored, and execution returns to Palacios. Palacios handles the exit and then executes a VM entry, which saves the host OS context, restores the guest context, and then resumes execution at the guest instruction where the VM exit occurred. As part of the VM exit, hardware interrupts may be delivered to the host OS. As part of the VM entry, software-generated interrupts (virtual interrupts) may be delivered to the guest. This architecture allows Palacios to virtualize a subset of the hardware resources in a way that is completely transparent to the guest environment. At a very high level, the Palacios kernel thread handling the guest looks like this:

```
guest_context = processor_reset_context;


while (1) {
        disable_host_interrupts();
```

```
        (exit_reason, guest_context) =
                              vm_enter(guest_context);

        enable_host_interrupts();

        guest_context =
           handle_exit(guest_context, exit_reason);
}
```

By far, the bulk of the Palacios code is involved in handling exits.

The notion of exceptional conditions that cause VM exits is critical to understand. Exceptional conditions are generally referred to either as exit conditions (Intel) or intercepts (AMD). The hardware defines a wide range of possible conditions. For example: writing or reading a control register, taking a page fault, taking a hardware interrupt, executing a privileged instruction, reading or writing a particular I/O port or MSR, etc. Palacios decides which of these possible conditions merits an exit from the guest. The hardware is responsible for exiting to Palacios when any of the selected conditions occur. Palacios is then responsible for handling those exits.

The above description is somewhat oversimplified, as there is actually a third context involved, the shadow context. It is important now to explain what is meant by each of host context, guest context, and shadow context. The host context is the context of the host OS kernel thread running Palacios. The guest context is the context which the guest VM believes it is running in. This includes register contents, internal CPU state, control structures and registers, as well as the actual software state of any code running in the VM. Code executing in the guest context has full access to change any part of the guest context, though it should be noted that certain changes can trigger VM exits. The important part to

keep in mind is that the guest context is actually abstract. It is only the context that the guest thinks it has, not the context that is actually instantiated. The actual context which is used to execute a VM is called the shadow context. The shadow context includes the execution state of a guest that is actually loaded onto the hardware. That is the shadow context is the actual processor context that is being used when the guest is running. This means that the guest does not really run in guest context, it just thinks it does. In reality, it is the shadow context, which is fully managed by Palacios, that runs the guest environment. This separation of the guest and shadow context is what makes hardware virtualization support necessary. The hardware is configured to cause a VM exit whenever a guest attempts to change an important component of the guest context. This VM exit traps into Palacios (the host context) where its operation is emulated such that the modification appears to occur in the guest context, while a modified version of the operation is executed on the shadow context. The reasoning behind this will become apparent during the discussion of shadow paging.

2.3.2 Resource hooks

During normal operation it is common for a guest to attempt to interact with the guest context in a manner that cannot be fully captured by the shadow context. In other words the guest will try to access a hardware component for which the hardware does not support a shadow version. This can include things such as hardware devices, special control registers, or specially handled memory regions. When such an event occurs, the only recourse is to emulate its behavior inside the VMM. To facilitate this common behavior, Palacios provides an extensive interface to allow VMM components to register to receive and handle these guest events.

Special handlers are implemented in Palacios to emulate the operations involving resources such as MSRs and CPUIDs, as well as hypercalls. Each of these resources is

actually a collection of resources, with an associated unique ID used to determine which MSR or which CPUID is being operated on. A special handler framework is implemented for each resource, that allows a VMM component to register itself as a handler for a specific resource ID. When a handler is registered, it includes a set of callback functions which are called whenever the resource needs to be emulated. When an operation is performed such as an RDMSR or WRMSR, the framework determines the appropriate handler for the MSR in question, and invokes the callback functions to handle the emulation.

Because Palacios is capable of trapping and emulating such a large number of possible actions both inside and outside of the guest execution context, it is fully capable of emulating a wide range of possible behaviors. This functionality makes it possible to construct a large and diverse set of different guest environments.

2.4 Memory architecture

Achieving decent performance in a VM requires efficient virtualization of the guest environment's memory address space. All modern OSes already include support for a single address virtualization layer, referred to as virtual memory. In this case an OS uses page tables loaded in hardware to automatically translate virtual memory addresses to physical addresses that are used to access physical memory. These page tables are structured as a tree hierarchy that is indexed via specific segments of the virtual address being translated. This translation is performed automatically by hardware, and includes a number of optimizations including a special cache called a Translation Lookaside Buffer (TLB). Much of the complexity needed in a VMM architecture is a result of having to virtualize this address translation hardware.

A VMM cannot allow the guest to establish a virtual address mapping to any physical address, as this could allow the guest to conflict with Palacios, the host OS, or other guests.

At the same time, VMMs must maintain the illusion that the guest is running, by itself, on an actual machine in which mappings to any physical address are permitted. This requires a new memory address abstraction layer that is located between the physical hardware and what the guest perceives as hardware. Conceptually, this new abstraction is implemented using two levels of memory address mappings. Virtual addresses in the guest ("guest virtual addresses") map to "physical" addresses in the guest ("guest physical addresses") using the guest's page tables, and these guest physical addresses map to "host physical addresses" (real physical addresses) using a separates set of page tables created by the VMM.

There are two standard methods for VMMs to implement this new virtualized paging abstraction. The first method, required by the early versions of SVM and VT, is called shadow paging, and requires the VMM to fully emulate the address translation hardware in software. The second method, introduced in current versions of SVM and VT, is called nested paging, and uses new hardware extensions to perform the additional address translation directly in hardware.

I will now discuss in more detail how Palacios virtualizes a guest environments memory address space, as well as examine how Palacios implements both shadow and nested paging.

2.4.1 Memory map

Palacios maintains its own representation of the mapping between guest physical addresses and host physical addresses, which will be referred to as the *memory map*. This memory map separates the actual address translation mapping from the architectural representation used for either shadow or nested paging. This memory map contains the translations between every guest physical address and the corresponding host physical address that has been allocated by Palacios. As shown in Figure 2.4.1, the memory map is designed to be as

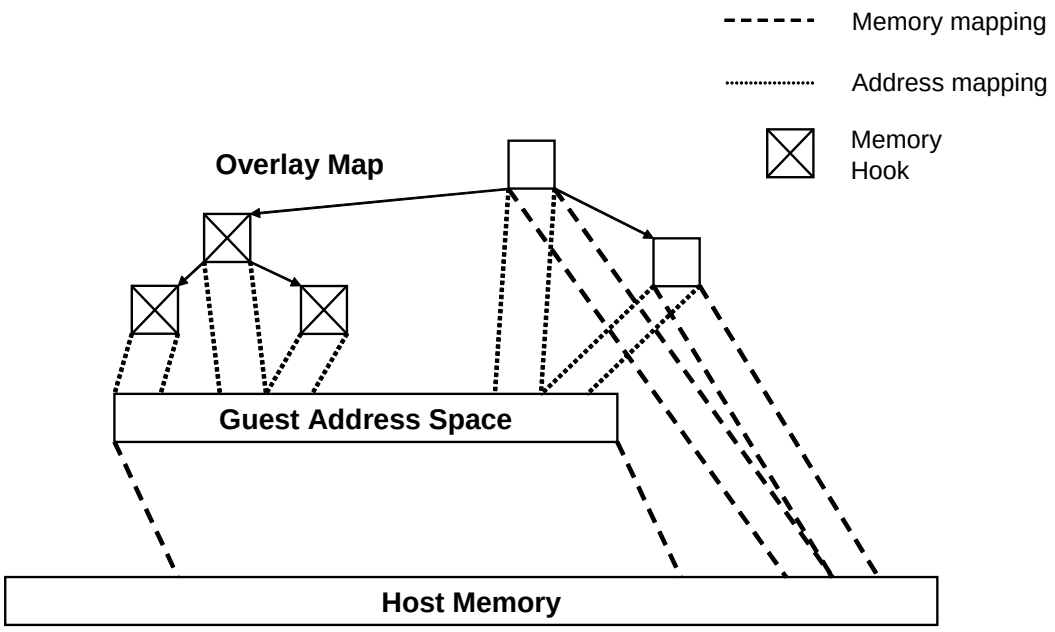


Figure 2.3: The Palacios memory map architecture. Guest memory is contained in a single preallocated physically contiguous memory region. Additional memory regions can be specified and stored in a red black tree. These sub regions provide guest memory redirections as well as emulated memory regions via memory hooks.

efficient as possible in both the space and time domains. This means that the data structures needed for the map are compact and translation lookups can be done very quickly.

The most important aspect of Palacios' memory management system is that guest memory is preallocated in a single physically contiguous chunk. This means that at the basic level, a guests memory address space is a single block that is located at some offset in the host's physical memory space. Memory translations are thus accomplished by simply adding a single offset value to every guest physical address. This contiguous chunk of memory is referred to as the base memory region, and is responsible for the translations corresponding to the vast majority of all guest memory accesses.

While a single memory segment is capable of handling most address translations, most guest environments require special handling for a small set of specific address ranges. These ranges belong to such things as emulated devices that are accessed via memory mapped I/O, as well as special mappings for physical devices that the guest has been given direct access to. To support these special ranges, Palacios implements a second memory map that is overlaid onto the base memory region. This overlay map is implemented as an ordered red black tree keyed to the lowest guest physical address contained in a region. It should be noted that generally the contents of this red black tree are very few in number, generally no more than 5 for standard guest environments. These regions use the same data structure as the base region, and include the size of the region, the guest physical address it maps from, and the host physical address it maps to.

Address translations are accomplished by first performing a lookup of a guest physical address in the overlay map. If a special region containing that address is found it is returned to the caller. If the address is not contained in any special region, the lookup falls through and the base memory region is returned. It is possible to also create special regions that have no corresponding host physical addresses. These regions instead contain a reference to a special function handler that is called whenever an address in that region

is accessed. The most common use case for these regions is a memory hook, in which the VMM emulates the accessing operation in software. Memory hooks will be discussed in more detail later.

2.4.2 Shadow paging

As stated earlier, some hardware features are not directly supported by the SVM and VT hardware virtualization extensions, which requires a VMM architecture to implement a special shadow context to ensure correct execution. Early versions of these extensions lacked this support for paging features, and so required that page tables be implemented inside the shadow context. This approach is commonly referred to as *Shadow Paging*. Palacios supports shadow paging as the default virtual paging mechanism, since it is guaranteed to be supported by all CPU versions.

Shadow paging is implemented via a second set of page tables that exist in the shadow context. These are the page tables that are actually loaded into hardware when a guest begins executing. These shadow page tables map guest virtual addresses directly to host physical physical addresses, as configured in the VM's memory map. These shadow page tables are generated via a translation process of the guest page tables (the page tables located in the guest context). Recall that the guest page tables translate guest virtual addresses to guest physical addresses, which are not valid memory locations since the guest memory space is located at some offset in physical memory. Therefore when a guest attempts to activate its set of guest page tables, the operation is trapped into a VMM where a set of shadow page tables is loaded into the hardware instead. Furthermore, any changes made to the guest's page tables are trapped by Palacios, which makes corresponding changes to the shadow page tables. This propagation of information happens through two mechanisms: page faults and reads/writes to paging-related control registers, both of which cause VM exits.

The core idea behind Palacios's current shadow paging support is that it is designed to act as a *virtual TLB*. When a guest initially loads a new set of page tables, it is trapped by Palacios which instead loads an empty page table into hardware. This is equivalent to a TLB flush operation that occurs whenever page tables are switched on real hardware. As the guest continues to execute it accesses memory addresses that trigger page faults, due to the lack of any page table entries. These page faults are trapped by Palacios, which in turn must update the shadow page tables to allow the access to succeed. This is done by first determining the virtual address causing the page fault, and then looking up the guest's page table entry associated with that virtual address. This guest page table entry includes a guest physical address that must be translated to a host physical address via a lookup in the VM's memory map. Once the host physical address is located, the page permissions are reconciled between the guest page table entries and the permissions included in the memory map. This information is then used to generate a shadow page table entry that is accessible by the hardware. This behavior is consistent with the behavior of a hardware TLB.

Of course, some page faults need to be handled by the guest itself, and so a page fault (on the shadow page tables) which causes an exit may result in the handler delivering a page fault (based on the guest page tables) to the guest. For example, a page table entry in the guest may be marked as not present because the corresponding page has been swapped to disk. An access to that page would result in a hardware page fault, which would result in a VM exit. The handler would notice that the shadow page table entry was in sync with the guest, and therefore the guest needed to handle the fault. It would then assert that a page fault for the guest physical address that originally faulted should be injected into the guest on the next entry. This injection would then cause the guest's page fault handler to run, where it would presumably schedule a read of the page from disk.

2.4.3 Nested paging

More recent CPUs actually include hardware support for virtualized paging. This support takes the form of a second layer of page tables that translates from guest physical addresses to host physical addresses. This is commonly known as *Nested Paging*. Nested paging requires n^2 page table lookups in hardware to translate first from a guest virtual address to a guest physical address, and finally from a guest physical address to a host physical address. In essence nested page tables are a direct hardware representation of the VM's memory map that is readable by the CPU. This allows a guest OS to directly load its own page tables and handle most page faults without requiring VM exits to first determine if shadow page table updates are necessary. Unfortunately this requires an additional hardware cost for each page table lookup, since it now must traverse two page tables instead of just one. The impacts this has on performance are well documented, and several potential solutions have been proposed [9, 34]. In general the software complexity needed for nested paging is considerably less than for shadow paging.

2.5 I/O architecture and virtual devices

There are generally three types of devices in modern x86 based systems. The first set of devices are those that have collectively come to define the x86 hardware platform. These devices include components such as interrupt controllers, hardware timers, northbridge, and southbridge chipsets. Because this set of devices will always be present in any system, modern OSes have all been designed around the assumption that they will be available. The second set of devices are almost as common as the first type but are not necessarily required. These devices include the PCI and IDE buses, and generally serve as interconnects which other devices use to interface with the OS and other devices. While these devices are not necessarily required for the OS to function correctly, their absence would introduce

severe limitations on OS functionality. The final set of devices is made up of a very diverse collection of hardware. These include common I/O devices such as network and graphics cards. Each of these devices exists as a non standard component that can only be accessed via a specific and often proprietary interface. While the first two types of devices are designed according to well established standards, the final set of devices generally have no standard interface at all. It should be noted that the majority of hardware devices fall into the third group.

The differentiation in types of devices is very important for VMMs. In order for a VMM to provide a virtual environment that is capable of supporting a standard OS, it must include the first set of devices that the OS expects to find. A VMM must also support the second type of devices if it intends to support OSes that provide more than just a very basic set of features. Finally to support most of the features available in an OS, such as a file system and network connectivity, the VMM must provide at least a limited set of devices from the third set. That is the VMM must provide at least one virtual network card, and at least one type of block storage device. Furthermore, because hardware virtualization extensions currently support virtualization of the CPU and only very limited virtualization of devices, the VMM itself must handle the virtualization of each of the required devices, typically through emulation. Every full featured VMM currently available includes a set of virtual devices, that meets the criteria listed above. Palacios follows this pattern by including a set of emulated devices consisting of common hardware components. Each of the devices included with Palacios has been implemented from scratch as part of the project.

Due to the fact that virtualized devices are so important to the operation of a virtualized guest OS, Palacios provides an extensive support for incorporating virtual devices into the system. This support consists of a framework that exports a set of common interfaces which allow virtual devices to easily be integrated into the code base as well as instantiated

and configured for any VM.

The virtual device interface itself includes specific support for each of the device classes I have described. Because the first class of required devices interacts closely with the virtual environment, these devices are designed to interact directly with the core VMM through specialized interfaces designed specifically for each device. The second class of interconnect devices each export their own interface to allow other devices to connect to them. For instance the emulated PCI bus provides a number of different functions to allow other devices to register themselves onto the bus. Finally, the third type of devices are designed to focus purely on data transfer, and are designed to allow the utmost flexibility in how the data transfer is accomplished.

Each device from the third set is architected using a split device model. This architecture allows the separation of the actual device interface from the mechanisms used to actually transfer the data. This allows a large degree of flexibility in how virtual devices are actually implemented by the VMM. The device frontend implements the actual emulation of the hardware interface. It is the frontend that the guest actually interacts with directly by sending control commands and receiving interrupts. The device backend implements the actual data transfer, and interacts with the frontend via standard interfaces implemented for each general class of device. For instance, storage devices transfer data using read and write functions that operate on parameters that include the offset and length of the data on the actual storage medium. The underlying method used by the backend to transfer data is fully independent of the frontend behavior, and can be implemented in any number of ways. For example, Palacios supports virtual ATA disk devices via a virtualized IDE bus. The disk frontend implements a fully implemented ATA command interface, and translates each transfer command into a general read/write operation. These read/write functions are implemented in a set of backends which can either operate on an in memory ramdisk image, over network to a remote disk, or to a disk image located on a local file

system. This split architecture allows a wide variety of device behaviors to be implemented using generic interfaces that do not have to implement full hardware specifications.

The current list of included virtual devices is (not including device backends):

- NVRAM and RTC
- Keyboard/Mouse [PS2]
- Programmable Interrupt Controller (PIC) [based on Intel 8259]
- Programmable Interval Timer (PIT) [based on Intel 8024]
- PCI Bus
- Multi Channel IDE BUS [ATA/ATAPI command support]
- Local APIC / IOAPIC
- Northbridge [based on Intel 440FX chipset]
- Southbridge [based on Intel PIIX 3]
- Serial Port [Based on generic UART spec]

VirtIO Palacios supports a suite of virtual devices based on the Linux VirtIO interface [82]. VirtIO is a minimalist device interface designed specifically for virtualized software environments. VirtIO devices are structured as a set of ring buffers located inside a guest's address space. These ring buffers store DMA descriptors which a virtual device uses to transfer data. VirtIO is a general framework which can be used by many classes of virtual devices. The Linux kernel includes drivers for VirtIO based network, block, console and balloon devices. Palacios includes support for the standard set of Linux VirtIO devices, and also uses VirtIO to implement more specific devices.

Virtual device interface The implementation of virtual devices in Palacios is facilitated by its virtualized paging, I/O port hooking, and interrupt injection support. Palacios's shadow paging support provides the ability to associate ("hook") arbitrary regions of guest physical memory addresses with software handlers that are linked with Palacios. This is the mechanism used to enable memory-mapped virtual devices. Palacios handles the details of dealing with the arbitrary instructions that can generate memory addresses on x86. Similarly, Palacios allows software handlers to be hooked to I/O ports in the guest, and it handles the details of the different kinds of I/O port instructions the guest could use. This is the mechanism used to support I/O-space mapped virtual devices. As we previously discussed, Palacios handlers can intercept hardware interrupts, and can inject interrupts into the guest. This provides the mechanism needed for interrupt-driven devices. I will now explain the registration and configuration process used by the device framework. The resource hook interfaces will be explained in later in Section 2.5.2.

Each device in Palacios is instantiated only in response to its existence in a VM's configuration file. Each guest configuration includes a special section that includes an enumeration of the devices the guest will use as well as their associated configuration options. Each device is defined using a subtree in the XML configuration file which is passed directly to the device implementation. This means that every device can include its own configuration parameters and not rely on a global configuration syntax. This allows a large degree of configurability in device behavior, as each device can include special options in an ad hoc manner. The only constant configuration syntax is the top level tag which specifies the global device type. When a VM is being initialized, Palacios iterates over these tags and performs a lookup of the global device type. This lookup returns the device descriptor which is used to instantiate the device and connect it to the new VM.

When the device's initialization function is called it is responsible for creating a device instance and attaching it to the VM. This is done by first calling `v3_allocate_device()`,

which takes as arguments a device ID, a collection of function pointers which define the generic virtual device interface, and an opaque pointer used to store instance specific state. This function returns a pointer to the newly created device in the form of a `struct vm_device *` data type. Once the device has been allocated, it must be attached to the VM context via a call to `v3_attach_device()` whose arguments are the pointer returned from `v3_allocate_device` and the VM context it is being attached to.

2.5.1 Interrupts

All modern hardware devices rely heavily on hardware generated interrupts in order to function correctly. Interrupts are used to signal the OS that some event occurred, such as a network packet arrival or the completion of a particular I/O operation. Interrupts themselves are differentiated by an index value that is generally uniquely assigned to each hardware device. The guest OS is responsible for providing a special handler for each interrupt that reacts to the event that occurred. For instance, when an interrupt occurs for a received packet from a network card, the interrupt handler copies the received packet into a memory buffer. Obviously, interrupts are an integral component of all modern hardware devices, so any VMM must include some facility to inject interrupts into a running guest OS.

Interrupt delivery in the presence of a VMM like Palacios introduces some complexity. Both Intel and AMD include extensive support for virtualizing both hardware and software interrupts. This allows a VMM a large degree of freedom in how it responds to actual physical interrupts as well as how it forwards interrupts to the guest context. Palacios is currently designed to automatically exit whenever any hardware interrupt (IRQ) occurs. This exit occurs before any IRQ acknowledgment occurs, so once Palacios exits the interrupt is still pending in the host context. Therefore, as soon as Palacios re-enables interrupts

following the exit, control is vectored to the host operating system's interrupt handler. This is necessary to ensure that the host OS is capable of responding to hardware events in a timely manner.

Palacios supports a wide variety of interrupt sources. For instance interrupts injected into a guest may originate from a virtual device implementation, a core VMM component, or an actual hardware device. To support all of these possibilities, Palacios includes a general framework that allows any VMM component to raise any possible interrupt. In practice this is rarely used, and interrupts are instead injected via secondary interfaces. This interface routes interrupts through emulated interrupt controller hardware, and tracks the interrupt vectors that are waiting to be injected. Before each VM entry, Palacios queries the interrupt framework to determine if there are any pending interrupts that need to be injected. If so, Palacios configures the entry state of the guest to perform a hardware injection of the appropriate interrupt. Both SVM and VT provide support for injecting virtualized interrupts in such a way as they appear to be actual hardware interrupts inside the guest context. The hardware automatically vectors control of the VM to the appropriate interrupt handler defined by the guest OS. A very similar mechanism is used to inject hardware exceptions.

It is important to note that raising an interrupt in Palacios does not guarantee that the associated guest interrupt handler will be called immediately upon the next VM entry. Raising an interrupt in Palacios only marks it as pending, with the actual injection dependent on a number of situations. For instance, *when* an interrupt is actually injected into the guest depends on the guest's current configuration of the interrupt controller hardware, whether it has interrupts disabled, and the priorities of other pending interrupts.

In order to allow direct access to a device from a VM, Palacios also supports an interface that allows particular hardware interrupts to be forwarded directly to the guest context. This interface is part of the external interface implemented by the host OS. The current im-

plementation requires that Palacios explicitly forward the interrupt to the guest, instead of reconfiguring the hardware to handle it. These interrupts are treated exactly the same as virtual interrupts, and use the framework described above.

Host events

Virtual devices implemented in Palacios are fully capable of configuring physical interrupts to be routed to Palacios via callback functions. For example, a virtual keyboard controller device could hook interrupts for the physical keyboard so that it receives keystroke data. The Palacios convention, however, is generally to avoid direct interrupt hooking for virtual devices. Instead, we typically have virtual devices export custom callback functions that can then be called in the relevant place in the host, at the host's convenience. For example, our keyboard controller virtual device is implemented as follows. When the physical keyboard controller notes a key-up or key-down event, it generates an interrupt. This interrupt causes an exit to Palacios. This interrupt then triggers the handler implemented inside the host OS' keyboard device driver. The driver reads the keyboard controller, and returns the keystroke data to the host OS. The host OS determines whether the keystroke was meant for one of its own processes or a Palacios VM. Any keystroke data meant for a VM is delivered directly to Palacios where it is forwarded to the emulated keyboard device. Upon receiving the data, the virtual device updates its internal state and raises the appropriate virtual interrupt through Palacios. This will then trigger the guest's keyboard interrupt handler, which will then read the keystroke from the virtual device.

The host event framework is the interface that enables a host OS to forward higher level events, such as a full keystroke, into Palacios. The framework implements a publish/subscribe mechanism where the host OS publishes an event that is then delivered to every Palacios component that has subscribed to it. Currently, Palacios supports keyboard, mouse, and timer event notifications. These events must be generated from inside the host

OS, and currently target specific virtual devices that forward the interactive events to a VM.

2.5.2 Emulated I/O

Modern hardware devices implement three common mechanisms that can be used to control a device and transfer data between it and the OS. These three mechanisms are memory mapped I/O, Direct Memory Access (DMA), and I/O ports. Each of these interfaces must be fully virtualized and emulated in order to support a complete virtual device framework. Palacios handles the emulation details for these operations, and translates them into standard callback procedures that are individually implemented by each device. This allows virtual devices to associate a given software handler to each I/O port and memory region in use by the device. Each device typically "hooks" a common set of either I/O ports or memory regions to enable interaction with the guest OS. For many devices these ports and regions are generally at well known locations and so are often hardcoded into the devices implementation. For other devices, the virtual PCI device handles the allocation and registration of the I/O ports and memory regions used by a device.

DMA operations, the first method of device interaction, are left to each device to implement internally. The reason for this is that they consist entirely of simple memory copies to/from guest physical memory. Palacios provides an address translation interface that allows a device to easily generate a host address from a given guest address. Therefore, all that is needed to implement a DMA operation is a translation from the guest physical address given in a DMA descriptor to a host virtual address that can be operated on directly by the virtual device implementation. The DMA operation is then typically achieved via a `memcpy()`, which does not warrant further discussion. Instead the remainder of this section will focus on I/O ports and memory mapped I/O.

I/O ports

I/O ports are the traditional x86 method of interacting with hardware devices. The x86 architecture includes a 16-bit *I/O* address space (the addresses are called *I/O ports*) that is accessed with a special set of IN and OUT instructions. These IN/OUT instructions operate on small amounts of data loaded to and from registers. As such, I/O ports are more suitable for device configuration and control instead of actual data transfer. Even though most new devices are moving to memory mapped I/O, I/O ports are still used by a number of common devices. It is thus necessary to allow virtual devices the ability to use I/O ports for communication with a guest OS.

Palacios provides an interface whereby a device can register itself as a handler for a given set of I/O ports. This process is called *I/O hooking*, and involves hooking a pair of software handlers to a specific I/O port. In this manner, whenever the guest OS performs an IN or OUT operation on a I/O port, that operation is translated to a read or write function call to a specified handler. Each handler is responsible for emulating whatever action the I/O operation was meant to accomplish.

The I/O hook interface consists of a registration function, and a set of read and write handlers. For each port that is to be hooked, a call is made to `v3_hook_io_port` which takes as arguments the port number and two function pointers referring to the read/write handlers. Each guest I/O operation will result in one call to the appropriate read/write function, where the read handles IN instructions (reading data into the guest) and the write handles OUT instructions (writing data from the guest). Palacios also supports the dynamic hooking and unhooking of I/O ports, since some devices (such as PCI) allow an OS to remap I/O ports between devices.

The actual translation and emulation of the I/O operations is accomplished by Palacios before the handlers are called. Furthermore, once the handlers return, Palacios updates

the guest register state to reflect any side effect of the operation such as loading a value from a device into a guest hardware register. This is relatively easy because virtualized I/O operations generally enjoy a large amount of hardware support via the virtualization extensions, such as operand decoding. The hooks themselves are stored in a red-black tree that is keyed to the port number associated with each hook, which allows fast hook lookups. When the guest performs an I/O operation, the hardware is able to determine whether or not an exit should occur (SVM allows VM exits to be configured per port). Palacios will then dispatch the exit to the generic I/O handlers, which will query the list of hooks to find the appropriate callback functions. Palacios then translates the I/O operation to an appropriate read or write call that is dispatched to the appropriate hook callback function. Once the operation is handled, Palacios ensures that the guest state is updated to reflect a successful emulation of the operation.

Memory mapped I/O

The second I/O method consists of memory mapped operations in the guest's address space. Memory mapped I/O relies on the actual hardware to reroute memory operations in a given address range to an associated device. This allows an OS to configure a device using standard memory instructions, instead of the specialized IN/OUT operations. This is beneficial because it allows a larger configuration space that can be manipulated using standard memory operations. Unfortunately, while memory mapped I/O is easier to use in an OS, it is much harder to handle in a VMM. There are a number of reasons for this, such as an increased number of potential instructions performing the I/O operation, and a larger address space which they can operate in. Because of this increased complexity, both VT and SVM provide minimal support for these operations relative to port based I/O.

Conceptually, the behavior of a memory mapped I/O operation is almost identical to a I/O port operation, when viewed from a virtual devices viewpoint. Both operations are hid-

den behind a *hook* abstraction, that allows a virtual device to register a set of read and write handlers to a given region of guest memory. Whenever a guest performs an operation on the region a VM exit occurs, wherein Palacios decodes and translates the operation into one of the two function handlers associated with the memory hook. The registration and handlers are essentially the same: Hooks are registered via either `v3_hook_full_mem()` or `v3_hook_write_mem()` (whose differences will be discussed shortly), and are invoked through either a read or write callback function.

The underlying framework that enables memory hooks is rather complex and consists of components that support hook registration, memory operation trapping, instruction decoding, and instruction emulation. The basic building block for memory hooks is the 4KB memory page due to the fact that the memory hook framework is based around the virtualized paging infrastructure. For every memory hook registered with Palacios, at least one guest physical memory page is reserved for hooked operations. These pages are treated as a special case by the virtualized paging system, and configured to always generate a page fault when they are accessed. These page faults are trapped by Palacios which then translates the operation to be dispatched to the hook's callback functions.

The memory hooked pages are managed by Palacios' memory map and virtualized paging system. When a memory hook is created, a new memory region is created in the memory map that includes the range of pages associated with the hook. These regions are tagged as being hooked regions, which the paging system uses when constructing either the shadow or nested page tables. As stated earlier there are two types of memory hooks: write only hooks which only trap write operations but allow read operations to proceed without emulation, and full hooks where both read and write operations are fully emulated by Palacios. Page table entries associated with full memory hooks are marked as not present, which forces a VM exit due to a page fault whenever the memory is read or written. Memory write hooks retain an actual page of memory which the guest can read

directly from, however these pages are marked as read-only so any write operation will trigger an exiting page fault. As described in Section 2.4, when Palacios detects a page fault has occurred, it performs a lookup in the memory map to determine whether any action needs to be taken by the VMM. If it finds that a hooked page is being accessed, it begins the translation and emulation process.

I will now describe the implementation of a write operation performed on a hooked memory region, a read operation is simply the inverse of the steps I describe below. In order to determine what action is being performed on the hooked memory region, Palacios must first decode and then emulate the instruction which caused the fault. The decode phase is handled by the integrated Xed [60] instruction decoder. The decoder is used to return the instruction type as well as memory addresses for each of the instruction's operands. Note that the entire guest state has been copied into host memory at this point, so register operands can be referenced via a pointer to the in memory version of the registers. Once the instruction and operands have been determined Palacios must then emulate the instruction to determine what the final value would be. This emulation is necessary because not all memory references are simple MOV operations, they can include arithmetic and boolean operators among many others. The emulation is accomplished using an emulation framework which performs the operation on the in memory versions of the decoded operands. Once the final value has been determined it is passed as an argument to the hooks write callback function, along with the guest address of the operation. This allows a virtual device handling the hooked operations to determine exactly what operation was performed on the hardware device being implemented. For a read operation the above steps are the same with the difference being that the callback function is called first to read the value used in the memory operation. Once this value has been read from the virtual device, the faulting instruction is emulated using that returned value. Once the memory operation has been fully handled, execution is returned to the guest.

Similar to the I/O port hooks, memory hooks can be dynamically hooked and unhooked. This is a bit more complicated for memory than I/O ports, as it requires an invalidation of any existing page tables which have been previously configured based on the initial hook configuration. In shadow paging this requires deleting all of the shadow page tables, as the only alternative is an exhaustive search for any entry pointing to the hooked regions. Nested paging simply requires a TLB invalidation of the nested page table entry referring to the hooked regions.

Passthrough I/O

In addition to hooking memory locations or I/O ports, it is also possible in Palacios to allow a guest to have direct access, without exits, to given physical memory locations or I/O ports. This, combined with the ability to revector hardware interrupts back into the guest, makes it possible to assign particular physical I/O devices directly to particular guests. While this can achieve maximum I/O performance (because minimal VM exits occur) and maximum flexibility (because no host device driver or Palacios virtual device is needed), it requires that (a) the guest be mapped so that its guest physical addresses align with the host physical addresses the I/O device uses, and (b) that we trust the guest not to ask the device to read or write memory the guest does not own. A full description of Passthrough I/O will be presented in Chapter 5.

2.6 Currently supported host operating systems

As I mentioned, one of the central design goals of Palacios to implement an OS independent VMM that is widely compatible with a large collection of host operating systems. So far Palacios has been integrated into a significant set of diverse OSes by myself and others.

The initial OS we targeted was a very small and simple teaching OS called GeekOS.

GeekOS provides only the very basic services required by an OS, and as such proved to be an ideal candidate for our initial development effort. Because GeekOS had so few moving parts as it were, we were able to very easily modify and extend it to support Palacios. In essence it did just enough to boot the machine, and then got completely out of the way. By starting with GeekOS I was able to see early on that a fully featured VMM architecture could very well be designed with minimal requirements of the host OS it is integrated into. The compatibility and OS independent nature of Palacios can be traced directly to its origins as an extension to GeekOS.

The second OS Palacios was integrated with was the Kitten lightweight kernel developed at Sandia National Labs. Kitten is a new OS borrowing heavily from the Linux codebase, but focused on HPC and supercomputing environments. The integration of Palacios and Kitten was done over two days at a joint development meeting between the Kitten and V3VEE developers. Kitten is a minimalistic OS, with slightly more features that GeekOS. The integration with Kitten firmly cemented design of the minimal OS interface as well as the decision to package Palacios as a static library that can be linked into an OS during compilation. Kitten is currently the primary development OS for Palacios.

Recently, Palacios has been integrated into the latest version of MINIX. MINIX is a full featured OS designed for commodity environments, and as such provides a platform for many of the advanced features in Palacios. These include such things as the socket and file system interfaces.

Finally, there is an ongoing effort among a group of Northwestern undergraduate students to integrate Palacios into current versions of the Linux kernel. While this is still in an early stage of development, its possibility serves to demonstrate that Palacios is capable of running in a very diverse range of host OS environments.

2.7 Contributors

Palacios is a large project with many contributors spread across Northwestern University, the University of New Mexico, Sandia National Labs, and several other sites. While I have been the primary designer and developer, Palacios would hardly be where it is today without the contributions, advice, and guidance of many others. Everyone who has been a part of this project has my sincere gratitude. The following is a list of the main contributors.

Much of the original core of Palacios, including the low level hardware support, was originally developed in close collaboration with my advisor, Peter Dinda. He is also responsible for numerous bug fixes, architectural decisions, and component implementations including the keyboard, mouse, NVRAM, and RTC devices.

Patrick Bridges has been the source of countless pieces of advice and guidance, bug fixes, and getting Palacios to run on new hardware environments.

Many other graduate students have also been involved in Palacios' design. The original virtual PCI bus was implemented by Lei Xia and Chang Bae. The port of VNET into Palacios involved Lei Xia, Zheng Cui, and Yuan Tang, which included the VNET routing core, the VirtIO based virtual network devices, and Dom0 guest support. Lei Xia also did much of the early work to incorporate the XED decoding library. The first virtual IDE layer was implemented by Zheng Cui. Chang Bae explored the use of shadow paging optimizations.

