OPERATING SYSTEM MECHANISMS FOR PERFORMANCE ISOLATION BETWEEN CO-LOCATED APPLICATIONS

by

Jiannan Ouyang

B.E. in Computer Science, University of Science and Technology of China, 2010

Submitted to the Graduate Faculty of
the Kenneth P. Dietrich School of Arts and Sciences,
Department of Computer Science in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2016

UNIVERSITY OF PITTSBURGH

KENNETH P. DIETRICH SCHOOL OF ARTS AND SCIENCES,

DEPARTMENT OF COMPUTER SCIENCE

This dissertation was presented

by

Jiannan Ouyang

It was defended on

June 22nd 2016

and approved by

Dr. John Lange, Department of Computer Science, University of Pittsburgh

Dr. Daniel Mosse, Department of Computer Science, University of Pittsburgh

Dr. Youtao Zhang, Department of Computer Science, University of Pittsburgh

Dr. Zhiqiang Lin, Department of Computer Science, University of Texas at Dallas

Dissertation Director: Dr. John Lange, Department of Computer Science, University of Pittsburgh

OPERATING SYSTEM MECHANISMS FOR PERFORMANCE ISOLATION BETWEEN CO-LOCATED APPLICATIONS

Jiannan Ouyang, PhD

University of Pittsburgh, 2016

The efficient sharing of a single server node between multiple co-located applications is increasingly important in modern large-scale datacenters and supercomputers. However, existing operating system (OS) architectures fall short in ensuring the performance isolation between co-located applications, which impedes the efficiency of large-scale computing infrastructures as well as the scalability of large-scale applications. The goal of this research is to improve the performance isolation capability of existing OS kernels, as well as to develop novel OS architectures aiming to provide isolated and optimized execution environments for workloads with disparate runtime requirements.

We first studied the performance interference problem between time-shared virtual machines as seen in the cloud. We conducted comprehensive performance analyses to identify the root causes of the kernel performance degradation problem in shared virtual environments. To address this problem, we designed and implemented two synchronization techniques optimized for shared virtual environments: the preemptable ticket spinlock (pmtlock) algorithm and the Shoot4U paravirtual TLB shootdown scheme. Our evaluation demonstrates that both techniques significantly reduce the performance interference between co-located virtual machines.

Besides improving existing OS kernels, we also looked at the design of operating systems aiming to provide isolated and optimized execution environments for in-situ analysis applications as seen in modern high performance computing (HPC). We designed and implemented the Pisces lightweight co-kernel architecture, which allows multiple independent lightweight

co-kernels to be deployed side-by-side with Linux on isolated hardware partitions. Each co-kernel can be optimized for the local HPC workload, while the performance isolation between co-kernels is enforced at both the software and hardware level. Our evaluation shows better application scalability on the co-kernel architecture compared with native Linux.

Finally, to support high performance I/O on lightweight co-kernels, we developed HobbesIO: an I/O delegation service on lightweight co-kernels that allows the application transparent I/O delegation from a co-kernel process to an I/O service processes deployed on arbitrary native or virtual Linux enclaves. Our evaluation shows that HobbesIO achieves comparable performance with native Linux.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF CODES

For my family, mentors and friends.

1.0 INTRODUCTION

Large-scale computing infrastructures, for instance warehouse scale datacenters and supercomputers, consisting of tens of thousands of networked servers, form the backbone of modern computing industry. They serve as the material foundation of web services including search engines and social networks, as well as scientific computations that allow scientists and researchers to tackle cutting-edge problems. Advances in large-scale computing infrastructures during the past decade enable novel computing paradigms such as cloud computing, big data as well as large-scale machine learning, which lead to dramatic improvements in our social, economy and everyday life. In turn, these improvements shed light on new problems and opportunities that require even more computational resources. This demand poses tough challenges to computer system researchers to continually offer more computational resources with sustainable performance and reliability.

With the increasing amount of computational resources available on each compute node, and the exploding amount of data generated by large-scale applications, it is increasingly important to efficiently share a single server node among multiple applications or services in modern large-scale datacenters and supercomputers. However, the performance interference problem, where the performance of one application is impacted by the system software and co-located applications sharing the same computational resources, impedes the effectiveness and scalability of application co-location and resource sharing [Lo et al., 2015].

In particular, several studies have established that the average server utilization in most modern datacenters is low, ranging from 10% to 50% [Lo et al., 2015; Barroso et al., 2013; Kaplan et al., 2008; Gartner, 2012; Reiss et al., 2012; Ma et al., 2015], which significantly hurts the cost and energy efficiency of large-scale datacenters. Specifically, Google's typical online-service datacenters exhibit only about 30% CPU utilization on average [Barroso et al.,

2013], while the industry wide average is even lower: only between 6% [Kaplan et al., 2008] to 12% [Gartner, 2012]. The root cause of this problem is that to guarantee Quality-of-Service (QoS) of latency critical applications, data center operators or developers tend to avoid sharing by either dedicating computational resources to applications, or exaggerating resource reservations for applications in shared environments [Lo et al., 2015; Reiss et al., 2012]. To address this problem and improve the efficiency of sharing, better performance isolation between applications must be ensured at all levels in the computing infrastructure.

On the other hand, the emerging trend of in-situ data processing [Ma et al., 2007; Zheng et al., 2013] in modern high performance computing (HPC) requires the system software to provide optimized and isolated execution environments, for co-located applications with disparate requirements. In-situ data processing co-locates data generation (simulation) and processing (analysis/visualization) workloads on the same compute node and processes the generated data locally. This model greatly reduces the data volume transferred over interconnects and therefore achieves better energy efficiency and scalability, compared with traditional HPC models that use a dedicated cluster for each workload, and move data between clusters over interconnects [Kogge et al., 2008]. However, the in-situ model also poses significant challenges on the performance isolation capability of the node level system software. Because tightly-coupled high performance computing (HPC) applications are known to be sensitive to performance variance [Petrini et al., 2003; Ferreira et al., 2008; Hoefler et al., 2010]. For example, Petrini et al. showed that application performance could be improved by a factor of 2 on 8192 processors by reducing interference of system daemons [Petrini et al., 2003]. As a result, specialized operating systems are built for these applications to provide isolated execution environments [Kaplan, 2007; Giampapa et al., 2010]. However, these optimized operating systems often lack the OS functionalities or features required by the data processing applications.

We observe that modern system software architecture is largely based on a *shared homogeneous* system software layer, which typically is an operating system or a hypervisor deployed on the bare mental hardware. However, we argue that *a shared homogeneous system software architecture fall short in ensuring the performance isolation between co-located applications*, which impedes the efficiency of large-scale computing infrastructures as well as

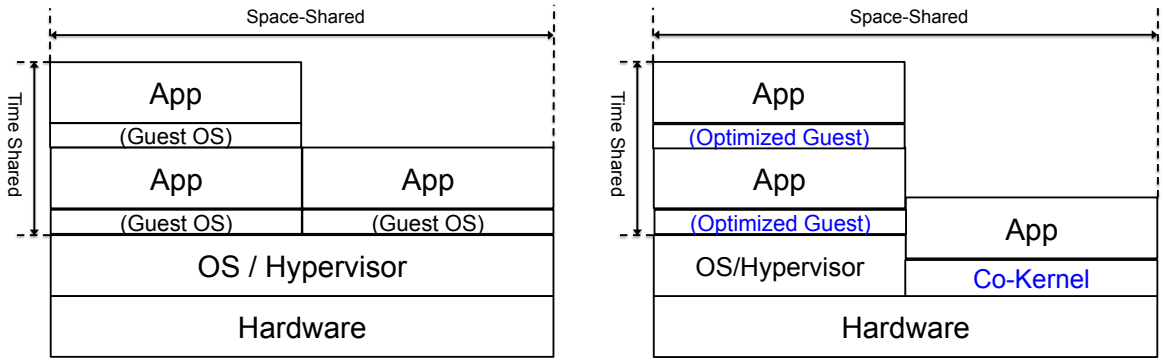
the scalability of large-scale applications.

Because on one hand, homogeneous kernels do not suit for the heterogeneous computing environments in modern computing infrastructures. In particular, we will show that OS kernels designed for native environments can experience dramatic performance degradation in shared virtual environments. Thus, kernels with virtualization specific optimizations should be used in virtual environments. On the other hand, a homogeneous system software layer cannot meet disparate application requirements. For example, in the in-situ data processing model, the data generation workloads are typically tightly-coupled large-scale applications, which require optimized HPC kernels to ensure consistent performance; while the data processing workloads demand generic OS services, programming languages and tools supported by Linux. It is very difficult for a homogeneous system software layer to provide HPC optimized environments and Linux compatibility on a single node at the same time. Finally, a shared system software layer cannot prevent the performance interference resulting from contentions on shared data structures and code paths at the system software level.

Therefore, in this dissertation, we advocate for *isolated heterogeneous kernels* on the same node. We aim to provide sustainable and scalable performance isolation at the system software layer through optimizing existing OS mechanisms, as well as rethink the OS designs and explore novel OS architectures.

1.1 RESEARCH OVERVIEW

Figure 1 (a) illustrates the state-of-the-art system organization that shares a single node among multiple co-located applications. A shared host operating system, or hypervisor in case of full virtualization based approaches, is deployed on the bare mental hardware. Applications can be directly deployed on the shared system software layer, or encapsulated inside a virtual machine or a container [Soltesz et al., 2007]. A virtual machine based approach requires a guest OS, typically Linux, to be deployed inside the VM. In case of containers, applications are deployed on top of the native host operating system, who is



(a) Current Approach (b) Proposed Approach

Figure 1: System Software Architecture for Application Co-Location

responsible for ensuring the isolation of namespaces and performance between containers. However, all three approaches has a shared homogeneous system software layer: the operating system or the hypervisor. Therefore, they share the drawbacks of shared homogeneous system software as we discussed previously.

In this work, we advocate for a isolated heterogeneous kernels based approach as depicted in Figure 1 (b). First of all, heterogeneous kernels should be used to adapt to heterogeneous computing environments. In particular, we claim that operating systems designed for physical environments should be optimized specifically for virtual environments to sustain good performance. Therefore, we looked at guest operating system optimizations to ensure the performance isolation between time-shared virtual machines. Besides, we propose that heterogeneous system software stacks can be deployed on isolated hardware partitions on the same node. Each software stack is optimized for certain class of applications, and independently manages a dedicated set of hardware resources. No dependency between software stacks is allowed, so that performance isolation between stacks is enforced at both the hardware and software layer. This architecture, which we call *the co-kernel architecture*, allows the creation of isolated execution environments optimized for certain workloads. In this work, we apply the co-kernel architecture to in-situ analysis HPC workloads using lightweight ker-

nels [Lange et al., 2010].