The collaboration with the Kitten developers has been a large part of the project from very early on. Kevin Pedretti and Trammell Hudson in particular have been a tremendous source of help in working on the Palacios/Kitten interface, tracking down bugs, and offering much advice on large scale system development.

Numerous undergraduates have also contributed to the design of Palacios. Andy Gocke implemented the original support for Intel's VT extensions. Philip Soltero has fixed a

number of bugs, built multiple guest environment configurations, and performed numerous performance studies. Steven Jaconette built the direct paging framework as well as the virtual CGA console device together with Rob Deloatch. The virtual serial port emulation was completed by Rumou Duan. Jason Lee, Madhav Suresh, and Brad Weinberger are currently working on the port of Palacios to Linux. And many others have contributed reference implementations and exploratory work. Matt Wojcik and Peter Kamm worked on networking support for our early targeted host OS.

Externally, Erik van der Kouwe has ported Palacios to MINIX, contributed the console interface, and sent along numerous bug fixes.

2.8 Conclusion

This chapter has introduced the design and implementation of the Palacios virtual machine monitor. Palacios is an OS independent embeddable VMM architecture, that is widely portable to many environments. Palacios has been deployed on a wide range of hardware including supercomputers, HPC clusters, and commodity desktop hardware. Palacios has also been ported to a wide range of different host environments such as the Kitten lightweight kernel, MINIX, and Linux. Palacios is very configurable, allowing it to target specific environments, and includes many optional features that can be deployed as needed. Palacios is also thoroughly extensible, allowing new virtual environments to be implemented and deployed relatively easily.

The nature of Palacios makes it an ideal candidate for deployment on HPC systems. In order to evaluate Palacios as an HPC VMM we have collaborated with Sandia National Laboratories, and analyzed Palacios running on the RedStorm Cray XT supercomputer. The next chapter will detail the results of this evaluation.

Chapter 3

Palacios as an HPC VMM

As explained earlier, one of the primary design goals for Palacios was to create a VMM architecture that could effectively virtualize high performance computing resources. HPC environments pose many challenges for system designers that are not seen in commodity systems. Extremely large scales and tightly integrated parallel execution are very sensitive to small performance losses. In fact, minor performance loss at only a single node can result in ripple effects across the entire system. Systems that are not specifically designed to achieve tightly coupled scalability will not be able to effectively function in these environments.

HPC is a ripe area for virtualization because there is a substantial amount of interest in using virtualization features to improve the reliability, maintainability, and usability of HPC resources. While there has been effort put into adapting existing virtualization tools for HPC environments, no virtualization architecture has yet been designed specifically for this area.

There are two standard approaches for building VMM architectures. The first approach is to build an OS service that runs in the context of a host OS. The second design relies on a hypervisor architecture, where the VMM has full control over hardware, and implements minimal OS services such that all applications must run inside a VM. Current host

OS based architectures target commodity environments that are running full featured commodity operating systems. This requires that any system running these tools must also run a large OS, such as Linux. This approach is untenable on large scale systems due to the poor scalability that commodity operating systems exhibit. Second, virtualized architectures that implement their own hypervisor such as Xen and VMWare ESX do not allow native execution for any OS. This means that every application running on these systems must execute inside a virtual context with all the associated overheads. Palacios is the first VMM architecture designed to integrate with a lightweight kernel host OS.

3.1 Introduction

This chapter describes the use of Palacios as a high performance virtual machine monitor (VMM) architecture embedded into Kitten, a high performance supercomputing operating system (OS). Together, Palacios and Kitten provide a flexible, high performance virtualized system software platform for HPC systems. This platform broadens the applicability and usability of HPC systems by:

- providing access to advanced virtualization features such as migration, full system checkpointing, and debugging;
- allowing system owners to support a wider range of applications and to more easily support legacy applications and programming models when changing the underlying hardware platform;
- enabling system users to incrementally port their codes from small-scale development systems to large-scale supercomputer systems while carefully balancing their performance and system software service requirements with application porting effort; and

- providing system hardware and software architects with a platform for exploring hardware and system software enhancements without disrupting other applications.

Kitten is an OS being developed at Sandia National Laboratories that is being used to investigate system software techniques for better leveraging multicore processors and hardware virtualization in the context of capability supercomputers. Kitten is designed in the spirit of *lightweight kernels* [79], such as Sandia's Catamount [46] and IBM's CNK [85], that are well known to perform better than commodity kernels for HPC. The simple framework provided by Kitten and other lightweight kernels facilitates experimentation, has led to novel techniques for reducing the memory bandwidth requirements of intra-node message passing [10], and is being used to explore system-level options for improving resiliency to hardware faults.

Kitten and Palacios together provide a scalable, flexible HPC system software platform that addresses the challenges laid out earlier and by others [63]. Applications ported to Kitten will be able to achieve maximum performance on a given machine. Furthermore, Kitten is itself portable and open, propagating the benefits of such porting efforts to multiple machines. Palacios provides the ability to run existing, unmodified applications and their operating systems, requiring no porting. Furthermore, as Palacios has quite low overhead, it could potentially be used to manage a machine, allowing a mixture of workloads running on commodity and more specialized OSes, and could even run ported applications on more generic hardware.

In the remainder of this chapter, I will describe the design and implementation of Kitten, examine how Palacios and Kitten function together, and evaluate the performance of the two. The core contributions are the following:

- And introduction and description the Kitten HPC OS.

- A motivation for using virtualization on supercomputers and how Palacios and Kitten

can provide an incremental path to using many different kinds of HPC resources for the mutual benefit of users and machine owners.

- Evaluations of parallel application and benchmark performance and overheads using virtualization on high-end computing resources. The overheads we see, particularly using hardware nested paging, are typically less than 5%.

3.2 Motivation

Palacios and Kitten are parts of larger projects that have numerous motivations. Here we consider their joint motivation in the context of high performance computing, particularly on large scale machines.

Maximizing performance through lightweight kernels Lightweight compute node OSes maximize the resources delivered to applications in order to maximize their performance. As such, a lightweight kernel does not implement much of the functionality of a traditional operating system; instead, it provides mechanisms that allow system services to be implemented *outside* the OS, for example in a library linked to the application. As a result, they also require that applications be carefully ported to their minimalist interfaces.

Increasing portability and compatibility through commodity interfaces Standardized application interfaces, for example partial or full Linux ABI compatibility, would make it easier to port parallel applications to a lightweight kernel. However, a lightweight kernel cannot support the full functionality of a commodity kernel without losing the benefits noted above. This means that some applications cannot be run without modification.

Achieving full application and OS compatibility through virtualization Full system virtualization provides full compatibility at the hardware level, allowing existing unmod-

ified applications and OSes to run. The machine is thus immediately available to be used by any application code, increasing system utilization when ported application jobs are not available. The performance of the full system virtualization implementation (the VMM) partially drives the choice of either using the VMM or porting an application to the lightweight kernel. Lowering the overhead of the VMM, particularly in communication, allows more of the workload of the machine to consist of VMMs.

Preserving and enabling investment in ported applications through virtualization A VMM which can run a lightweight kernel provides straightforward portability to applications where the lightweight kernel is not available natively. Virtualization makes it possible to emulate a large scale machine on a small machine, desktop, or cluster. This emulation ability makes commodity hardware useful for developing and debugging applications for lightweight kernels running on large scale machines.

Managing the machine through virtualization Full system virtualization would allow a site to dynamically configure nodes to run a full OS or a lightweight OS without requiring rebooting the whole machine on a per-job basis. Management based on virtualization would also make it possible to backfill work on the machine using loosely-coupled programming jobs or other low priority work. A batch-submission or grid computing system could be run on a collection of nodes where a new OS stack could be dynamically launched; this system could also be brought up and torn down as needed.

Augmenting the machine through virtualization Virtualization offers the option to enhance the underlying machine with new capabilities or better functionality. Virtualized lightweight kernels can be extended at runtime with specific features that would otherwise be too costly to implement. Legacy applications and OSes would be able to use features

such as migration that they would otherwise be unable to support. Virtualization also provides new opportunities for fault tolerance, a critical area that is receiving more attention as the mean time between system failures continues to decrease.

Enhancing systems software research in HPC and elsewhere The combination of Palacios and Kitten provides an open source toolset for HPC systems software research that can run existing codes without the need for victim hardware. Palacios and Kitten enable new systems research into areas such as fault-tolerant system software, checkpointing, overlays, multicore parallelism, and the integration of high-end computing and grid computing.

3.3 Palacios as a HPC VMM

Part of the motivation behind Palacios's design is that it be well suited for high performance computing environments, both on the small scale (e.g., multicores) and large scale parallel machines. Palacios is designed to interfere with the guest as little as possible, allowing it to achieve maximum performance.

Palacios is currently designed for distributed memory parallel computing environments. This naturally maps to conventional cluster and HPC architectures. Multicore CPUs are currently virtualized as a set of independent compute nodes that run separate guest contexts. Support for single image multicore environments (i.e., multicore guests) is currently under development.

Several aspects of Palacios's design are suited for HPC:

- Minimalist interface: Palacios does not require extensive host OS features, which allows it to be easily embedded into even small kernels, such as Kitten and GeekOS [35].
- Full system virtualization: Palacios does not require guest OS changes. This allows

it to run existing kernels without any porting, including Linux kernels and whole distributions, and lightweight kernels [79] like Kitten, Catamount, Cray CNL [45] and IBM's CNK [85].

- Contiguous memory preallocation: Palacios preallocates guest memory as a physically contiguous region. This vastly simplifies the virtualized memory implementation, and provides deterministic performance for most memory operations.

- Passthrough resources and resource partitioning: Palacios allows host resources to be easily mapped directly into a guest environment. This allows a guest to use high performance devices, with existing device drivers, with no virtualization overhead.

- Low noise: Palacios minimizes the amount of OS noise [21] injected by the VMM layer. Palacios makes no use of internal timers, nor does it accumulate deferred work.

- Extensive compile time configurability: Palacios can be configured with a minimum set of required features to produce a highly optimized VMM for specific environments. This allows lightweight kernels to include only the features that are deemed necessary and remove any overhead that is not specifically needed.

3.3.1 Architecture

Configurability Palacios is designed to be highly modular to support the generation of specialized VMM architectures. The modularity allows VMM features and subsystems to be selected at compile time to generate a VMM that is specific to the target environment. The configuration system is also used to select from the set of available OS interfaces, in order to enable Palacios to run on a large number of OS architectures. The build and

compile time configuration system is based on a modified version of KBuild ported from Linux.

Palacios also includes a runtime configuration system that allows guest environments to specifically configure the VMM to suit their environments. Virtual devices are implemented as independent modules that are inserted into a runtime generated hash table that is keyed to a device's ID. The guest configuration also allows a guest to specify core configuration options such as the scheduling quantum and the mechanism used for shadow memory.

The combination of the compile time and runtime configurations make it possible to construct a wide range of guest environments that can be targeted for a large range of host OS and hardware environments.

Interrupts Palacios includes two models for hardware interrupts, passthrough interrupts and specific event notifications. Furthermore, Palacios is capable of disabling local and global interrupts in order to have interrupt processing on a core run at times it chooses. The interrupt method used is determined by the virtual device connected to the guest.

For high performance devices, such as network interconnects, Palacios supports passthrough operation which allows the guest to interact directly with the hardware. For this mechanism no host OS driver is needed. In this case, Palacios creates a special virtual passthrough device that interfaces with the host to register for a given device's interrupt. The host OS creates a generic interrupt handler that first masks the interrupt pin, acks the interrupt to the hardware interrupt controller, and then raises a virtual interrupt in Palacios. When the guest environment acks the virtual interrupt, Palacios notifies the host, which then unmasks the interrupt pin. This interface allows direct device IO to and from the guest environment with only a small increase to the interrupt latency that is dominated by the hardware's world context switch latency.

3.4 Kitten

Kitten is an open-source OS designed specifically for high performance computing. It employs the same "lightweight" philosophy as its predecessors—SUNMOS, Puma, Cougar, and Catamount[1]—to achieve superior scalability on massively parallel supercomputers while at the same time exposing a more familiar and flexible environment to application developers, addressing one of the primary criticisms of previous lightweight kernels. Kitten provides partial Linux API and ABI compatibility so that standard compiler tool-chains and system libraries (e.g., Glibc) can be used without modification. The resulting ELF executables can be run on either Linux or Kitten unchanged. In cases where Kitten's partial Linux API and ABI compatibility is not sufficient, the combination of Kitten and Palacios enables unmodified guest OSes and applications to be loaded on demand.

The general philosophy being used to develop Kitten is to borrow heavily from the Linux kernel when doing so does not compromise scalability or performance (e.g., adapting the Linux bootstrap code for Kitten). Performance critical subsystems, such as memory management and task scheduling, are replaced with code written from scratch for Kitten. To avoid potential licensing issues, no code from prior Sandia-developed lightweight kernels is used. Like Linux, the Kitten code base is structured to facilitate portability to new architectures. Currently only the x86_64 architecture is officially supported, but NEC has recently ported Kitten to the NEC SX vector architecture for research purposes[24]. Kitten is publicly available from `http://software.sandia.gov/trac/kitten` (or `http://code.google.com/p/kitten/`) and is released under the terms of the GNU Public License (GPL) version 2.

[1]The name Kitten continues the cat naming theme, but indicates a new beginning.

3.4.1 Architecture

Kitten (Figure 3.1) is a monolithic kernel that runs symmetrically on all processors in the system. Straightforward locking techniques are used to protect access to shared data structures. At system boot-up, the kernel enumerates and initializes all hardware resources (processors, memory, and network interfaces) and then launches the initial user-level task, which runs with elevated privilege (the equivalent of root). This process is responsible for interfacing with the outside world to load jobs onto the system, which may either be native Kitten applications or guest operating systems. The Kitten kernel exposes a set of resource management system calls that the initial task uses to create virtual address spaces, allocate physical memory, create additional native Kitten tasks, and launch guest operating systems.

The Kitten kernel supports a subset of the Linux system call API and adheres to the Linux ABI to support native user-level tasks. Compatibility includes system call calling conventions, user-level stack and heap layout, thread-local storage conventions, and a variety of standard system calls such as `read()`, `write()`, `mmap()`, `clone()`, and `futex()`. The subset of system calls that Kitten implements natively is intended to support the requirements of existing scalable scientific computing applications in use at Sandia. The subset is also sufficient to support Glibc's NPTL POSIX threads implementation and GCC's OpenMP implementation without modification. Implementing additional system calls is a relatively straightforward process.

The Kitten kernel contains functionality aimed at easing the task of porting of Linux device drivers to Kitten. Many device drivers and user-level interface libraries create or require local files under `/dev`, `/proc`, and `/sys`. Kitten provides limited support for such files. When a device driver is initialized, it can register a set of callback operations to be used for a given file name. The `open()` system call handler then inspects a table of the

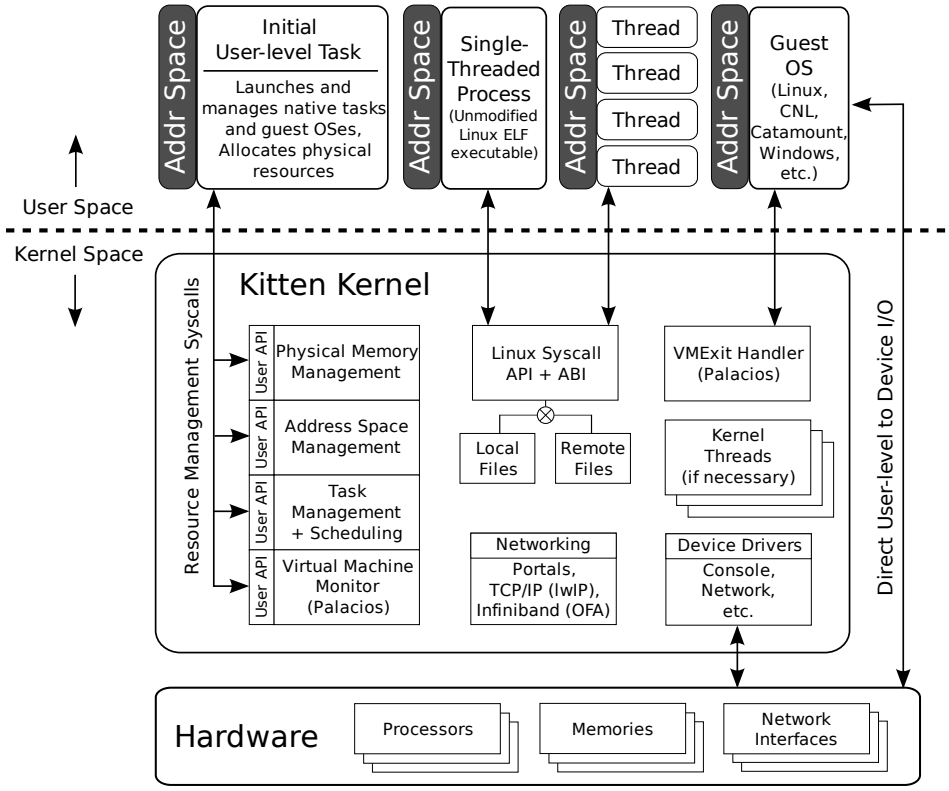


Figure 3.1: High level architecture of the Kitten lightweight kernel.

registered local file names to determine how to handle each open request. Remote files are forwarded to a user-level proxy task for servicing. Kitten also provides support for kernel threads, interrupt registration, and one-shot timers since they are required by many Linux drivers. The Open Fabrics Alliance (OFA) Infiniband stack was recently ported to Kitten without making any significant changes to the OFA code.

3.4.2 Memory management

Unlike traditional general-purpose kernels, Kitten delegates most virtual and physical memory management to user-space. The initial task allocates memory to new native applications and Palacios virtual machines by making a series of system calls to create an address

space, create virtual memory regions, and bind physical memory to those regions. Memory topology information (i.e., NUMA) is provided to the initial-task so it can make intelligent decisions about how memory should be allocated.

Memory is bound to a context of execution before it starts executing and a contiguous linear mapping is used between virtual and physical addresses. The use of a regular mapping greatly simplifies virtual to physical address translation compared to demand-paged schemes, which result in an unpredictable mapping with complex performance implications. Networking hardware and software can take advantage of the simple mapping to increase performance (which is the case on Cray XT) and potentially decrease cost by eliminating the need for translation table memory and table walk hardware on the network interface. The simple mapping also enables straightforward pass-through of physical devices to para-virtualized guest drivers.

3.4.3 Task scheduling

All contexts of execution on Kitten, including Palacios virtual machines, are represented by a task structure. Tasks that have their own exclusive address space are considered processes and tasks that share an address space are threads. Processes and threads are identical from a scheduling standpoint. Each processor has its own run queue of ready tasks that are preemptively scheduled in a round-robin fashion. Currently Kitten does not automatically migrate tasks to maintain load balance. This is sufficient for the expected common usage model of one MPI task or OpenMP thread per processor.

The privileged initial task that is started at boot time allocates a set of processors to each user application task (process) that it creates. An application task may then spawn additional tasks on its set of processors via the `clone()` system call. By default spawned tasks are spread out to minimize the number of tasks per processor but a Kitten-specific task creation system call can be used to specify the exact processor that a task should be

Component	Lines of Code
Kitten	
Kitten Core (C)	17,995
Kitten Arch Code (C+Assembly)	14,604
Misc. Contrib Code (Kbuild/lwIP)	27,973
Palacios Glue Module (C)	286
Total	60,858
Grand Total	88,970

Figure 3.2: Lines of code in Kitten with Palacios integration as measured with the SLOC-Count tool.

spawned on.

3.5 Integrating Palacios and Kitten

Palacios was designed to be easily integrated with different operating systems. This leads to an extremely simple integration with Kitten consisting of an interface file of less than 300 lines of code. The integration includes no internal changes in either Kitten or Palacios, and the interface code is encapsulated with the Palacios library in an optional compile time module for Kitten. This makes Palacios a natural virtualization solution for Kitten when considered against existing solutions that target a specific OS with extensive dependencies on internal OS infrastructures.

Kitten exposes the Palacios control functions via a system call interface available from user space. This allows user level tasks to instantiate virtual machine images directly from user memory. This interface allows VMs to be loaded and controlled via processes received from the job loader. A VM image can thus be linked into a standard job that includes loading and control functionality.

SeaStar passthrough support Because Palacios provides support for passthrough I/O, it is possible to support high performance, partitioned access to particular communication devices. We do this for the SeaStar communication hardware on the Red Storm machine. The SeaStar is a high performance network interface that utilizes the AMD HyperTransport Interface and proprietary mesh interconnect for data transfers between Cray XT nodes [11]. At the hardware layer the data transfers take the form of arbitrary physical-addressed DMA operations. To support a virtualized SeaStar the physical DMA addresses must be translated from the guest's address space. However, to ensure high performance the SeaStar's command queue must be directly exposed to the guest. This requires the implementation of a simple high performance translation mechanism. Both Palacios and Kitten include a simple memory model that makes such support straightforward.

The programmable SeaStar architecture provides several possible avenues for optimizing DMA translations. These include a self-virtualizable firmware as well as an explicitly virtualized guest driver. In the performance study we conducted for this chapter we chose to modify the SeaStar driver running in the guest to support Palacios's passthrough I/O. This allows the guest to have exclusive and direct access to the SeaStar device. Palacios uses the large contiguous physical memory allocations supported by Kitten to map contiguous guest memory at a known offset. The SeaStar driver has a tiny modification that incorporates this offset into the DMA commands sent to the SeaStar. This allows the SeaStar to execute actual memory operations with no performance loss due to virtualization overhead. Because each Cray XT node contains a single SeaStar device, the passthrough configuration means that only a single guest is capable of operating the SeaStar at any given time.

Besides memory-mapped I/O, the SeaStar also directly uses an APIC interrupt line to notify the host of transfer completions as well as message arrivals. Currently, Palacios exits from the guest on all interrupts. For SeaStar interrupts, we immediately inject such interrupts into the guest and resume. While this introduces an VM exit/entry cost to each

SeaStar interrupt, in practice this only results in a small increase in latency. We also note that the SeaStar interrupts are relatively synchronized, which does not result in a significant increase in noise. We are investigating the use of next generation SVM hardware that supports selective interrupt exiting to eliminate this already small cost.

3.6 Performance

We conducted a careful performance evaluation of the combination of Palacios and Kitten on diverse hardware, and at scales up to 48 nodes. We focus the presentation of our evaluation on the Red Storm machine and widely recognized applications/benchmarks considered critical to its success. As far as we are aware, ours is the largest scale evaluation of parallel applications/benchmarks in virtualization to date, particularly for those with significant communication. It also appears to be the first evaluation on petaflop-capable hardware. Finally, we show performance numbers for native lightweight kernels, which create a very high bar for the performance of virtualization. The main takeaways from our evaluation are the following.

1. The combination of Palacios and Kitten is generally able to provide near-native performance. This is the case even with large amounts of complex communication, and even when running guest OSes that themselves use lightweight kernels to maximize performance.

2. It is generally preferable for a VMM to use nested paging (a hardware feature of AMD SVM and Intel VT) over shadow paging (a software approach) for guest physical memory virtualization. However, for guest OSes that use simple, high performance address space management, such as lightweight kernels, shadow paging can sometimes be preferable due to its being more TLB-friendly.

The typical overhead for virtualization is $\leq 5\%$.

3.6.1 Testbed

We evaluated the performance and scaling of Palacios running on Kitten on the development system *rsqual*, part of the Red Storm machine at Sandia National Laboratories. Each XT4 node on this machine contains a quad-core AMD Budapest processor running at 2.2 GHz with 4 GB of RAM. The nodes are interconnected with a Cray SeaStar 2.2 mesh network [11]. Each node can simultaneously send and receive at a rate of 2.1 GB/s via MPI. The measured node to node MPI-level latency ranges from 4.8 μsec (using the Catamount [46] operating system) to 7.0 μsec (using the native CNL [45] operating system). As we stated earlier, even though we can run multiple guests on a multicore Cray XT node by instantiating them on separate cores, our current implementation only allows the SeaStar to be exposed to a single guest context. Due to this limitation, our performance evaluation is restricted to a single guest per Cray XT node.

In addition, we used two dual-socket quad-core 2.3 GHz AMD Shanghai systems with 32GB of memory for communication benchmark testing on commodity HPC hardware. Nodes in this system are connected with Mellanox ConnectX QDR Infiniband NICs and a Mellanox Infiniscale-IV 24 port switch. When not running Kitten, these systems run Linux 2.6.27 and the OpenFabrics 1.4 Infiniband stack.

All benchmark timing in this chapter is done using the AMD cycle counter. When virtualization is used, the cycle counter is direct mapped to the guest and not virtualized. Every benchmark receives the same accurate view of the passage of real time regardless of whether virtualization is in use or not.

3.6.2 Guests

We evaluated Palacios running on Kitten with two guest environments:

- Cray Compute Node Linux (CNL). This is Cray's stripped down Linux operating

system customized for Cray XT hardware. CNL is a minimized Linux (2.6 kernel) that leverages BusyBox [105] and other embedded OS tools/mechanism. This OS is also known as Unicos/LC and the Cray Linux Environment (CLE).

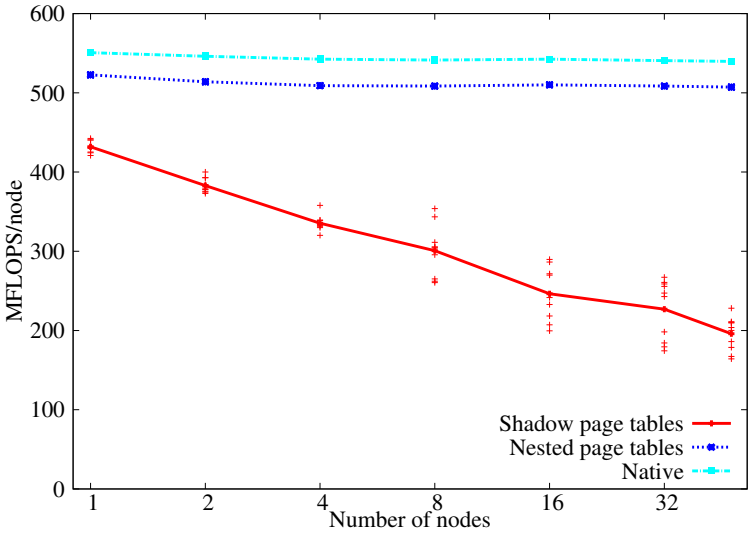
- Catamount. Catamount is a lightweight kernel descended from the SUNMOS and PUMA operating systems developed at Sandia National Labs and the University of New Mexico [87][3]. These OSes, and Catamount, were developed, from-scratch, in reaction to the heavyweight operating systems for parallel computers that began to proliferate in the 1990s. Catamount provides a simple memory model with a physically-contiguous virtual memory layout, parallel job launch, and message passing facilities.

We also use Kitten as a guest for our Infiniband tests. It is important to note that Palacios runs a much wider range of guests than reported in this evaluation. Any modern x86 or x86_64 guest can be booted.

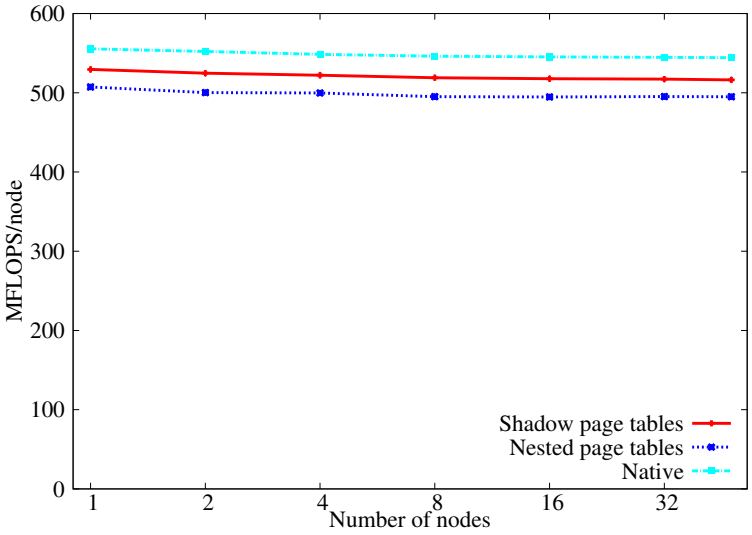
3.6.3 HPCCG benchmark results

We used the HPCCG benchmark to evaluate the impact of virtualization on application performance and scaling. HPCCG [33] is a simple conjugate gradient solver that represents an important workload for Sandia. It is commonly used to characterize the performance of new hardware platforms that are under evaluation. The majority of its runtime is spent in a sparse matrix-vector multiply kernel.

We ran HPCCG on top of CNL and Catamount on Red Storm, considering scales from 1 to 48 nodes. A fixed-size problem per node was used to obtain these results. The specific HPCCG input arguments were “100 100 100”, requiring approximately 380 MB per node. This software stack was compiled with the Portland Group pgicc compiler version 7, and was run both directly on the machine and on top of Palacios. Both shadow paging and



(a) CNL Guest



(b) Catamount Guest

Figure 3.3: HPCCG benchmark comparing scaling for virtualization with shadow paging, virtualization with nested paging, and no virtualization. Palacios/Kitten can provide scaling to 48 nodes with less than 5% performance degradation.

nested paging cases were considered. Communication was done using the passthrough-mapped SeaStar interface, as described earlier.

Figures 3.3(a) and 3.3(b) show the results for CNL and Catamount guests. Each graph compares the performance and scaling of the native OS, the virtualized OS with shadow paging, and the virtualized OS with nested paging. The graph shows both the raw measurements of multiple runs and the averages of those runs. The most important result is that the overhead of virtualization is less than 5% and this overhead remains essentially constant at the scales we considered, despite the growing amount of communication. Note further that the variance in performance for both native CNL and virtualized CNL (with nested paging) is minuscule and independent of scale. For Catamount, all variances are tiny and independent, even with shadow paging.

The figure also illustrates the relative effectiveness of Palacios's shadow and nested paging approaches to virtualizing memory. Clearly, nested paging is preferable for this benchmark running on a CNL guest, both for scaling and for low variation in performance. There are two effects at work here. First, shadow paging results in more VM exits than nested paging. On a single node, this overhead results in a 13% performance degradation compared to native performance. The second effect is that the variance in single node performance compounds as we scale, resulting in an increasing performance difference.

Surprisingly, shadow paging is slightly preferable to nested paging for the benchmark running on the Catamount guest. In Catamount the guest page tables change very infrequently, avoiding the exits for shadow page table refills that happen with CNL. Additionally, instead of the deep nested page walk ($O(nm)$ for n-deep guest and m-deep host page tables) needed on a TLB miss with nested pages, only a regular m-deep host page table walk occurs on a TLB miss with shadow paging. These two effects explain the very different performance of shadow and nested paging with CNL and Catamount guests.

It is important to point out that the version of Palacios's shadow paging implementation

we tested only performs on demand updates of the shadow paging state. With optimizations, such as caching, the differences between nested and shadow paging are likely to be smaller.

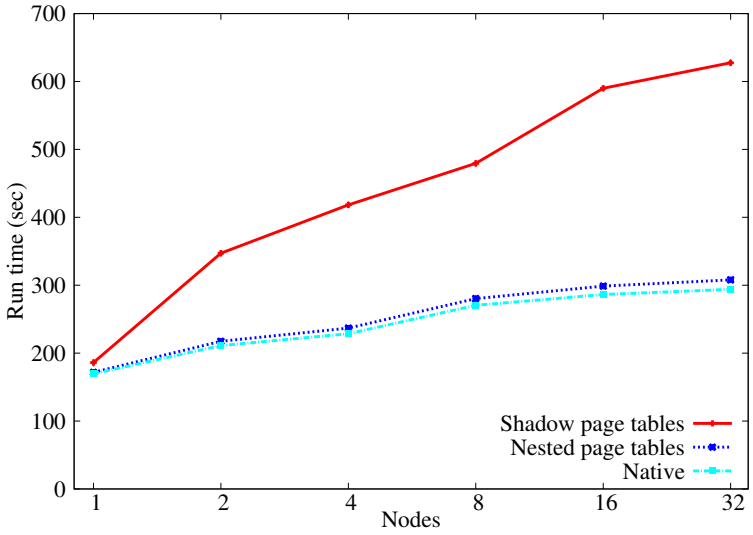
3.6.4 CTH application benchmark

CTH [20] is a multi-material, large deformation, strong shock wave, solid mechanics code developed by Sandia National Laboratories with models for multi-phase, elastic viscoplastic, porous, and explosive materials. CTH supports three-dimensional rectangular meshes; two-dimensional rectangular, and cylindrical meshes; and one-dimensional rectilinear, cylindrical, and spherical meshes, and uses second-order accurate numerical methods to reduce dispersion and dissipation and to produce accurate, efficient results. It is used for studying armor/anti-armor interactions, warhead design, high explosive initiation physics, and weapons safety issues.

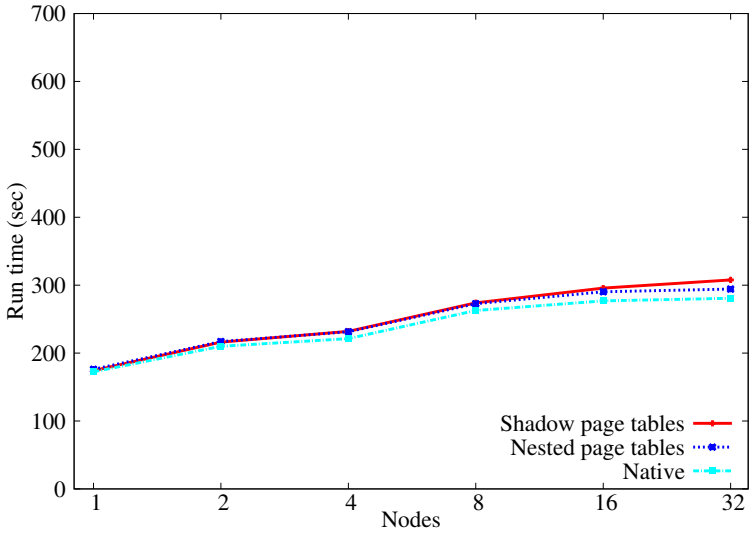
Figures 3.4(a) and 3.4(b) show the results using the CNL and Catamount guests. We can see that adding virtualization, provided the appropriate choice of shadow or nested paging is made, has virtually no effect on performance or scaling. For this highly communication intensive benchmark, virtualization is essentially free.

3.6.5 Intel MPI benchmarks

The Intel MPI Benchmarks (IMB) [39], formerly known as PALLAS, are designed to characterize the MPI communication performance of a system. IMB employs a range of MPI primitive and collective communication operations, at a range of message sizes and scales to produce numerous performance characteristics. We ran IMB on top of CNL and Catamount on Red Storm using SeaStar at scales from 2 to 48 nodes. We compared native performance, virtualized performance using shadow paging, and virtualized performance using nested paging. IMB generates large quantities of data. Figures 3.5 through 3.6

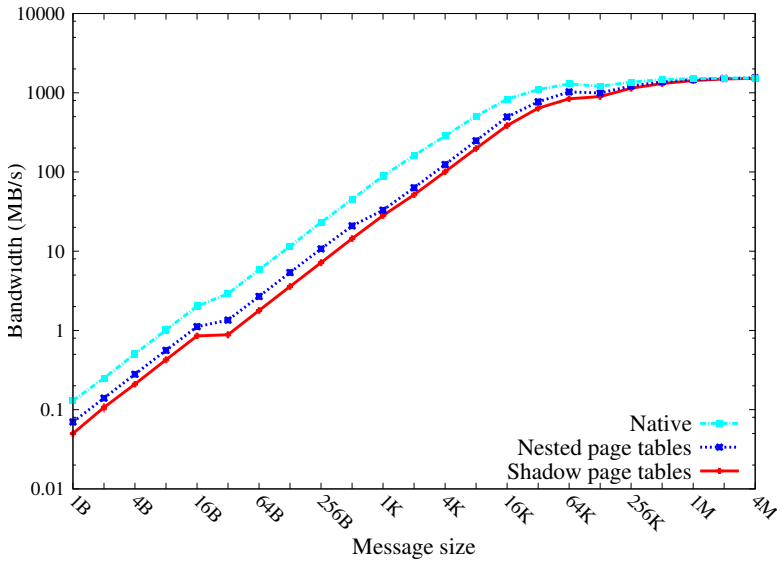


(a) CNL Guest

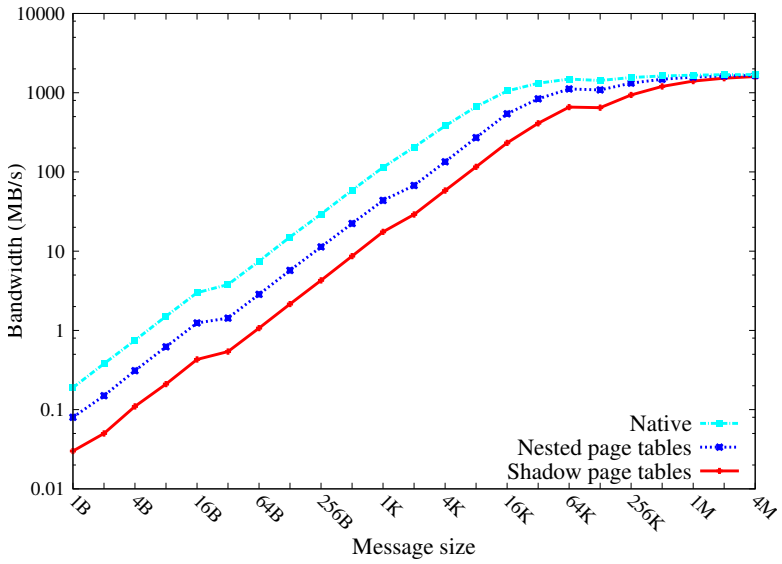


(b) Catamount Guest

Figure 3.4: CTH application benchmark comparing scaling for virtualization with shadow paging, virtualization with nested paging, and no virtualization. Palacios/Kitten can provide scaling to 32 nodes with less than 5% performance degradation.

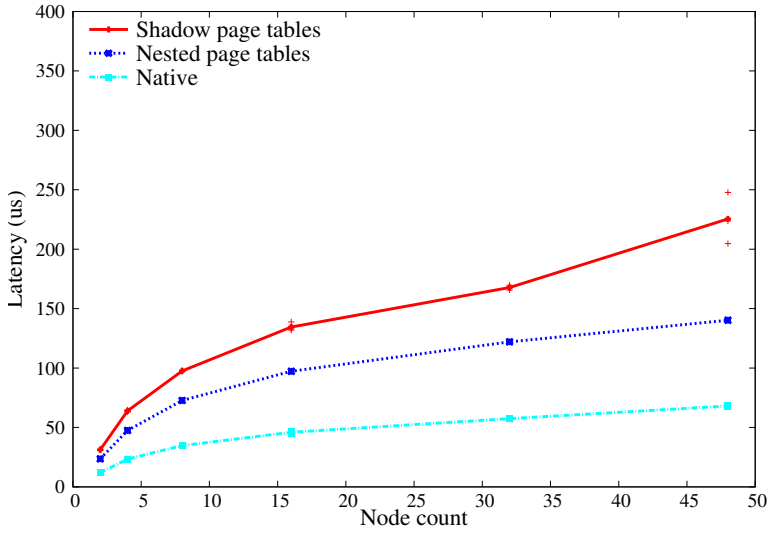


(a) CNL Guest

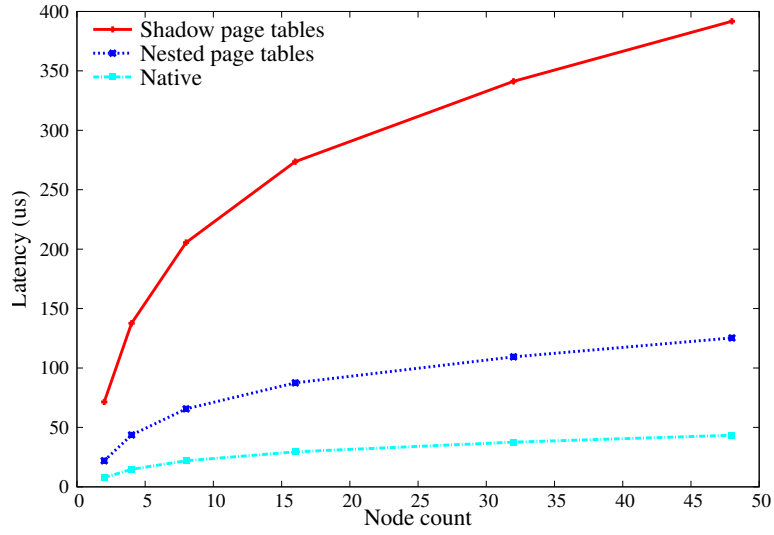


(b) Catamount Guest

Figure 3.5: IMB PingPong Bandwidth in MB/sec as a function of message size



(a) CNL Guest



(b) Catamount Guest

Figure 3.6: IMB Allreduce 16 byte latency in μsec as a function of nodes up to 48 nodes

illustrate the most salient data on CNL and Catamount.

Figure 3.5 shows the bandwidth of a ping-pong test between two nodes for different message sizes. For large messages, bandwidth performance is identical for virtualized and native operating systems. For small messages where ping-pong bandwidth is latency-bound, the latency costs of virtualization reduce ping-pong bandwidth. We have measured the extra latency introduced by virtualization as either 5 μsec (nested paging) or 11 μsec (shadow paging) for the CNL guest. For the Catamount guest, shadow paging has a higher overhead. Although the SeaStar is accessed via passthrough I/O, interrupts are virtualized. When the SeaStar raises an interrupt, a VM exit is induced. Palacios quickly transforms the hardware interrupt into a virtual interrupt that it injects into the guest on VM entry. The guest will quickly cause another VM exit/entry interaction when it acknowledges the interrupt to its (virtual) APIC. Shadow paging introduces additional overhead because of the need to refill the TLB after these entries/exits. This effect is especially pronounced in Catamount since, other than capacity misses, there is no other reason for TLB refills; in addition, Catamount has a somewhat more complex interrupt path that causes two additional VM exits per interrupt. Avoiding all of these VM exits via nested paging allows us to measure the raw overhead of the interrupt exiting process.

In Figure 3.6, we fix the message size at 16 bytes and examine the effect on an IMB All-Reduce as we scale from 2 to 48 nodes. We can see that the performance impacts of nested and shadow paging diverges as we add more nodes—nested paging is superior here.

The upshot of these figures and the numerous IMB results which we have excluded for space reasons is that the performance of a passthrough device, such as the SeaStar, in Palacios is in line with the expected hardware overheads due to interrupt virtualization. This overhead is quite small. Virtualized interrupts could be avoided using the AMD SVM interrupt handling features, which we expect would bring IMB performance with nested paging-based virtualization in line with native performance. However, at this point, we

	Latency (μsec)	Bandwidth (Gb/sec)
Kitten (Native)	5.24	12.40
Kitten (Virtualized)	5.25	12.40
Linux	4.28	12.37

Figure 3.7: Bandwidth and latency of node-to-node Infiniband on Kitten, comparing native performance with guest performance. Linux numbers are provided for reference.

expect that doing so would require minor guest changes.

3.6.6 Infiniband microbenchmarks

To quantify the overhead of Palacios virtualization on a commodity NIC, we ported OpenIB MLX4 (ConnectX) drivers to Kitten along with the associated Linux driver. We also implemented passthrough I/O support for these drivers in Palacios. We then measured round-trip latency for 1 byte messages averaged over 100000 round trips and 1 megabyte message round trip bandwidth averaged over 10000 trips using a ported version of the OpenFabrics `ibv_rc_pingpong`. The server system ran Linux 2.6.27, while the client machine ran either Kitten natively, Kitten as a guest on Palacios using shadow paging, or Linux.

As can be seen in Figure 5.8, Palacios's pass-through virtualization imposes almost no measurable overhead on Infiniband message passing. Compared to Linux, Kitten both native and virtualized using Palacios slightly outperform Linux in terms of end-to-end bandwidth, but suffers a 1 μsec/round trip latency penalty. We believe this is due to a combination of the lack of support for message-signaled interrupts (MSI) in our current Linux driver support code, as well as our use of a comparatively old version of the OpenIB driver stack. We are currently updating Linux driver support and the OpenIB stack used in in Kitten to address this issue.

	HPCCG MFLOPS
Native CNL	588.0
Palacios/Kitten + CNL Guest	556.4
KVM/CNL + CNL Guest	546.4
% Diff Palacios vs. KVM	1.8%

Figure 3.8: Comparison of Palacios to KVM for HPCCG benchmark.

3.6.7 Comparison with KVM

To get a feel for the overhead of Palacios compared to existing virtualization platforms, we ran the HPCCG benchmark in a CNL guest under both KVM running on a Linux host and Palacios running on a Kitten host. KVM (Kernel-based Virtual Machine) is a popular virtualization platform for Linux that is part of the core Linux kernel as of version 2.6.20. Due to time constraints we were not able to expose the SeaStar to KVM guest environments, so only single node experiments were performed. The same "100 100 100" test problem that was used in Section 3.6.3 was run on a single Cray XT compute node. HPCCG was compiled in serial mode (non-MPI) leading to slightly different performance results. As can be seen in Figure 3.8, Palacios delivers approximately 1.8% better performance than KVM for this benchmark. Each result is an average of three trials and has a standard deviation less of than 0.66. Note that small performance differences at the single node level typically magnify as the application and system are scaled up.

3.7 Conclusion

Palacios and Kitten are new open source tools that support virtualized and native supercomputing on diverse hardware. We described the design and implementation of both Palacios and Kitten, and evaluated their performance. Virtualization support, such as Pala-

cios's, that combines hardware features such as nested paging with passthrough access to communication devices can support even the highest performing guest environments with minimal performance impact, even at relatively large scale. Palacios and Kitten provide an incremental path to using supercomputer resources that has few compromises for performance.

Our analysis has determined that in order to deliver the best possible performance, a VMM must have detailed knowledge of a guest environment's architecture, behavior, and internal state. Unfortunately, existing virtualization interfaces do not offer a method of exchanging this information directly. To address this issue I have explored symbiotic virtualization, a new approach to virtualized architectures that implements a set of high level virtual interfaces. In the next chapter I will describe in detail symbiotic virtualization and introduce a very basic symbiotic interface.

Chapter 4

Symbiotic Virtualization

While our experiences have shown that it is indeed possible to virtualize large scale HPC systems with minimal overhead, we have found that doing so requires cooperation between the guest and VMM. Traditional virtualization approaches that implement VMs as opaque entities are not able to achieve acceptable performance in HPC settings. Maximizing the potential performance in an HPC setting requires that a VMM have detailed knowledge of the internal state and behavior of the guest environment in order to fully optimize the operation of the VMM and hardware. I have found that this knowledge is only accessible when there is increased communication and trust across the VMM/guest interface. Simply put, the relationship between the VMM and the guest needs to be *symbiotic*. In this chapter I will examine existing techniques for collecting knowledge about virtual machine state, and explore a new approach called *Symbiotic Virtualization*.

4.1 Introduction

As discussed previously, virtualization provides a great deal of flexibility for guest environments. This flexibility (such as machine consolidation, resource sharing, and migration) can be leveraged in a large number of ways to optimize the virtual environments themselves as well as the computational resources they are running on. There are many

examples of how this flexibility can be used to add new features to existing unmodified operating systems and applications. This ability to operate on any virtualized environment, irrespective of what is running inside the VM, makes these approaches very appealing. These approaches are typically referred to as black box methods because they require no detailed knowledge of the internal execution state of a guest environment.

The black box approach has been used by virtualization architectures from the beginning, and is a natural architectural emergence given the goals of virtualization. The success of virtualization can be directly attributed to its ability to support existing OSes and applications with no modifications. The only way to ensure this compatibility, was for virtual machines to provide exactly the same interfaces as expected by the legacy operating systems which targeted physical hardware. While this interface is universally compatible with existing code bases, it creates a number of limitations because VMMs are much more capable computing substrates than actual hardware. These limitations are due to the use of an interface designed for a substrate that lacked the capabilities and dynamism of a VMM. The two most common interfaces used today are either a fully virtualized hardware interface (e.g., VMWare [108], KVM [76]) or a paravirtualized hypercall interface designed to resemble the hardware interface as much as possible (e.g. Xen [6]).

Black box methods are enabled by the encapsulation that is afforded by virtual machines. This encapsulation allows black box mechanisms to operate with any guest environment, as well as alter the underlying resources in a manner that is transparent to the guest environment. These control mechanisms include such things as VM placement, overlay topology and routing, network and processor reservations, and interposing transparent services. Myself and others explored these techniques in the context of the Virtuoso project, which focused on automatic adaptation of distributed virtual machines. Specifically my contributions focused on using black box methods to dramatically increase network performance in these environments.

While our work in Virtuoso demonstrated the effectiveness of using black box techniques, it did so at only a course grained level. While treating a virtual machine as an opaque container does allow universal compatibility with any existing OS, it precludes any mechanism that interacts with the internal state of the VM. This means that any mechanism developed with this method can only perform operations on whole VMs or any data generated as a side effect of the VM's execution. As the Virtuoso project demonstrated, these approaches excel in the context of distributed computing due to the fact that the limiting resource is usually the network and not the local compute resources. Because virtualization allows for the removal of any local resource constraints, adaptation mechanisms are able to freely configure the network and locate VMs in a way that optimizes the network configuration.

While these optimizations are effective at optimizing a distributed environment, they have a limited capability to optimize the execution of the VM on local compute resources. This is due to the fact that the guest OS often assumes it is the lowest software layer sitting directly on the physical hardware, even though it is actually an abstraction layer when running as a VM. This assumption has resulted in OS architectures that are not ideally suited to virtual environments. The fundamental issue here is that virtualization allows the creation of extremely dynamic and flexible hardware environments, while modern operating systems are still designed around a very static and exclusive hardware model. Consider memory as an example. In native environments a machine has a static amount of RAM that is fully managed by the OS. If the OS does not have any use for the physical memory, it simply stores it in a free list where it remains unused until needed. This is acceptable in these environments because the OS is managing all of the physical hardware resources, and so is not holding onto memory that could be used for another purpose. Contrast this with virtualized environments, where multiple OSes are executing inside guest contexts. Because each guest OS is designed around the assumption that it has static physical mem-

ory, it is extremely difficult for the VMM to dynamically optimize the memory resources among the different VMs. If one guest over commits its allocated memory there is no basic OS mechanism that can be used by the VMM to give it access to memory allocated but unused by another guest OS.

As a consequence of using hardware based interfaces, current virtualization interfaces are largely designed to be purely unidirectional. The only way a VMM can signal a guest OS is through interrupts or interfaces built on top of a hardware device abstraction. The unidirectional nature of the virtualization interface has created what is called the *semantic gap* [13]. The semantic gap refers to the lack of semantic knowledge available to a VMM due to the fact that the virtualization interface is at a low enough level that very little semantic information is able to travel across it. Because of this, the VMM has no way of collecting detailed knowledge about the internal state of a guest environment. Considerable effort has been put into using gray box techniques to bridge the *semantic gap* [42, 43, 26]. Gray box techniques rely on the fact that the VMM has complete access to the entirety of the VMs memory. This access allows the VMM to attempt to reconstruct the internal guest state by parsing the raw VM image in order to locate and parse elements of the internal guest state.

The advantage of gray box approaches is that they maintain the advantages of black box approaches by requiring no modifications to the OS or applications. However, there are a number of drawbacks, including the fact that the information gleaned from such techniques is semantically poor, which restricts the kinds of decision making that the VMM can do or services it can offer. Furthermore, the effort is a significant burden, given the organizational complexity of the internal guest state. These approaches also suffer from compatibility issues, because while they do not require guest cooperation they do require a priori knowledge of the guest architecture and organization. This makes these approaches susceptible to any changes made to the OS implementations. These drawbacks compound

to make gray box approaches an extremely labor intensive approach, especially considering that the state being reconstructed is probably already readily available inside the guest environment. Clearly an alternative is needed in order to fully enable virtualized environments to achieve all that they are capable of.

My thesis focuses on a new approach to virtualization called *symbiotic virtualization*. Symbiotic virtualization is an approach to designing VMMs and OSes such that both support, but neither requires, the other. A symbiotic OS targets a native hardware interface, but also exposes a software interface, usable by a symbiotic VMM, if present, to optimize performance and increase functionality. A symbiotically virtualized architecture consists of a generalized set of *symbiotic interfaces* that provide VMM↔guest information flow which can be leveraged to improve the functionality and performance of virtualized environments. This new set of interfaces preserves the benefits of black box methods by maintaining hardware compatibility with legacy operating systems, while also providing new virtualization interfaces that give a VMM access to high level semantic information. Symbiotic virtualization consists of several components and interfaces that provide both passive and functional interfaces between a guest and VMM. In this chapter I will explain in more detail my early work with black box methods, and explore in more detail the symbiotic approach to virtualization, and finally I will examine the passive interface. The functional interface will be discussed in chapter 6.

4.2 Virtuoso

Adaptive parallel and distributed computing with the intent of improving performance or achieving particular levels of quality of service has been an important goal since the 1990s. More recently, autonomic computing, adaptive computing with the intent of reducing the management burden of complex software systems, has garnered significant commercial

and academic interest. The deployment of adaptive computing has historically been limited, in part due to the need to change systems software and applications to support it. With the growth in availability and performance of virtualization technologies, in particular virtual machine monitors for x86 processors, in the 2000s, it became possible to consider adaptive computing using existing, unmodified applications and systems software.

The Virtuoso Project (virtuoso.cs.northwestern.edu) explored inference, adaptation, and reservations for parallel and distributed applications running in virtualized distributed environments at cluster, data center, and wide-area scales. A key insight was that the virtualization layer provides an excellent location for the major elements of adaptive computing. We demonstrated that the virtualization layer can be effectively used to:

- Monitor the application's traffic to automatically and cheaply produce a view of the application's network and CPU demands as well as to detect parallel imbalance due to internal or external issues [32, 90, 31, 75].

- Monitor the performance of the underlying physical network by use the application's own traffic to automatically and cheaply probe it, and then use the probes to produce characterizations [30, 31].

- Formalize performance optimization and adaptation problems in clean, simple ways that facilitate understanding their asymptotic difficulty [91, 94, 92, 93].

- Adapt the application to the network to make it run faster or more cost-effectively with algorithms that make use of network performance information and mechanisms such as VM→host mapping, scheduling of VMs on individual hosts, and overlay network topology and routing [90, 30, 94],

- Reserve resources, when possible, to improve performance [50, 55, 57, 56]. Automatic reservations of CPU, including parallel gang scheduling through real-time

methods, and optical network light paths were demonstrated.

- Transparently add network services to unmodified applications and OSes [49].

Our work focused on adaptation for *existing, unmodified applications and the software stacks they run on*; our techniques can retrofit adaptation to applications without any software changes. My contributions to the Virtuoso project were a system for network reservations and transparent network services.

4.2.1 Network reservations

The VRESERVE component of Virtuoso makes it possible for an unmodified commodity application that is unaware of network reservations to nonetheless make use of them. Adaptation agents in Virtuoso can potentially make such use *automatic* by driving VRESERVE themselves.

We primarily focused on reservations of optical network components that support circuit-switching [12], such as the OMNInet testbed network, a large scale, circuit switched, dense wave division multiplexed (DWDM) optical network deployed around the Chicago metropolitan area. The OMNInet network was implemented with dedicated fiber connected to Nortel OPTera Metro DWDM platforms. OMNInet light paths are reserved via the ODIN [61] network reservation service designed for and deployed on the OMNInet testbed. ODIN translates high level path reservation requests into configuration commands for optical switches at each hop.

At any point in time, given a pair of VMs connected by an overlay network, VRESERVE determines whether it is possible to reserve an fiber optic light path between them. If so, VRESERVE interfaces with the network reservation system (e.g., ODIN) to instantiate the optical light path, and then signals the overlay network to modify its overlay topology to use the newly available optical link. The choice of overlay links to enhance with

reservations is typically made in conjunction with the adaptation agents and the inferred traffic load matrix.

4.2.2 Transparent services

With Virtuoso we also introduced the notion of *transparent network services* for virtual execution environments. A transparent network service can not only monitor traffic, and control its routing, but it can also *manipulate the data and signaling of a flow or connection.* It can statefully manipulate the packet stream entering or exiting a VM at the data link, network, transport, and (to a limited extent) application layers. However, despite this dramatic freedom, a transparent network service must work with existing, unmodified applications. Furthermore, it must not require any changes to the guest OS, its configuration, or other aspects of the VM.

Transparent network services are implemented using our Virtual Traffic Layer (VTL) framework. VTL is cross-platform (Unix and Windows), VMM-agnostic framework that provides APIs for packet acquisition and serialization, packet inspection and modification, maintenance of connection state, and utility functions for common compositions of the previous three. These APIs are used to construct modules that implement services.

Using VTL we have designed the following transparent network services:

- Tor-VTL: This service bridges the applications running in a VM to the Tor overlay network [17], resulting in networking being anonymous.
- Subnet Tunneling: This service alters the default routing behavior between two VMs that are on the same physical network, requiring network and data link layer packet manipulation, resulting in enhanced performance.
- Local Acknowledgments: This service generates TCP acknowledgments locally to improve TCP performance on high reliability networks.

- Split-TCP: This service improves TCP performance by splitting a connection into multiple connections (e.g.[95]).
- Protocol Transformation: This service transforms TCP connections into high-performance protocol connections, such as UDT [29] connections.
- Stateful Firewall: This service is a firewall that is unmodifiable by code in the VM because it exists outside of the VM.
- TCP Keep-Alives: This service maintains TCP connections in a stable, open state despite a long duration network disconnection.
- Vortex: This service provides wormholing of traffic on the unused ports of volunteer machines back to an intrusion detection system [48].

4.2.3 Benefits

We found that it is feasible to infer various useful demands and behaviors of an application running inside a collection of VMs to a significant degree using a black box model of the VM contents. The techniques we developed for application inference fall essentially into two categories. The first category is *characterization of the resource demands of the application*. The most important result here is the dynamic inference of application topology and the traffic load matrix. The second category is *discovery of application performance problems.* Here, we describe techniques to infer the runtime performance, its slowdown due to external load, and its global bottlenecks. We can also answer the question of how fast a BSP-style parallel program could run if the current bottleneck were removed.

Using a black-box approach to inference may seem restrictive, but our results show that information sufficient for effective adaptation can be gleaned from this approach. This

is coupled with the clear advantage of the black-box approach in terms of adoption—no assumptions need be made and no code need be modified.

4.2.4 Limitations

While we have shown that black box approaches are capable of driving optimization mechanisms in a distributed context, other environments are not so amenable. HPC environments in particular pose a problem for these approaches due to the strong requirements they have for performance. The main limitation with black box approaches is that they rely on heuristic driven decisions based on collections of measurements. As described, black box approaches function by monitoring the execution of a VM and collecting measurements based on its macro behavior and side effects. This monitoring is not without a cost, and can impose a substantial overhead on a VM's execution. This overhead could be considered a source of noise in the system, and could potentially have negative consequences on overall system performance [21].

The dependence on collections of events also poses a problem for HPC. A poorly configured VMM can result in very poor performance at scale, and a black box approach would not be able to correct the issue until it has collected enough information to make an informed decision. This delay in fixing the problem could result in a very noticeable loss in performance, which would be unacceptable on a large scale system. Finally, because black box methods rely on heuristics, it is possible that the heuristics can generate an incorrect decision. This uncertainty is extremely unpalatable to managers of large scale HPC systems. The issues with black box methods can be summarized as overhead, latency, and lack of information. That is they are too expensive, take too long, and might make things worse because they have incomplete knowledge. Each of these problems are addressed using a symbiotic approach.

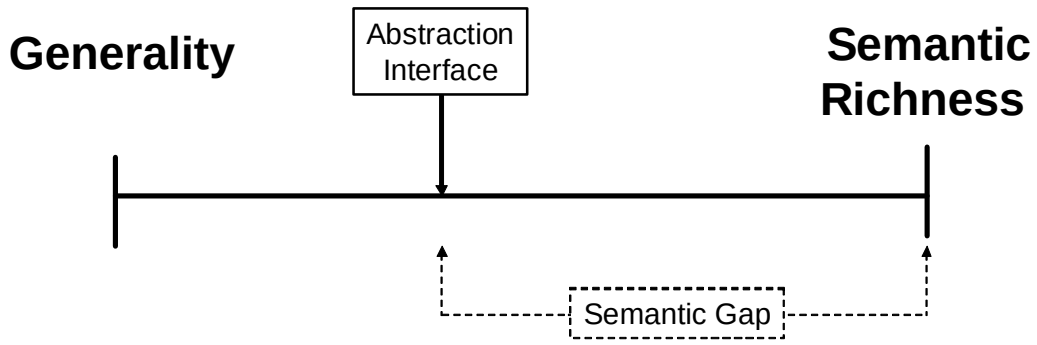


Figure 4.1: The semantic gap. Every abstraction interface must be designed as a compromise between generality and semantic richness. Every interface exists somewhere in this spectrum, with the semantic gap increasing as the interface moves more towards generality.

4.3 Symbiotic virtualization

Any abstraction interface must be designed around two competing goals. The first is generality, which ensures that the interface is compatible and usable by a wide range of architectures. If an interface includes information or behaves in a manner that is too specific, it precludes a large number of potential users. The second goal is semantic richness. A semantically rich interface allows the different layers to optimize themselves depending on the state of the system. Both of these goals exist in mutual contention, that is, an interface that is very semantically rich tends to lack in generality. Designing an optimal abstraction layer depends entirely on maximizing both generality and semantic richness, without sacrificing too much of either.

Every interface must compromise between generality and semantic richness, as shown in Figure 4.1. An inescapable side effect of this compromise is the lack of semantic information across the interface layer. This loss of semantic information is called the *semantic gap*, and can very in degree depending on how general the interface is designed to be. A large semantic gap reduces considerably the options each layer has for optimizations, since the amount of information is severely limited. As a result of the lack of cross layer

information sharing, most optimizations tend to be heuristic driven and can often produce bad results.

As an example, consider the Nagle algorithm [67], a networking optimization technique. The Nagle algorithm is designed to limit the generation of small packets by bundling many small messages into one large message at the network transport layer. By combining small messages into a single large packet, the overhead wasted on duplicated packet headers is reduced. Unfortunately, this technique suffers from a very large semantic gap. The Nagle algorithm is implemented using heuristics to guess whether or not to delay a small packet in order to wait for more small packets to arrive. A heuristic approach is necessary because the transport layer does not have the information needed to determine whether or not an application intends to send additional small messages. This information is lost as the data is sent through the socket interface as a result of the semantic gap. Because the algorithm is only able to guess about whether it should wait for more messages, it can often produce incorrect guesses that have a significant detrimental impact on network performance.

The Nagle algorithm is a prime example of how the semantic gap can cause serious performance and behavioral issues across abstraction layers. In this case it is left up to the applications themselves to alter their behavior such that the Nagle algorithm is less likely to function incorrectly. I argue that this is a failed abstraction interface. In essence, the interface is designed specifically to hide the information that is necessary for performance optimization. And furthermore, the lack of the information can in fact induce a loss in performance.

Existing virtualization interfaces all suffer from this same problem. The virtualization interfaces in use today have all tried to maximize generality over semantic richness, and as a result have introduced extremely large semantic gaps between the VMM and the guest environment. At the time these interfaces were designed this decision was correct. In

order to be usable, VMMs needed to ensure compatibility with existing and unmodified legacy environments. This is in fact one of the central benefits that virtualization provides. However this does not negate the fact that the resultant semantic gap severely limits the behavior of VMM architectures. Because the interfaces were designed to resemble actual hardware as much as possible, VMMs are essentially limited to performing hardware emulation. Most optimizations that are implemented are done so in relation to the interactions with the host OS the VMM is running on, which has no semantic gap. The limitation on a VMMs ability to optimize itself for a guest environment will always exist unless a method can be found to bridge the semantic gap.

The design of a new virtualization interface that is capable of fully bridging the semantic gap is the focus of this dissertation. I denote this new approach to designing virtualization interfaces as *symbiotic virtualization*. Unlike existing virtualization approaches, symbiotic virtualization places an equal emphasis on both semantic richness and generality. The goal of symbiotic virtualization is to introduce a virtualization interface that provides access to high level semantic information while still retaining the universal compatibility of a virtual hardware interface. Symbiotic virtualization is an approach to designing VMMs and OSes such that both support, but neither requires, the other. A symbiotic OS targets a native hardware interface, but also exposes a software interface, usable by a symbiotic VMM, if present, to optimize performance and increase functionality. Symbiotic virtualization is neither full system virtualization nor paravirtualization, however it can be used with either approach.

A symbiotic OS exposes two types of interfaces. The passive interface allows a symbiotic VMM to simply read out structured information that the OS places in memory. This interface has extremely low overhead, as the VMM can readily read guest memory during an exit or from a different core. However, the information is necessarily provided asynchronously with respect to exits or other VMM events. Because of this, guest information

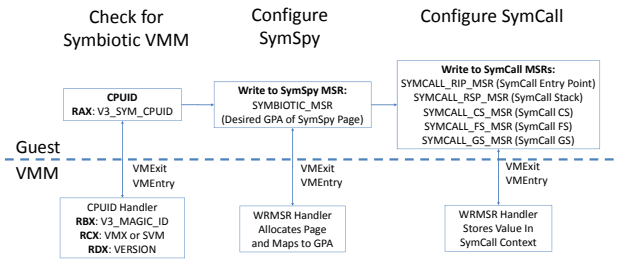


Figure 4.2: Symbiotic VMM discovery/configuration. First the guest executes a CPUID instruction to detect a symbiotic VMM. If found, it then configures SymSpy by writing to a virtualized MSR. Finally, to configure SymCall (described in Chapter 6), an execution environment is created and sent to the VMM by writing to additional virtual MSRs

that may be useful in handling the exit may not be available at the time of the exit.

The functional interface allows a symbiotic VMM to invoke the guest synchronously, during exit handling or from a separate core. However, these invocations have considerably higher costs compared to the passive interface. Furthermore, the implementation complexity may be much higher for two reasons. First, the VMM must be able to correctly support re-entry into the guest *in the process of handling a guest exit*. Second, from the guest's perspective, the functional interface provides an additional source of concurrency *that is not under guest control.* The VMM and guest must be carefully designed so this concurrency does not cause surprise race conditions or deadlocks.

In addition to the functional and passive interfaces, symbiotic virtualization requires a discovery protocol that the guest and VMM can run to determine which, if any, of the interfaces are available, and what data forms and entry points are available.

4.4 Discovery and configuration

One of the principal goals of symbiotic virtualization is to provide an enhanced interface between a VMM and an OS while still allowing compatibility with real hardware. In contrast to paravirtualization, symbiotic virtualization is designed to be enabled and config-

ured at run time without requiring any changes to the OS. As such, symbiotic interfaces are implemented using existing hardware features, such as CPUID values and Model Specific Registers (MSRs). When run on a symbiotic VMM, CPUID and MSR access is trapped and emulated, allowing the VMM to provide extended results. Due to this hardware-like model, the discovery protocol will also work correctly if no symbiotic VMM is being used; the guest will simply not find a symbiotic interface. This allows a guest to detect a symbiotic VMM at boot time and selectively enable symbiotic features that it supports. The discovery and configuration process is shown in Figure 4.2.

In order to indicate the presence of a symbiotic VMM we have created a virtualized CPUID value. The virtualized CPUID returns a value denoting a symbiotic VMM, an interface version number, as well as machine specific interface values to specify hypercall parameters. This maintains hardware compatibility because on real hardware the CPUID instruction simply returns an empty value indicating the non-presence of a symbiotic OS which will cause the OS to abort further symbiotic configurations[1]. If the guest does detect a symbiotic VMM then it proceeds to configure the symbiotic environment using a set of virtualized MSRs as well as the higher level SymSpy interface which we will now describe.

4.5 SymSpy passive interface

The most basic interface used in Symbiotic Virtualization is *SymSpy*. Configuring SymSpy is the second step in the symbiotic configuration process shown in Figure 4.2, and every other symbiotic interface builds on top of it. Other symbiotic interfaces I have developed will be described in following chapters.

The SymSpy interface provides a mechanism for the sharing of structured information

[1]We use CPUID instead of a virtual MSR because accesses to non-present MSRs generate a General Protection Fault

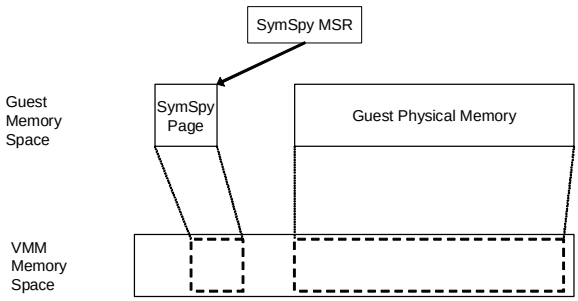


Figure 4.3: The SymSpy passive interface in the Palacios implementation of symbiotic virtualization. The guest reserves space in its physical address space which is used by the VMM to map in a shared memory region.

between the VMM and the guest OS. SymSpy builds on the widely used technique of a shared memory region that is accessible by both the VMM and guest. This shared memory is used by both the VMM and guest to expose semantically rich state information to each other, as well as to provide asynchronous communication channels. The data contained in the memory region is well structured and semantically rich, allowing it to be used for most general purpose cross layer communication. The precise semantics and layout of the data on the shared memory region depends on the symbiotic services that are discovered to be jointly available in the guest and the VMM. The structured data types and layout are enumerated during discovery. During normal operation, the guest can read and write this shared memory without causing an exit. The VMM can also directly access the page during its execution.

A high level view of the SymSpy infrastructure is shown in Figure 4.3. After a guest has detected the presence of a symbiotic VMM it chooses an available guest physical memory address that is not currently in use. This address does not have to be inside the guest's physical memory space. Once an address has been found, the guest writes it to the SymSpy MSR, which is a special virtual MSR implemented by the VMM. The symbiotic VMM intercepts this operation, allocates a new page, and maps it into the guest at the

location specified in the MSR. The SymSpy page is then mapped into both the host's and guest's virtual address space, where it is thereafter accessible to the VMM via a host virtual address, and to the guest via a guest virtual address. For guests that have been assigned multiple cores, SymSpy implements a second per core memory region. This page is used for information pertaining to a local core's execution, instead of global OS state. The mechanism for mapping in per core SymSpy pages is exactly the same as already described, just using a different virtual MSR. The SymSpy interface has been implemented in both Kitten and Linux guest kernels.