Thesis Statement An isolated heterogeneous kernels based approach allows the deployment of heterogeneous kernels on the same node; each kernel can be optimized for certain workloads or computing environments; the performance isolation between co-located applications can be enforced at both the hardware and the system software layer.

We first looked at a case where the operating system designed for physical computing environments experiences significant performance degradation in shared virtual environments because of performance interference between time-shared virtual machines. Virtualization breaks one fundamental assumption held by kernel developers: the (virtual) CPU is always progressing. However, a vCPU can be preempted and suspended by the VMM. These effects are especially harmful for timing sensitive kernel synchronization operations, in particular those busy-waiting based synchronization operations. We conducted comprehensive performance profiling and instrumentation to pin-point the kernel performance bottlenecks in shared virtual environments. We identified busy-waiting based kernel synchronization operations as the main source of performance overhead. Specifically, kernel spinlocks and TLB shootdown operations. To address these problem, we proposed the preemptable ticket spinlock algorithm and the Shoot4U paravirtual TLB shootdown scheme, which are specialized for virtual environments to tolerate vCPU preemptions. Our evaluation results show that our proposed techniques greatly reduce the performance degradation caused by vCPU preemptions.

The second part of this research rethinks the design of the node level system software architecture for in-situ analysis workloads in modern HPC. In-situ analysis workloads require a HPC optimized execution environment for the simulation workloads, a Linux environment for the analytic and visualization workloads, while the performance isolation between the two environments has to be enforced. These requirements can hardly be met by current shared homogeneous kernel based approaches. Therefore, we proposed the co-kernel architecture, in which a compute node is decomposed and viewed as a pool of computational resources, including CPU cores, memory blocks, and I/O devices. Those resources are then dynamically composed at runtime to form isolated hardware resource partitions. On each partition, an independent kernel as well as a software stack on top of it is deployed. Each software stack

directly manages the hardware resources belonging to its partition without any external dependencies. Compared with a shared homogeneous system software approach, our co-kernel architecture allows each software stack to be optimized to certain class of applications. Meanwhile, performance isolation is enforced between software stacks because sharing is restricted at both the hardware and software level.

We implemented the proposed co-kernel architecture based on the Linux kernel and the Kitten lightweight kernel [Lange et al., 2010]. We developed the Pisces kernel module to extend unmodified Linux kernels to support co-located kernels. We also modified the Kitten kernel to support a multi-instance environment. Our evaluation demonstrates that the Kitten co-kernel shows significantly more consistent performance than Linux, especially when competing co-located workload is deployed on the same node with dedicated hardware resources. It also shows that the co-kernel architecture achieves better performance isolation and scalability compared with the native Linux approach as we scale up the number of computing nodes.

Finally, we revisited the I/O mechanism on lightweight co-kernels, because the previously used lightweight virtualization based I/O approach introduces OS noises and kernel overheads from the guest Linux kernel. We investigated HobbesIO, an I/O delegation service on lightweight co-kernels that allows the application transparent I/O delegation from a co-kernel process to an I/O service processes deployed on arbitrary native or virtual Linux enclaves. Our evaluation shows that HobbesIO achieves comparable performance with native Linux.

1.2 CONTRIBUTIONS

The contributions of this research include,

- Investigated the performance problems caused by preempted virtual CPUs. Pin-pointed the busy-waiting based synchronization as an important source of the performance degradation problem in shared virtual environments.
- Identified the lock waiter preemption problem. Proposed the preemptable ticket spinlock

(pmtlock) algorithm to address this problem, and demonstrated significant performance improvement over the state-of-the-art ticket spinlocks.

- Proposed the Shoot4U paravirtual TLB shootdown scheme to address the TLB shootdown preemption problem. Demonstrated superior performance over both the baseline and a state-of-the-art optimization.
- Developed the Pisces co-kernel architecture that allows multiple optimized and isolated heterogeneous kernels to be deployed on the same node.
- Demonstrated that lightweight co-kernels can achieve better performance isolation and scalability for in-situ workloads compared with a shared Linux approach.
- Designed and studied the HobbesIO I/O delegation service for lightweight kernels that enables transparently delegating I/O system calls to an arbitrary native or virtual Linux instance.

As a result of this work, I also contributed to the following open source software,

PMTLOCK A Linux implementation of the preemptable ticket spinlock algorithm.

Shoot4U A Linux implementation of the Shoot4U paravirtual TLB shootdown scheme.

Pisces Designed and implemented the fundamental mechanisms of the Pisces kernel module that enable the co-kernel support on Linux.

Kitten Augmented the Kitten lightweight kernel to support the co-kernel architecture. Added SATA device support.

Palacios Augmented the PCI passthrough feature for the co-kernel architecture.

HobbesIO Designed and implemented a I/O delegation mechanism for lightweight co-kernels.

1.3 OUTLINE

The rest of this dissertation is organized as follows:

In Chapter 2, I will review the literature on the topics of kernel synchronization overhead in virtual environments, operating system design for HPC systems and I/O service delegation.

In Chapter 3, I will look at the performance interference problem between time-shared virtual machines caused by preempted virtual CPUs. I will show detailed profiling and instrumentation results to demonstrate that busy-waiting based synchronization operations are an important cause of the performance degradation problem. After that, I will describe two optimized kernel synchronization schemes designed for shared virtual environments: the preemptable ticket spinlock algorithm and the Shoot4U paravirtual TLB shootdown scheme. Finally, I will show empirical results to demonstrate the efficiency of the proposed schemes.

Chapter 4 introduces the lightweight co-kernel architecture, our proposed system software architecture for in-situ data processing workloads in modern HPC systems. Details about our motivations, design goals, system architecture as well as the implementation will be covered. Finally, a comprehensive evaluation of our system will be shown on both a single node and a multi-node HPC cluster.

In Chapter 5, I will introduce HobbesIO, an application transparent I/O delegation service on lightweight co-kernels. I will discuss the limitation our previous I/O solution on co-kernels, and compare existing I/O approaches to motivate an I/O delegation based solution. The design, implementation and evaluation of HobbesIO will be covered.

Finally, I will conclude in Chapter 6.

2.0 LITERATURE REVIEW

2.1 KERNEL SYNCHRONIZATION OVERHEAD IN VIRTUAL ENVIRONMENTS

Numerous research works have looked at the problem of kernel synchronization overhead in virtual environments. Most of them have focused on spinlocks and the *lock holder preemption* problem, originally identified by V. Uhlig et al. in 2004 [Uhlig et al., 2004]. Lock holder preemption happens when a virtual CPU (vCPU) holding a spinlock is preempted by the VMM, which dramatically increases the waiting time of other vCPUs requesting the same lock. Uhlig et al. [Uhlig et al., 2004] proposed a paravirtualization based approach in which the guest OS provides scheduling hints to the underlying VMM. These hints demarcated non-preemptable regions of guest execution that corresponded to critical sections in which a spinlock was held. T. Friebel and S. Biemueller [Friebel, 2008] proposed a paravirtual spinlock approach, which was later adopted by Xen and KVM [Raghavendra and Fitzhardinge, 2012]. In their scheme a vCPU notifies the VMM via a hypercall if it has been waiting longer than a threshold. The VMM then blocks the spinning vCPU until the requested lock is released.

Besides the spinlock preemption problems, a few previous works have looked into the *TLB shootdown preemption* problem. In particular, H. Kim et al. [Kim et al., 2013] studied the performance degradation caused by both spinlock and TLB shootdown preemptions. They proposed the use of TLB shootdown IPIs as a VMM scheduling heuristic in order to reduce the delay introduced by a preempted vCPU. While their approach does help alleviate the delays imposed by TLB shootdowns on preempted vCPUs, it does not address the underlying problem directly. In contrast, our Shoot4U mechanism addresses the source of the problem directly by eliminating the necessity for busy-waiting inside the VM. A paravirtual remote

flush TLB scheme (kvmtlb) has also been developed for KVM to address the TLB shootdown preemption problem [Dadhania, 2012]. This scheme maintains the preemption state of all vCPUs inside the VMM and shares this information with the guest. When initiating TLB operations, if the remote vCPU is running, then the conventional shootdown approach is used. Otherwise, if the remote vCPU is preempted, a *should_flush* flag is set on that remote vCPU and an IPI is not sent. When rescheduling a vCPU, the VMM checks the *should_flush* flag. If set, the VMM invalidates all TLB entries of that vCPU. The primary limitation of this technique is that the preemption state of a vCPU can change after its state has been checked by the invoking CPU but before the IPI is actually delivered. In this case, the involving vCPU could result in busy-waiting a preempted vCPU again.

Other approaches to improving VM performance in the face of cross core synchronizations include improving VMM scheduling policy. In particular, co-scheduling [Ousterhout, 1982] requires all vCPUs from one VM to be scheduled at the same time. This approach as well as its variants such as relaxed co-scheduling was adopted by the virtual machine scheduler in VMware ESX [VMware, 2010]. Other co-scheduling variants include adaptive co-scheduling schemes [Weng et al., 2011; Zhang et al., 2012] that allow the VMM scheduler to dynamically alternate between co-scheduling and asynchronous scheduling for a particular VM, as well as balanced scheduling [Sukwong and Kim, 2011] which associates a VM's individual vCPUs with dedicated physical CPUs and does not require that the vCPUs be co-scheduled. H. Kim et al. [Kim et al., 2013] proposed the demand-based coordinated scheduling that controls time-sharing in response to inter-processor interrupts (IPIs) between virtual CPUs. While each of these approaches alleviate the problems caused by intra-VM synchronizations, they do so by providing workarounds as opposed to addressing the underlying issues.

Hardware assisted approaches have also been developed for busy-waiting preemption problem. The Pause-Loop Exiting [Intel, 2016] feature in Intel processors as well as the Pause Filter feature [AMD, 2011] in AMD processors allow the hardware to detect spinning vCPUs and trigger a VMexit event if a vCPU spins longer than a threshold. Major hypervisors have added the support for these features. However, because of the semantic gap [Bauman et al., 2015; Fu et al., 2014] between the guest and the VMM, it is still an open question how to utilize and tune these features effectively.