4.6 Conclusion

This chapter introduced *symbiotic virtualization*, a new approach to designing virtualization interfaces. A symbiotic architecture supports symbiotic virtual interfaces to allow high level semantic information to be shared across the VMM/guest boundary. These interfaces are necessary because existing virtualization interfaces (both hardware emulation and paravirtualization) are specifically designed to minimize the amount of semantic information available. This has resulted in a large semantic gap between a VMM and a guest. Symbiotic virtualization bridges that gap while still maintaining the compatibility features of the current interfaces. Symbiotic interfaces are entirely optional both for a guest and a VMM, but if they are supported by both they can be used for mutual benefit. The basic interface is SymSpy, a communication channel built on top of shared memory. This allows a guest and VMM to exchange structured information asynchronously in order to expose internal state information in an easy to access manner.

Symbiotic virtualization is of particular use in HPC environments, because it provides detailed information with relatively low overhead compared to existing techniques. In the next chapter I will examine how symbiotic virtualization, and SymSpy in particular, can

be used to optimize the performance of a virtualized guest environment deployed at large scale on the RedStorm Cray XT system.

Chapter 5

Symbiotic Virtualization for High Performance Computing

The evaluation results from Chapter 3 included limited performance studies on 32–48 nodes of a Cray XT system that was virtualized using an early version of Palacios. This chapter continues that evaluation using an enhanced version of Palacios running at much larger scale, up to 4096 nodes on a Cray XT system. In addition to considering the much larger scale, this chapter focuses on symbiotic virtualization techniques needed to achieve scalable virtualization at scale. While these techniques are evaluated specifically on an HPC system, the should generalize beyond both HPC and Palacios.

The essential symbiotic techniques needed to achieve low overhead virtualization at these scales are passthrough I/O, workload-sensitive selection of paging mechanisms, and carefully controlled preemption. Passthrough I/O provides direct guest/application access to the specialized communication hardware of the machine. This in turn enables not only high bandwidth communication, but also preserves the extremely low latency properties of this hardware, which is essential in scalable collective communication.

The second technique we have determined to be essential to low overhead virtualization at scale is the workload-sensitive selection of thes paging mechanisms used to implement the guest physical to host physical address translation. Palacios supports a range of

approaches, from those with significant hardware assistance (e.g., nested paging, which has several implementations across Intel and AMD hardware), to those that do not (e.g., shadow paging, which has numerous variants). There is no single best paging mechanism; the choice is workload dependent, primarily on guest context switching behavior and the memory reference pattern.

The final technique we found to be essential to low overhead virtualization at scale is carefully controlled preemption within the VMM. By preemption, we mean both interrupt handling and thread scheduling. Palacios carefully controls when interrupts are handled, and internally only does cooperative thread context switches. This control means that it mostly avoids introducing timing variation in the environment that the guest OS sees. This in turn means that carefully tuned collective communication behavior in the application remains effective.

What our techniques effectively accomplish is keeping the virtual machine as true to the physical machine as possible in terms of its communication and timing properties. This in turn allows the guest OS's and application's assumptions about the physical machine it is designed for to continue to apply to the virtual machine environment. In the virtualization of a commodity machine such authenticity is not needed. However, if a machine is part of a scalable computer, the disparity between guest OS and application assumptions and the behavior of the actual virtual environment, leads to a performance impact that grows with scale.

We generalize beyond the three specific techniques described above to argue that to truly provide scalable performance for virtualized HPC environments, the black box approach of commodity VMMs should be abandoned in favor of a symbiotic virtualization model. In the symbiotic virtualization model, the guest OS and VMM cooperate in order to function in a way that optimizes performance. Our specific techniques are examples of symbiotic techniques, and are, in fact, built on the SymSpy passive symbiotic interface in

Palacios.

5.1 Virtualization at scale

We conducted a detailed performance study by virtualizing the Red Storm Cray XT supercomputer using the combination of Palacios and Kitten. The study included both application and microbenchmarks, and was run at the largest scales possible on the machine (at least 4096 nodes, sometimes 6240 nodes). The upshot of our results is that it is possible to virtualize a large scale supercomputer with $\leq$5% performance penalties, even when running communication-intensive, tightly coupled applications. In the subsequent sections, we explain how.

5.1.1 Hardware platform

Testing was performed during an eight hour window of dedicated system time on the Red Storm system at Sandia National Labs. We used the same Red Storm nodes from the early experiment, only changing the scale of the studies performed.

5.1.2 Software environment

Each test was performed in at least three different system software configurations: native, guest with nested paging, and guest with shadow paging. In the native configuration, the test application or microbenchmark is run using the native Red Storm operating system, Catamount [46], running on the bare hardware. This is the same environment that users normally use on Red Storm. Some tests were also run, at much smaller scales, using Cray's CNL [45].

The environment that we label *Guest, Nested Paging* in our figures consists of Kitten and Palacios running on the bare hardware, managing an instance of Catamount running

as a guest operating system in a virtual machine environment provided by Palacios. In this mode, the AMD processor's nested paging memory management hardware is used to implement the guest physical address to host physical address mapping that is chosen by Palacios. The guest's page tables and a second set of page tables managed by Palacios are used for translation. Palacios does not need to track guest page table manipulations. However, every virtual address in the guest is translated using a *two dimensional* page walk involving both sets of page tables [9]. This expensive process is sped up through the use of a range of hardware-level TLB and page walk caching structures.

In contrast, the *Guest, Shadow Paging* mode uses software-based memory management provided by Palacios and disables the processor's nested paging hardware. Shadow paging avoids the need for a two dimensional page walk, but it requires that Palacios track guest page tables. Every update to the guest's page tables causes an exit to Palacios, which must then validate the request and commit it to a set of protected *shadow* page tables, which are the actual page tables used by the hardware. We elaborate on the choice of paging mechanism later in the chapter, but generally it is quite dependent on the guest OS. With Catamount, the paging modes give nearly identical performance.

Virtualizing I/O devices is critical to VM performance, and, here, the critical device is the SeaStar communication interface. We provide guest access to the SeaStar using passthrough I/O, an approach we elaborate on later. We consider two ways of using the SeaStar, the default way, which is unnamed in our figures, and an alternative approach called *Accelerated Portals*. The default approach uses interrupt-driven I/O.

In the version of AMD SVM available on Red Storm, intercepting any interrupt requires that all interrupts be intercepted. Because a variety of non-SeaStar interrupts must be intercepted by Palacios, our implementation adds a VM exit cost to SeaStar interrupts. Essentially, when Palacios detects an exit has occurred due to SeaStar interrupt, it immediately re-enters the guest, re-injecting the SeaStar interrupt as a software interrupt. This

process requires O(1000) cycles, resulting in interrupt-driven SeaStar performance having a higher latency when virtualized rather than native.

In *Accelerated Portals*, the guest uses user-level polling instead of interrupts for message transmission and reception. This is the fastest way of using SeaStar natively. Because interrupts are not involved, the interrupt exit cost described above does not occur when the guest is virtualized. The upshot is that virtualized accelerated portals performance is nearly identical to native accelerated portals performance.

It is important to point out that more recent versions of AMD's SVM hardware (and of Intel's VT hardware) can support much more selective interrupt exiting. If such hardware were available, we would use it to avoid exiting on SeaStar interrupts, which should make interrupt-driven SeaStar performance under virtualization identical to that without virtualization.

The guest Catamount OS image we used was based on the same Cray XT 2.0.62 Catamount image used for the native experiments. Minor changes were required to port Catamount to the PC-compatible virtual machine environment provided by Palacios (the native Cray XT environment is not fully PC-compatible). Additionally, the SeaStar portals driver was updated to allow passthrough operation as described in Section 5.2.

5.1.3 MPI microbenchmarks

The Intel MPI Benchmark Suite version 3.0 [39] was used to evaluate point-to-point messaging performance and scalability of collective operations.

Point-to-point performance

Figure 5.1 shows the results of a ping-pong test between two adjacent nodes. Small message latency, shown in Figure 5.1(a), is approximately 2.5 times worse with nested or shadow guest environments compared to native. This is a result of the larger interrupt

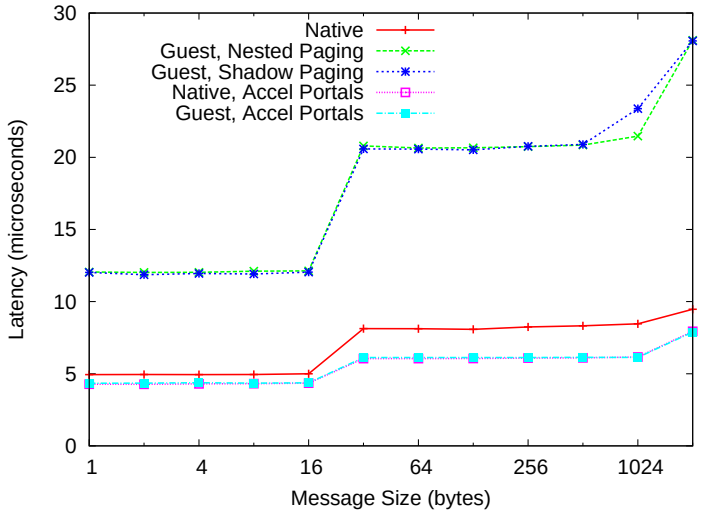
overhead in the virtualized environment. However, note that in absolute terms, for the smallest messages, the latency for the virtualized case is already a relatively low 12 μs, compared to the native 5 μs. Eliminating this virtualized interrupt overhead, as is the case with accelerated portals, and would be the case with more recent AMD SVM hardware implementations, results in virtually identical performance in native and guest environments.

Figure 5.1(b) plots the same data but extends the domain of the x-axis to show the full bandwidth curves. The nested and shadow guest environments show degraded performance for mid-range messages compared to native, but eventually reach the same asymptotic bandwidth once the higher interrupt cost is fully amortized. Bandwidth approaches 1.7 GByte/s. Avoiding the interrupt virtualization cost with accelerated portals results again in similar native and guest performance.

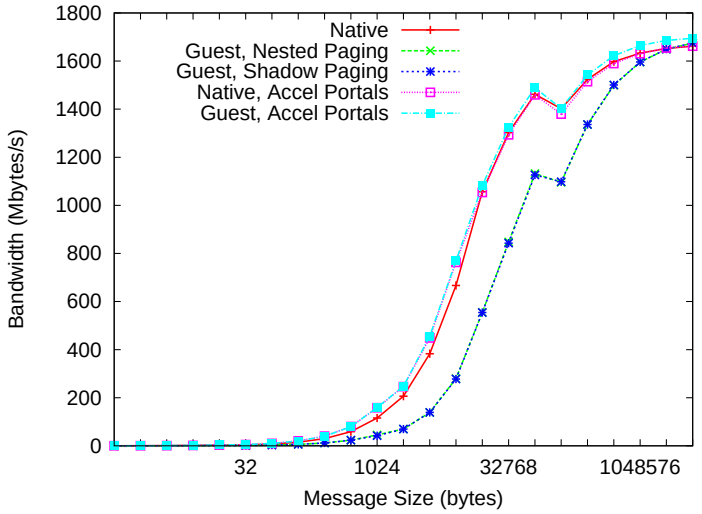
Collective performance

Figures 5.2, 5.3, and 5.4 show the performance of the MPI Barrier, Allreduce, and Alltoall operations, respectively. The operations that have data associated with them, Allreduce and Alltoall, are plotted for the 16-byte message size since a common usage pattern in HPC applications is to perform an operation on a single double-precision number (8 bytes) or a complex double precision number (16 bytes).

Both Barrier and Allreduce scale logarithmically with node count, with Allreduce having slightly higher latency at all points. In contrast, Alltoall scales quadratically and is therefore plotted with a log y-axis. In all cases, the choice of nested vs. shadow paging does not appear to matter. What does matter, however, is the use of an interrupt-driven versus a polling-based communication in the guest environment. Similarly to what was observed in the point-to-point benchmarks, eliminating network interrupts by using the polling-based accelerated portals network stack results in near native performance. As noted previously, more recent AMD SVM implementations support selective interrupt ex-



(a) Latency



(b) Bandwidth

Figure 5.1: MPI PingPong microbenchmark measuring (a) latency and (b) bandwidth Both interrupt-driven access and accelerated portals-based access to the SeaStar NIC are shown. For interrupt-driven access, virtualization adds an additional 8–16 μs to latency and achieves lower bandwidth for small messages. With the AMD SVM virtualization hardware available on Red Storm, passthrough interrupt delivery requires a VM exit. More recent hardware relaxes this requirement and should result in native performance. For accelerated portals, virtualized and native performance are already nearly identical.

iting, which would make the virtualized interrupt-driven performance identical to the native or virtualized accelerated portals numbers. Still, even with this limitation, virtualized interrupt-driven communication is quite fast in absolute terms, with a 6240 node barrier or all-reduce taking less than 275 μs to perform.

The Alltoall operation is interesting because the size of the messages exchanged between nodes increases with node count. This causes all of the configurations to converge at high node counts, since the operation becomes bandwidth limited, and the cost of interrupt virtualization is amortized.

5.1.4 HPCCG application

HPCCG [33] is a simple conjugate gradient solver that is intended to mimic the characteristics of a broad class of HPC applications in use at Sandia, while at the same time being simple to understand and run. A large portion of its runtime is spent performing sparse matrix-vector multiplies, which is a memory bandwidth intensive operation.

HPCCG was used in weak-scaling mode with a "100x100x100" subproblem on each node, using approximately 380 MB of memory per node. This configuration is representative of typical usage, and results in relatively few and relatively large messages being communicated between neighboring nodes. Every iteration of the CG algorithm performs an 8-byte Allreduce, and there are 149 iterations during the test problem's approximately 30 second runtime. The portion of runtime consumed by communication is reported by the benchmark to be less than 5% in all cases. Interrupt-driven communication was used for this and other application benchmarks. Recall that the microbenchmarks show virtualized interrupt-driven communication is the slower of the two options we considered.

As shown in Figure 5.5, HPCCG scales extremely well in both guest and native environments. Performance with shadow paging is essentially identical to native performance, while performance with nested paging is 2.5% worse at 2048 nodes.

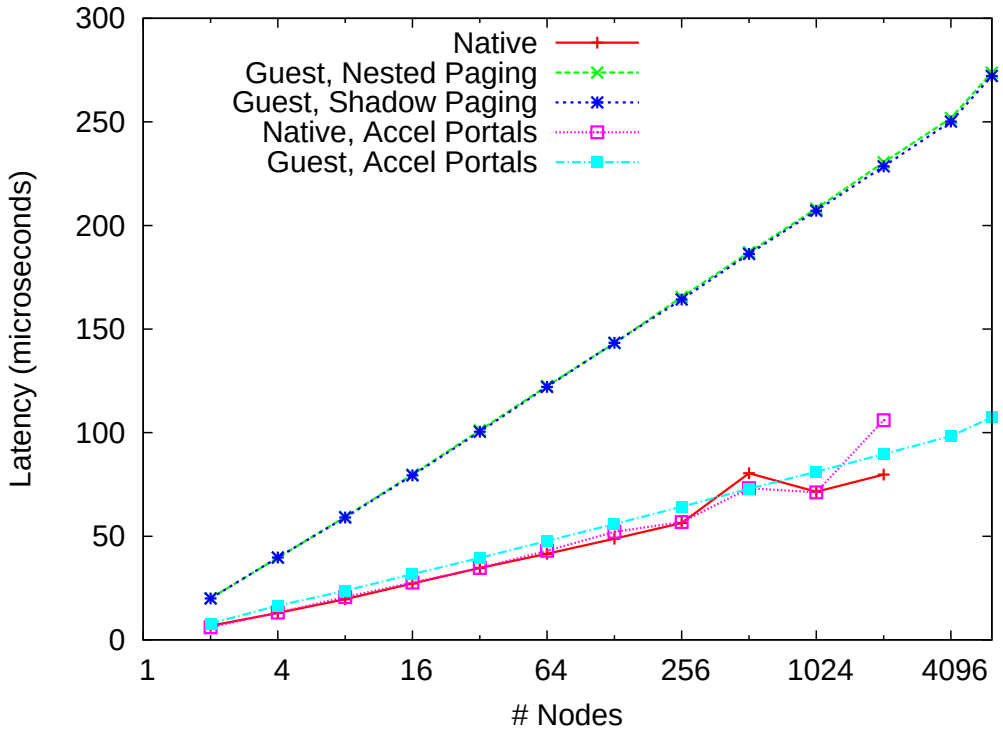


Figure 5.2: MPI barrier scaling microbenchmark results measuring the latency of a full barrier. Both interrupt-driven access and accelerated portals-based access to the SeaStar NIC are shown. For interrupt-driven access, the additional interrupt latency overhead compounds as the barrier scales. With the AMD SVM virtualization hardware available on Red Storm, passthrough interrupt delivery requires a VM exit. More recent hardware relaxes this requirement and thus should provide native performance. For accelerated portals, virtualized and native performance are already nearly identical.

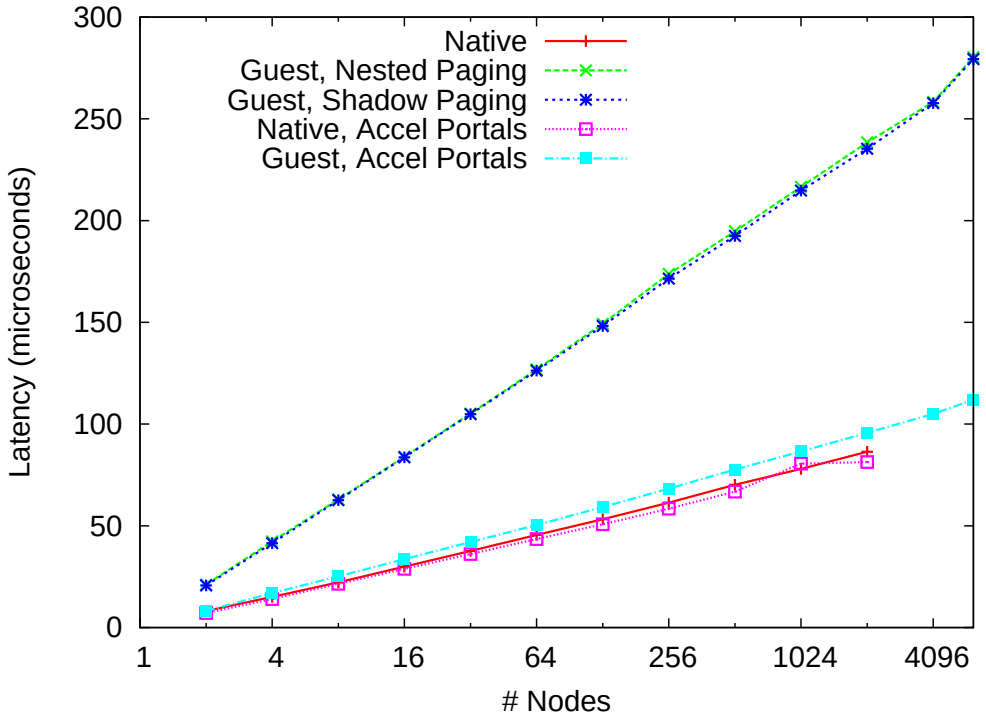


Figure 5.3: MPI Allreduce scaling microbenchmark results measuring the latency of a 16 byte all-reduce operation. Both interrupt-driven access and accelerated portals-based access to the SeaStar NIC are shown. For interrupt-driven access, the additional interrupt latency overhead compounds as the number of nodes in the reduction scales. With the AMD SVM virtualization hardware available on Red Storm, passthrough interrupt delivery requires a VM exit. More recent hardware relaxes this requirement and thus should provide native performance. For accelerated portals, virtualized and native performance are already nearly identical.

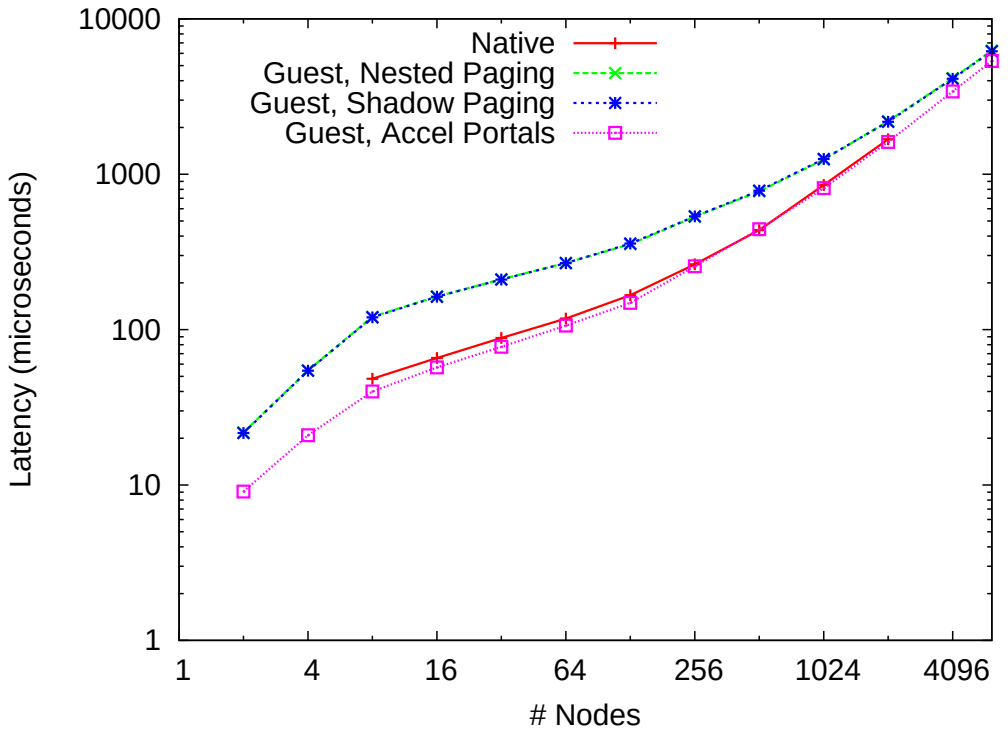


Figure 5.4: MPI AlltoAll scaling microbenchmark results measuring the latency of a 16 byte all-to-all operation. Both interrupt-driven access and accelerated portals-based access to the SeaStar NIC are shown. For interrupt-driven access, the effect of the additional interrupt latency in virtualization declines as we scale up, with performance converging at about 2048 nodes. Virtualized access to accelerated portals performs identically to native.

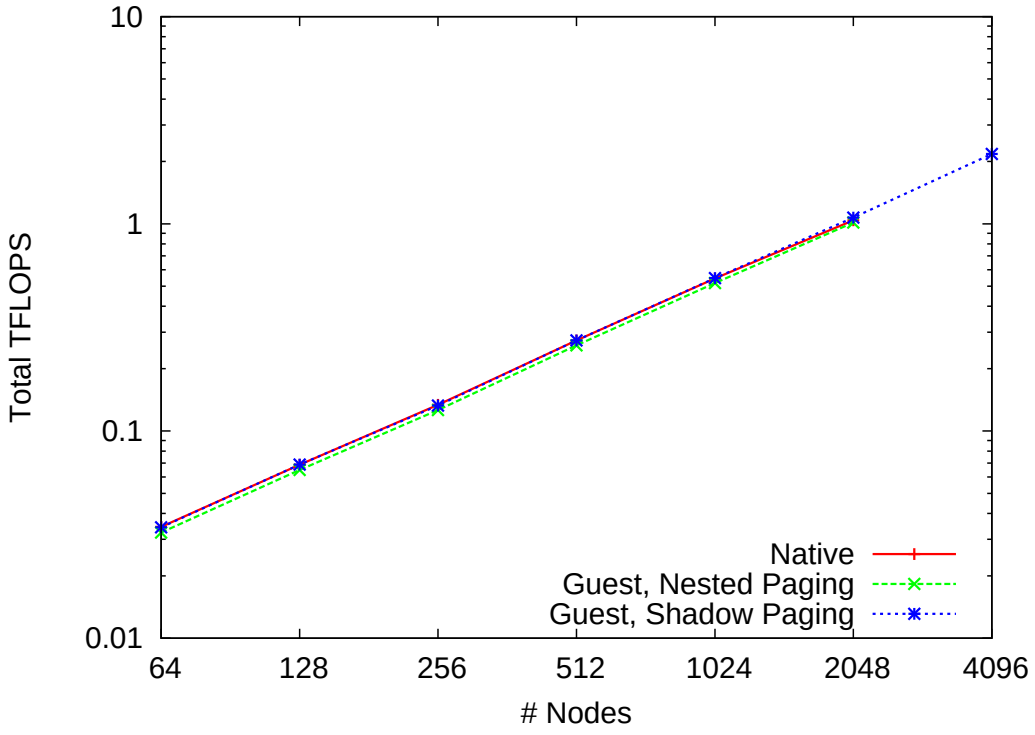


Figure 5.5: HPCCG application benchmark performance. Weak scaling is measured. Virtualized performance is within 5% of native.

5.1.5 CTH application

CTH [20] is a multi-material, large deformation, strong shock wave, solid mechanics code developed by Sandia National Laboratories. It is used for studying armor/anti-armor interactions, warhead design, high explosive initiation physics, and weapons safety issues.

A shaped charge test problem was used to perform a weak scaling study in both native and guest environments. As reported in [21], which used the same test problem, at 512 nodes approximately 40% of the application's runtime is due to MPI communication, 30% of which is due to `MPI_Allreduce` operations with an average size of 32 bytes. The application performs significant point-to-point communication with nearest neighbors using large messages.

Figure 5.6 shows the results of the scaling study for native and guest environments. At 2048 nodes, the guest environment with shadow paging is 3% slower than native, while the nested paging configuration is 5.5% slower. Since network performance is virtually identical with either shadow or nested paging, the performance advantage of shadow paging is likely due to the faster TLB miss processing that it provides.

5.1.6 SAGE application

SAGE (SAIC's Adaptive Grid Eulerian hydrocode) is a multidimensional hydrodynamics code with adaptive mesh refinement developed at Los Alamos National Laboratory [47]. The timing_c input deck was used to perform a weak scaling study. As reported in [21], which used the same test problem, at 512 nodes approximately 45% of the application's runtime is due to MPI communication, of which roughly 50% is due to `MPI_Allreduce` operations with an average size of 8 bytes. In the same report, Sage is measured to be 3.5 times more sensitive to noise than CTH at 2048 nodes.

Figure 5.7 shows the results of executing the scaling study in the native and virtualized

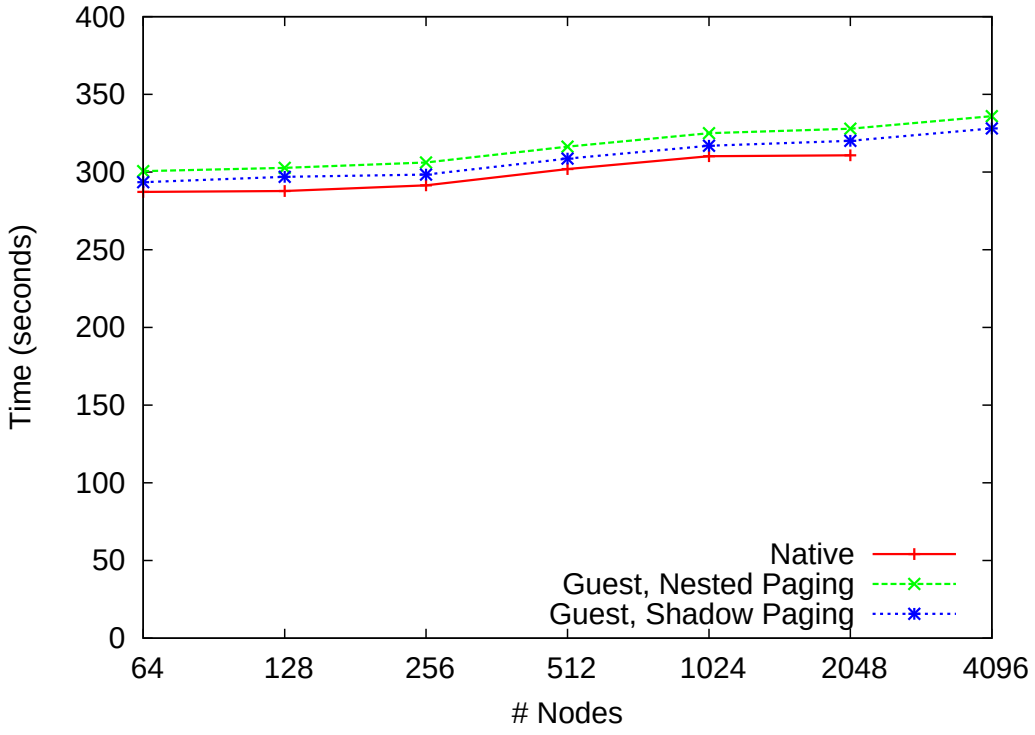


Figure 5.6: CTH application benchmark performance. Weak scaling is measured. Virtualized performance is within 5% of native.

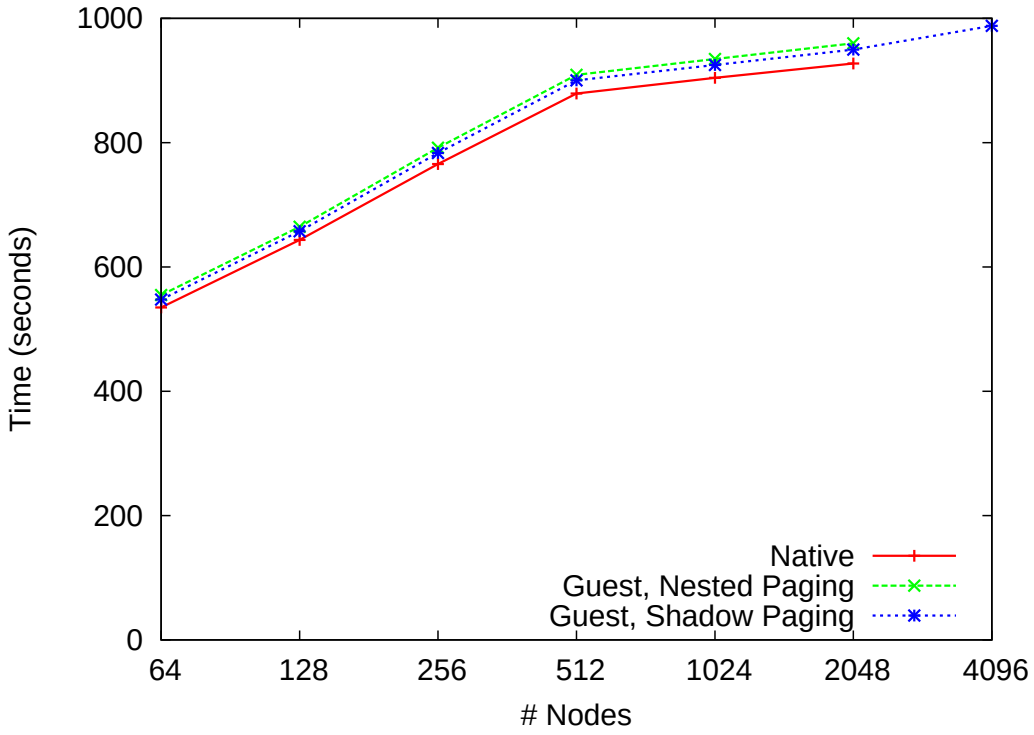


Figure 5.7: Sage application benchmark performance. Weak scaling is measured. Virtualized performance is within 5% of native.

environments. At 2048 nodes, shadow paging is 2.4% slower compared to native while nested paging is 3.5% slower. As with CTH, the slightly better performance of shadow paging is believed to be due to its faster TLB miss processing.

5.2 Passthrough I/O

One of the principle goals in designing Palacios was to allow a large amount of configurability in order to allow targeting of multiple and diverse environments. This allows us to enable a number of configuration options that are specific to HPC environments, in order to minimize any overheads and maximize performance. The special HPC configuration of Palacios makes a number of fundamental choices in order to provide guest access to

hardware devices with as little overhead as possible. These choices were reflected both in the architecture of Palacios, as configured for HPC, as well as two assumptions about the environment Palacios executes in.

The first assumption we make for HPC environments is that only a single guest will be running on a node at any given time. Node here refers to some specific partition of the physical resources, be that a single CPU core, a single multicore CPU, or a collection of multicore CPUs. Restricting each partition to run a single guest environment ensures that there is no resource contention between multiple VMs. This is the common case for capability supercomputers as each application requires dedicated access to the entirety of the system resources. It is also the common case for space-shared capacity machines. The restriction vastly simplifies device management because Palacios does not need to support sharing of physical devices between competing guests; Palacios can directly map an I/O device into a guest domain without having to manage the device itself.

The second assumption we make for HPC environments is that we can place considerable trust in the guest OS because HPC system operators typically have full control over the entire software stack. Under this assumption, the guest OS is unlikely to attempt to compromise the VMM intentionally, and may even be designed to help protect the VMM from any errors.

5.2.1 Passthrough I/O implementation

In Palacios, passthrough I/O is based on a virtualized PCI bus. The virtual bus is implemented as an emulation layer inside Palacios, and has the capability of providing access to both virtual as well as physical (passthrough) PCI devices. When a guest is configured to use a passthrough device directly, Palacios scans the physical PCI bus searching for the appropriate device and then attaches a virtual instance of that device to the virtual PCI bus. Any changes that a guest makes to the device's configuration space are applied only to the

virtualized version. These changes are exposed to the physical device via reconfigurations of the guest environment to map the virtual configuration space onto the physical one.

As an example, consider a PCI Base Address Register (BAR) that contains a memory region that is used for memory-mapped access to the device. Whenever a guest tries to change this setting by overwriting the BAR's contents, instead of updating the physical device's BAR, Palacios instead updates the virtual device's BAR and reconfigures the guest's physical memory layout so that the relevant guest physical memory addresses are redirected to the host physical memory addresses mapped by the real BAR register. In this way, Palacios virtualizes configuration operations but not the actual data transfer.

Most devices do not rely on the PCI BAR registers to define DMA regions for I/O. Instead the BAR registers typically point to additional, non-standard, configuration spaces, that themselves contain locations of DMA descriptors. Palacios makes no attempt to virtualize these regions, and instead relies on the guest OS to supply valid DMA addresses for its own physical address space. While this requires that Palacios trust the guest OS to use correct DMA addresses as they appear in the host, it is designed such that there is a a high assurance that the DMA addresses used by the guest are valid.

The key design choice that provides high assurance of secure DMA address translation from the guest physical addresses to the host physical addresses is the shape of the guest's physical address space. A Palacios guest is initially configured with a physically contiguous block of memory that maps into the contiguous portion of the guest's physical address space that contains memory. This allows the guest to compute a host physical address from a guest physical address by simply adding an offset value. This means that a passthrough DMA address can be immediately calculated as long as the guest knows what offset the memory in its physical address space begins at. Furthermore, the guest can know definitively if the address is within the bounds of its memory by checking that it does not exceed the range of guest physical addresses that contain memory, information that is

readily available to the guest via the e820 map and other standard mechanisms. Because guest physical to host physical address translation for actual physical memory is so simple, DMA addresses can be calculated and used with a high degree of certainty that they are correct and will not compromise the host or VMM. The code required is literally just a few lines and very difficult to get wrong.