Besides busy-waiting preemption problems, researchers have also found other performance problems caused by CPU overcommitment. For example, Gleaner [Ding et al., 2014] looked at the cost caused by the intervention from the VMM during synchronization-induced idling in the application, guest OS, or supporting libraries, which was denoted as the blocked-waiter wake up problem.

2.2 OPERATING SYSTEM DESIGN FOR SUPERCOMPUTERS

In the past decade, HPC systems have converged to use Linux as the preferred node operating system. This has led Linux to emerge as the dominant environment for many modern HPC systems [Yoshii et al., 2009; Kaplan, 2007] due to its support of extensive feature sets, ease of programmability, familiarity to application developers, and general ubiquity. While Linux environments provide tangible benefits to both usability and maintainability, they suffer from fundamental limitations when it comes to providing *predictable performance* as well as effective *performance isolation* that are required by tightly coupled HPC applications. This is because commodity systems, such as Linux, are designed to maximize a set of design goals that conflict with those required to provide predicable performance and complete isolation. Specifically, commodity systems are almost always designed to maximize resource utilization, ensure fairness, and most importantly, gracefully degrade in the face of increasing loads. These goals often result in non-predictable performance and software level interference that has a significant impact on HPC application performance at a large scale.

As a result, two separate philosophies have emerged over recent years concerning the development of operating systems specialized for supercomputers. On the one hand, a series of projects have investigated the ability to configure and adapt Linux for supercomputing environments by selecting removing unused features to create a more lightweight kernel (LWK). Alternatively, other work has investigated the development of lightweight operating systems from scratch with a consistent focus on maintaining a high performance environment.

Perhaps the most prominent example of a Linux-based supercomputing OS is Compute Node Linux (CNL) [Kaplan, 2007], part of the larger Cray Linux Environment. CNL has been

deployed on a variety of Cray supercomputers in recent years, including the multi-petaflop Titan system at Oak Ridge National Laboratory. Additional examples of the Linux-based approach can be seen in efforts to port Linux-like environments to the IBM BlueGene/L and BlueGene/P systems [ZeptoOS, 2016; Appavoo et al., 2008]. Alternatively, examples using non-Linux based OS deployment can be seen in IBM's Compute Node Kernel (CNK) [Giampapa et al., 2010] and several projects being led by Sandia National Laboratories, including the Kitten [Lange et al., 2010] project. While CNK and Kitten both incorporate lightweight design philosophies that directly attempt to limit OS interference by limiting many general-purpose features found in Linux environments, both CNK and Kitten address one of the primary weaknesses of previous lightweight OSes by providing an environment that is somewhat Linux-compatible and can execute a variety of applications built for Linux.

Deploying multiple operating systems on the same node has been explored previously with SHIMOS [Shimosawa and Ishikawa, 2009], whereby multiple modified Linux kernels can co-exist and manage partitioned hardware resources. However, this project was motivated by considerations such as physical device sharing between co-kernels and thus required significant effort in optimizing cross kernel communication with kernel-level message passing and page sharing, as well as shared resources such as page allocators that required kernel-level synchronization. Our approach is based more fundamentally on the concept of strict isolation between lightweight co-kernels that manage all resources assigned to them without cross dependencies.

Others have looked at deploying replicated kernels on the same node or limiting data sharing in the kernel to improve OS scalability. Hurricane [Unrau et al., 1995] and Hive [Chapin et al., 1995] organize an operating system as multiple independent kernels, which communicates with each other for resource management to provide better reliability and scalability. Barrelfish [Baumann et al., 2009] treats the machine as a network of independent cores assuming no inter-core sharing, and implements the OS services with distributed processes that communicate via messages. The factored operating system [Wentzlaff and Agarwal, 2009] uses space sharing instead time sharing to increase scalability and structures OS services as collections of distributed servers. Corey [Boyd-Wickizer et al., 2008] exposes resource sharing control abstractions and allows the application specify resource sharing requirements

explicitly. K42 [Krieger et al., 2006] and Tornado [Gamsa et al., 1999] uses clustered objects to reduce contention and improve locality.

The most relevant efforts to our co-kernel approach are FusedOS from IBM [Park et al., 2012], mOS from Intel [Wisniewski et al., 2014], and McKernel from the University of Tokyo and RIKEN AICS [Tomita et al., 2014]. FusedOS partitions a compute node into separate Linux and LWK-like partitions, where each partition runs on its own dedicated set of cores. The LWK partition depends on the Linux partition for various services, with all system calls, exceptions, and other OS requests being forwarded to Linux cores from the LWK partition. Similar to FusedOS, McKernel deploys a LWK-like operating environment on heterogeneous (co)processors, such as the Intel Xeon Phi, and delegates a variety of system calls to a Linux service environment running on separate cores. Unlike FusedOS, the LWK environments proposed by mOS and McKernel allow for the native execution of some system calls, such as those related to memory management and thread creation, while more complicated system calls are delegated to the Linux cores. These approaches emphasize compatibility and legacy support with existing Linux based environments, to provide environments that are portable from the standpoint of existing Linux applications. In contrast to these approaches, our co-kernel architecture places a greater focus on performance isolation by deploying co-kernels as fully isolated OS instances that provide standalone core OS services and resource management. In addition, our approach also supports dynamic enclave resource allocation and revocation, with the ability to grow and shrink enclaves at runtime, as well as virtualization capabilities through the use of the Palacios VMM.

2.3 I/O SERVICE DELEGATION

I/O service delegation based approaches have been proposed for driver compatibility and fault isolation purposes [Xen, 2016; LeVasseur et al., 2004; Nikolaev and Back, 2013]. Depending on how the I/O stack is decomposed between the client and server domains, these approaches delegate I/O services at three different abstraction levels: virtual device level [LeVasseur et al., 2004; Xen, 2016], device file level [Amiri Sani et al., 2014], and system call

level [Nikolaev and Back, 2013].

The driver domain [Xen, 2016] approach is the most widely adopted I/O architecture in modern cloud computing [Barham et al., 2003]. In this approach, the hypervisor creates privileged VMs, or driver domains, that directly manage physical hardware. A virtual device is decomposed into a frontend component that resides in the client VM, and a backend component that resides in the driver domain. I/O requests sent to the frontend are forwarded to the backend via event channels, and handled by the Linux device driver layer in the driver domain. The primary advantage of this approach is that it reuses Linux device drivers. However, delegating I/O requests at the device driver layer assumes the client VM has the full I/O stack support above device driver, which is not the case for lightweight kernels. Moreover, delegating I/O requests to a driver domain introduces performance interference due to the shared driver domain. Though the creation of multiple driver domains can mitigate I/O contentions, performance interference can still result from contentions on other shared software components, such as the communication channels as well as the hypervisor.

Different from driver domains, Paradice [Amiri Sani et al., 2014] delegates I/O services at device file level. This approach is designed for the virtualization of devices that are normally accessed directly through the device file interface (e.g. /dev/gpu), such as GPUs and cameras. By delegating file operations on a device file to a remote server VM, I/O operations sent to a virtual device can be processed by a remote Linux VM. The primary advantage of this approach is its generality. Because driver domains require different implementations for each class of devices (e.g. block devices and network devices), while Paradice supports any device that is accessible through the device file interface. However, this approach is not applicable if the devices are accessed through higher level I/O abstractions, e.g. TCP/IP sockets.

VirtuOS [Nikolaev and Back, 2013] decomposes kernel services into several server VMs and delegates I/O requests to remote VMs at the system call level. It adopts an exceptionless system call [Soares and Stumm, 2010] approach to reduce the cost of cross-VM system calls. However, to share data across VMs, it introduces an additional data copy on a shared buffer between the user process the corresponding service domain. While this design sidesteps the potential difficulties with designs that would provide a service domain with direct access to

a user processs memory, it potentially introduces considerable overhead on the data path, especially for large volume data transfers.

Existing works that combine Linux and a lightweight kernel on the same node, including FusedOS [Park et al., 2012], McKernel [Tomita et al., 2014] and mOS [Wisniewski et al., 2014], all adopt a I/O delegation approach that delegates I/O requests from the lightweight kernel to a remote Linux instance. Our work primarily focus on the performance and isolation of the I/O delegation service.

3.0 SYNCHRONIZATION IN VIRTUAL MACHINES

Several studies have established that the average server utilization in most datacenters is low, ranging from 10% to 50% [Barroso et al., 2013; Kaplan et al., 2008; Gartner, 2012; Reiss et al., 2012; Lo et al., 2015]. In particular, Google's typical online-service datacenters exhibit only about 30% CPU utilization on average, as reported in 2013 [Barroso et al., 2013]. The industry wide average utilization is even lower: between 6% [Kaplan et al., 2008] to 12% [Gartner, 2012]. This low utilization problem result in significant computational resource as well as capital waste.

A promising way to improve efficiency is to co-locate multiple virtual machines on the same node in the cloud, such that the available computational resources can be allocated and utilized by other co-located applications, and achieve higher overall system utilization. However, virtual machine (VM) based approaches to workload consolidation, as seen in IaaS cloud as well as datacenter platforms, have long had to contend with performance degradation caused by synchronization primitives inside the guest environments. Among the challenges that virtualization poses to OS designers is the fact that the underlying virtual hardware can be arbitrarily scheduled by the underlying VMM. This has serious consequences for timing sensitive operations in the guest OS, which can be affected by virtual CPU preemptions by the host scheduler and introduce delays that are orders of magnitude longer than those primitives were designed for.

In this chapter, we will focus on the performance problems of *busy-waiting based synchronization operations* in the guest kernel, in particular spinlocks and TLB shootdown operations. We will first demonstrate how busy-waiting based synchronization operations impact guest performance, and pin-point the design defects of state-of-the-art mechanisms. Then we will propose our optimized schemes for virtual environments, specifically the pre-

emptable ticket spinlock algorithm and the Shoot4U paravirtual TLB shootdown scheme. Finally, we will empirically study the performance of the proposed schemes and demonstrate their effectiveness in virtual environments.

3.1 BUSY-WAITING BASED SYNCHRONIZATION IN VIRTUAL ENVIRONMENTS

In this work, we looked at three performance problems related to busy-waiting based synchronization operations: the lock holder preemption problem, the lock waiter preemption problem and the TLB shootdown preemption problem.

3.1.1 The Lock Holder Preemption Problem

Spinlocks are used in the kernel to protect code regions that require atomic execution or exclusive access to the underlying hardware. These regions are generally protected by disabling interrupts and acquiring a spinlock, based on the assumption that the operations will be short in duration and so won't result in long delays for other contending threads.