It is also important to point out that as long as the guest uses physical addresses valid with respect to its memory map, it is impossible for it to affect the VMM or other passthrough or virtual devices with a DMA request on a passthrough device.

To allow the guest to determine when a DMA address needs to be translated (by offsetting) for passthrough access, Palacios uses SymSpy to advertise which PCI devices are in fact configured as passthrough. Each PCI bus location tuple (bus ID, device ID, and function number) is combined to form an index into a bitmap. If a device is configured as passthrough the bit at its given index will be set by the VMM and read by the guest OS. This bitmap allows the guest OS to selectively offset DMA addresses, allowing for compatibility with both passthrough devices (which require offsetting) and virtual devices (which do not). Furthermore, when the guest is run without the VMM in place, this mechanism naturally turns off offsetting for all devices.

Comparison with other approaches to high performance virtualized I/O: Due to both the increased trust and control over the guest environments as well as the simplified mechanism for DMA address translation, Palacios can rely on the guest to correctly interact with the passthrough devices. The passthrough I/O technique allows direct interaction with hardware devices with as little overhead as possible. In contrast, other approaches designed to provide passthrough I/O access must add additional overhead. For example, VMM-Bypass [59], as designed for the Xen Hypervisor, does not provide the same guarantees in terms of address space contiguousness. Furthermore, their usage model assumes that the guest environments are not fully trusted entities. The result is that the implementation

complexity is much higher for VMM-Bypass, and further overheads are added due to the need for the VMM to validate the device configurations. Furthermore, their technique is highly device specific (specifically Infiniband) whereas our passthrough architecture is capable of working with any unmodified PCI device driver.

Self-Virtualization [78] is a technique to allow device sharing without the need for a separate virtual driver domain. While self virtualization does permit direct guest interaction with hardware devices it does via a simplified virtual interface which places a limit on the usable capabilities of the device. This approach also requires specially architected hardware, while our passthrough implementation supports any existing PCI device.

5.2.2 Current implementations

We have currently implemented passthrough I/O for both a collection of HPC OSes, such as Catamount and Kitten, as well as for commodity Linux kernels. The Catamount OS specifically targets the Cray SeaStar as its only supported I/O device, therefore Catamount did not require a general passthrough framework. However, Kitten and Linux are designed for more diverse environments so we have implemented the full passthrough architecture in each of them. In each case, the implementation is built on the SymSpy guest implementation (Section 4.3), which consists of about 300 lines of C and assembler. The actual DMA address offsetting and bounds checking implementation is about 20 lines of C.

Both Kitten and Linux include the concept of a DMA address space that is conceptually separate from the address space of core memory. This allows a large degree of compatibility between different architectures that might implement a separate DMA address space. The environment exposed by Palacios is such an architecture. Every time a device driver intends to perform a DMA operation it must first transform a memory address into a DMA address via a DMA mapping service. Our guest versions of both Linux and Kitten include a modified mapping service that selectively adds the address offset to each DMA address

if the device requesting the DMA translation is configured for passthrough. Our modifications also perform a sanity check to ensure that the calculated DMA address resides inside the guests memory space, thus protecting the VMM from any malformed DMA operations. These modifications are small, easy to understand, and all encompassing, meaning that the VMM can have a high degree of confidence that even a complicated OS such as Linux will not compromise the VMM via malformed DMA operations.

5.2.3 Infiniband passthrough

To quantify the overhead of Palacios virtualization with a commodity NIC, we examined the performance of Mellanox MLX4 (ConnectX) cards configured for passthrough in Linux. Passthrough support for these 64-bit PCI Express devices was provided through the PCI passthrough support described above.

We measured round-trip latency for 1 byte messages averaged over 100000 round trips and 4 megabyte message round trip bandwidth averaged over 10000 trips using the OpenFabrics `ibv_rc_pingpong` program. The client system which performed the timings ran native Fedora 11 with Linux kernel 2.6.30, and the client machine ran a diskless Linux BusyBox image that also used Linux kernel 2.6.30 with symbiotic extensions either natively or virtualized in Palacios using shadow paging.

As can be seen in Figure 5.8, Palacios's pass-through virtualization imposes almost no measurable overhead on Infiniband message passing in both Kitten and Linux. In particular, Palacios's passthrough PCI support enables virtualized Linux to almost perfectly match the bandwidth of native Linux on Infiniband, and because Infiniband does not use interrupts for high-speed message passing with reliable-connected channels, the 1-byte message latencies with and without virtualization are identical.

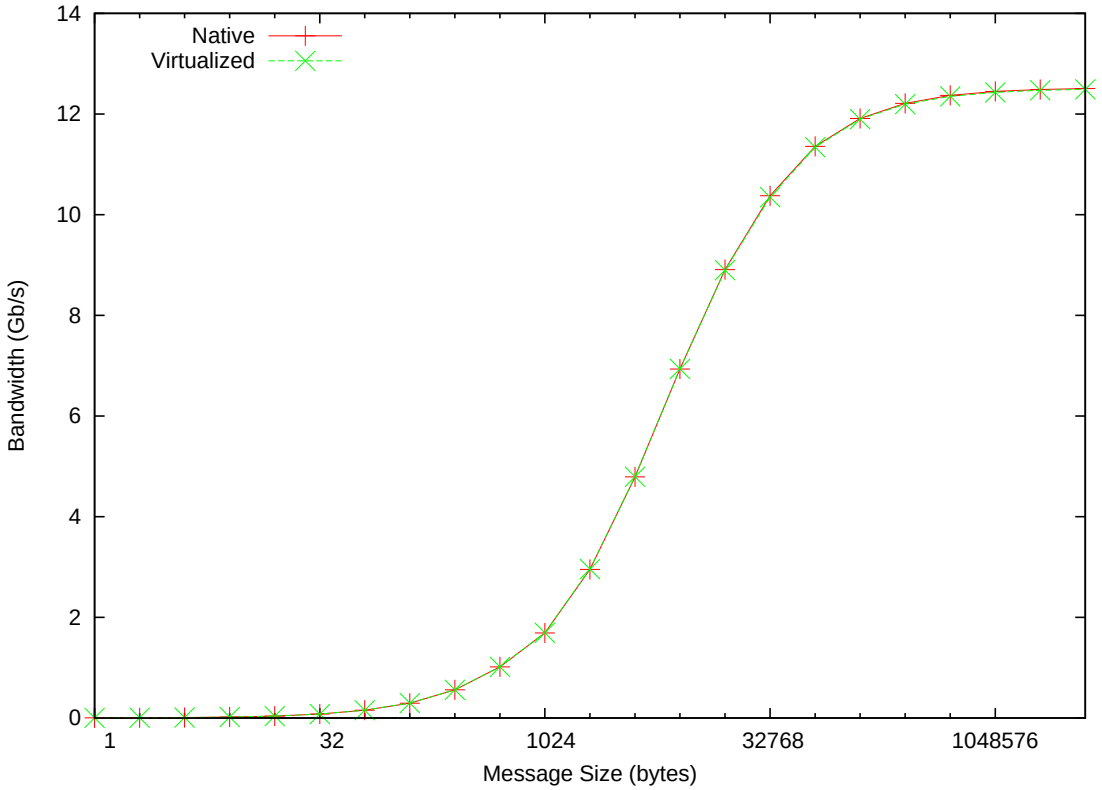


Figure 5.8: Infiniband bandwidth at message sizes from 1 byte to 4 megabytes. 1-byte round-trip latency both native and virtualized was 6.46μsec, with peak bandwidth for 4MB messages at 12.49 Gb/s on Linux virtualized with Palacios compared to 12.51 Gb/s for native Linux.

5.2.4 Future extensions

Future advances in hardware virtualization support may obviate the need for the passthrough techniques described above. For example, AMD's IOMMU adds hardware support for guest DMA translations. However, we should note that our approach includes a very minimal amount of overhead and it is not clear that hardware techniques will necessarily perform better. An IOMMU would introduce additional performance overhead in the form of page table lookups, something which our approach completely avoids. As we will show in the next section and others have demonstrated [1], with the appropriate assumptions software approaches can often demonstrably operate with less overhead than hardware approaches.

5.3 Workload-sensitive paging mechanisms

In our evaluations we have focused on two standard techniques for virtualizing the paging hardware: Shadow Paging and Nested Paging as described in Section 5.1.2. We have found that the best performing technique is heavily dependent on the application workload as well as the architecture of the guest OS. As an example, Catamount performs a minimal number of page table operations, and never fully flushes the TLB or switches between different page tables. This means that very few operations are required to emulate the guest page tables with shadow paging. Because the overhead of shadow paging is so small, shadow paging performs better than nested paging due to the better use of the hardware TLB. In contrast, Compute Node Linux (CNL) another HPC OS, uses multiple sets of page tables to handle multitasking and so frequently flushes the TLB. For this OS there is a great deal more overhead in emulating the page table operations and any improvement in TLB performance is masked by the frequent flush operations. For this case nested paging is clearly the superior choice.

Based on our earlier evaluation in Chapter 3, we demonstrated that the behavior of the guest OS and applications have a critical impact on the performance of the virtualized paging implementation. We have found this to be true in the broader server consolidation context [4] as well as the HPC context we discuss here. Those results showed that the choice of virtual paging techniques is critically important to ensuring scalable performance in HPC environments and that the best technique varies across OSes and applications. This suggests that an HPC VMM should provide a mechanism for specifying the initial paging technique as well as for switching between techniques during execution. Furthermore, an HPC VMM should provide a range of paging techniques to choose from. Palacios incorporates a modular architecture for paging architectures. New techniques can be created and linked into the VMM in a straightforward manner, with each guest being able to dynamically select among all the available techniques at runtime.

5.4 Controlled preemption

It is well understood that background noise can have a serious performance impact on large scale parallel applications. This has led to much work in designing OSes such that the amount of noise they inject into the system is minimized. Palacios is designed not only minimize the amount of overhead due to virtualization, but also to concentrate necessary overheads and work into deterministic points in time in an effort to minimize the amount of noise added to the system by virtualization.

Palacios runs as a non-preemptible kernel thread in Kitten. Only interrupts and explicit yields by Palacios can change control flow. Palacios controls the global interrupt flag and guest interrupt exiting and uses this control to allow interrupts to happen only at specific points during exit handling. This combination of behaviors allows Palacios to guarantee well-controlled availability of CPU resources to the guest. Background pro-

cesses and deferred work are only allowed to proceed when their impact on performance will be negligible.

When a guest is configured it is allowed to specify its execution quantum which determines the frequency at which it will yield the CPU to the Kitten scheduler. It is important to note that the quantum configured by Palacios is separate from the scheduling quantum used by Kitten for task scheduling. This separation allows each guest to override the host OS scheduler in order to prevent the host OS from introducing additional OS noise. Furthermore this quantum can be overridden at runtime such that a guest can specify critical sections where Palacios should not under any circumstances yield the CPU to another host process.

5.4.1 Future extensions

An extant issue in HPC environments is the overhead induced via timer interrupts. A large goal of Kitten is to implement a system with no dependence on periodic interrupts, and instead rely entirely on on-demand one shot timers. However, periodic timers are occasionally necessary when running a guest environment with Palacios, in order to ensure that time advances in the guest OS. Because some guest OSes do require periodic timer interrupts at a specified frequency, the VMM needs to ensure that the interrupts can be delivered to the guest environment at the appropriate rate. We are developing a method in which the guest OS is capable of both enabling/disabling as well as altering the frequency of the host's periodic timer. This would allow a guest OS to specify its time sensitivity[1], which will allow Palacios and Kitten to adapt the timer behavior to best match the current workload.

[1] You can think of this as being loosely correlated to the guest's timer frequency setting

5.5 Conclusion

Our primary contribution has been to demonstrate that it is possible to virtualize the largest parallel supercomputers in the world[2] at very large scales with minimal performance overheads. Even tightly coupled, communication-intensive applications running on specialized lightweight OSes that provide maximum hardware capabilities to them can run in a virtualized environment with $\leq$5% performance overhead at scales in excess of 4096 nodes. This result suggests that such machines can reap the many benefits of virtualization that have been articulated before (e.g., [37, 22]). One benefit not previously noted is that virtualization could open the range of applications of the machines by making it possible to use commodity OSes on them in capacity modes when they are not needed for capability purposes.

We believe our results represent the largest scale study of HPC virtualization by at least two orders of magnitude, and we have described how such performance is possible. Scalable high performance rests on passthrough I/O, workload sensitive selection of paging mechanisms, and carefully controlled preemption. These techniques are made possible via a symbiotic interface between the VMM and the guest, an interface we have generalized with SymSpy.

Beyond supercomputers, our experiences with these symbiotic techniques are increasingly relevant to system software for general-purpose and enterprise computing systems. For example, the increasing scale of multicore desktop and enterprise systems has led OS designers to consider treating multicore systems like tightly-coupled distributed systems. As these systems continue to scale up toward hundreds or thousands of cores with distributed memory hierarchies and substantial inter-core communication delays, lessons learned in designing scalable system software for tightly-coupled distributed memory su-

[2]Red Storm is currently the 17th fastest machine in the world.

percomputers will be increasingly relevant to them. In many ways these systems will contain increased complexity, because they will run be designed with much more diverse software and hardware environments. The SymSpy symbiotic interface is effective in HPC settings due to the relatively small amount of state information needed for optimization. However, in commodity environments the increased complexity of both the hardware and software will require more detailed and complex information that SymSpy is capable of exposing. As a solution to this the next chapter will discuss SymCall, a functional symbiotic interface that allows a VMM to request that a guest execute state queries on the VMM's behalf.

Chapter 6

Symbiotic Upcalls

Having demonstrated the utility of symbiotic virtualization in the realm of HPC, I will now shift focus commodity environments. While the impetus for supporting symbiotic interfaces in HPC systems is quite large, it also is of equal use for more standard system architectures. As I have explained, there is a great deal of interest in optimizing virtualized performance in data center environments, and several techniques have been developed for bridging the semantic gap present in those systems. However, with the wide deployment of complex OS environments such as Linux, the amount and complexity of OS state information is quite formidable. Even a symbiotic interface such as SymSpy would be incapable of handling the amount of information necessary to optimize many of these systems. This increased complexity requires a new approach to collecting state information, one that does not require a guest environment to preemptively expose an overwhelming amount of data. Instead of relying on the guest OS to provide the data in an easily accessible manner, it is possible for it to support a functional interface that allows a VMM to run queries against it. This would allow the guest OS to organize its internal state however it wanted, and still provide a mechanism whereby a VMM could easily access it. This chapter will evaluate SymCall, a symbiotic interface that provides a VMM with functional access to a guest's internal context.

6.1 Introduction

The SymCall interface is designed to allow the VMM to make synchronous upcalls into the guest kernel. SymCall essentially makes it possible for a guest to easily provide an efficient and safe system call interface to the VMM. These calls can then be used during the handling of a guest exit. That is, the VMM can invoke the guest *during the handling of a guest exit.* This chapter describes the design and implementation of SymCall in Palacios in considerable detail, and evaluates the latency of SymCall.

Using the SymCall interface, I designed, implemented, and evaluated a proof-of-concept symbiotic service in Palacios. This service, *SwapBypass*, uses shadow paging to reconsider swapping decisions made by a symbiotic Linux guest running in a VM. If the guest is experiencing high memory pressure relative to its memory partition, it may decide to swap a page out. However, if the VMM has available physical memory, this is unnecessary. Although a page may be swapped out and marked unavailable in the guest page table, SwapBypass can also keep the page in memory and mark it available in the shadow page table. The effect is that access to the "swapped" page is at main memory speeds, and that the guest is using more physical memory than initially allotted, even if it is incapable of dynamically adapting to changing physical memory size.

Implementing SwapBypass requires information about the mapping of swap IDs to swap devices, which is readily provided via a SymCall, but extremely challenging to glean otherwise. I evaluated SwapBypass both through performance benchmarks, and through an examination of its implementation complexity.

6.2 SymCall functional interface

SymCalls are a new VMM/guest interface by which a VMM can make synchronous upcalls into a running guest OS. In a guest OS, this interface is designed to resemble the existing

system call interface as much as possible, both in terms of the hardware interface presented to the guest, as well as the internal upcall implementations. Based on the similarity to system calls I refer to symbiotic upcalls as symcalls.

The x86 architecture has a several well defined frameworks for supporting OS system calls. These interfaces allow a system call to be executed via a special instruction that instantiates a system call context defined at initialization time by the OS. The interfaces also define a separate instruction to return from a system call and reinstate the calling process context. Two versions of these instructions exist: SYSENTER/SYSEXIT and SYSCALL/SYSRET. As part of it's initialization, the OS writes to a series of MSRs that collectively define the environment that will be created when the system call instruction is executed. While the two interfaces vary slightly, in general the environments they create are the same. The environmental context consists of an instruction and stack pointer, as well as code and stack segment selectors. There is also support for auxiliary segment selectors for the FS and GS segments, which are used by the OS to reference CPU local storage areas. These components are written to a special set of MSRs. When a System call instruction is executed, the context variables are copied out of the MSRs and instantiated on the hardware. When execution resumes the CPU is running a special OS code path that dispatches to the correct system call handler. When a system call returns it executes the corresponding exit instructions that reverse this procedure.

Due to the conceptual similarity between symcalls and system calls I designed our implementation to be architecturally similar as well. Just as with system calls, the guest OS is responsible for enabling and configuring the environment which the symcalls will execute in. It does this using a set of virtualized MSRs that are based on the actual MSRs used for the SYSCALL and SYSRET interface. When the VMM makes a symbiotic upcall, it configures the guest environment according to the values given by the guest OS. The next time the guest executes it will be running in the SymCall dispatch routine that invokes

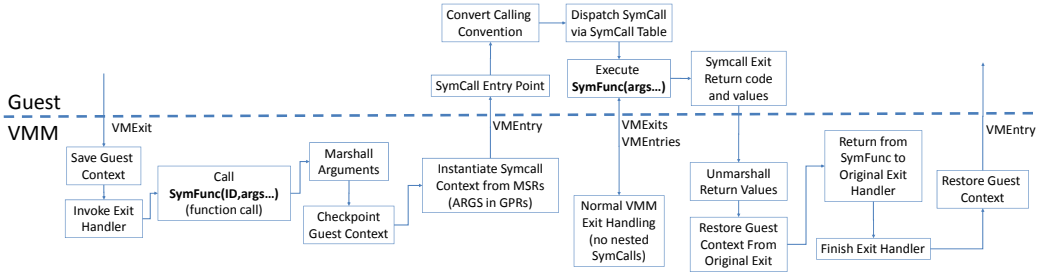


Figure 6.1: The execution path of the SymCall functional interface in the Palacios VMM. The execution follows 3 high level stages. First the VMM reconfigures the guest context to execute a symcall and performs a VM entry. While the SymCall is executing it can take additional exits that are handled by the VMM identically to normal exits. Finally when the symcall returns the VMM rolls the guest context back to the state before the symcall was called.

the handler for the specific symcall. This configuration is the third step of the symbiotic configuration procedure in Figure 4.2.

6.2.1 Virtual hardware support

The SymCall virtual hardware interface consists of a set of MSRs that are a union of the MSRs used for the SYSENTER and SYSCALL frameworks[1]. I combine both the MSR sets to provide a single interface that is compatible for both the Protected (32 bit) and Long (64 bit) operating modes. The set of symbiotic MSRs are:

- *SYMCALL_RIP:* The value to be loaded into the guest's RIP/EIP register. This is the address of the entry point for symcalls in the guest kernel
- *SYMCALL_RSP:* The value to be loaded into the guest's RSP/ESP register. This is the address of the top of the stack that will be used when entering a symcall.

[1]The execution model however more closely resembles the SYSCALL behavior

- *SYMCALL_CS:* The location of the code segment to be loaded during a symcall. This is the code segment that will be used during the symcall. The stack segment is required to immediately follow the code segment, and so can be referenced via this MSR.

- *SYMCALL_GS:* The GS segment base address to be loaded during a symcall.

- *SYMCALL_FS:* The FS segment base address to be loaded during a symcall. The GS or FS segments are used to point to kernel-level context for the symcall.

The RIP, RSP, and CS(+SS) MSRs are needed to create the execution context for the symbiotic upcall. The FS and GS MSRs typically hold the address of the local storage on a given CPU core. FS or GS is typically used based on the operating mode of the processor.

As I stated earlier the execution model for a symbiotic upcall is based on system calls. The one notable difference is that symbiotic upcalls always store the guest state before the call is executed and reload it when the symcall returns. Furthermore the state is saved inside the VMM's address space and so is inaccessible to the guest OS. This is largely a safety precaution due to the fact that the guest OS has much less control over when a symbiotic call is executed. For example, a system call can only be executed when a process is running, but a symcall can also occur when the guest is executing in the kernel.

As described, the system call return process copies back the context that existed before the system call was made (but possibly modified afterwards). Returning from a symbiotic upcall is the same with the exception being that the symbiotic call always returns to the context immediately before the symcall was made. This is because the calling state is not saved in the guest environment, but instead stored by the VMM. Because there is no special instruction to return from a symcall the guest instead executes a special hypercall indicating the return value.

Component	Lines of code
VMM infrastructure	300(C)
Guest infrastructure	211(C) + 129(ASM)
Total	511(C) + 129(ASM)

Figure 6.2: Lines of code needed to implement the SymCall infrastructure as measured by SLOCcount

The virtual hardware interface I have developed follows the system call design to minimize the behavioral changes of a guest OS. Our other objective was to create an interface that would be implementable in physical hardware. Existing hardware implementations could be extended to provide hardware versions of the MSRs that would only be accessible while the CPU is executing in a VM context. A second type of VM entry could be defined which launches into the state defined by the MSRs and automatically saves the previous guest state in the virtual machine control structures. And finally a new instruction could implemented to return from a symbiotic upcall and reload the saved guest state.

6.2.2 Symbiotic upcall interface

Using the virtual hardware support, I have implemented a symbiotic upcall facility in the Palacios VMM. Furthermore I have implemented symbiotic upcall support for two guest OSes: 32 bit Linux and the 64 bit Kitten OS. Our SymCall framework supports both the Intel VMX and AMD SVM virtualization architectures. The symcalls are designed to resemble the Linux system call interface as closely as possible. The description will focus on the Linux implementation.

Implementing the SymCall interface required modifications to both the Palacios VMM as well as the Linux kernel running as a guest. The scale of the changes is shown in Figure 6.2. The modifications to the guest OS consisted of 211 lines of C and 129 lines of assembly as measured by SLOCcount. This code consisted of the generic SymCall infrastructure and did not include the implementation of any symcall handlers. The VMM

infrastructure consisted of an additional 300 lines of C implemented as a compile time module.

Guest OS support The Linux guest implementation of the symbiotic upcall interface shares much commonality with the system call infrastructure. Symbiotic upcalls are designed to be implemented in much the same manner as a normal system call. Each symbiotic upcall is associated with a given call index number that is used to lookup the appropriate call handler inside a global array. The OS loads the *SYMCALL_RIP* MSR with a pointer to the SymCall handler, which uses the value of the *RAX* General Purpose Register (GPR) as the call number. The arguments to the symcall are supplied in the remaining GPRs, which limits each symbiotic upcall to at most 5 arguments. Our current implementation does not support any form of argument overflow, though there is no inherent reason why this would not be possible. The arguments are passed by value. Return values are passed in the same way, with the error code passed in *RAX* and additional return values in the remaining GPRs. Any kernel component can register a symbiotic upcall in exactly the same way as it would register a system call.

One notable difference between symcalls and normal system calls is the location of the stack during execution. Normal system calls execute on what is known as the kernel mode stack. Every process on the system has its own copy of a kernel mode stack to handle its own system calls and possibly also interrupts. Among other things this allows context switching and kernel preemption, because each execution path running in the kernel is guaranteed to have its own dedicated stack space. This assurance is possible because processes are unable to make multiple simultaneous system calls. Symbiotic upcalls on the other hand can occur at any time, and so cannot use the current process' kernel stack. In our implementation the guest OS allocates a symbiotic stack at initialization. Every symbiotic upcall that is made then begins its execution with *RSP* loaded with the last address of the stack frame. Furthermore it is mandated that symbiotic upcalls cannot nest,

that is the VMM cannot perform a symcall while another symcall is running. This also means that symbiotic upcalls are an independent thread of execution inside the OS. This decision has ramifications that place a number of restrictions on symcall behavior, which I will elaborate on in Section 6.2.3.

VMM support From the VMM perspective symbiotic upcalls are accessed as standard function calls, but are executed inside the guest context. This requires modifications to the standard behavior of a conventional VMM. The modifications to the Palacios VMM required not only additional functionality but also changes and new requirements to the low level guest entry/exit implementation.

As I stated earlier the VMM is responsible for saving and restoring the guest execution state before and after a symbiotic upcall is executed. Only a single instance of the guest state is saved, which means that only one symcall can be active at any given time. This means that symbiotic upcalls cannot nest. Our design does not perform a full checkpoint of the guest state but rather only saves the minimal amount of state needed. This allows symbiotic upcalls some leeway in modifying the current guest context. For example the guest OS is not prevented from modifying the contents of the control registers. In general the saved state corresponds to the state that is overwritten by values specified in the symcall MSRs.

The guest state that is saved by the VMM includes:

- *RIP:* The instruction pointer that the guest was executing before the exit that led to the symbiotic upcall.
- *Flags Register:* The system flags register
- *GPRs:* The full set of available General Purpose registers, including the Stack Pointer (*RSP*) used for argument passing.

- *Code Segment Descriptor/Selector:* The selector and cached descriptor of the code segment
- *Stack Segment Descriptor/Selector:* The selector and cached descriptor of the Stack segment
- *FS and GS Segment Bases:* The base addresses for both the FS and GS segments. These are used by the guest OS to store the address of the local processor data area.
- *CPU Privilege Level:* The AMD virtualization architecture requires the CPU Privilege level be saved as a separate entity, even though it is specified by the lower bits of the CS and SS segment selectors. For simplicity it is saved separately when running on SVM.

Because symbiotic upcalls are executed in guest context the VMM had to be modified to perform a nested VM entry when a symcall is executed. VMM architectures are based on an event model. The VMM executes a guest in a special CPU operating mode until an exceptional event occurs, a special action is taken or an external event occurs. This causes the CPU to perform a VM exit that resumes inside the VMM context at a given instruction address. The VMM is then responsible for determining what caused the exit event and taking the appropriate action. This generally entails either emulating a certain instruction, handling an interrupt, modifying the guest state to address the exception, or servicing a request. This leads most VMMs to be implemented as event-dispatch loops where VM entries are made implicitly. That is a VM entry occurs automatically as part of a loop, and exit handlers do not need to be written to explicitly re-enter the guest.

For symbiotic upcalls I had to make VM entries available as an explicit function while also retaining their implicit nature. To do this I had to make the main event loop as well as the exit handlers re-entrant. Re-entrancy is necessary because it is not only possible

but entirely likely that the guest will generate additional exits in the course of executing a symbiotic upcall. I found that it was fairly straightforward to modify the exit handlers to be re-entrant, however the dispatch function was considerably more complicated.

Implementing re-entrancy centered around ensuring safe access to two global data structures: The guest state structure which contains the state needed by the VMM to operate on a given guest environment and the virtualization control structures that store the hardware representation of the guest context. The guest state needed by the VMM is deserialized and serialized atomically before and after a VM entry/exit. This structure is re-entrant safe because the VMM checkpoints the necessary state before and after a symbiotic call is made. This ensures that the guest will safely be able to re-enter the guest after the symbiotic upcall returns, because the guest state is copied back to the hardware structures before every entry. However it does not store the hardware state containing the exit information. In practice the exit information is small enough to store on the stack and pass as arguments to the dispatch function.

6.2.3 Current restrictions

In our design, symbiotic upcalls are meant to be used for relatively short synchronous state queries. Using symcalls to modify internal guest state is much more complicated and potentially dangerous. Since our current implementation is based on this fairly narrow focus, I made a number of design choices that limit the behavior of the symcall handler in the guest OS. These requirements ensure that only a single symcall will be executed at any given time and it will run to completion with no interruptions, i.e. it will not block.

The reasoning behind restricting the symcall behavior is to allow a simplified implementation as well as a provide behavioral guarantees to the VMM executing a symbiotic upcall. If symbiotic upcalls were permitted to block, the synchronous model would essentially be broken, because a guest OS would be able to defer the upcall's execution

indefinitely. Furthermore it would increase the likelihood that when a symbiotic upcall did return, the original reasons for making the upcall would no longer be valid. This is in contrast to system calls where blocking is a necessary feature that allows the appearance of synchronicity to applications.

In order to ensure this behavior, a symcall handler in the guest OS is not allowed to sleep, invoke the OS scheduler, or take any other action that results in a context switch. Furthermore while the guest is executing a symbiotic upcall the VMM actively prevents the injection of any external interrupts such as those generated by hardware clocks. Our implementation also blocks the injection of hardware exceptions, and mandates that symcall handlers do not take any action that generates a processor exception that must be handled by the guest OS. While this might seem restrictive, I note that, in general, exceptions generated in a kernel code path are considered fatal.

The requirement that symcall handlers not block has further ramifications in how they deal with atomic data structures. This is particularly true because, as I stated earlier, a VMM can execute a symbiotic upcall at any point in the guest's execution. This means that it is possible for a symcall to occur while other kernel code paths are holding locks. This, and the fact that symcalls cannot block, mean that symcalls must be very careful to avoid deadlocks. For instance, if a kernel control path is holding a spinlock while it modifies internal state it can be pre-empted by a symbiotic upcall that tries to read that same state. If the symcall ignores the lock it will end up reading inconsistent state, however if it tries to acquire the spinlock it will deadlock the system. This is because the symcall will never complete which in turn means the process holding the lock will never run because symcalls must run to completion and cannot be interrupted.

In order to avoid deadlock scenarios while still ensuring data integrity, special care must be taken when dealing with protected data structures. Currently our implementation allows symbiotic upcalls to acquire locks, however they cannot wait on that lock if it is

not available. If a symcall attempts to acquire a lock and detects that it is unavailable, it must immediately return an error code similar to the POSIX error *EWOULDBLOCK*. In multiprocessor environments I relax the locking requirements in that symbiotic upcall handlers can wait for a lock as long as it is held by a thread on another CPU.

6.3 SwapBypass example service

I will now show how symcalls make possible optimizations that would otherwise be intractable given existing approaches by examining SwapBypass, a VMM extension designed to bypass the Linux swap subsystem. SwapBypass allows a VMM to give a VM direct access to memory that has been swapped out to disk, without requiring it be swapped in by the guest OS.

SwapBypass uses a modified disk cache that intercepts the I/O operations to a swap disk and caches swapped out pages in the VMM. SwapBypass then leverages a VM's shadow page tables to redirect swapped out guest virtual addresses to the versions in the VMM's cache. SwapBypass uses a single symcall to determine the internal state and permissions of a virtual memory address. The information returned by the symcall is necessary to correctly map the page and would be extremely difficult to gather with existing approaches.

I will now give a brief overview of the Linux swap architecture, and describe the SwapBypass architecture.

6.3.1 Swap operation

The Linux swap subsystem is responsible for reducing memory pressure by moving memory pages out of main memory and onto secondary storage, generally on disk. The swap architecture is only designed to handle pages that are assigned to anonymous memory regions in the process address space, as opposed to memory used for memory mapped

files. The swap architecture consists of a number of components such as the collection of swap disks, the swap cache, and a special page fault handler that is invoked by faults to a swapped out memory page. The swap subsystem is driven by two scenarios: low memory conditions that drive the system to swap out pages, and page faults that force pages to be swapped back into main memory.

Swap storage The components that make up the swap storage architecture include the collection of swap devices as well as the swap cache. The swap devices consist of storage locations that are segmented into an array of page sized storage locations. This allows them to be accessed using a simple index value that specifies the location in the storage array where a given page is located. In Linux this index is called the *Swap Offset*. The swap devices themselves are registered as members of a global array, and are themselves identified by another index value, which Linux calls the *Swap Type*. This means that a tuple consisting of the *Swap Offset* and *Swap Type* is sufficient for determining the storage location for any swapped out page.

As pages are swapped out, the kernel writes them to available swap locations and records their location. As a side effect of swapping out the page, any virtual address that refers to that page is no longer valid and furthermore the physical memory location is most likely being used by something else. To prevent accesses to the old virtual address from operating on incorrect data, Linux marks the page table entries pointing to the swapped out page as not present. This is accomplished by unsetting the *Present* bit in the page table entry (PTE). Because marking a page invalid only requires a single bit, the rest of the page table entry is ignored by the hardware. Linux takes advantage of this fact and stores the swap location tuple into the available PTE bits. I refer to PTEs that are marked not present and store the swap location tuple as *Swapped PTEs*.

As a performance optimization Linux also incorporates a special cache that stores memory pages while they are waiting to be swapped out. Because anonymous memory

is capable of being shared between processes and thus referenced by multiple virtual addresses, Linux must wait until all the PTEs that refer to the page are marked as swapped PTEs before it can safely move the page out of main memory and onto the appropriate swap device. Tracking down all the references to the page and changing them to Swapped PTEs is typically done in the background to minimize the impact swapping has on overall system performance. Thus it is possible for pages to remain resident in the cache for a relatively long period of time, and furthermore it is possible that one set of PTEs will point to a page in the swap cache while another set will be marked as Swapped PTEs. This means that just because a PTE is a Swapped PTE does not mean the page it refers to is actually located at the location indicated by the Swapped PTE. It is important to note that every page in the swap cache has a reserved location on a swap device, this means that every page in the swap cache can be referenced by its Swapped PTE. The swap cache itself is implemented as a special substructure in the kernel's general page cache, this is a complex internal kernel data structure organized as a radix tree.

Swapped page faults As I mentioned earlier Linux marks the page table entries of swapped out pages as invalid and stores the swap location of the page into the remaining bits. This causes any attempted access to a swapped out virtual address to result in a page fault. Linux uses these page faults to determine when to swap pages back into main memory. When a page fault occurs the kernel exception handler checks if the faulting virtual address corresponds to a Swapped PTE. If so, it first checks if the page is resident in the swap cache. If the page is found in the page cache then the handler simply updates the PTE with the physical memory address of the page in the swap cache and indicates that there is a new reference to the page. If the page is not found in the swap cache then the Swapped PTE contains the location of the page in the collection of swap devices. This triggers a swap in event, where the swap subsystem reads the page from the swap device and copies it to an available physical memory location. This operation could itself trigger

additional swap out events in order to make a location in main memory available for the swapped in page. Once the page is copied into main memory it is added to the swap cache, because its possible that other Swapped PTEs reference that page and have not been updated with its new physical address. Once all references have been updated the page is removed from the swap cache.

Finally it should be noted that after a page has been swapped in a copy of the page remains on the swap device. This means that if a process swaps in a page only in order to read it, that the page can simply be deleted from the swap cache without writing it to the swap device. The next time the page is referenced it will simply be copied back into the swap cache. Also note that a page can be swapped in from disk and written to while still in the swap cache, and then swapped out again. In this case the version is the swap cache must be written back to the swap device. This makes it possible for a swapped out page to be desynchronized from its copy on the swap device. This behavior is important and has ramifications for SwapBypass that I will discuss later.