Spinlocks are heavily used in the kernel as the fundamental synchronization operation. Therefore, they are carefully designed to be simple and fast. Busy-waiting based spinlocks require kernel developers to carefully use spinlocks only in situations where the critical section is short or blocking the current thread is not an option. However, a "short" critical section is based on the assumption that the code is executed natively and the underlying CPU is always making progress, which is not true in virtual environments where *a virtual CPU can be preempted and suspended arbitrarily for an undetermined length of time*. As a result, the *lock holder preemption problem* could happen.

Numerous research works have studied the lock holder preemption problem, originally identified by V. Uhlig et al. in 2004 [Uhlig et al., 2004]. Lock holder preemption occurs whenever a virtual machine's (VM's) virtual CPU (vCPU) is scheduled off of a physical CPU while a lock is held inside the VM's context. The result is that when the VM's other

vCPUs are attempting to acquire the lock they must wait until the vCPU holding the lock is scheduled back in by the VMM so it can release the lock. As kernel level synchronization is most often accomplished using spinlocks, the time spent waiting on a lock is wasted in a busy loop. This wait time is typically orders of magnitude longer than the designed wait time, and could result in significant performance degradation.

3.1.2 The Lock Waiter Preemption Problem

While most of the previous works have focused on the lock holder preemption problem, in this work, we identified the *lock waiter preemption problem* that is associated with a queue based fair spinlock algorithm named ticket spinlocks.

Ticket spinlocks are a relatively recent modification to the global spinlock architecture found in Linux. Introduced in kernel version 2.6.25, ticket spinlocks are designed to improve lock fairness and prevent starvation. The lock is always granted to the next lock waiter in the queue, thus guaranteeing that locks are dispatched in FIFO order and no thread will ever experience starvation. In particular, when a thread attempts to acquire a ticket spinlock that is currently held by another thread, it is granted a ticket which determines the order among all outstanding lock requests. In this manner each thread must wait to acquire a lock until after it has been held by every other thread that previously tried to acquire it. This ensures that a given thread is never preempted by another thread while trying to acquire the same lock, and thus guarantees that well behaved threads will all acquire the lock in a timely manner.

While ticket spinlocks have been shown to provide advantages to performance and consistency for native OS environments, they pose a new challenge for virtualized environments. This is due to the fact that besides the lock holder preemption problem, a preempted waiter in a FIFO queue can cause similar or even worse performance degradation. Restricting lock acquisitions to a FIFO schedule expands the lock holder preemption problem by creating an environment where anyone with an earlier position in a lock's queue is effectively holding the lock as far as threads later in the queue are concerned. Thus, when executing inside a virtual machine environment, if a vCPU currently holding a ticket is preempted, all subse-

quent ticket holders must wait for the preempted vCPU to be rescheduled. This can result in execution being blocked by lock contention even when the lock in question is available. For example, if the previous lock holder released the lock, later waiters in queue would still have to wait for the previous preempted waiter to be rescheduled and granted the lock, resulting in a scenario where there is contention over an idle resource. In other words, any waiter in queue is preempted the progress is whole queue would be blocked. We denote this situation as the *lock waiter preemption problem.*

3.1.3 The TLB Shootdown Preemption Problem

Besides the spinlock preemption problems including the lock holder and waiter preemption problems, in this work, we also looked the performance problem of another busy-waiting based synchronization operation: the TLB shootdown operation.

Translation Look-aside Buffers (TLBs) are a critical hardware component for virtual memory based systems, however they still require explicit management by the Operating System (OS) in order to maintain cache coherence. This requires the OS itself to directly manage the contents to the TLB caches on each CPU core in the system by ensuring that stale entries are removed before they can be accessed by any hosted applications. This is especially a problem for multi-threaded applications as they leverage shared page tables both as a space saving optimization as well as a way to amortize address space management overheads. Cache coherence is managed by the OS through the use of invalidation and flushing operations that remove one or more entries from a local TLB cache. The operations are propagated to other cores in the system via Inter-Processor Interrupt (IPI) based signalling that directly invoke a given TLB operation on a remotely targeted CPU, a mechanism that is canonically referred to as a TLB shootdown.

Modern operating systems consider TLB shootdown operations to be performance critical and so optimize them to exhibit very low latency. The implementation of these operations is therefore architected to ensure that shootdowns can be completed with very low latencies through the use of IPI based signalling. As such, TLB shootdowns can be implemented using a busy wait based stall of the invoking CPU while the operation is handled on each of

the remote CPUs. Unfortunately, the low latency provided by IPI handlers is only ensured when the target CPU is available to handle the resulting interrupt. While this is generally a reasonable assumption in native environments, it does not carry over to virtualized environments as the availability of a given vCPU to handle interrupts is entirely dependent on the behavior of the underlying host scheduler.

The *TLB shootdown preemption problem* is the result of a vCPU invoking a TLB shootdown on a remote vCPU that is currently preempted by the host scheduler and therefore not available to handle the resulting IPI interrupt. In this case the invoking vCPU will block in a polling based loop until the target vCPU is rescheduled and returns to an active state. The scheduling delays, or the time between the preemption and rescheduling of a vCPU, are often orders of magnitude larger than the latency that TLB shootdown operations were designed for. This is especially true when multiple vCPUs are sharing the same underlying physical CPU. These unexpected delays can cause significant impacts on application performance depending on the workload, with particularly dramatic effects seen on multi-threaded workloads that require large amounts of address space modifications.

3.2 PERFORMANCE ANALYSIS

In order to better understand the effects that vCPU preemptions have on kernel synchronization operations, we measured the performance degradation caused by CPU overcommitment on the PARSEC [Bienia, 2011] benchmark suite. We choose the PARSEC benchmark suite because it includes representative multi-threaded emerging workloads, including both desktop and server applications as can be seen in cloud computing platforms. Besides, the PARSEC benchmark suite has also been used in several related works [Kim et al., 2013; Ding et al., 2014].

We used the Linux perf [Perf, 2016] tool set to profile the percentage of CPU time spent in various kernel functions in order to identify the performance bottlenecks. Then we conducted detailed performance instrumentation on the major problematic kernel operations we identified.

We first measured the PARSEC performance using a single KVM based guest without overcommitting resources (1-VM), before adding a second KVM guest on the same machine running a CPU bounded workload using sysbench [Sysbench, 2016] (2-VMs). Each VM was configured to use the same number of vCPUs (12) evenly distributed on the underlying physical CPUs, so that each physical core was shared by two vCPU cores from different VMs. Equal time sharing between vCPUs was ensured using Linux cgroups [Cgroups, 2016] and the Pause-Loop Exiting (PLE) [Riel, 2011] feature was disabled.

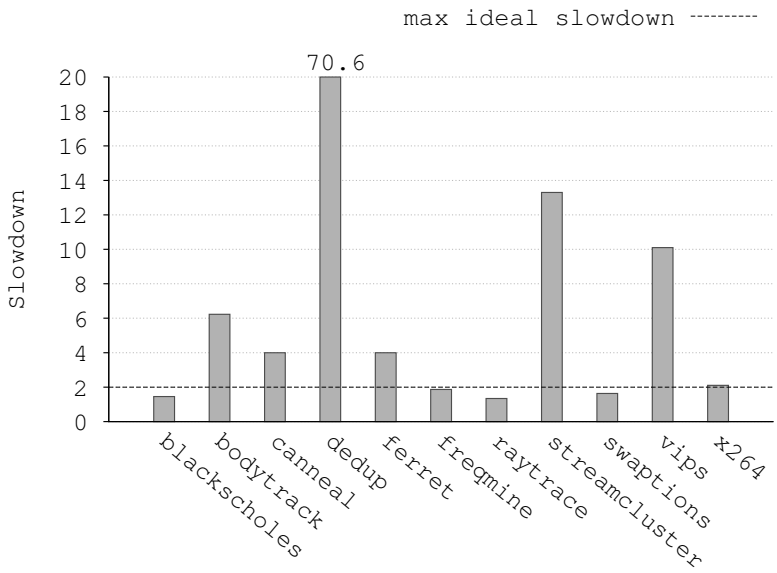


Figure 2: Performance Slowdown of CPU Overcommitment

Figure 2 shows the benchmark results for the 1-VM and 2-VMs configurations. The ideal slowdown would be 2x, due to the equal time sharing configuration of each physical CPU. As the results show 6 out of the 11 benchmarks have performance slowdowns of over 4x; 3 exhibit more than 10x slowdown; and the dedup benchmark has a slowdown of 70.6x.

For each of the applications we separated out the overheads resulting from TLB shootdowns (k:tlb) as well as spinlocks (k:lock), being the two most common causes of preemption based performance problems. The remaining overhead was split between other kernel level functions (k:other) and time spent in userspace (u:*). Figure 3 shows these results: with the exception of dedup, all benchmarks spent the majority of time in userspace for the 1-VM scenario, indicating that most PARSEC benchmarks are userspace intensive work-

loads. However, for the 2-VMs case a significant number of benchmarks exhibit noticeable increases in kernel based overheads. In particular, dedup spent 64% of the CPU time on TLB shootdown operations, while vips 75% of the CPU time on spinlocks. As the results show, overheads resulting from spinlocks and TLB shootdowns account for the majority of the added overhead.

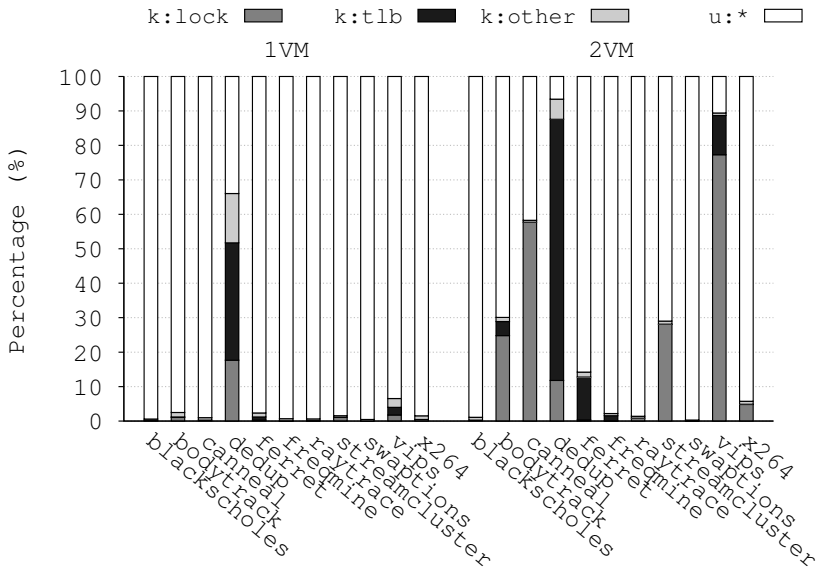


Figure 3: CPU Usage Profiling

For the spinlock performance problems in particular, we instrumented the Linux ticket spinlock to profile the number of lock holder preemptions and lock waiter preemptions. Lock preemption was identified by detecting inordinately long wait times for a given lock, where "long wait times" were conservatively chosen to be 2048 iterations of the inner loop of a busy waiting spinlock. On our machine, 2048 iterations corresponded to roughly 1μs, an amount of time that exceeds the time a thread would spend holding a lock according to statistics [Friebel, 2008]. Next we separated the lock waiter preemption scenarios from the set of detected preemptions, by checking whether the stalled lock was in fact available. To make this determination we modified the existing spinlock structure to include a `holder_id` variable that served as an indicator of lock availability. The value of `holder_id` was set to the thread id of a given lock holder on acquisition and cleared when the lock was released.

Table 1 shows the results of our analysis after running the hackbench [Hackbench, 2008]

	N	$N_{holder} + N_{waiter}$	N_{waiter}	$\frac{N_{waiter}}{N_{holder}+N_{waiter}}$
1 VM	$1.11E8$	1,089	452	41.5%
2 VMs	$9.65E7$	44,342	39,221	88.5%

Table 1: Profiling of Lock Holder and Waiter Preemptions

benchmark with 1 and 2 VMs, in which N is the total number of spinlock acquisitions, N_{holder} and N_{waiter} are the total number of holder and waiter preemptions respectively. Column 3 shows the number of preemptions in which lock acquisitions were delayed because of either lock holder or lock waiter preemption. While these delays were infrequent when compared to the total number of lock acquisitions in column 2, it should be noted that even a limited degree of preemption can cause significant performance degradation [Friebel, 2008]. Furthermore, while the total number of lock acquisitions declined when additional VMs were added, the number of preemptions resulting in stalled lock acquisitions actually increased. More critically, the stalled lock operations were predominately due to a preempted lock waiter and *not* a preempted lock holder. The degree of the issue is shown more clearly in column 4, which provides the percentage of stalled lock acquisitions resulting from a preempted lock waiter. As can be seen, even when a physical machine is overcommitted by a factor of only 2, lock waiter preemption becomes the dominant source of synchronization overhead.

Next, we measured the latency of TLB shootdown operations when running the dedup benchmark, by instrumenting the Linux kernel using ktap [Ktap, 2016]. Figure 4 shows the cumulative distribution function (CDF) of TLB shootdown latencies using a logarithmic x-axis. As the figure shows, the 2-VMs case exhibits a significant increase in the average operation latency with the 90th percentile increasing by two orders of magnitude over the 1-VM configuration. It demonstrated the dramatic slowdown on TLB shootdown operations caused by sharing the physical CPUs.

In summary, our performance analysis results indicates that: firstly, equally sharing the physical CPUs between two VMs would not result in half of the performance in each VM;

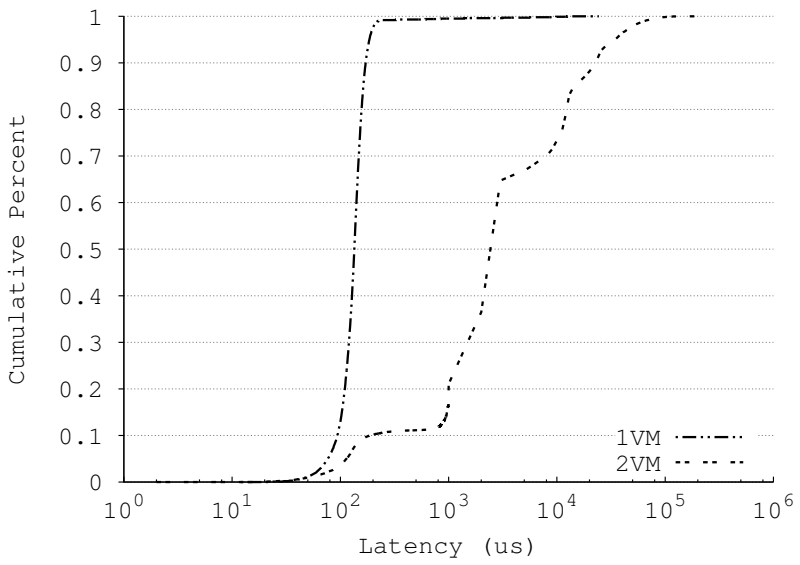


Figure 4: CDF of TLB Shootdown Latency

instead, up to 70.6 times slowdown was observed in the PARSEC benchmark suite. Secondly, item The majority of the slowdown comes from kernel performance degradation, especially the spinlock and TLB shootdown operations according to profiling. Thirdly, lock waiter preemption is the dominant source of spinlock preemption overhead when multiple VMs are deployed. Lastly, the 90th percentile latency of the TLB shootdown operation increased by two orders of magnitude when a second VM is deployed.

The rest of this chapter will focus on how to address the two major performance problems when over-subscribing CPUs, specifically the lock waiter preemption problem and the TLB shootdown preemption problem.

3.3 THE PREEMPTABLE TICKET SPINLOCK ALGORITHM

With the use of FIFO order spinlocks, i.e. ticket spinlocks, virtual CPUs must also contend with lock waiters being preempted before they are able to acquire the lock. This has the

effect of blocking access to a lock, even if the lock itself is available. This problem exists whenever a VMM preempts a waiter that has not yet acquired the lock. In this case even if the lock is released, no other thread is allowed to acquire it until the next waiter is allowed to run, resulting in a scenario where there is contention over an idle resource. We denote this situation as the *lock waiter preemption* problem. In order to solve this problem, we introduce the *preemptable ticket spinlock* (pmtlock), a new locking primitive that is designed to enable a VM to always make forward progress by relaxing the ordering guarantees offered by ticket spinlocks.

Preemptable ticket spinlock improves the performance of traditional ticket spinlock by allowing preemption of a waiter that has been detected to be unresponsive. Unresponsiveness is determined via a linearly increasing timeout that allows earlier waiters a window of opportunity in which they can acquire a lock before the lock is offered to later waiters. It can provides performance benefits when running without VMM support, while it can also utilize VMM support via a specialized paravirtual interface to provide overall superior performance.

Preemptable ticket spinlocks are based on the observation that forward progress is preferable to fairness in the face of contention. In the case where lock waiter preemption is preventing a guest from acquiring a lock, then a thread waiting on the lock should be able to preempt the next thread in line if that thread is incapable of acquiring the lock in a reasonable amount of time. While preemptable ticket spinlocks do allow preemption, which technically breaks the ordering guarantees of standard ticket spinlocks, it does so in a way that minimizes the loss of fairness by always granting priority to earlier lock waiters. Priority is granted via a time based window which gradually increases the number of ticket values capable of acquiring the lock. The choice of a time based window is based on the observation that VMM level preemption typically results in large periods of unresponsiveness, while other causes of unresponsiveness typically result in periods orders of magnitude smaller. This means that a timeout based detection approach can be implemented with high accuracy and relatively low overhead. The use of timeouts also allows the implementation of linearly expanding exclusivity windows, which ensure that if an earlier ticket holder is able to acquire a lock it will do so before a later ticket holder is offered the chance. In this way early ticket holders are only preempted if they are inactive for a long period of time, almost

always as the result of the vCPU being preempted by the VMM.

3.3.1 Design

Preemptable ticket spinlocks is a hybrid spinlock architecture that combines the features of both ticket and generic spinlocks in order to preserve the fairness of ticket spinlocks while avoiding the lock waiter preemption problem. The intuition behind it is that making forward progress is more important than ensuring fairness. It leverages the advantages of both generic spinlocks and ticket spinlocks in order to ensure fairness in the absence of preemption while also supporting out of order lock acquisition when the waiters in the queue are preempted. This is done via the use of a *proportional timeout threshold* that determines the ability of a thread to acquire a lock based on that thread's position among the set of threads currently waiting on the lock. In preemptable ticket spinlocks, a thread can acquire a lock out-of-order if it has been waiting longer than its *timeout threshold*. We denote such a thread as a timed-out waiter. The timeout threshold is calculated from a *unit timeout threshold* that is multiplied with the thread's current lock queue position index n, as shown in the following equation:

$$timeout\_threshold = n \times \tau \tag{3.1}$$

in which τ denotes the unit timeout threshold parameter, which tunes the fairness and performance of the preemptable ticket spinlock algorithm as we will discuss later.

To calculate the position index value n, two variables are maintained for each lock: `num_request` indicates the total number of lock requests of a lock, and `num_grant` indicates the total number of lock requests that have been granted. In addition, each thread has a local variable named `ticket`, which represents the queue position of the request. `num_request` and `num_grant` are maintained by each thread in a distributed fashion for each lock according to the locking algorithm. When acquiring a lock, the current `num_request` value is stored into the thread's local `ticket` variable, and then atomically incremented by 1. Conversely, when releasing a lock, `num_grant` is atomically incremented by 1. The position index n is then calculated based on a thread's `ticket` value as well as the current value of `num_grant`,

as shown below:

$$n = ticket - num\_grant \tag{3.2}$$

It indicates the number of waiters for a given lock before the current thread. Because `ticket` stores the number of outstanding lock requests at the request time, and `num_grant` contains the number of lock requests that have been granted, we can determine the number of pending requests before current thread (and thus the thread's queue position) as `(ticket - num_grant)`. Note that it is possible for a thread to have a negative position in the queue (`ticket` $<$ `num_grant`). This can result from whenever a lock has been preemptively acquired and the preempted core is then rescheduled at a later point in time. In this case a negative position indicates that later threads have violated the lock order, in which case the preempted thread should attempt to acquire the lock immediately.