6.3.2 SwapBypass implementation

SwapBypass uses shadow page tables to redirect the swapped PTEs in a guest's page table to pages that are stored in a special cache located in the VMM. This allows a guest application to directly reference memory that has been swapped out to disk by it's OS. An example set of page table hierarchies are shown in Figure 6.3. In this case the guest OS has swapped out 3 pages that are referenced by the current set of page tables. As I described earlier it has marked the Swapped PTEs as not present in order to force a page fault when they are accessed. However, when a VMM is using shadow paging all page faults cause VMExits, which allows a VMM to handle page faults before the guest OS. In many cases the VMM updates its shadow page tables to reflect the guest page tables and continues execution in the guest, other times the VMM must forward the page fault exception to the

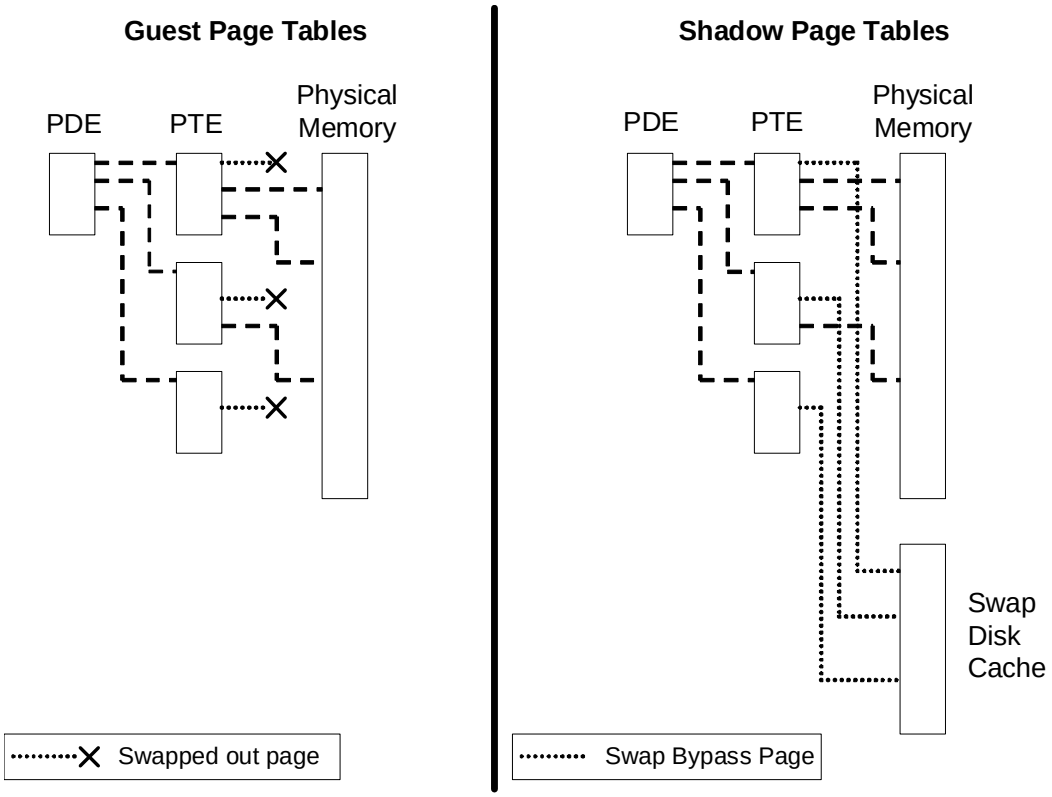


Figure 6.3: The guest and shadow page table configuration needed to provide a guest access to memory it has swapped out. With standard shadow paging a swapped out page triggers a page fault that is injected into the guest environment that causes the page to be swapped in. However, with SwapBypass swapped out pages are remapped by the VMM to point to pages located in the swap disk cache in the VMM's memory space. Essentially, the guest's virtual address space is virtualized to allow the VMM to redirect virtual address accesses.

Component	Lines of Code
Swap disk cache	373(C)
Page fault handler	47(C)
SwapBypass core	182(C)
Guest SymCall functions	53(C)
Total	655(C)

Figure 6.4: Lines of code needed to implement SwapBypass as measured by SLOCcount

VM so it can be handled by the guest OS. Without SwapBypass the VMM would only see that the guest marked its page table entries as invalid, and thus forward the page fault to the guest OS. However when SwapBypass is active it is able to detect that the guest's PTE is in fact a Swapped PTE[2], and set the shadow PTE to point at the page in the cache. SwapBypass uses a special symcall to inspect the internal swap state of the guest Linux kernel as well as to determine the access permissions of the virtual address containing the swapped PTE.

SwapBypass is implemented with several components. A single symcall that returns the state of a guest virtual address, a special swap device cache that intercepts I/O operations to a swap disk, a new edge case that is added to the shadow page fault handler, and the SwapBypass core that provides the interface between the symcall, swap disk cache, and shadow page table hierarchy.

Figure 6.4 shows the implementation complexity of the different components in lines of code as measured by SLOCCount. All together SwapBypass consists of 655 lines of code.

Swap disk cache The first component of SwapBypass is a cache that is located inside the VMM between a guest and its swap disk. This cache intercepts all I/O operations and caches swapped out pages as they are written to disk. As swapped pages are written to disk

[2] A Swapped PTE contains a set of flags to indicate a swapped page without any additional information

they are first inserted into the cache, if the cache is full then victim pages are chosen and flushed to disk according to a Least Recently Used (LRU) policy. When pages are read from disk they are copied from cache if found, otherwise the pages are read directly from disk and not inserted into the cache.

During initialization the swap disk cache registers itself with SwapBypass, and supplies its *Swap Type* identifier as well as a special function that SwapBypass uses to query the cache contents. This function takes as an argument the *Swap Offset* of a page located on disk and returns the physical address of the page if it is present in the cache. In order for the swap disk cache to determine its *Swap Type* identifier I had to modify Linux to add the *Swap Type* to the swap header that is written to the first page entry of every swap device. The swap disk cache intercepts this write and parses the header to determine its *Swap Type*.

The swap disk cache is also responsible for notifying SwapBypass of disk reads which correspond to swap in events. These events drive invalidations that I will discuss in more detail later.

SwapBypass symcall Implementing SwapBypass requires knowledge of the state of a swapped out page and the permissions that the current process has on the virtual address referring to that page. This information is extremely difficult to determine from outside the guest context. Furthermore this information cannot be collected asynchronously. The reason for this is that if the VMM does not immediately modify the shadow page tables to point to a page in the cache then it *must* inject a page fault into the guest. The page fault would then cause the guest to swap the page back into its memory space, and modify the Swapped PTE to point at the new location. By the time the asynchronous upcall completed the reason for calling it would no longer exist. Furthermore, because symcalls are executed synchronously they execute in the process context that existed when the exit leading to the symcall occurred. In the case of SwapBypass this means that the symcall executes as the process that generated the page fault on the swapped PTE. An asynchronous approach

could not provide this guarantee, which would greatly complicate the implementation.

The symcall takes two arguments: a *guest virtual address* and a *Swapped PTE*. The guest virtual address is the virtual address that caused the page fault while the Swapped PTE is the guest PTE for that virtual address. The symcall returns a set of three flags that mirror the permission bits in the hardware PTEs (Present, Read/Write, and User/System). These bits indicate whether the virtual address is valid, whether the page is writable, and finally whether it can be accessed by user processes.

The first action taken by the symcall is to find the task descriptor of the process which generated the page fault. Linux stores the current task descriptor in a per-CPU data area whose location is stored in the FS segment selector. This means that the task descriptor is found by simply calling `get_current()`, because the FS segment is loaded as part of the symcall entry.

Next, the symcall determines if the page referenced by the Swapped PTE is in fact swapped out or if it is present in the kernel's swap cache. As I stated before, Linux does not immediately update all the Swapped PTEs referencing a given page, so it is possible for the PTEs to be out of date. In this case the guest's page fault handler would simply redirect the PTE to the page's location in the swap cache and return. Therefore, if the symcall detects that the page is present in the swap cache, it immediately returns with a value indicating that the page is not present. This will cause SwapBypass to abort and continue normal execution by injecting a page fault into the guest. SwapBypass cannot operate on pages in the swap cache, even if they are available in the SwapBypass cache because of the synchronization issues mentioned earlier.

If the swapped PTE does not refer to a page in the swap cache, then it can be redirected by SwapBypass. In this case it is necessary to determine what access permissions the current process has for the virtual address used to access the page. Linux does not cache the page table access permissions for swapped out pages, so it is necessary to query the

process' virtual memory map. The memory map is stored as a list of virtual memory areas that make up the process' address space. The symcall scans the memory map searching for a virtual memory area that contains the virtual address passed as an argument to the symcall. Once the region is located, it checks if the region is writable and if so sets the writable flag in the return value.

Finally the symcall checks if the virtual address is below the 3GB boundary, and if so sets the user flag in the return value.

Shadow page fault handler Similar to the Linux swap subsystem, SwapBypass is driven by page faults that occur when a guest tries to access a swapped out page. When operating normally, the shadow page fault handler parses the guest page tables in order to create a shadow page table hierarchy. If the shadow handler determines that the guest page tables are invalid, then it simply injects a page fault into the guest.

For SwapBypass to function correctly the shadow page fault handler must be able to detect when a guest page fault was generated by a swapped PTE. This can be determined by simply checking several bits in the swapped PTE. If this check succeeds, then the shadow page fault handler invokes SwapBypass. Otherwise it continues normally and injects a page fault. The important take away here is that the shadow page fault handler can determine if a fault is caused by a swapped PTE by simply checking a couple of bits that are already available to it. This means that there is *essentially no additional overhead* added to the shadow paging system in the normal case.

When the shadow page fault handler invokes SwapBypass it supplies the virtual address and the swapped PTE from the guest page tables. SwapBypass returns to the shadow page fault handler the physical address where the swapped page is located and a set of page permissions. The shadow page fault handler then uses this information to construct a shadow PTE that points to the swapped out page. This allows the guest to continue execution and operate on the swapped out page as if it was resident in the guest's address

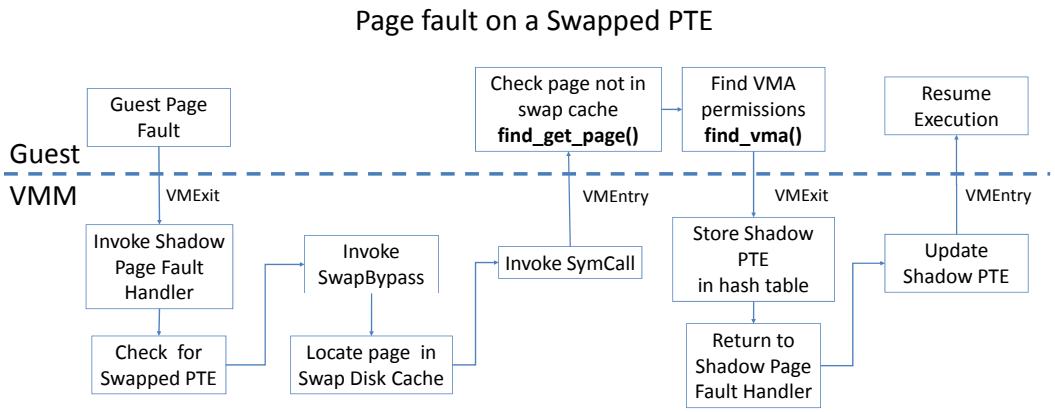


Figure 6.5: The execution path of SwapBypass in response to a guest page fault on a swapped PTE. First SwapBypass checks whether the faulting address corresponds to a swapped out page that is present in the swap disk cache. If so, it invokes the find_get_page() symcall to determine the permissions for the swapped page. Finally, SwapBypass updates the shadow page table to point the swapped out entry to the page stored in the swap disk cache.

space. If the swapped page is unavailable to SwapBypass then the shadow page fault handler falls back to the default operation and injects a page fault into the guest.

SwapBypass core The SwapBypass core interfaces with the swap disk cache, the symcall, and the Shadow page fault handler and tracks the swapped PTEs that have been successfully redirected to the swap disk cache. SwapBypass is driven by two guest events: page faults to swapped PTEs and I/O read operations to the swap disk cache. Page faults create mappings of swapped pages in the shadow page tables, while read operations drive the invalidation of those mappings. The execution path resulting from a page fault is shown in Figure 6.5, and the execution path for disk reads is shown in Figure 6.6.

Page faults When a guest page fault occurs and the shadow page fault handler determines that it was caused by an access to a swapped PTE, SwapBypass is invoked and passed the faulting virtual address and guest PTE. First SwapBypass determines which swap device the swapped PTE refers to and the location of the page on that device. Next,

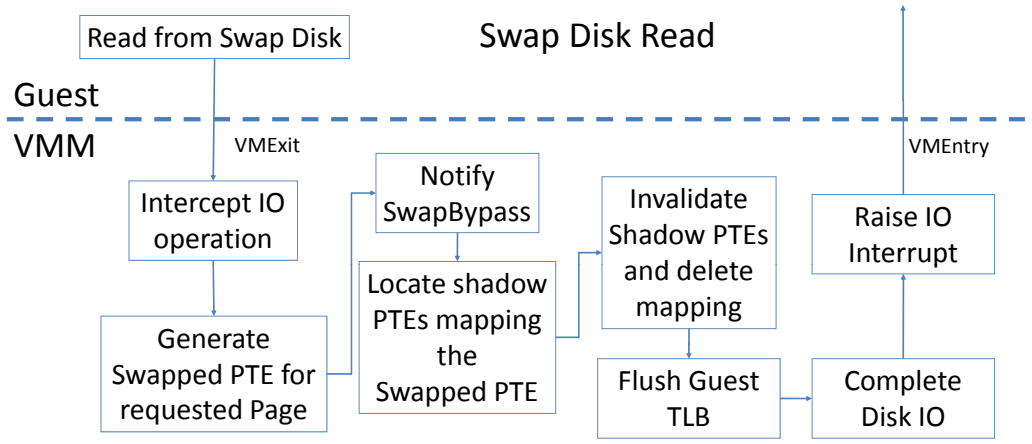


Figure 6.6: The execution path of SwapBypass in response to an I/O read operation to a swap device. First the page being requested from disk is used to generate the possible page table entries to reference it. This entry is then used as a key in a hash table lookup to find all existing SwapBypass page table mappings pointing to the cached page. These entries are all deleted from the shadow page table, the TLB is flushed. and the I/O operation completes.

it queries the swap disk cache to determine if that page is present in the memory cache. If the page is present, SwapBypass makes a symcall into the guest passing in the virtual address and swapped PTE value. The symcall returns whether the swapped page is in fact located on disk, and the permissions of the virtual address.

If the page is present in the swap disk cache and the symcall indicates that the page on disk is valid, then SwapBypass adds the virtual address onto a linked list that is stored in a hash table keyed to the swapped PTE value. This allows SwapBypass to quickly determine all the shadow page table mappings currently active for a swapped page. Finally SwapBypass returns the permissions and physical address of the swapped page to the shadow page fault handler.

Disk reads Read operations from a swap disk result in the guest OS copying a page off the swap device and storing it in the swap cache. When this operation completes the OS

will begin updating the swapped PTEs to reference the page in memory. When this occurs SwapBypass must remove any existing shadow page table entries that reference the page. If the shadow page table entries were not invalidated, then the guest could see two different versions of the same memory page. One version would be in the guest's swap cache and be referenced by any new page table entries created by the guest, while any old swapped PTEs would still only see the version on disk.

When the swap disk cache detects a read operation occurring, it combines it's *Swap Type* with the page index being read to generate the swapped PTE that would be used to reference that page. The swap disk cache then notifies SwapBypass that the page referenced by the swapped PTE has been read. SwapBypass then locates the list of shadow page table mappings for that swapped PTE in the previously mentioned hash table. Each shadow page table entry is invalidated and the swapped PTE is deleted from the hash table. SwapBypass then returns to the swap disk cache which completes the I/O operation.

6.3.3 Alternatives

I believe that SwapBypass is a compelling argument for Symbiotic Virtualization and SymCall in particular. Especially when considered against current alternatives. Consider two other approaches that could be taken based on current techniques. The gray-box/introspection approach would require that the VMM read and parse the internal guest state to determine whether a page was capable of being remapped by SwapBypass. Even if the guest was modified to include the read/write and user/system bits in the swapped PTE format, the VMM would still have to access the swap cache directly. This would be a very complex procedure that would need to locate and access a number of nested data structures.

The second approach would be to use the current upcall implementations that are based on hardware interrupts and guest device drivers. This approach has two problems: inter-

Latency for echo() SymCall			
First (“cold”)	5 VMExits	63455 cycles	35 μs
Next (“warm”)	0 VMExits	15771 cycles	9 μs

Figure 6.7: SymCall latency for a simple echo() symcall. The first call takes longer because it generates 5 shadow page faults as a result of its execution. The second call is shorter because the pages are already mapped in, and so it does not generate any nested exits.

rupts are asynchronous by nature and Linux uses a return from an interrupt handler as an opportunity to reschedule the current task. The asynchronous issue could be dealt with in the VMM by first ensuring that the guest context was configured to immediately handle the interrupt if it was injected, however this would be complex and might result in some upcalls being aborted. However it would also require changes to the Linux interrupt handling architecture to forbid context switches for certain interrupt classes.

Finally, a simple disk cache might be used instead of SwapBypass to speed up accesses to swapped pages. While this does benefit performance, SwapBypass is capable of completely eliminating the overhead of the swap system in the Linux kernel. As our evaluation shows, this dramatically improves performance, even over an ideal swap device with no I/O penalty.

6.4 Evaluation

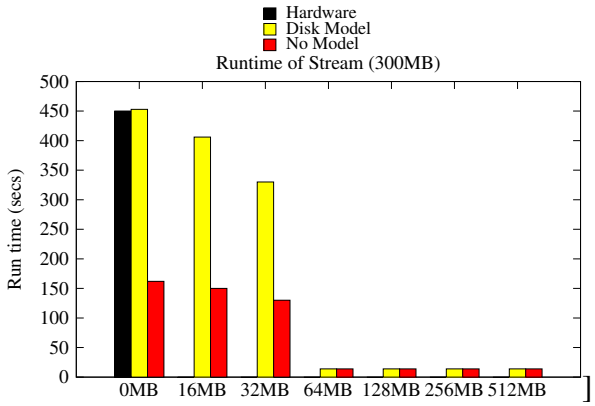
I evaluated both the performance of the SymCall implementation as well as the implementation of SwapBypass. These tests were run on a Dell SC-440 server with a 1.8GHz Intel Core 2 Duo Processor and 4GB of RAM. The guest OS implementation was based on Linux 2.6.30.4.

6.4.1 SymCall latency

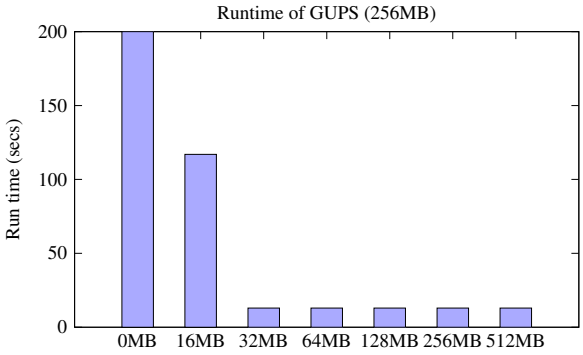
The first test measured the latency in making a symcall. For this test I implemented an `echo()` symcall, that simply returned the arguments as return values. First I measured the latency of a symcall made for the first time. When a symcall is first executed, or "cold", it will access a number of locations in kernel memory that are not present in the shadow page tables. The guest will generate shadow page faults until all the memory locations are accessible. For a simple symcall with no external references this requires 5 shadow page faults. I also ran a second test of a symcall after its memory regions have been added to the shadow page tables. In this "warm" case the symcall generated no exits. The results shown in Figure 6.7 are an average of 10 test calls. The latency for a "cold" symcall is 64 thousand CPU cycles, which on our test machine equates to around 35 microseconds. The "warm" symcall completed in $\sim$16 thousand cycles or 9 microseconds.

6.4.2 SwapBypass performance

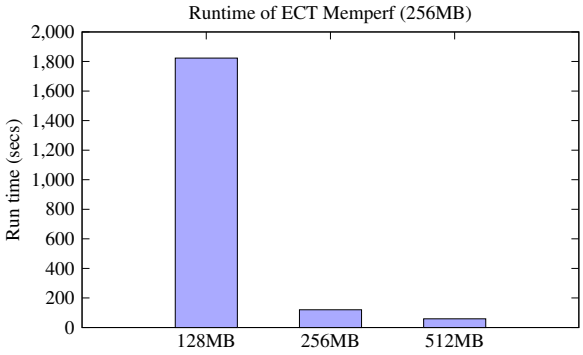
I have evaluated the effectiveness of SwapBypass using a set of memory benchmarks that operate on anonymous memory regions. These benchmarks include the microbenchmarks Stream [62] (small vector kernel) configured to use 300MB of memory and GUPS [72] (random access) configured to use 256MB of memory. Stream and GUPS are part of the HPC Challenge benchmark suite. I also used the ECT memperf benchmark [89] configured to use 256MB of memory. ECT memperf is designed to characterize a memory system as a function of working set size, and spatial and temporal locality. Each benchmark was run in a guest configured with 256MB of memory and a 512MB swap disk combined with a swap disk cache in the Palacios VMM. The performance of each benchmark was measured as a function of the size of the swap disk cache. The benchmarks were timed using an external time source.



(a) (a) Stream (300MB) Performance

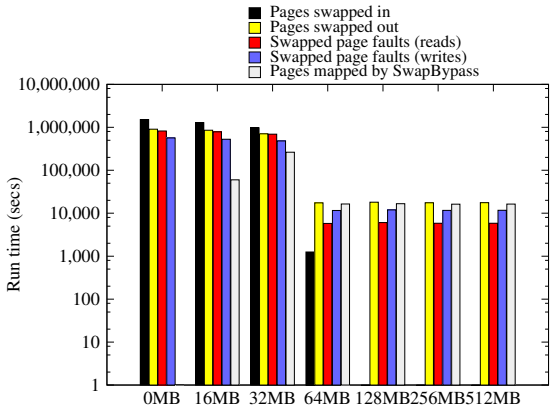


(b) (b) GUPS (256MB) Performance

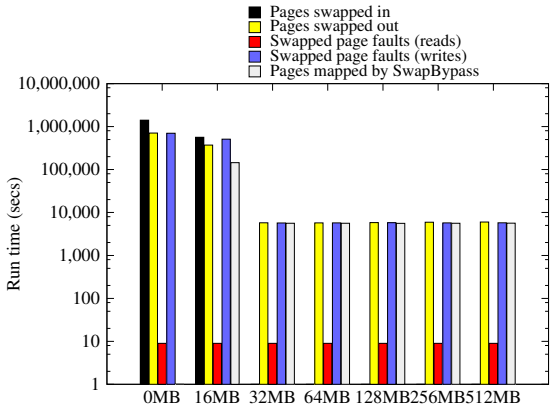


(c) (c) Memperf (256) Performance

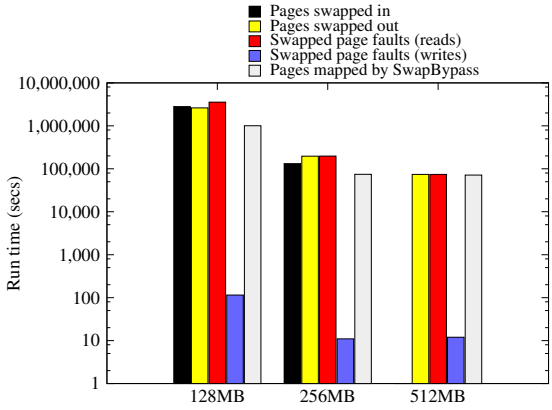
Figure 6.8: Performance results for Stream, GUPS, and ECT Memperf benchmarks. The benchmark runtime was measured for variable sized swap disk caches.



(a) (a) Stream (300MB) Swap Statistics



(b) (b) GUPS (256MB) Swap Statistics



(c) (c) Memperf (256) Swap Statistics

Figure 6.9: Hardware event statistics for Stream, GUPS, and ECT Memperf benchmarks. Events were counted for each benchmark run using variable sized swap disk caches.

For the Stream benchmark I ran tests using a hardware swap disk, a virtual swap disk implemented using a simple disk performance model, and a pure RAM disk implemented in VMM memory. The hardware disk was a 7200RPM SATA disk partitioned with a 512MB swap partition. Our harddisk model used a simplistic average seek delay based on the hardware specifications of the SATA disk. For the other benchmarks I only used the RAM disk without a disk model. Our reason for concentrating our evaluation on a RAM-based swap disk is to generate ideal I/O conditions for the Linux swap system. With the RAM-based swap disk all disk I/O is eliminated and data is transferred at the speed of system memory. This means that the Linux swap architecture is the *sole* source of overhead, assuring that any performance benefits gained by SwapBypass are *not* simply the result of implementing a disk cache in RAM. Our inclusion of the hardware swap disk and disk model evaluations is to help illustrate the effects of SwapBypass with non-ideal I/O.

Stream Our initial benchmark is Stream, configured to use a 300MB region of memory. Figure 6.8(a) shows the runtime of the Stream benchmark for exponentially increasing swap disk cache sizes. I first ran the benchmark using the hardware swap disk as well as with SwapBypass configured with no swap disk cache. Without a cache, SwapBypass flushes all swapped pages to disk and so cannot access them, meaning that SwapBypass will have no beneficial effect on the performance. As the figures show both the hardware and disk model configurations performed comparably with runtimes of around 450 seconds or 7.5 minutes. The configuration using the RAM disk swap device completed in only around 150 seconds or 2.5 minutes due to the lack of disk I/O.

I then began to scale up the size of the swap disk cache exponentially to determine the impact SwapBypass would have on performance. As the cache size increases the runtime begins to decrease until the combined size of the cache and the VM's physical memory partition exceeds the working set size of the benchmark. As soon as this threshold is

reached the runtime drops off dramatically and the performance of both disk model and RAM disk configurations are essentially identical at 14 seconds. At this point and beyond, SwapBypass is able to satisfy every swapped page fault by mapping the shadow page tables to the page in the swap disk cache—the Linux swap system is completely bypassed and is essentially cut out of the guest's execution path.

The effectiveness of SwapBypass at bypassing the swap system is demonstrated in Figure 6.9(a), which provides a hardware level view of Linux swap system. For each benchmark run I collected the number of pages transferred to and from the swap device (*Pages swapped in* and *Pages swapped out*), the number of page faults generated by swapped out pages (*Swapped page faults (reads)* and *Swapped page faults (writes)*), and also the number of pages that SwapBypass was able to map into the guest from the swap disk cache (*Pages mapped by SwapBypass*). As the swap disk cache size initially increases the number of page faults and swap I/O operations does not change much but the number of pages mapped by SwapBypass increases substantially. However, when the cache size plus the guest memory partition size reaches the benchmark's working set size, all the measurements decrease dramatically. Also, the number of pages swapped in by the guest OS goes to 0.

GUPS GUPS exhibits behavior similar to that of Stream. The GUPS results are shown in Figures 6.8(b) & 6.9(b)

ECT Memperf ECT Memperf results are shown in Figures 6.8(c) & 6.9(c). The memperf results are limited to swap disk cache sizes of 128MB and greater because the execution time for lower cache sizes was too large to measure. The execution time for the 128MB cache size was around 1800 seconds or 30 minutes, and the test run for the 64MB cache size was terminated after 6 hours.

Summary of results A summary of the speedups that SwapBypass can provide for the different benchmarks is shown in Figure 6.10. The reason for the dramatic increase in

Benchmark	speedup
Stream (No model)	11.5
Stream (disk model)	32.4
GUPS	15.4
ECT Memperf	30.9

Figure 6.10: Performance speedup factors of SwapBypass. The speedup of ECT memperf is measured over the 128MB swap disk cache configuration

performance once the working set size threshold is reached is due to a compounding factor. When an OS is in a low memory situation, swapping in a page necessitates swapping another page out, which will need to be swapped in at a later time, which will in turn force a page to be swapped out, and so on. Therefore when SwapBypass is able to avoid a swap in operation, it is also avoiding a swap out that would be needed to make memory available for the swapped in page. SwapBypass is therefore able to prevent the guest from trashing, which unsurprisingly improves performance dramatically.

Our results show that it is possible to artificially and transparently expand a guest's physical memory space using a VMM service. Furthermore, the availability of a symbiotic interface makes implementing this feature relatively easy, while existing approaches would require deep guest introspection or substantial modifications to the guest OS. Being able to easily implement SwapBypass suggests that there are other extensions and optimizations that could be built using symbiotic interfaces.

6.5 Conclusion

In this chapter I presented the design and implementation of the SymCall functional up-call interface. This framework was implemented in the Palacios VMM and a Linux guest kernel. Using the symbiotic interfaces I implemented SwapBypass, a new method of decreasing memory pressure on a guest OS. Furthermore, I showed how SwapBypass is only

possible when using a symbiotic interface. Finally, I evaluated SwapBypass showing it improved swap performance by avoiding thrashing scenarios resulting in 11–32x benchmark speedups.

While SymCall allows a VMM to extract detailed state information from a guest environment, it is limited in only being able to use queries that have been explicitly implemented and made available by the guest OS. If the guest OS does not implement a symcall that performs an operation needed by the VMM, then the VMM has not other recourse for performing that operation or obtaining the information. In order to provide a VMM with the utmost flexibility in interacting with a guest environment, a mechanism is required that allows a VMM to execute arbitrary code inside the guest context. The next chapter will examine symbiotic modules, a framework that allows just that. A symbiotic modules allows a VMM to inject blocks of code that run in the guest's context and are able to interface with a guest OS using the guest's own internal API.

Chapter 7

Symbiotic Modules

Symbiotic virtualization as described to this point depends on the guest environment making its context available to the VMM explicitly. Both SymSpy and SymCall require that the guest OS implement a set of explicit interfaces that the VMM can use to access internal state in the guest environment. While I have demonstrated the usefulness of this approach, it does retain a shortcoming in that the interfaces are static and specific. A VMM is completely dependent on a guest OS providing a specific piece of state information or a specific symcall. If the VMM needs information not provided by the guest OS, or needs a different symbiotic interface, the VMM has no other recourse. What is needed is a method in which a VMM can interact arbitrarily with the guest context, and create new symbiotic interfaces and communication channels dynamically.

I have developed symbiotic modules (SymMod) as a solution to this problem. SymMod is a mechanism that allows a VMM to run arbitrary blocks of code inside the guest context. These code blocks are injected into the guest context as modules and are able to implement special device drivers, security scanners, performance monitors, or any other functionality that is needed by the VMM. In essence these modules can be thought of as a special kind of loadable kernel module in Linux or a driver in Windows. I will explore three different types of symbiotic modules that interface with the running guest context in different ways.

The important thing to note is that symbiotic modules vastly minimize the semantic gap because they actually operate inside the guest context instead of through a generic external interface. Because of their location inside the guest context, a guest OS can access the module's functionality with negligible overhead, since the VM does not need to trap into the VMM whenever the module is invoked.

In this chapter I will describe three different types of symbiotic modules that each have different behaviors and uses:

- **OS specific drivers** – Standard OS drivers that interface with an OS in the same manner as legacy drivers
- **Symbiotic Modules** – Specialized modules that are protected from the guest environment but accessible by it
- **Secure Symbiotic Modules** – Fully secure modules that cannot be detected or accessed by the guest environment, and can only be invoked by the VMM

Each of these module types can be used to extend VMM and guest OS functionality in different ways and for different purposes. I have implemented SymMod, a symbiotic module framework in the Palacios VMM and a Linux guest OS, which allows dynamic injection of each of class of symbiotic module. The rest of this chapter will describe SymMod.

7.1 Motivation

Operating systems have long implemented support for extensibility and modularity as a means of managing the complexity of diverse hardware environments. These mechanisms allow new functional components to be added to an OS at runtime and are most commonly used for device drivers. This modularity allows an OS to dynamically reconfigure itself to

support new hardware and optimize its behavior for specific workloads and environments. This is as true for virtualized environments as it is for those running directly on hardware.

Device drivers are of particular importance because, virtualized I/O is currently the predominant performance bottleneck for virtualized architectures. Thus the mechanisms used to communicate with devices are a particularly important component of the virtualized environment, as their performance has a very large impact on overall system performance. These mechanisms include a combination of device driver behavior as well as the interfaces exposed by the virtual devices themselves. Virtualized devices are currently the greatest source of tension when choosing between compatibility and performance. While emulated virtual devices based on common physical hardware provide a large degree of compatibility for existing operating systems, the mechanisms used to emulate the devices results in substantial performance penalties.

The central problem with using emulated hardware devices for virtual I/O is the overhead associated with device interactions. Emulation is driven by traps into the VMM anytime the device driver interacts with the emulated device. These interactions take the form of either an x86 *in*/*out* instruction to a device's IO port or a memory operation to a address range that is mapped by the device. The VMM configures itself to exit into the VMM whenever one of these operations occur, thus resulting in a VM exit and entry cycle for every operation. As stated earlier, VM entries and exits are extremely expensive hardware operations, much more expensive than the actual native device I/O operations. However, due to the fact that these operations are relatively cheap on real hardware, existing device drivers have been implemented to perform these operations frequently. As a result many of these drivers generate large overheads in a virtual machine environment, often for non-critical operations.

The fact that legacy device I/O cannot be done without imposing significant overhead in a virtual machine has inspired the adoption of paravirtual I/O interfaces. Paravirtual

devices expunge and consolidate device operations to create a streamlined interface that minimizes the overhead imposed by VM entries and exits. While paravirtual I/O does improve I/O performance, it requires all virtualized OSes to implement special device drivers designed specifically for the paravirtual devices. This means that every OS would need to implement a set of device drivers for every VMM architecture that it runs on. This has been widely adopted by both VMMs and operating systems, as the possible combinations of VMMs and OSes are relatively small. OSes that use these devices incorporate them in such a way that paravirtual devices appear and behave exactly the same as traditional hardware devices. This allows the OS architecture to stay the same, and for paravirtual device drivers to use the same kernel interfaces as existing hardware drivers.

While this approach has been very effective in enabling OSes to use paravirtual devices, it inherits a number of legacy design decisions that impose a set of restrictions when running in a virtualized context. This is due to the fact that OSes have traditionally been designed such that they always run on static hardware environments with a set of devices that are permanent and known a priori. Furthermore, it has been assumed that devices themselves expose extremely stable interfaces due to the fact that any changes to the interface requires changes to the actual device hardware. However, with virtual environments these assumptions are no longer valid. First, because virtual devices are implemented in software, the interfaces can be changed quickly and frequently. Second, because VMs can be moved between different VMM environments, the available devices given to a VMM can in fact change dynamically as the OS is running.