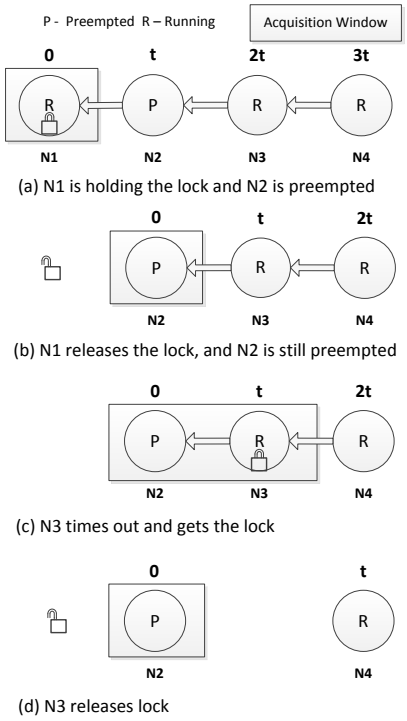


Figure 5: Illustration of Preemptable Ticket Spinlock Algorithm

Figure 5 provides an illustrative example of the functionality of preemptable spinlocks. The timeout threshold is indicated by the number above each node, and is set proportionally

based on the node's position in the queue. In the initial stage (a), four nodes are waiting while $N1$ is holding the lock. At this point the vCPU hosting $N2$ is preempted by VMM. In the following stage (b), $N1$ releases the lock, causing the timeout threshold to be updated for each node. At this point the lock is available but no node can acquire it because the next waiter in the queue $N2$ is currently preempted. This is the lock waiter preemption problem. In stage (c) node $N3$ reaches the timeout threshold and acquires the lock out-of-order before $N2$. Finally at stage (d), $N3$ releases the lock, causing $N4$ to update it's timeout threshold. At this point, $N4$ has still not reached the timeout threshold, so $N2$ is able to immediately acquire the lock without contention.

3.3.2 Properties

3.3.2.1 Preemption Adaptivity Preemptable ticket spinlocks are a hybrid lock algorithm that combines the benefits of generic spinlocks and ticket spinlocks by relaxing the ordering guarantees provided by ticket spinlocks. The underlying feature of preemptable ticket spinlocks is a timeout threshold that controls when a given waiter can acquire the lock out of order. The timeout threshold is derived from a tunable constant named *unit timeout threshold* denoted as τ, combined with a waiter's queue position index n. The behavior of a preemptable ticket spinlock can be tuned to match the behavior of either a generic spinlock, a ticket spinlock, or a combination of the two depending on the value assigned to τ. The following equation shows the behavior of preemptable ticket spinlocks for different values of τ.

$$lock = \begin{cases} generic \quad spinlock & \tau = 0 \\ pmtlock \quad ticket \quad spinlock & 0 < \tau < \infty \\ ticket \quad spinlock & \tau = \infty \end{cases} \tag{3.3}$$

A τ value of 0 results in an immediate timeout that mimics the behavior of a generic spinlock, while setting $\tau = \infty$ will prevent a timeout from ever occurring and so generate the strict ordering behavior of a standard ticket spinlock. Preemptable ticket spinlocks are thus able to tune their behavior by trading off between aggressiveness and fairness depending on the state of the system and the behavior of the underlying VMM scheduler.

A well chosen τ value can provide both good performance and fairness. Fairness is ensured when τ is large enough that lock waiters will not time out prematurely. Performance is ensured when τ is small enough that a lock waiter is able to promptly detect when an earlier waiter is preempted. According to previous work [Friebel, 2008], the lock holding time and preemption time in fact differ by orders of magnitude. Typically lock holding time is less than $1\mu s$, while the time between a vCPU's preemption and rescheduling is typically in the order of millisecond. A good practice it to choose a value of τ that is slightly larger than typical lock holding time, e.g. around $2\mu s$, so that it is large enough to preserve fairness in the absence of preemption and small enough to quickly respond to preemptions. An empirical study of this parameter will be provided in the evaluation section.

3.3.2.2 Fairness The standard technique for achieving fairness is to ensure that locks are granted in the same order in which the requests were made, i.e. FIFO ordering. With generic spinlocks, all waiters have an equal chance of acquiring a lock, regardless of when the waiter first requested it. This makes generic spinlocks an "unfair" locking algorithm. Ticket spinlocks implement strict ordering that enforced via the use of tickets assigned consecutively to new lock requests. An earlier waiter with smaller ticket value always get the lock before a waiter that requested the lock at a later point in time.

In contrast to these locking behaviors, preemptable ticket spinlocks ensure that,

- For all waiters yet to reach their timeout threshold, strict ordering is preserved
- Waiters that have reached their timeout threshold have priority over those who have not
- All waiters that have reached their timeout threshold have equal priority among themselves

The first point holds because the timeout threshold is proportional to a waiter's queue position. Earlier waiters have smaller thresholds, and thus they time out earlier than those who are later in the queue. The second point holds because according to equation 3.2, the position index n of a non-timed-out waiter is larger or equal than the number of timed out waiters. Thus all non-timed-out waiters have timeout thresholds no less than $x \times \tau$, where x is the number of timed-out waiters . With a proper value of τ the threshold is large enough

for every timed out waiter to complete their critical sections in the absence of preemption. In other words, every thread that has reached its timeout threshold will have time to acquire and release the lock before the next waiter times out. This ensures that priority is given to a preempted waiter immediately after it is rescheduled, and furthermore all non-timed-out waiters will wait until every thread that has timed out has acquired the lock. Thus while ordering is violated, the violations are minimized to those vCPUs that have been preempted by the VMM.

Based on the description above, it is straight forward to show that the number of ordering violations experienced by a given lock is bounded by the number of vCPUs assigned to a VM. Furthermore, the probability that a preempted waiter is unable to immediately acquire the lock after rescheduling is given by $P(x) = x/R$, where x is the number of lock acquisitions that have occurred since a lock waiter was preempted, and R is the number of outstanding waiters that have been preempted.

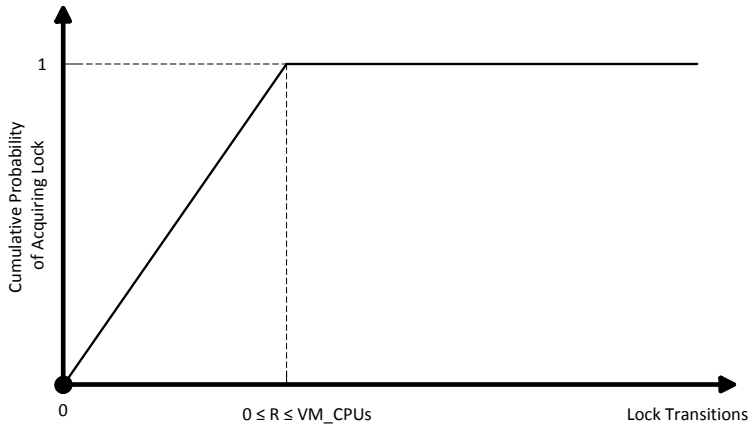


Figure 6: CDF of Getting the Lock after Rescheduling

Thus for a simple case, where a set of waiters are preempted and rescheduled simultaneously, we can derive the cumulative density function shown in Figure 6. The x-axis represents the number of lock acquisitions necessary before the preempted waiter is able to acquire the lock. R is the number of preempted waiters currently contending for the lock. VM_CPUS is the number of vCPUs assigned to the VM.

From this figure we can see that the probability that a preempted waiter has acquired the lock increases linearly based on the number of waiters that were simultaneously preempted by the VMM. The number of ordering violations is limited by the number of lock waiters preempted by the VMM at any given point at time, and in the worst case is bounded to the number of active vCPUs assigned to the VM. While more dynamic scheduling cases will alter the shape of the CDF, they will not change the worst case bounds.

It is important to note that the above discussion holds for a lock waiter preemption case in which the time spent holding a lock is less than the unit timeout threshold τ. In the case of lock holder preemption, all lock waiters will time out due to the fact that preemption time is considerably larger than the timeout threshold. Lock holder preemption represents the worst case in regards to fairness, in that every waiter in the queue will reach its timeout threshold and compete equally for the lock. In this case our approach will degenerate to a generic spinlock behavior. However the order violation will still be bounded by the number of vCPUs.

3.3.2.3 Host Independence Unlike previous solutions [Friebel, 2008; Raghavendra and Fitzhardinge, 2012; Riel, 2011; Sukwong and Kim, 2011; Uhlig et al., 2004; Wells et al., 2006; Weng et al., 2011; Zhang et al., 2012], preemptable ticket spinlocks work in the absence of any VMM side support, which makes it a solution where host side modifications such as paravirtualization are not feasible. Preemptable ticket spinlocks can be implemented entirely inside a guest OS and are capable of detecting lock waiter preemption adaptively based on a single timeout directly measurable by the guest. However, while preemptable ticket spinlocks are capable of operating independently without VMM support, it is possible to further improve their performance by combining them with existing lock holder preemption solutions.

3.3.3 Implementation

In order to evaluate the efficacy of preemptable ticket spinlocks, we have implemented them inside version 4.0.0 of the Linux kernel. Our implementation acts as a drop in replacement

for the standard ticket spinlock implementation currently supported by the kernel. The implementation consisted of only ∼60 lines of C and assembly code, and consists of a modified spinlock data type as well as modifications to the lock, unlock, islocked and trylock operations. The implementation resides entirely in the guest kernel and does not require any additional VMM side support in order to function correctly.

Code 3.1: Lock and Unlock Operations

```
1  #define TIMEOUT_SHIFT 14
2  void __ticket_spin_lock(arch_spinlock_t *lock)
3  {
4    register struct __raw_tickets inc={.tail=1};
5    unsigned int timeout = 0;
6    __ticket_t current_head;
7
8    inc = xadd(&lock->tickets,inc);
9    if (likely(inc.head == inc.tail))
10     goto spin;
11
12   timeout = (__ticket_t)(inc.tail - inc.head)
13       << TIMEOUT_SHIFT;
14   do {
15     current_head =
16         ACCESS_ONCE(lock->tickets.head);
17     if (inc.tail == current_head) {
18       goto spin;
19     } else if (inc.head != current_head
20       && ((s16)(inc.tail - current_head) > 0)) {
21       inc.head = current_head;
22       timeout = (__ticket_t)(inc.tail - inc.head)
23           << TIMEOUT_SHIFT;
24     }
25     cpu_relax();
26   } while (timeout--);
27
28 spin:
29   for (;;) {
30     if (xchg(&lock->acquired, 1) == 0)
31       goto out;
32     cpu_relax();
33   }
34 out: barrier();
35 }
36
37 void __ticket_spin_unlock(arch_spinlock_t *lock) {
38   __add(&lock->tickets.head, 1, UNLOCK_LOCK_PREFIX);
39   xchg(&lock->acquired, 0);
40 }
```

Code 3.1 shows the implementation of *lock* and *unlock* operations. As part of our modifications we added code to maintain the timeout threshold, while also changing the semantics of some of the existing data fields. The existing kernel spinlock data structure contains variables to track the head and tail of a queue in order to implement the proper ticket semantics. In order to detect a preempted lock waiter we have added another field named `acquired` which indicates the availability of the lock.

In the lock function, we declare a local struct `inc` which acts as a local copy of the `head` and `tail` values, `timeout` which is used as the timeout threshold, and `current_head` which is another local copy of `head` used to detect changes to the value of `head`. At line 8 the code atomically updates `inc` in order to increase the value of `head`. At this point `inc.tail` is regarded as the "ticket" of the current thread. Line 9-10 shows the fast path, which handles the case of an uncontended lock acquisition. Lines 12-26 implement the core of the proportional timeout functionality. The timeout threshold is initialized in line 12, and updated in line 22 whenever `head`'s value changes. This ensures that the timeout threshold is always proportional to the number of pending lock requests that arrived previously, according to equation 3.2. A timed out thread will break out of the loop at line 26. A thread should also break out of the loop when it's ticket is equal to the current head, meaning that it is due to acquire the lock based on the ticket ordering. Besides, a thread breaks out of the loop if it's ticket is equal to the current value of `head`. Finally, line 29–34 implement a generic spinlock which is invoked by every thread that is allowed past the FIFO order queue.

It is important to note that when calculating the timeout threshold using $(tail - head) \times \tau$, we need to take care of the *integer wrap-around problem* as shown in line 12. Because *tail* and *head* are unsigned integers, they could wrap-around when reaching the maximum value given their size. Consider the case where *head* is approaching the maximum value of its size and tail is wrapped-around, then *tail* is smaller than *head*; $tail - head$ would result in a negative value that is converted to a very large unsigned integer because the timeout threshold is an unsigned integer with larger size. A very large timeout threshold would result in starvation. To correctly handle this case, we have to cast $tail - head$ to a signed value with the same size of *head* and *tail*. The C99 standard guarantees the correctness of the subtraction of unsigned values.

Another important note is that *dynamic timeout threshold update* is used in this implementation. A potential problem is that a preempted waiter's ticket (*tail*) is possible to be less than the current *head*, it could result in starvation if not handled correctly. For example, if a waiter with ticket number *tail* is preempted, preemptable ticket spinlock allows later waiters to timeout and acquire the lock out of order. Each lock release operation of these waiters would increase the *head* value by 1. Eventually, the *head* would become greater than *tail*. As a result, when dynamically updating the timeout threshold once the preempted waiter is rescheduled, the result of $tail - head$ is again a negative value that is converted to a very large unsigned integer. Consequently, this waiter get a very large timeout threshold and results in a starvation. To fix to this problem, we only update timeout threshold if the ticket is larger than the current head as shown in line 20.

The unlock operation is relatively simple, and is implemented by combining the unlock operations of both ticket and generic spinlocks. The operation atomically increments `head` by 1 and clears the `acquired` flag.

Code 3.2: isLocked and Trylock Operations

```
1  int __ticket_spin_is_locked(arch_spinlock_t *lock) {
2    struct __raw_tickets tmp = ACCESS_ONCE(lock->tickets);
3    return (tmp.tail != tmp.head)
4    || (ACCESS_ONCE(lock->acquired)==1);
5  }
6
7  int __ticket_spin_trylock(
8          arch_spinlock_t *lock) {
9    arch_spinlock_t old, new;
10   *(u64 *)&old = ACCESS_ONCE(*(u64 *)lock);
11   if (old.tickets.head != old.tickets.tail)
12     return 0;
13   if (ACCESS_ONCE(lock->acquired) == 1)
14     return 0;
15   new.head_tail = old.head_tail +
16       (1 << TICKET_SHIFT);
17   new.acquired = 1;
18   /* cmpxchg is a full barrier */
19   if (cmpxchg((u64 *)lock, *(u64 *)&old,
20         *(u64 *)&new) == *(u64 *)&old) {
21     return 1;
22   } else return 0;
23 }
```

While the lock and unlock operations provide the necessary functionality for basic locking, the Linux kernel also requires additional locking semantics for certain cases. In particular Linux makes consistent use of other spinlock primitives such as islocked and trylock. In order to fully support preemptable ticket spinlocks through the kernel, we had to modify these operations as well. Code 3.2 shows the implementation of these primitives. At line 3 our code modifications return true if it detects the presence of earlier waiters for the lock or if the lock is currently not available. The trylock operation attempts to acquire a given lock, but immediately returns 0 if the lock is not available. In order to support preemptable ticket spinlocks we modified the implementation at lines 19-20, where we added an atomic check to determine whether the lock is free and if there are no earlier waiters. Other than these minimal changes, the existing implementations were left as originally written.

Next, we will discuss two optimizations that could be applied to reduce lock size, simplify the implementation and improve performance.

3.3.3.1 Compact Locks The implementation described above increased the lock size by introducing a lock availability field. It also introduces one more atomic operation in both the lock and unlock operations to maintain the lock availability field. Considering the wide usage of spinlocks in kernel data structures, as well as the importance of spinlocks to the overall system performance, these overheads can cause measurable kernel size increase and system level performance slowdown, especially when the system is under-committed and preemptable ticket spinlock does not have a significant advantage over ticket spinlock.

The idea of the compact preemptable ticket spinlock optimization is to consolidate the lock availability field with the tickets fields, by using the lowest significant bit (LSB) of the tickets fields as the lock availability flag. Specifically, if we increase the ticket number by 2 instead of 1 when requesting a lock, we can reserve the lowest bit to represent availability of the lock. In this way, we can consolidate the queuing info and lock availability info into the same bytes without increasing the lock size. Meanwhile, this approach simplifies the unlock operation to atomically add 1 to the lock, which clears the lowest bit and advances the tail number with one atomic operation. This optimization should keep the size of preemptable ticket spinlocks the same with ticket spinlock, and reduce the overhead of unlock.

3.3.3.2 Static Timeout Threshold

Our next optimization tries to simplify the maintenance of timeout threshold by using a static timeout threshold, i.e. calculate the timeout threshold once upon lock request, and do not update the threshold with the dynamic of the FIFO queue. A static timeout threshold would respond to lock waiter preemptions slower compared to dynamic thresholds, because the timeout thresholds are not updated with a thread's most recent position in queue, which would result in longer spinning. However, as we have previously discussed, dynamic timeout threshold update makes it more difficult to reason the correctness of the locking algorithm. Therefore, we propose to simply calculate the timeout threshold once upon lock request. As we will show in the evaluation section, a simplifed implementation can achieve the same level of fairness and performance compared with the basic implementation.

3.3.4 Evaluation

We evaluated the preemptable ticket spinlock algorithm on a dual socket Dell R450 server configured with Intel "Ivy-Bridge" Xeon processors (6 cores each) with hyperthreading enabled and 24 GB of RAM split across two NUMA domains. Each server was running CentOS 7 with Linux Kernel 3.16. We performed the evaluation using 2 separate VMs each with 12 vCPUs both mapped to the same socket. Each vCPU was pinned to a single hyperthreaded CPU core, so that each core was shared by 2 vCPUs. The Linux cgroups [Cgroups, 2016] interface was used to allocate an equal share of CPU time to each VM. We used CentOS 7 with Linux Kernel 3.16 as the guest OS in both VMs, and modified the guest kernel to implement our proposed algorithms. We deployed the benchmarks in one VM and measured their performances, while ran a CPU bounded competing workload based on sysbench [Sysbench, 2016].

Previously, we discussed two optimizations on top of the basic preemptable ticket spinlock algorithm, namely the compact preemptable ticket spinlock and the static timeout threshold. The goal of the experiments in this section is to evaluate the basic implementation, as well as the effectiveness of the two optimizations. In the following experiments, we denote the basic preemptable ticket spinlock implementation as *pmt-basic*, the implementation with the

compact optimization as *pmt-cpt*, and the implementation with both the compact and the static timeout threshold optimizations as *pmt-cpt-st*.

3.3.4.1 Parameter and Fairness Analysis In Section 3.3.2, we theoretically analyzed how does the *unit timeout threshold* parameter affect the behavior, performance and fairness of the preemptable ticket spinlock algorithm; the goal of our first experiment here is to empirically evaluate the impact of the unit timeout threshold parameter on the performance and fairness of the preemptable ticket spinlock and its variants.

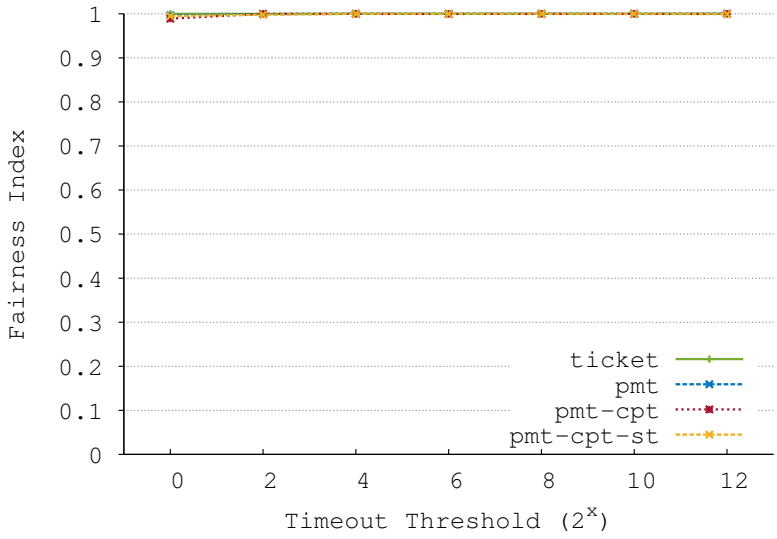


Figure 7: Fairness Index v.s. Unit Timeout Threshold (1VM)

Our metric of spinlock fairness is based on the assumption that for a strict fair lock, if N threads request the same lock in parallel from N cores for a sufficient amount of times, the number of iterations each thread gets the lock should be a *uniform distribution*. Therefore, we can compare the fairness of spinlock algorithms by measuring uniformness of the lock acquisition distribution. In this work, we use the Jain's fairness index [Jain et al., 1984], i.e. $(\sum x_i)^2/(n\sum x_i^2)$, to measure lock fairness. In the Jain's fairnes index, the distribution of the resource is more fair with the fairness index getting closer to 1. To measure the fairness index, we developed a synthetic kernel spinlock fairness benchmark named *fairbench*, which creates N kernel threads in parallel on N cores competing for the same spinlock for a total number

of M iterations, and reports the fairness index. Specifically, we set $N = 12, M = N \times 10^5$ in our experiment.