Most virtual hardware is used as a surrogate for existing hardware device types, such as network and block devices. Therefore, one approach to the issues mentioned above is to create a standard set of I/O interfaces for common device types that can be widely adopted across both VMM and OS architectures. Linux implements such an interface called VirtIO [82]. While a standardized interface does provide a common architecture that

can be targeted by both operating systems and VMMs, it prevents any improvements to the interface or other diverse interfaces designed for more specific environments. Therefore, standardization does improve the situation, but it does not provide a complete solution to the problems discussed.

It should also be noted that virtual devices can include more exotic functionality than standard hardware. For instance it is common practice for VMMs to include a special set of *guest tools* or special device drivers that provide optimized I/O interfaces as well as advanced functionality to provide better integration with the underlying virtualization layer. One such example of these special drivers is the *balloon driver* [15]. This is a special device that allocates unused guest memory to compact a VM's memory footprint. This provides optimized behavior for VM migration as well as increased consolidation on shared hardware. Other specialized drivers allow for increased integration with a GUI [106, 70]. As new special devices are developed it will be necessary to continuously develop standardized interfaces for them.

7.2 Symbiotic Modules

The essential problem is that with current architectures OS users are required to include special device drivers for every virtual hardware configuration that the OS could conceivably run on. This creates a need for profligate inclusion of device drivers for all possible hardware environments. Loading these drivers either requires user intervention whenever a VM is run on a different VMM, or it requires the inclusion and preconfiguration of all the possible device drivers. The problem becomes even more prevalent when considering passthrough device access. With the introduction of self-virtualized [78] and SR-IOV [18] devices that provide hardware support for passthrough operation, the number of drivers needed to be included will grow to unmanageable sizes. In this environment, it is both

impractical and inefficient to require OSes to include full support for all the possible hardware combinations they will encounter. What is needed instead is a mechanism whereby a device driver for a specific piece of hardware can be loaded into a VM from an external source. Instead of requiring every VM to have prescient knowledge of the hardware environments it will run on, the guest OS should allow a VMM to provide the necessary drivers needed to interact with each environment. To achieve this goal I have developed *symbiotic modules*.

Symbiotic modules provide greater flexibility for virtualizing devices. For instance, with symbiotic modules a VMM can load specialized drivers for a given piece of hardware depending on factors outside a VM's realm of knowledge. As an example, a specialized device could support multiple modes of operation depending on whether it is being shared between multiple VMs or dedicated to a single VM. Depending on how the VMM has allocated the device, it can provide the OS a specialized driver that is optimized based on the current conditions. Furthermore, a VMM would be capable of changing the actual hardware a VM is interfacing with. An example of this would be a VM that is migrated to a remote site while still needing to retain its old IP address. In this case the VMM could replace a passthrough network card driver with one that uses an overlay network to route packets to and from the original home network. This would allow a VM to continue executing without being forced to reconfigure itself based on actions taken in layers beneath it.

In a wider sense symbiotic modules are a means of creating symbiotic communication channels that do not already exist. Whereas the symbiotic interfaces I have already described implement explicit interfaces exported by a guest OS, symbiotic modules allow a VMM to extend a guest OS to support any symbiotic interface that it currently lacks. Symbiotic modules also allow a VMM to extend the functionality of a guest OS in an almost arbitrary manner. OS modules are implemented based on very high level semantic inter-

faces built into the core of the OS. These interfaces allow drivers and modules to interact with many different components of the OS kernel in much the same way as standard kernel code. This direct access to the internal kernel API allows modules to interact with many kernel subsystems, and to extend the OS functionality to a very large degree.

One drawback to this approach is that a VMM needs to incorporate modules that are specific to the OSes intended to run them. Linux modules cannot be loaded into Windows, and vice versa. However, there is no reason why a standard module format and interface cannot be developed. Both Linux and windows have invested large amounts of time and effort in maintaining a standard user space API for applications. It is not infeasible that such an API could be created to support externally loaded modules.

It might seem that this argument for a standard module interface is contradictory, given that I have previously stated that standard interfaces are too stifling to be widely adopted for virtual hardware. However, there is an important distinction between the interface that crosses the VMM/guest boundary and an interface implemented inside a guest kernel. Device interfaces that cross the VMM/guest boundary are necessarily semantically poor, due to the fact that interface operations are prohibitively expensive. A complex and semantically rich interface will quickly become unusable due to the overhead of sending that information across the VMM/guest boundary. In other words, the semantic gap for virtual hardware interfaces is necessarily very large. However, if a standard interface is implemented inside a guest kernel context, it can be much more complex. The reason for this is that the increased complexity does not impose additional performance penalties resulting from VMM/guest context switches. This leads to a small semantic gap. By creating a very rich standardized internal interface, VMMs could still implement highly specialized drivers and modules while not losing too much generality.

7.3 Symbiotic device drivers

As I explained earlier, allowing a VMM to dynamically replace a guest's device drivers enables a number of possibilities that until now were not feasible. For one, it allows a VMM to give a guest direct hardware access in a manner that is secure. Furthermore, it allows a VMM to dynamically switch the underlying hardware from passthrough to virtual without interrupting the guest. This mechanism can be used to enable secure passthrough devices, secure passthrough device sharing between multiple guests, migration of VMs using passthrough devices, and access to devices for which the guest lacked an appropriate driver.

7.3.1 Current Kernel Module Architectures

This dissertation considers symbiotic modules in the context of Linux, and so it is necessary to first describes the Linux kernel module architecture. Linux allows a large degree of extensibility via dynamically loaded modules, and as such exports a common API and loading mechanism. All kernel modules have a common binary format that enables the running kernel to parse and link them into the active OS. This format consists of a slightly modified ELF relocatable object file [101] with a few special sections.

Figure 7.1 demonstrates the loading procedure for a standard Linux kernel module. When the module is loaded from user space, the object file is first copied from user space into the kernel's address space and unpacked into available memory regions by the module loader. This module load occurs as either an explicit user space operation or from a specialized probe operation instigated by the kernel. The module itself is located on some storage medium that is then read by the kernel. After the module is unpacked the kernel's module linker then begins to link the relocatable symbols from the unpacked module to exported internal kernel symbols. These kernel linkage symbols are themselves included

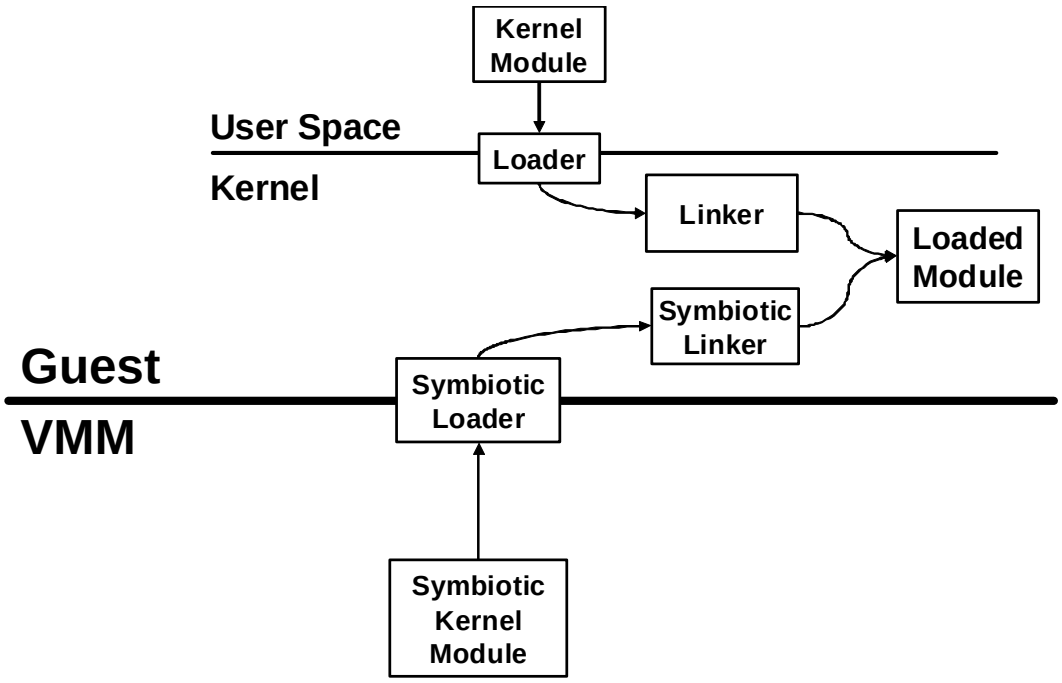


Figure 7.1: Symbiotic device driver architecture

in a special kernel section that contains all of the available exported functions and variables that make up the module API. During this process the kernel module is also capable of exporting its own symbols that future modules can link against.

Each module has a small number of special symbols that refer to functions and variables that are used as part of the linking process. The most important symbols are the initialization and deinitialization functions. After a module has been fully linked into the kernel the loading mechanism explicitly invokes the module's initialization function to activate the module. This function is responsible for registering the module's functionality with the internal kernel APIs. For instance a device driver would register itself with the device layer and advertise the hardware devices it is meant to interface with. Because the module's initialization function is always called and has complete access to the kernel's memory space, the module is essentially unbounded in what it can do.

7.3.2 Guest Architecture

The most basic form of a symbiotic module is a standard kernel module that uses a specialized loading mechanism. Instead of being loaded from user space, a module is instead copied from the VMM's context into the guest context. This allows the VMM to maintain a collection of kernel modules that can be dynamically loaded into any Linux kernel that is running inside a VM. This loading mechanism is based on a new virtual device (based on the VirtIO interface) that is dedicated to loading symbiotic modules.

This device supports several modes of operation. First a guest OS can request a specific module from the VMM. In this scenario a guest OS might detect that a new piece of hardware has become available, and does not have access to the appropriate driver. In response to this, the guest kernel can query the underlying VMM if the necessary driver is available. If so, the guest kernel begins the loading process as shown in Figure 7.1. The symbiotic loading process is very similar to the operations performed for user space modules, with the exception that the module is loaded from the specialized device instead of from user space. In the second case, a VMM can explicitly load a module into a running guest, even if the guest OS has not requested it. In this case the VMM notifies the guest OS that a module is about to be loaded, at which time the guest kernel allocates the necessary space and returns the load address back to the VMM in the form of a set of DMA descriptors. Once the destination has been received by the VMM, the module is copied into the memory referenced by the DMA descriptors and the module load process follows the previous case.

7.3.3 VMM Architecture

Implementing symbiotic modules in Palacios required not just a specialized module loading device but also a framework for including, accessing, and tracking the symbiotic mod-

ules inside the VMM context. In order to manage the collection of symbiotic modules, Palacios adds an additional layer of encapsulation to the already encapsulated module objects. This encapsulation allows kernel modules to be linked into Palacios' library image as black box binary objects. The encapsulation layer added by Palacios includes metadata about the module such as the target OS, the target architecture (i386 or x86_64), and the module's type, name and size. These symbiotic modules are created during the build process and then placed in a special Palacios build directory. During compilation the build process scans this directory and links in any object files that are found. This allows any symbiotic module to be loaded into Palacios by simply doing a file system copy of the module file into the Palacios build tree.

After the linking stage of Palacios' build process, the symbiotic modules are located in a special ELF section of the Palacios Library. When Palacios is initialized by the host OS, this section is scanned and the encapsulated metadata is parsed. The module object and associated metadata are then stored in an internal hash table which is queried whenever a module load request is made.

7.4 General Symbiotic Modules

While basic symbiotic device drivers offer a basic mechanism that can be used to extend existing Linux kernels, they only scratch the surface of what can be achieved with a fully featured symbiotic module architecture. The existing kernel module framework is still designed around the legacy assumptions about the environment the OS will be running in. One major issue with this is that the Linux development process does not consider the internal kernel API a standard interface that will stay constant across different versions. This means that, to guarantee correctness, every kernel module must explicitly target a specific kernel version. This means that the VMM must not only be able to determine the

version of the running kernel, but it also must store different device driver implementations for every possible kernel version that will be running. This inverts the initial problem of requiring systems to track an ever larger collection of device drivers for every possible scenario, and shifts the responsibility from the guest OS to the VMM. To fully address this problem a new API is needed; one that is guaranteed to remain semantically stable for different versions of the same kernel.

Providing a stable user space API is one of the basic guidelines that OS developers try to follow. It is widely known that successful and widely deployed operating systems place a very strong emphasis on ensuring that any internal changes to the kernel do not cause an application to stop working. In effect the user space API is regarded as a permanent fixture, that is guaranteed to always stay the same. Therefore, an application written for a given OS version, will always work for any future versions of the kernel. This consistency, however, does not propagate to internal OS interfaces. This creates a problem, because, as stated above, those are the interfaces that a symbiotic module would use. In order for symbiotic modules to be widely usable there needs to exist a standard internal module API which remains consistent across different OS versions.

The problem with module interface consistency exists because the internal API is in effect not an official interface at all. Modules are actually linking directly to the internal kernel data structures and functions, which must be able to change in order to support new OS features. In order to successfully provide a stable and consistent module API, an actual interface must be constructed that serves as a wrapper around the internal kernel structure. This interface must be considered to be as formal as the user space API, which will guarantee that any module written for it is guaranteed to work indefinitely. It should be noted that this will necessarily create a new semantic gap between the internal kernel state and the semantic information exposed via the new API. However this semantic gap will be much smaller than existing interfaces, due to its implementation inside the kernel,

which will make accessing the interface an inexpensive operation.

One important point to keep in mind is that this new API is not a simple set of wrapper functions that handle data marshaling between internal kernel structures and the interface specifications. This API is in fact a fully symbiotic interface, in other words it is designed fully in keeping with the goals of symbiotic virtualization enumerated in Chapter 4. As a symbiotic interface, the API is specifically geared towards the assumption that the OS is running in a virtual environment. As a result this would include a re-architecting of the OS in order to expose a feature set that would be usable by a VMM.

As an example of one such interface, consider the earlier cases of hardware devices and the associated device drivers. As mentioned previously, migration poses a significant problem for device access in a virtualized OS. This problem comes to the fore even in a relatively simple case of migrating VMs between hosts located on the same LAN but with different passthrough network devices. A fundamental question is what *should* happen in this case, and what such a scenario would look like to a guest OS. Currently, the only way to handle this is to disable the network interface from the original host and reconfigure a new network interface with a new driver once the guest has started running at the destination host. In current OS architectures this will result in a disconnection of any open network connections, which leads to the breakage of the desired virtualization abstraction. The desired behavior would be a transparent switch to a new hardware device while leaving the logical network interface intact. Supporting this behavior is not possible with the current module framework in Linux, however with the addition of a symbiotic module interface this becomes fairly straightforward.

In essence the device driver is split between a hardware interface component and an OS interface component. Using this separation, the hardware interface layer can be swapped in a manner that is transparent to the OS, while the OS state pertaining to a specific network interface remains intact. In this case the symbiotic interface corresponds to the interface

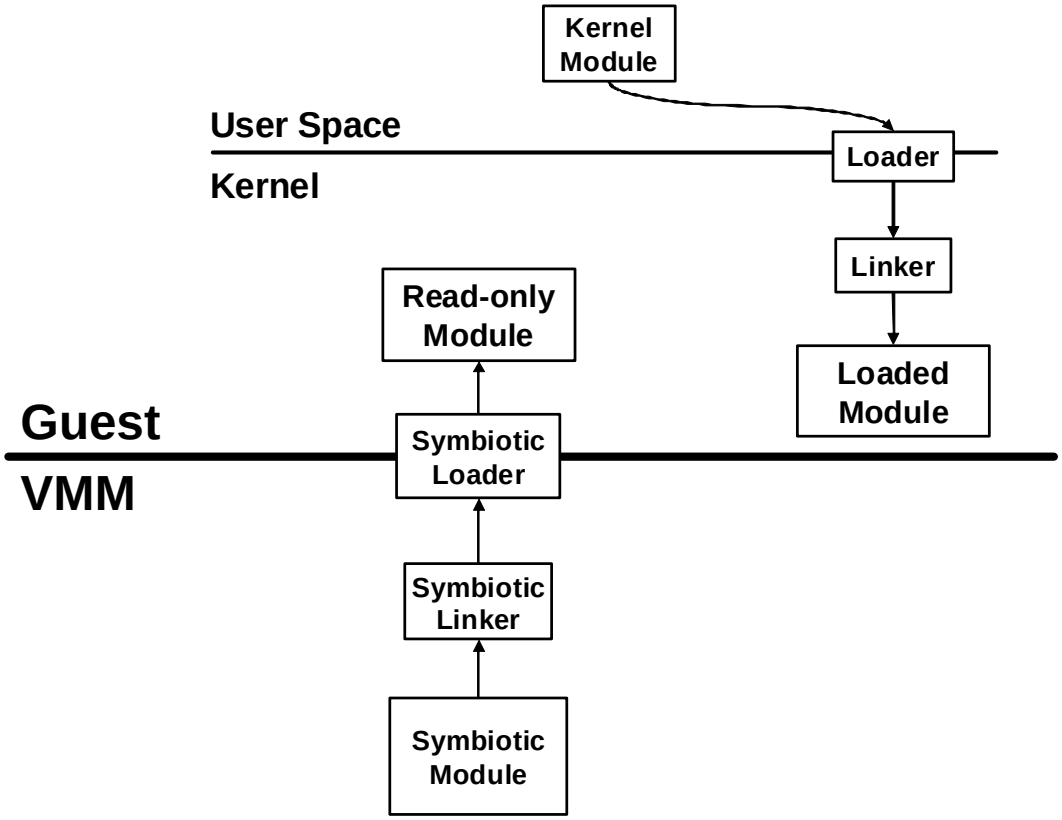


Figure 7.2: Basic symbiotic module architecture

separating the hardware and OS components of the driver. As long as this is a common API, specialized hardware drivers can be loaded into the OS by the VMM, and connected to the active high level network interface. In order to demonstrate the feasibility of using these types of symbiotic modules, I have incorporated this extended feature set into the architecture for symbiotic device drivers explained earlier.

7.4.1 Architecture and Operation

I have implemented a general symbiotic module framework inside both Palacios and Linux as an extension to the symbiotic device driver framework. The extensions include a new symbiotic module API inside the Linux kernel comprising a general purpose symbiotic

module interface. This framework also supports a constrained protection model in which the module itself is protected from being modified by the guest environment. This is achieved by loading each module into a read only memory region which a guest OS can read and execute but not modify. To support this protection mode, I have also added a new linkage mechanism that relies on the VMM itself to link the module into a running kernel. The architecture of this new framework is illustrated in Figure 7.2.

The key feature that enables symbiotic modules is the inclusion of a new symbiotic API into a guest OS. This interface must include internal OS functions and data structures while being general enough to remain consistent as the rest of the OS changes. This requires that the interface remain separate from the internal kernel API, while still providing proximal functionality. The symbiotic interface I have developed is similar to, but separate from, the current module API implemented in Linux. The interface itself consists of a set of common data structures, global variables, and accessible functions with standardized calling formats. The variables and functions are explicitly exported as interface components inside the kernel. As a result of being exported, the link location for each is stored in a special section of the kernel's binary image. During the kernel's initialization process, the section's location is advertised to the VMM by way of the SymSpy interface.

Publishing the API symbol table to the VMM is what allows the linking process to be offloaded into the VMM itself. The symbiotic modules are compiled into an intermediate ELF format, containing special sections denoting unresolved external symbols. The VMM is then able to perform the final linkage by updating the unpacked module object with the internal kernel locations of the exported API symbols. The end result is a block of code that can be loaded and run inside the guest context, with direct access to the symbiotic API supported by the guest kernel.

Up to now, I've explained how it is possible to load an executable module to a state where it is executable inside the guest environment. However, such an approach is not

obviously superior to the existing linking infrastructure that is already implemented by the Linux kernel. The reason for delegating the linking phase to the VMM is to support protection mechanisms. Recall that a symbiotic module is protected from any writes originating from the guest OS, this precludes the guest OS from performing the linking steps due to the requirement that it modify the module itself to update the locations corresponding to external symbols. This leaves only the VMM with the ability to perform the necessary updates to the module image to handle symbol resolution.

While linking is taken care of by the VMM, there are still a small number of requirements placed on the guest OS to support this procedure. Besides exporting a symbiotic module API, the guest OS needs to reserve virtual memory for the module itself to be loaded, as well as ensuring that the module is properly initialized. In my implementation of SymMod, the guest OS is responsible for dynamically allocating a physically contiguous region of guest physical memory as well as an associated region of virtual memory in the kernel address space. When a module is loaded, the VMM notifies the guest OS of a pending module injection and the amount of memory needed to contain it. The guest OS allocates this memory, and returns the virtual address back to the VMM. The VMM then copies the module data into the allocated memory region and performs the linking process. Once the process is complete the VMM notifies the guest OS that the module has been loaded. The fact that a special contiguous physical memory region has been allocated by the guest allows the VMM to use page table protection mechanisms to ensure that nothing in the guest context will be able to modify any contents of the module's memory region. This is accomplished by updating the memory map associated with the given VM, which ensures that the page tables will always disallow any write operations occurring from the guest context. Finally, once the guest receives notification that the load has completed, it activates the module by calling a special initialization function specified by the VMM. This initialization process then registers the module with the internal kernel services.

It should be noted that while the module is protected from being modified by the guest OS, there is no mechanism or policy in place to prevent other types of misuse. For instance the guest OS is still capable of jumping to any address inside the module and executing from there. I have chosen not to address this issue in the current version, but note that there are possible approaches that can be taken. Because a VMM can fully virtualize the CPU, it is possible for the VMM to put in place certain safeguards that would disallow the OS from jumping to arbitrary code locations in the module. For instance, if an OS was designed to operate at a lower privilege level, then the VMM could create virtual call gates. This would fully restrict the guest OS from entering into the module except at predefined locations. Another approach could rely on page faults generated by instruction fetches whenever the guest OS tries to call into the module. Implementing this feature is left as future work.

7.5 Secure Symbiotic Modules

The two forms of the symbiotic modules described so far have focused on providing additional functionality, such as device drivers, to a guest OS. In almost all situations, these modules deliver an obvious benefit to the guest context and so we can rely to some degree on the guest OS supporting them. That is to say that we can trust the guest to cooperate to a certain degree during the module's loading, initialization, and execution. For instance, a small module that simply acts as a notification service for changes made at the VMM layer would be beneficial to the execution of the guest. The guest would have no reason to try to subvert it, other than as an attack vector targeting the VMM itself. While the use of a symbiotic module as a cross layer attack vector might seem dangerous, it is important to note that existing VMM interface layers suffer from this same problem [69].

There are a certain class of modules, however, that do require stronger guarantees about

the security of their execution. For instance, modules designed to implement security scanning features need to ensure that a compromised guest environment will be unable to interfere with their correct execution. For these types of modules, the framework presented thus far does not provide the guarantees necessary to ensure correct functionality. To address this issue, I have developed a third type of symbiotic module called a *Secure Symbiotic Module*. A secure symbiotic module is very similar to the previously described symbiotic modules, with differences only made to the interactions with the guest environment and the assumed execution context.

7.5.1 Environmental assumptions

Secure symbiotic modules are designed with the assumption that an OS is always initialized from a known valid state. That is, at boot time it is verifiable that the OS being loaded has not already been compromised. This assumption is fairly straightforward to enforce. One method is to load the OS image from a non volatile medium such as a CD, over the network via PXE, or even from the VMM itself. In these cases any successful attacks would only be able to compromise the local, in memory, version of the kernel. These local modifications would be lost the next time the machine is rebooted. Additionally the VMM could use checksumming to ensure that only a known valid kernel image is being loaded. Regardless of the method used, this ensures that the OS is secure until the user space initialization begins. This provides a window of guaranteed uncompromised execution which the VMM can rely upon to handle the necessary configuration needed for secure symbiotic modules. Once this configuration is completed, the VMM no longer has to rely on any cooperation from the guest OS to successfully execute the symbiotic module in a secure context. Furthermore, secure symbiotic modules can be loaded at any point thereafter, and still enjoy the same security guarantees.

It is also assumed that attacks do not result in the large scale modification to the run-

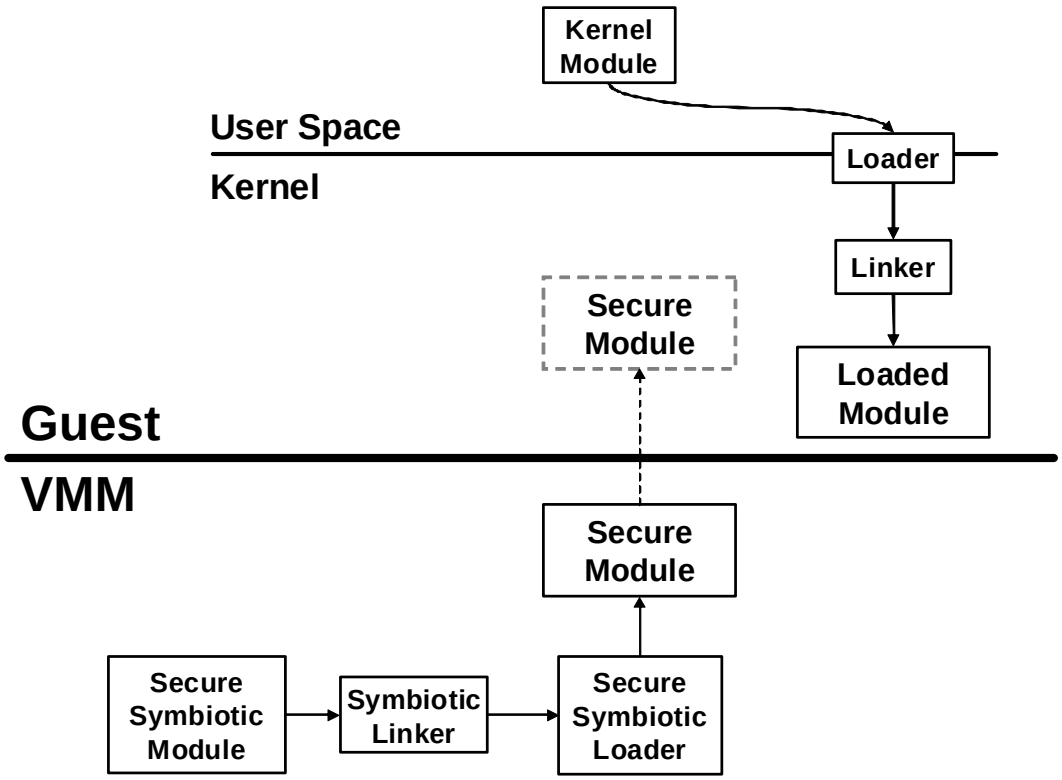


Figure 7.3: Secure symbiotic module architecture

ning guest kernel. Such an attack would essentially install its own OS kernel that emulates the expected behavior of the current kernel while hiding an active attack payload. While this does create a vector of attack, it should be noted that an attack of this scale would be exceedingly complicated and would likely be detectable due to the sheer scale of modifications done to the guest context.

7.5.2 Architecture

The architecture for secure symbiotic modules is given in Figure 7.3. Secure symbiotic modules use the same linking process as regular symbiotic modules, with a few modifications. Normal symbiotic modules require the guest OS to dynamically allocate memory

to contain the module itself as well as perform the requisite initialization. This is due to the fact that normal symbiotic modules are assimilated into the running kernel image, and once loaded effectively become part of the guest OS. In contrast, secure symbiotic modules operate as entirely separate entities that are never accessed by the guest OS or any part of the guest context. This is accomplished by removing the secure modules from the guest context at all times, except when the VMM wants them to be executed. The architecture ensures that the guest context is never even aware that a secure symbiotic module is present or activated.

While still running within the secure execution window during initialization the OS performs two crucial steps that provide the environment needed by the secure symbiotic module framework. The first operation is to provide the VMM with the necessary API that will be used to access the kernel functionality. This API is created in the same manner as the regular symbiotic module API, with the only difference being how it is advertised to the VMM. While regular symbiotic modules export the symbol table passively via SymSpy, secure symbiotic modules use a hypercall to send the VMM a copy of the API symbols during the secure execution window. This prevents any attack from compromising the module loading process by rewriting the symbol table to point to corrupted entries. This ensures that the symbols used to link a secure symbiotic module are valid. It should be noted that the API might still be subverted by corrupting the API implementation functions themselves. The solution to this, which I am leaving as future work, is to sequester the implementation of the secure API into a single memory region. This memory region is then passed to the VMM during the secure initialization window, where it is write protected by the VMM. In this way the API is fully protected from being modified by anything running in the guest context.

The second necessary step taken during the initialization window is to reserve an address space that will be used by any secure symbiotic module loaded by the VMM. The

kernel accomplishes this by reserving a region of its virtual address space, and marking it as unusable. It is important to note that this only reserves virtual memory and not physical memory. Because the secure symbiotic module is otherwise invisible to the guest context there is no need to locate it inside the guest's physical memory space. This also allows the VMM to multiplex a wide range of modules using the same memory region. This memory region is implemented using a special extension to the shadow paging architecture inside Palacios called a virtual memory overlay.

Virtual memory overlay Once the guest OS allocates a virtual address region, it signals its availability to Palacios. Once Palacios receives the information about this region it creates a special *virtual memory overlay* map. While the normal memory map used by Palacios maps guest physical memory addresses, the virtual memory map operates on guest virtual addresses. This map is only usable when the guest is running with shadow paging and operates in a similar manner to the mapping used for SwapBypass as discussed in Chapter 6.

Having a special virtual memory region controlled entirely by the VMM allows Palacios to ensure that a secure symbiotic module linked into that destination will always run correctly. Moreover it allows Palacios to dynamically map arbitrary pages into the guest memory space at runtime. The map is implemented as a clone of the physical memory map, with the exception that it operates on virtual addresses instead of physical ones. The shadow paging handlers are also modified to query the virtual memory overlay map at the very beginning of the page fault handler. If a page fault corresponds to a map entry, the handler calls a special function that fixes up the page fault based on the virtual address region. Once that function returns the handler returns immediately to the guest, and completely bypasses the standard shadow paging handlers.

To understand why this is necessary consider an example. If a guest OS were to re-

serve a physical address region, instead of a virtual address region, then the corresponding virtual addresses used to access the region would be whatever the guest page tables were configured to. This would allow an attacker to disable a secure symbiotic module simply by removing any page table entries that point to the physical memory reserved for the modules. Even if the guest OS modified the virtual addresses corresponding to the reserved region, the API would most likely break if the symbols were linked using relative addresses. To prevent this from occurring the VMM cannot depend on the guest OS to provide the virtual address lookups needed to execute a secure symbiotic module. This means that the VMM must be responsible for providing the virtual address to physical address conversion using shadow page tables. This in turn allows the VMM to redirect the virtual addresses to any physical addresses in host memory, which leads to the ability to multiplex modules into the same virtual memory region.

SymCall So far I have described a framework for linking in a block of code that is otherwise completely inaccessible to the environment it is intended to run in. While this might seem like a pointless exercise, it is actually a crucial design component. Based on the assumptions I have been using, the guest OS cannot be trusted in any manner once it has initialized its user space environment. At this point an attacker could very possibly have exploited a vulnerability in the kernel itself and gained control over the system. This means that the VMM cannot rely on the guest OS to actually use any of the secure features implemented inside the module. Naturally, one of the attacker's first actions will be to disable the security features that might detect or hinder them. Therefore, any secure module that requires invocation from the guest OS itself is vulnerable to being disabled. Since the OS cannot be trusted to invoke the module, that leaves only the VMM. Furthermore, the module must always be protected from an adversarial guest context during its execution.

In order to protect the secure symbiotic module from a compromised guest context, the

executing environment needs the following properties:

- The module code must be inaccessible from the guest context at all times, unless it has been explicitly invoked
- Invocation must vector immediately into the module, with no code path that involves anything under the control of the guest OS
- The module's execution must never be interrupted in a way that causes the execution to vector out of the module and into the guest OS.

Fortunately, all of these requirements can be met by invoking the module using a SymCall. As you might recall, symcalls can result in an immediate vectoring into a specific location inside the guest context. Furthermore, during the execution of a SymCall the VMM ensures that all external events are suppressed, which guarantees that the symcall will always run to completion and never block.

7.5.3 Operation

When everything is put together, Palacios is capable of dynamically invoking an arbitrary module at any point of a guest's standard operation. Once the environment is initially configured by the guest OS, a set of modules can be transparently loaded in the background and remain ready to be run whenever the VMM chooses. Each module is linked and loaded in a reserved location of the host's memory, where it is kept unused until needed. When the VMM decides to use a module's functionality, as part of a periodic scan or in response to suspicious behavior, Palacios reconfigures the virtual address overlay map to point to the module's location in host memory. This effectively activates the module inside the guest context, the module can now access any part of the guest image and the module is visible to the guest environment. However, instead of returning control back to the guest itself,

Palacios instead invokes a symcall that vectors control into the module. At this point the module performs its function and finally returns from the symcall, which in turn returns control back to the VMM. At this point the VMM unmaps the module from the virtual memory overlay and performs any necessary actions based on the result of the secure modules execution. If no problems were detected Palacios returns execution to the guest from where it originally left off. This method of operation is the same for every secure symbiotic module, and any module can be activated in this way simply by reconfiguring the virtual memory overlay to point to its location in host memory.

7.6 Conclusion

In this chapter I introduced symbiotic modules, and described the SymMod framework I have implemented in both Palacios and Linux. Symbiotic modules provide solutions to several problems prevalent in virtualized environments. First, symbiotic device drivers are a mechanism for avoiding the need to preconfigure VM's for every possible execution environment they might run on. Symbiotic device drivers allow a VMM to dynamically inject kernel modules and device drivers into a running kernel to extend or provide services that are missing from the guest kernel. Second, generic symbiotic modules serve to answer many of the problems not directly solved by symbiotic device drivers, at the expense of significant modifications to a guest kernel. These modules require a new internal OS API that is designed to remain stable across OS versions. These modules also serve as a mechanism for adding new symbiotic communication channels that do not already exist in a guest environment. Finally, secure symbiotic modules provide a secure execution environment for symbiotic modules whose usefulness relies on them being protected from the guest environment.

Chapter 8

Related Work

8.1 Virtualization Approaches

Pre-virtualization [54] is a technique of transforming an existing OS to use a paravirtualized interface. This approach automates the development effort needed to convert an OS from using a hardware interface to a paravirtualized one. In many ways pre-virtualization closely resembles symbiotic virtualization, it modifies an OS to allow a runtime VMM environment to modify its behavior to extend and optimize its performance in a virtual environment. However, pre-virtualization diverges from symbiotic virtualization in the techniques it uses. Pre-virtualization uses compile time binary modification to transform a guest OS such that it can be operated on by the VMM. Symbiotic virtualization is an approach that transforms an OS design such that it allows VMM modifications.

There are many current examples of virtualization tools that implement both full system virtualization [108, 70, 103] and paravirtualization [6, 76, 109]. In fact it is quickly becoming the case that these approaches are no longer mutually exclusive. Despite the blurring of the boundaries between both of these methods there has not been a significant departure from either. Symbiotic virtualization is a new interface approach for virtual environments by introducing a guest interface that a VMM can use.

Other virtualization techniques involve partitioning the internal OS state in such a way that separate user space environments can operate independently and concurrently [74, 58, 104]. While symbiotic techniques could possibly be applied to these environments, they lie outside of the context of my thesis due to the higher semantic layer at which they operate. While at a high level these approaches appear similar to the other virtualization architectures, they are in fact quite different and do not expose the same functionality as the full OS approaches.

8.2 Bridging the semantic gap

Considerable effort has been put into better bridging the *semantic gap* of the VMM↔OS interface and leveraging the information that flows across it [42, 41, 49, 96, 51, 75, 31]. However, the information gleaned from such black-box and gray-box approaches is still semantically poor, and thus constrains the decision making that the VMM can do. Further, it goes one way; the OS learns nothing from the VMM. In symbiotic virtualization the OS would make its internal state information easily accessible to the VMM and understandable. In addition to such a passive information interface, the OS would also also provide a functional interface that the VMM could use to access OS functionality. The VMM would use the passive information interface and the functional interface to augment OS functionality and improve performance via optimizations of the virtual environment.

Currently one of the most compelling uses for bridging the semantic gap is virtual machine introspection, most commonly used in security applications [14, 113, 26, 43, 25, 77, 40, 5, 44]. These approaches are notable because they attempt to generate semantic information of an untrusted guest environment. While symbiotic virtualization does not solve or completely address the problems that these tools aim to solve, it is the case that symbiotic approaches could facilitate these tools operation. Security introspection architectures

typically monitor common data structures and OS state present in a virtual machine. A symbiotic OS would provide this state in an accessible manner to a security scanner. If the security scanner begins operation at a known correct state, it can detect changes to these structures much more efficiently in a symbiotic context. However, it is also important to note that these structures could not be trusted completely, and other safeguards would have to be in place to ensure their integrity.

Others have also explored bridging the semantic gap to modify internal OS state in order to improve performance. FoxyTechnique [111] adapts the behavior of the underlying virtual hardware in order to elicit the desired responses by the guest OS. While similar to symbiotic virtualization this approach is still limited by the semantic gap, where as a symbiotic approach would not have those limitations. My thesis will be examining how to remove the semantic gap in order to facilitate the development of examples such as this.

8.3 SymCall

While SymCall is a new interface for invoking upcalls into a running guest environment, providing upcall support for a guest OS is not a new concept. However the standard approaches are generally based on notification signals as opposed to true upcall interfaces [16]. These notifications usually take the form of hardware interrupts that are assigned to special vectors and injected by the VMM. Because interrupts can be masked by a guest OS, these upcall interfaces are typically asynchronous. Furthermore, existing upcalls consist of only a notification signal and rely on a virtual device or event queue to supply any arguments. Symcalls in contrast are always synchronous and do not need to be disabled with the same frequency as interrupts. Furthermore they allow argument passing directly into the upcall, which enables the VMM to expose them as normal function calls.

8.4 Virtual device drivers

Many people have explored how to provide direct access to hardware devices for performance reasons. These approaches either require specialized hardware [78, 84], or specialized drivers [59]. My approach of symbiotic device drivers is complementary to these approaches. Symbiotic device drivers allow a VMM to provide a specialized device driver that provides VMM specific functionality, such as safety and reusability guarantees [110]. Furthermore symbiotic device drivers allow a VMM to provide a specific device driver to interface with local hardware, without requiring a VM to include large driver sets or rely on virtualized device interfaces such as [6, 53, 82].

8.5 Virtualization in HPC

Recent research activities on operating systems for large-scale supercomputers generally fall into two categories: those that are Linux-based and those that are not. A number of research projects are exploring approaches for configuring and adapting Linux to be more lightweight. Alternatively, there are a few research projects investigating non-Linux approaches, using either custom lightweight kernels or adapting other existing open-source OSes for HPC.

The Cray Linux Environment [45] is the most prominent example of using a stripped-down Linux system in an HPC system, and is currently being used on the petaflop-class Jaguar system at Oak Ridge National Laboratories. Other examples of this approach are the efforts to port Linux to the IBM BlueGene/L and BlueGene/P systems [86, 7]. Since a full Linux distribution is not used, this approach suffers many of the same functionality weaknesses as non-Linux approaches. In some cases, these systems have also encountered performance issues, for example due to the mismatch between the platform's memory management hardware and the Linux memory management subsystem.

Examples of the non-Linux approach include IBM's Compute Node Kernel (CNK) [65] and several projects being led by Sandia, including the Catamount [79] and Kitten projects as well as an effort using Plan9 [64]. Both CNK and Kitten address one of the primary weaknesses of previous lightweight operating systems by providing an environment that is largely compatible with Linux. Kitten differs from CNK in that it supports commodity x86_64 hardware, is being developed in the open under the GPL license, and provides the ability to run full-featured guest operating systems when linked with Palacios.

The desire to preserve the benefits of a lightweight environment but provide support a richer feature set has also led other lightweight kernel developers to explore more full-featured alternatives [85]. We have also explored other means of providing a more full-featured set of system services [99], but the complexity of building a framework for application-specific OSes is significantly greater than simply using an existing full-featured virtualized OS, especially if the performance impact is minimal.

There has been considerable interest, both recently and historically, in applying existing virtualization tools to HPC environments [80, 19, 27, 37, 97, 98, 112]. However, most of the recent work has been exclusively in the context of adapting or evaluating Xen and Linux on cluster platforms. Palacios and Kitten are a new OS/VMM solution developed specifically for HPC systems and applications. There are many examples of the benefits available from a virtualization layer [66] for HPC. There is nothing inherently restrictive about the virtualization tools used for these implementations, so these approaches could be directly applied to this Palacios and Kitten.

Chapter 9

Conclusion

In this dissertation I introduced *symbiotic virtualization* and described its implementation in the Palacios virtual machine monitor. an approach to designing virtualized architectures such that high level semantic information is available across the virtualization interface. Symbiotic virtualization bridges the semantic gap via a bidirectional set of synchronous and asynchronous communication channels.

Palacios is a OS independent embeddable VMM designed to target diverse architectures and environments, of which I am the primary developer of. Palacios was developed in response to the fact that existing virtualization solutions target highly specific data center and desktop environments, without focusing on other areas such as HPC and education. Palacios supports a wide range of both compile time and run time configuration options that generates a specialized architecture for each environment it is used in. Palacios currently supports multiple host OS environments such as the Kitten Lightweight Kernel for HPC, MINIX, and Linux. To date Palacios has successfully virtualized a very wide range of hardware, including a Cray XT supercomputer, both Infiniband and Ethernet clusters, as well as standard desktop and server machines.

As part of a collaboration with Sandia National Laboratories, Palacios has been evaluated using the RedStorm Cray XT3. This evaluation demonstrated that with a correctly

designed and configured architecture supercomputing applications can successfully run in a virtualized context with negligible overhead. Using common HPC benchmarks as well specialized applications developed at Sandia, we were able to perform a large scale evaluation which delivered performance within 5% of native. Furthermore, our evaluations showed that the low level interaction between the VMM and guest OS can have profound effects on system performance at scale, and that there is no single correct VMM architecture. A VMM that delivers performance as close to native as possible, must take into account a guest's internal behavior and state. Specifically we found that the choice of virtual paging implementations can produce dramatically difference results depending on the guest's behavior and implementation.

As a result of our evaluation I discovered that correctly optimizing a VMM requires detailed knowledge of the internal state of a guest environment. However current virtualization solutions make it a point to hide this information as much as possible, instead implementing extremely low level interfaces that make almost no semantic information available. The use of these interfaces makes collecting internal state information extremely difficult and costly. In order to solve these problems, I developed SymSpy a basic symbiotic interface that all other symbiotic interfaces are built on top of. SymSpy used shared memory regions to allow a VMM and a guest to asynchronously exchange well structured data. This interface allows a guest to advertise and access high level state information that would otherwise be difficult to obtain.

To evaluate the impact of symbiotic virtualization in HPC settings, I implemented a set of symbiotic interfaces targeting HPC. These interfaces were built using the SymSpy framework. One application of symbiotic virtualization was a feature called *PCI Passthrough*. This feature allows a host OS to fully relinquish control of a PCI device over to a guest OS. If the guest OS contains a symbiotic framework it can access the device directly without incurring any performance loss as a consequence of running as a

VM. Using the PCI passthrough interface we were able to show that a guest environment can perform network I/O using Infiniband hardware at native speeds. Furthermore, we were able to expand our earlier evaluation on RedStorm to 4096 nodes, the largest scale virtualization study performed to date.

While our results showed great promise for symbiotic virtualization in the context HPC, symbiotic interfaces can also be used in commodity environments. To explore further how symbiotic virtualization can be used to improve existing VMM architectures I developed SymCall. SymCall is a functional interface that allows a VMM to perform a synchronous upcall into a guest environment while executing an exit handler. This upcall framework allows the VMM to examine complex state information that is impossible to expose via a shared memory interface. Much of an OS's internal state is scattered among various data structures and memory locations, requiring a procedural interface to answer basic questions about it. SymCall allows the implementation of such an interface in such a way that its operation is invisible to the guest itself. Using SymCall I implemented SwapBypass, an example VMM service that optimizes the performance of the Linux swap subsystem. SwapBypass uses shadow paging techniques, an intelligent virtual disk, and a single SymCall to transparently expand a guest's memory space. In evaluations I was able to show that SwapBypass can effectively bypass the entire Linux swapping architecture providing access to swapped out memory at core memory speeds.

Finally, I designed and implemented the SymMod framework, which allows a VMM to dynamically load blocks of code that run in a guest context. This framework allows a VMM to extend an operating system's functionality at run time by injecting functional extensions that interface directly with the guest OS' internal API. With SymMod a VMM includes a set of symbiotic modules that target a given interface implemented inside a guest OS. These modules are then loaded into the running VM where they are capable of augmenting the guest's functionality in arbitrary ways. This mechanism allows a VMM

to directly control a guest's implementation in order to optimize its behavior or ensure its integrity. I examined three different mechanisms for loading symbiotic modules into a guest environment. First I developed a new mechanism whereby standard Linux kernel modules can be loaded directly into a running Linux kernel from the VMM instead of user space. Second, I explored a new internal kernel interface which load themselves into a guest environment in a protected context which cannot be altered by the guest OS itself. Finally, I implemented a fully secure mechanism whereby modules can be loaded without any guest interaction and in fact remain wholly transparent to the guest itself.

9.1 Summary of contributions

- **Development of symbiotic virtualization** I have developed symbiotic virtualization, a new approach to virtualized architectures that expose high level semantic information. With symbiotic virtualization a VMM and guest environment communicate high level semantic information across a bidirectional set of synchronous and asynchronous communication channels. These interfaces are optional and are compatible with non symbiotic environments as well as native hardware.

- **Palacios** I am the primary designer and implementer of Palacios, an OS independent embeddable VMM primarily for use in HPC environments. Palacios is an open source project that is publicly available under the BSD license.

- **Implementation of a symbiotic virtualization architecture** I have designed and implemented a symbiotic virtualization framework. The VMM component of the framework is implemented inside the Palacios VMM, while guest frameworks have been implemented in both Linux and Kitten.

- **SymSpy** I have designed and implemented SymSpy, passive, asynchronous symbi-

otic interface based on shared memory.

- **Symbiotic upcalls** I have implemented the SymCall framework which allows a VMM to make synchronous upcalls into a running guest OS.

- **SwapBypass** I implemented a symbiotic swapping extension that prevents a guest OS from thrashing in low memory situations. This example will use shadow paging mechanisms to transparently extend the amount of memory available to a guest environment.

- **Symbiotic modules** I implemented a framework that allows a symbiotic VMM to dynamically inject device drivers directly into a guest environment. These drivers will allow a guest OS to have direct access to hardware in a secure manner. Furthermore, the drivers can be changed in response to configuration changes or migration events.

- **Kitten: a lightweight HPC OS** I have assisted in the development of the Kitten lightweight kernel being developed by Sandia National Laboratories. I have contributed several functional extensions and to several design decisions.

- **Palacios and Kitten** I have embedded Palacios in Kitten. This entailed implementing the Palacios/Kitten interface harness as well as the Kitten interface for controlling Palacios.

- **Evaluation of virtualization in HPC at scale** I have evaluated Palacios and Kitten in the largest scale virtualization benchmark to date on the Red Storm supercomputer at Sandia. The evaluations were done using standard Sandia benchmarks running inside Catamount [46] and Compute Node Linux [45], two OSes used in HPC contexts.

- **Evaluation of symbiotic virtualization in HPC** I have explored and evaluated how symbiotic interfaces and the symbiotic approach in general can be applied to HPC systems.

- **Passthrough PCI devices** I have used the symbiotic approach to implement a new architecture for passthrough device I/O for physical PCI devices. This architecture allows a guest environment to access physical devices with no overhead.

- **Black box methods to bridge the semantic gap** I have implemented and evaluated several mechanisms for bridging the semantic gap in virtual networks..

  - **Automatic network reservations for virtual machines** I have demonstrated that by bridging the virtual network semantic gap it is possible to provide dynamic runtime network reservations for unmodified OSes and applications. I have implemented *VRESERVE*, a system that provides optical network reservations dynamically using virtual network traffic inference.

  - **Virtual network services** I have designed and implemented VTL, a framework for building virtual network services that provide optimizations and extensions to unmodified applications and OSes. The VTL framework allows anyone to easily bridge the virtual network semantic gap in a generic way.

9.2 Future Work

This dissertation has described Palacios, a fully featured and widely used VMM, as well as symbiotic virtualization, a new approach to designing virtualized interfaces. While the results so far are quite promising, both Palacios and symbiotic virtualization contain a large amount of promise for future developments.

9.2.1 Palacios

Palacios continues to be a project of active development and research. In many ways this dissertation serves as the introduction of Palacios, in so far as we have demonstrated its capability and promise but have not yet fully explored is possibilities. As of the writing of this dissertation, the implementation in MINIX is still fairly recent and the implementation in Linux has only just begun. Furthermore the implementation of multicore guest support is still in progress, and the virtual paging framework is still under active development. For the foreseeable future, Palacios will continue to be extended and improved.

9.2.2 Virtualization Architectures

In addition to working on the Palacios VMM itself, our work has and will continue to provide new insights into the core architectures behind virtual machines. The current hardware virtualization extensions are still very new, by x86 ISA standards, and will doubtlessly be the subject of many future improvements. The experiences with Palacios have placed us in a unique position of being able to propose new mechanisms and architectures for future virtualization platforms. Of predominant interest is the future of virtualized I/O. Hardware support for virtualized I/O is just beginning to emerge, and there are many research questions regarding how best to use these new technologies. Features such as IOMMUs and virtualized PCI devices based on SRIOV have yet to be fully introduced, and an open problem exists on how best to make use of them.

9.2.3 Virtualization in HPC

There is also much work left in the virtualization of HPC systems. As of this dissertation the focus has been on evaluating Palacios as a proof of concept, to demonstrate that large scale HPC virtualization is feasible. As we continue on with Palacios, I expect that that

focus will gradually change as Palacios becomes more widely used in HPC centers.

9.2.4 Symbiotic virtualization

In this dissertation, my description of symbiotic virtualization has been centered predominantly around the interfaces themselves, at the expense of evaluating applications that use them. The symbiotic interfaces I have developed are primarily enabling technologies, that open the door for many more advanced features and applications that have yet to be developed. I believe there are many opportunities in exploring how to further use these interfaces in specific situations. For instance, an obvious future step is to design, implement, and evaluation a fully functional security analysis system implemented using the secure symbiotic module framework, or developing a dynamic driver layer built on standard symbiotic modules. Also it is worth exploring the use of symcalls for performance monitoring, such as implementing a kprobes interface based on the SymCall framework. These and many more research areas have been enabled by the presence of these interfaces, and there is a great deal of future work in exploring how to use them.

Bibliography

[1] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2006.

[2] AMD Corporation. Pacifica virtualization extensions. http://enterprise.amd.com/Enterprise/serverVirtualization.aspx, 2005.

[3] Rolf Riesen Arthur B. Maccabe, Kevin S. Mccurley and Stephen R. Wheat. SUNMOS for the Intel Paragon: A brief user's guide. In *Intel Supercomputer Users' Group. 1994 Annual North America Users' Conference*, pages 245–251, 1994.

[4] Chang Bae, John Lange, and Peter Dinda. Comparing approaches to virtualized page translation in modern VMMs. Technical Report NWU-EECS-10-07, Department of Electrical Engineering and Computer Science, Northwestern University, April 2010.

[5] Fabrizio Baiardi and Daniele Sgandurra. Building trustworthy intrusion detection through VM introspection. In *IAS '07: Proceedings of the Third International Symposium on Information Assurance and Security*, pages 209–214, Washington, DC, USA, 2007. IEEE Computer Society.

[6] 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 *19th ACM Symposium on Operating Systems Principles (SOSP)*, October 2003.

[7] Pete Beckman et al. ZeptoOS project website, `http://www.mcs.anl.gov/research/projects/zeptoos/`.

[8] Muli Ben-Yehuda, Jon Mason, Orran Krieger, Jimi Xenidis, Leendert Van Doorn, Assit Mallick, Jun Nakajima, and Elsie Wahlig. Utilizing IOMMUs for virtualization in Linux and Xen. Technical report, 2009.

[9] Ravi Bhargava, Ben Serebrin, Francesco Spanini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2008.

[10] Ron Brightwell, Trammell Hudson, and Kevin Pedretti. SMARTMAP: Operating system support for efficient data sharing among processes on a multi-core processor. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (Supercomputing'08)*, November 2008.

[11] Ron Brightwell, Trammell Hudson, Kevin T. Pedretti, and Keith D. Underwood. SeaStar Interconnect: Balanced bandwidth for scalable performance. *IEEE Micro*, 26(3):41–57, May/June 2006.

[12] Vincent W. S. Chan, Katherine L. Hall, Eytan Modiano, and Kristin A. Rauschenbach. Architectures and technologies for high-speed optical data networks. *Journal of Lightwave Technology*, 16(12):2146–2168, December 1998.

[13] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *The 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, 2001.

[14] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R. K. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*, Seattle, WA, USA, March 2008.

[15] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 273–286, 2005.

[16] David D. Clark. The structuring of systems using upcalls. In *Proceedings of the tenth ACM symposium on Operating systems principles (SOSP)*, 1985.

[17] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium, 2004*, 2004.

[18] Yaozu Dong, Zhao Yu, and Greg Rose. SR-IOV networking in Xen: Architecture, design and implementation. In *1st Workshop on IO Virtualization (WIOV)*, 2008.

[19] W. Emeneker and D. Stanzione. HPC cluster readiness of Xen and User Mode Linux. In *2006 IEEE Conference Cluster Computing (CLUSTER)*, pages 1–8, 2006.

[20] Jr. E.S. Hertel, R.L. Bell, M.G. Elrick, A.V. Farnsworth, G.I. Kerley, J.M. McGlaun, S.V. Petney, S.A. Silling, P.A. Taylor, and L. Yarrington. CTH: A Software Family for Multi-Dimensional Shock Physics Analysis. In *19th International Symposium on Shock Waves, held at Marseille, France*, pages 377–382, July 1993.

[21] Kurt B. Ferreira, Ron Brightwell, and Patrick G. Bridges. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (Supercomputing'08)*, November 2008.

[22] Renato Figueiredo, Peter A. Dinda, and Jose Fortes. A case for grid computing on virtual machines. In *23rd IEEE Conference on Distributed Computing (ICDCS 2003*, pages 550–559, May 2003.

[23] Renato Figueiredo, Peter A. Dinda, and Jose Fortes. Guest editors' introduction: Resource virtualization renaissance. *Computer*, 38(5):28–31, 2005.

[24] Erich Focht, Jaka Močnik, Fredrik Unger, Danny Sternkopf, Marko Novak, and Thomas Grossmann. *High Performance Computing on Vector Systems 2009*, chapter The SX-Linux Project: A Progress Report, pages 79–96. Springer Berlin Heidelberg, 2009.

[25] Timothy Fraser, Matthew R. Evenson, and William A. Arbaugh. Vici virtual machine introspection for cognitive immunity. In *ACSAC '08: Proceedings of the 2008 Annual Computer Security Applications Conference*, pages 87–96, Washington, DC, USA, 2008. IEEE Computer Society.

[26] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *In Proc. Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[27] Ada Gavrilovska, Sanjay Kumar, Himanshu Raj, Karsten Schwan, Vishakha Gupta, Ripal Nathuji, Radhika Niranjan, Adit Ranadive, and Purav Saraiya. High performance hypervisor architectures: Virtualization in HPC systems. In *1st Workshop on System-level Virtualization for High Performance Computing (HPCVirt)*, 2007.

[28] The Taneja Group. The true cost of virtual server solutions, 2009.

[29] Yunhong Gu and Robert L. Grossman. UDT: An application level transport protocol for grid computing. In *2nd International Workshop on Protocols for Long-Distance Networks (PFLDNet '04)*, February 2004.

[30] A. Gupta, M. Zangrilli, A. Sundararaj, A. Huang, P. Dinda, and B. Lowekamp. Free network measurement for virtual machine distributed computing. In *20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[31] Ashish Gupta. *Black Box Methods for Inferring Parallel Applications Properties in Virtual Environments*. PhD thesis, Northwestern University, Department of Electrical Engineering and Computer Science, March 2008.

[32] Ashish Gupta and Peter A. Dinda. Inferring the topology and traffic load of parallel programs running in a virtual machine environment. In *10th Workshop on Job Scheduling Strategies for Parallel Processing (JSPPS 2004*, June 2004.

[33] Mike Heroux. HPCCG MicroApp. `https://software.sandia.gov/mantevo/downloads/HPCCG-0.5.tar.gz`, July 2007.

[34] Gi Hoang, Chang Bae, John Lange, Li Zhang, Peter Dinda, and Russ Joseph. A case for alternative nested paging models for virtualized systems. In *Computer Architecture Letters (To Appear)*, 2010.

[35] David Hovenmeyer, Jeffrey Hollingsworth, and Bobby Bhattacharjee. Running on the bare metal with GeekOS. In *35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE)*, 2004.

[36] Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, Fabrice Pailet, Shailendra Jain, Tiju Jacob, Satish Yada, Sraven Marella, Praveen Salihundam, Vasantha Erraguntla, Michael Konow, Michael Riepen, Guido Droege, Joerg Lindemann, Matthias Gries, Thomas Apel, Kersten Henriss, Tor Lund-Larsen, Sebastian Steibl, Shekhar Borkar, Vivek De, Rob Van Der Wijngaart, and Timothy Mattson. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Proceedings of the 2010 IEEE International Solid State Circuits Conference (ISSCC 2010)*, February 2010.

[37] Wei Huang, Jiuxing Liu, Bulent Abali, and Dhabaleswar K. Panda. A case for high performance computing with virtual machines. In *20th Annual International Conference on Supercomputing (ICS)*, pages 125–134, 2006.

[38] Intel Corporation. Intel virtualization technology specification for the IA-32 Intel architecture, April 2005.

[39] Intel GmbH. Intel MPI benchmarks: Users guide and methodology description, 2004.

[40] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 128–138, New York, NY, USA, 2007. ACM.

[41] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Antfarm: tracking processes in a virtual machine environment. In *ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference*. USENIX Association, 2006.

[42] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 14–24, 2006.

[43] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. VMM-based hidden process detection and identification using Lycosid. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 91–100, 2008.

[44] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 91–104, New York, NY, USA, 2005. ACM.

[45] Larry Kaplan. Cray CNL. In *FastOS PI Meeting and Workshop*, June 2007.

[46] Suzanne Kelly and Ron Brightwell. Software architecture of the lightweight kernel, Catamount. In *2005 Cray Users' Group Annual Technical Conference*. Cray Users' Group, May 2005.

[47] D. Kerbyson, H. Alme, A. Hoisie, F. Petrini, H. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of ACM/IEEE Supercomputing*, November 2001.

[48] John Lange, Peter Dinda, and Fabian Bustamante. Vortex: Enabling cooperative selective wormholing for network security systems. In *Proceedings of the 10th International Sympossium on Recent Advances in Intrusion Detection, (RAID 2007)*, September 2007.

[49] John Lange and Peter A. Dinda. Transparent network services via a virtual traffic layer for virtual machines. In *In Proceedings of the 16th International Symposium on High Performance Distributed Computing (HPDC)*, 2007.

[50] John Lange, A. Sundararaj, and P. Dinda. Automatic dynamic run-time optical network reservations. In *14th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, July 2005.

[51] John Lange, Ananth I. Sundararaj, and Peter A. Dinda. Automatic dynamic run-time optical network reservations. In *In Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 255–264, 2005.

[52] Kevin Lawton. Bochs: The open source IA-32 emulation project. `http://bochs.sourceforge.net`.

[53] J. LeVasseur, V. Uhlig, J. Stoess, and S. Goetz. Unmodified device driver reuse and improved system dependability. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2004.

[54] Joshua LeVasseur, Volkmar Uhlig, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. Pre-virtualization: soft layering for virtual machines. Technical Report 2006-15, Fakultät für Informatik, Universität Karlsruhe (TH), July 2006.

[55] B. Lin and P. Dinda. Vsched: Mixing batch and interactive virtual machines using periodic real-time scheduling. In *Proceedings of ACM/IEEE SC (Supercomputing)*, November 2005.

[56] Bin Lin. *Human-driven Optimization*. PhD thesis, Northwestern University, July 2007. Technical Report NWU-EECS-07-04, Department of Electrical Engineering and Computer Science.

[57] Bin Lin, Ananth Sundararaj, and Peter Dinda. Time-sharing parallel applications with performance isolation and control. In *Proceedings of the 4th IEEE International Conference on Autonomic Computing (ICAC)*, June 2007. An extended version appears in the Journal of Cluster Computing, Volume 11, Number 3, September 2008.

[58] Linux Vserver Project. http://www.linux-vserver.org.

[59] 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.

[60] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN 2005 Conference on Programming Language Design and Implementatin (PLDI)*, June 2005.

[61] Joel Mambretti, David Lillethun, John Lange, and Jeremy Weinberger. Optical Dynamic Intelligent Network Services (ODIN): An Experimental Control-Plane Architecture for High-Performance Distributed Environments Based on Dynamic Lightpath Provisioning. *IEEE Communications Magazine*, 44(3), 2006.

[62] John D. McCalpin. A survey of memory bandwidth and machine balance in current high performance computers. In *Newsletter of the IEEE Technical Committee on Computer Architecture (TCCA)*, December 1995.

[63] Mark F. Mergen, Volkmar Uhlig, Orran Krieger, and Jimi Xenidis. Virtualization for high-performance computing. *Operating Systems Review*, 40(2):8–11, 2006.

[64] Ronald G. Minnich, Matthew J. Sottile, Sung-Eun Choi, Erik Hendriks, and Jim McKie. Right-weight kernels: an off-the-shelf alternative to custom light-weight kernels. *SIGOPS Oper. Syst. Rev.*, 40(2):22–28, 2006.

[65] José E. Moreira, Michael Brutman, José Castaños, Thomas Engelsiepen, Mark Giampapa, Tom Gooding, Roger Haskin, Todd Inglett, Derek Lieber, Pat McCarthy, Mike Mundy, Jeff Parker, and Brian Wallenfelt. Designing a highly-scalable operating system: The Blue Gene/L story. In *ACM/IEEE Supercomputing SC'2006 conference*, 2006.

[66] Arun Babu Nagarajan, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive fault tolerance for HPC with Xen virtualization. In *21st Annual International Conference on Supercomputing (ICS)*, pages 23–32, 2007.

[67] John Nagle. Congestion control in IP/TCP internetworks. *SIGCOMM Computer Communication Review*, 14(4):11–17, 1984.

[68] Hideo Nishimura, Naoya Maruyama, and Satoshi Matsuoka. Virtual clusters on the fly - fast, scalable, and flexible installation. In *7th IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 549–556, 2007.

[69] Tavis Ormandy. An empirical study into the security exposure to hosts of hostile virtualized environments.

[70] Parallels Corporation. http://www.parallels.com.

[71] Fabrizio Petrini, Darren Kerbyson, and Scott Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proceedings of SC'2003*, 2003.

[72] S. J. Plimpton, R. Brightwell, C. Vaughan, K. Underwood, and M. Davis. A simple synchronous distributed-memory algorithm for the HPCC randomaccess benchmark. In *Proceedngs of the IEEE International Conference on Cluster Computing (CLUSTER)*, September 2006.

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

[74] Daniel Price and Andrew Tucker. Solaris zones: Operating system support for consolidating commercial workloads. In *LISA '04: Proceedings of the 18th USENIX conference on System administration*, pages 241–254, Berkeley, CA, USA, 2004. USENIX Association.

[75] Benjamin Prosnitz. Black box no more: Reconstruction of internal virtual machine state. Technical Report NWU-EECS-07-01, Department of Electrical Engineering and Computer Science, Northwestern University, March 2007.

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

[77] Nguyen Anh Quynh and Yoshiyasu Takefuji. Towards a tamper-resistant kernel rootkit detector. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 276–283, New York, NY, USA, 2007. ACM.

[78] 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)*, July 2007.

[79] Rolf Riesen, Ron Brightwell, Patrick Bridges, Trammell Hudson, Arthur Maccabe, Patrick Widener, and Kurt Ferreira. Designing and implementing lightweight kernels for capability computing. *Concurrency and Computation: Practice and Experience*, 21(6):793–817, April 2009.

[80] Dennis M. Ritchie. A guest facility for Unicos. In *UNIX and Supercomputers Workshop Proceedings*, pages 19–24. USENIX, September 1988.

[81] John Scott Robin and Cynthia E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *SSYM'00: Proceedings of the 9th conference on USENIX Security Symposium*, pages 10–10, 2000.

[82] Rusty Russell. virtio: towards a de-facto standard for virtual I/O devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, 2008.

[83] Sandia National Labs. Introducing Red Storm. `www.sandia.gov/ASC/redstorm.html`.

[84] Jeffrey Shafer, David Carr, Aravind Menon, Scott Rixner, Alan L. Cox, Willy Zwaenepoel, and Paul Willmann. Concurrent direct network access for virtual machine monitors. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 306–317, Washington, DC, USA, 2007. IEEE Computer Society.

[85] Edi Shmueli, George Almasi, Jose Brunheroto, Jose Castanos, Gabor Dozsa, Sameer Kumar, and Derek Lieber. Evaluating the effect of replacing CNK with Linux on the compute-nodes of Blue Gene/L. In *roceedings of the 22nd International Conference on Supercomputing*, pages 165–174, New York, NY, USA, 2008. ACM.

[86] Edi Shmueli, George Almási, Jose Brunheroto, Jose Castaños, Gabor Dózsa, Sameer Kumar, and Derek Lieber. Evaluating the effect of replacing CNK with Linux on the compute-nodes of Blue Gene/L. In *22nd Annual International Conference on Supercomputing (ICS)*, pages 165–174, New York, NY, USA, 2008. ACM.

[87] Lance Shuler, Chu Jong, Rolf Riesen, David van Dresser, Arthur B Maccabe, Lee Ann Fisk, and T Mack Stallcup. The PUMA operating system for massively parallel computers. In *1995 Intel Supercomputer User's Group Conference*. Intel Supercomputer User's Group, 1995.

[88] James Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, 2005.

[89] T. Stricker and T. Gross. Optimizing memory system performance for communication in parallel computers. In *Proceedings of the 22nd annual international symposium on Computer architecture (ISCA)*, 1995.

[90] A. Sundararaj, A. Gupta, , and P. Dinda. Increasing application performance in virtual environments through run-time inference and adaptation. In *14th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, July 2005.

[91] Ananth Sundararaj, Manan Sanghi, John Lange, and Peter Dinda. An optimization problem in adaptive virtual environmnets. In *seventh Workshop on Mathematical Performance Modeling and Analysis (MAMA)*, June 2005.

[92] Ananth Sundararaj, Manan Sanghi, John Lange, and Peter Dinda. Hardness of approximation and greedy algorithms for the adaptation problem in virtual environments. In *3rd IEEE International Conference on Autonomic Computing (ICAC)*, 2006.

[93] Ananth Sundararaj, Manan Sanghi, John Lange, and Peter Dinda. Hardness of approximation and greedy algorithms for the adaptation problem in virtual environments. Technical Report NWU-EECS-06-06, Department of Electrical Engineering and Computer Science, Northwestern University, July 2006.

[94] Ananth I. Sundararaj. *Automatic, Run-time and Dynamic Adaptation of Distributed Applications Executing in Virtual Environments*. PhD thesis, Northwestern University, Department of Electrical Engineering and Computer Science, November 2006.

[95] Ananth I. Sundararaj and Dan Duchamp. Analytical characterization of the throughput of a split TCP connection. Technical report, Department of Computer Science, Stevens Institute of Technology, 2003.

[96] Ananth I. Sundararaj, Ashish Gupta, and Peter A. Dinda. Increasing application performance in virtual environments through run-time inference and adaptation. In *In Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 2005.

[97] Samuel Thibault and Tim Deegan. Improving performance by embedding HPC applications in lightweight Xen domains. In *2nd Workshop on System-level Virtualization for High Performance Computing (HPCVirt)*, pages 9–15, 2008.

[98] Anand Tikotekar, Geoffroy Vallée, Thomas Naughton, Hong Ong, Christian Engelmann, Stephen L Scott, and Anthony M Filippi. Effects of virtualization on a scientific application running a hyperspectral radiative transfer code on virtual machines. In *2nd Workshop on System-Level Virtualization for High Performance Computing (HPCVirt)*, pages 16–23, 2008.

[99] Jean-Charles Tournier, Patrick Bridges, Arthur B. Maccabe, Patrick Widener, Zaid Abudayyeh, Ron Brightwell, Rolf Riesen, and Trammell Hudson. Towards a framework for dedicated operating systems development in high-end computing systems. *ACM SIGOPS Operating Systems Review*, 40(2), April 2006.

[100] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy Santoni, Fernando Martin, Andrew Anderson, Steve Bennettt, Alain Kagi, Felix Leung, and Larry Smith. The architecture of virtual machines. *IEEE Computer*, pages 48–56, May 2005.

[101] CORPORATE UNIX Press. *System V application binary interface (3rd ed.)*. 1993.

[102] Geoffroy Vallee, Thomas Naughton, Christian Engelmann, Hong Ong, and Stephen L. Scott. System-level virtualization for high performance computing. In *PDP '08: Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 636–643, 2008.

[103] VirtualBox. http://www.virtualbox.org.

[104] Virtuozzo Corporation. http://www.swsoft.com.

[105] Denys Vlasenko. http://www.busybox.net.

[106] VMWare. http://www.vmware.com/products/fusion.

[107] VMWare Corporation. http://www.vmware.com.

[108] Carl Waldsburger. Memory resource management in VMware ESX Server. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[109] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the Denali isolation kernel. *SIGOPS Oper. Syst. Rev.*, 36(SI):195–209, 2002.

[110] Lei Xia, John Lange, and Peter Dinda. Towards virtual passthrough I/O on commodity devices. In *Proceedings of the Workshop on I/O Virtualization at OSDI*, December 2008.

[111] Hiroshi Yamada and Kenji Kono. Foxytechnique: tricking operating system policies with a virtual machine monitor. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 55–64, New York, NY, USA, 2007. ACM.

[112] Lamia Youseff, Rich Wolski, Brent Gorda, and Chandra Krintz. Evaluating the performance impact of Xen on MPI and process execution for HPC systems. In *2nd International Workshop on Virtualization Technology in Distributed Computing (VTDC)*, page 1, 2006.

[113] Yang Yu, Fanglu Guo, Susanta Nanda, Lap-chung Lam, and Tzi-cker Chiueh. A feather-weight virtual machine for windows applications. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 24–34, New York, NY, USA, 2006. ACM.