We use synthetic benchmark instead of instrumenting an application benchmark because instrumentation introduces extra latency to kernel spinlocks. Therefore, we cannot instrument and measure the lock acquisition distribution or fairness index without altering the timing of spinlocks and the locking behavior of application threads. However, by using a synthetic benchmark, we can measure lock acquisition distribution inside the benchmark without instrumenting kernel spinlocks, which gives us more actuate fairness results.

Figure 7 shows the *fairness index* when one VM is deployed on the hardware (1VM). The x-axis shows the unit timeout threshold in log scale, thus the actually unit timeout threshold is 2^x times of busy-loops. The y-axis shows the fairness index, higher is better. The ticket spinlock algorithm is used as the baseline and denoted as ticket. As can be seen, preemptable ticket spinlock variants (pmt*) show comparable fairness with the fair ticket spinlock algorithm under variance settings, which demonstrates that preemptable ticket spinlocks can provide good fairness in the absence of preemptions.

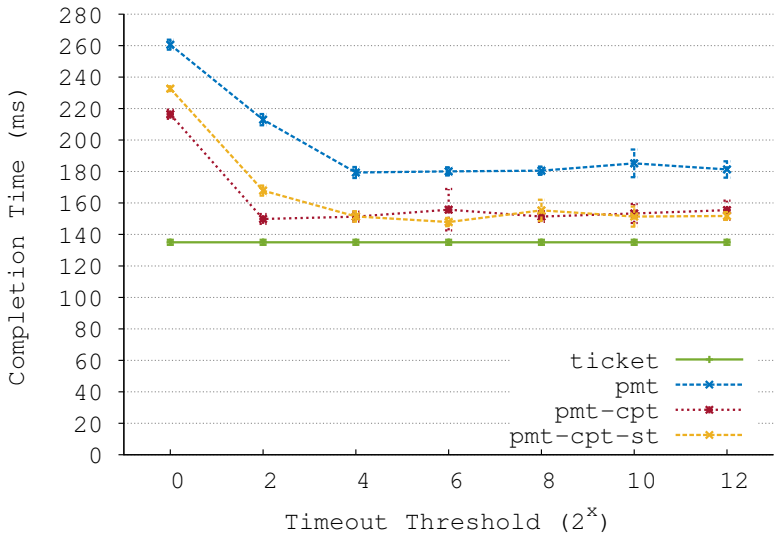


Figure 8: Fairbench Performance v.s. Unit Timeout Threshold (1VM)

Figure 8 shows the *completion time* of the fairbench benchmark with various unit timeout thresholds. The x-axis shows the unit timeout threshold in log scale, and the y-axis shows

the completion time of the benchmark in milliseconds. The ticket spinlock achieves the best performance because preemptable ticket spinlocks introduce overheads to maintain the lock availability information; and in the absence of preemptions, preemptable ticket spinlocks do not have much room to improve system performance. Therefore, this experiment is a "worst case" scenario for preemptable ticket spinlocks showing its overhead. However, note that this is the performance of a synthetic benchmark consists of purely lock and unlock operations, and the overhead for general applications should be much lower as we will show in later experiments.

On the other hand, comparing the preemptable ticket spinlock variants, locks with the compact optimization (pmt-cpt and pmt-cpt-st) show less overhead compared with pmt-basic. The reason is that the compact optimization reduces the number of atomic operations needed to maintain the lock availability flag. Besides, pmt-cpt-st shows comparable performance with pmt-cpt, demonstrating that the static threshold update optimization sustains the performance and fairness while simplifies the implementation.

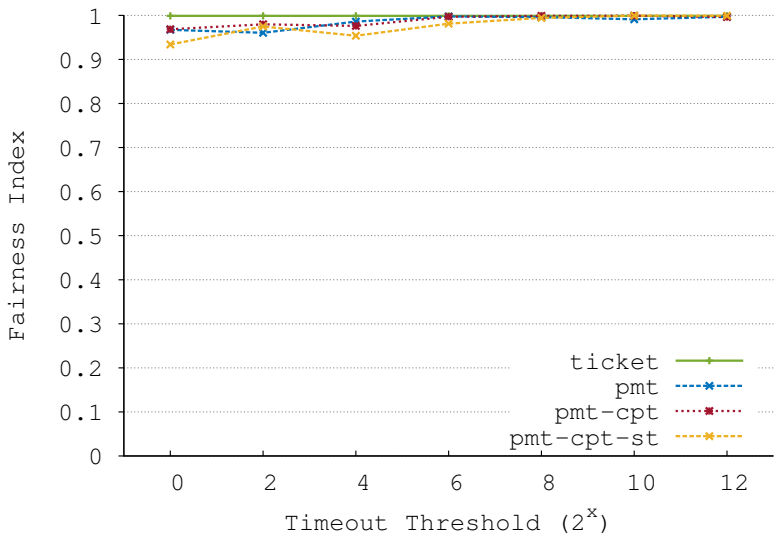


Figure 9: Fairness Index v.s. Unit Timeout Threshold (2VM)

Next, fairbench results in a 2VM case are shown, where two virtual machines with equal CPU shares are deployed on the same set of physical CPUs. Virtual CPUs (vCPU) are pinning on different physical CPUs, such that each physical CPU is shared by two vCPUs

from different VMs. Figure 9 reports the fairness index measured from one of the two VMs running fairbench, while the other VM was running a CPU intensive workload using the sysbench [Sysbench, 2016] benchmark.

The fairness properties enforced by various spinlock implementations are different as revealed in Figure 9, especially under small unit timeout thresholds, where preemptable ticket spinlock variants show less fairness. When the parameter becomes larger (e.g. $\geq 2^{10}$), preemptable ticket variants show comparable fairness with the ticket spinlock. The results demonstrate that the fairness level ensured preemptable ticket spinlock variants is tunable based on the unit timeout threshold parameter. Larger thresholds can provide more fairness, as discussed previously in equation 3.3.

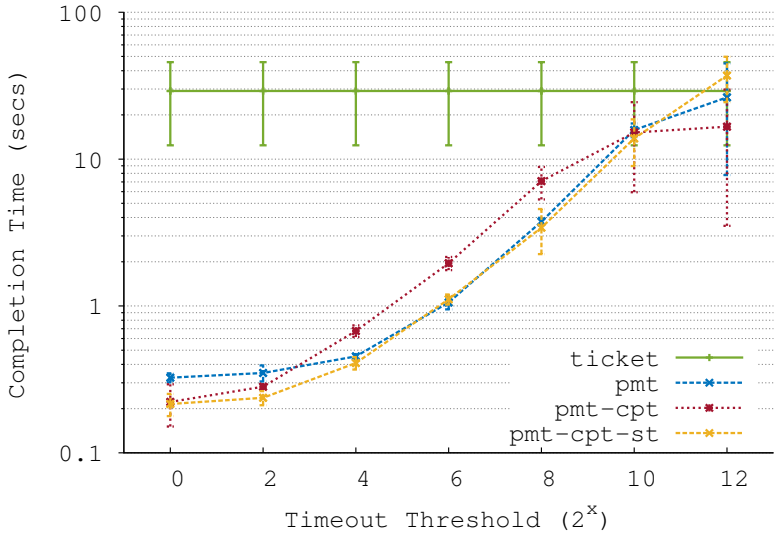


Figure 10: Fairbench Performance v.s. Unit Timeout Threshold (2VM)

While preemptable ticket spinlock and variants show good fairness with a sufficiently large threshold value, we need to make sure that preemptable ticket spinlocks still achieve good performance improvement under such a parameter. Figure 10 that shows the completion time of the fairbech benchmark in a 2VM setting. Note that the y-axis is the benchmark completion time in log-scale, lower is better. As can be seen, the smaller the threshold parameter is, the higher performance improvement preemptable ticket spinlock can achieve. And there is up to 300x difference between the performance improvement achieved by pre-

emptable ticket spinlock variants under different parameters. It is important to note that this synthetic workload is a "best case" scenario for preemptable ticket spinlocks, where the system is over-committed and there is a lot of preemptions. For generic workloads, the performance difference is much smaller as we will show next. Also note that pmt-cpt-st achieves the best performance under almost all parameters except the last, which demonstrates the efficiency of our optimizations.

In the following, we will show the hackbench [Hackbench, 2008] micro-benchmark performance under various parameters, in order to analyze the impact of the threshold parameter on a userspace program. Hackbench is a kernel intensive benchmark that uses spinlocks a lot. As we will see, the difference between performance improvements under various parameters is dramatically smaller compared with fairbench (2x v.s. 300x).

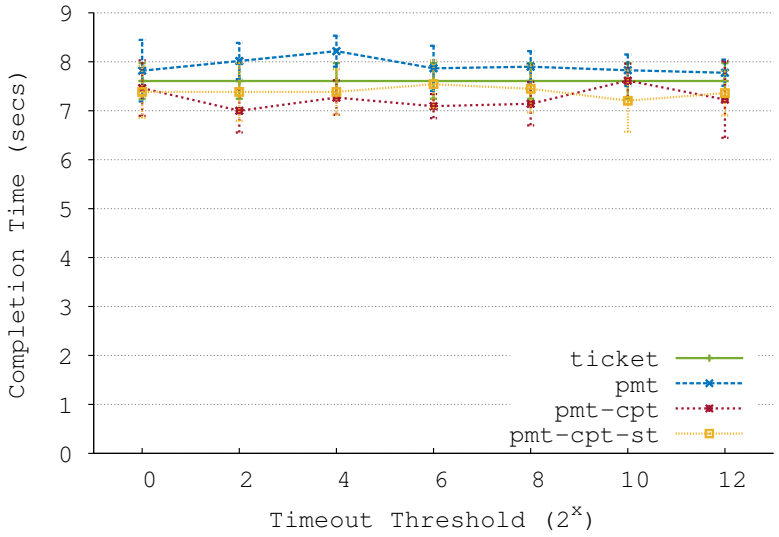


Figure 11: Hackbench Performance v.s. Unit Timeout Threshold (1VM)

Figure 11 shows the hackbench performance when one VM is deployed on the hardware (1VM). The x-axis shows the unit timeout threshold in log scale, and the y-axis shows the completion time of the benchmark in seconds. As can be seen, across all settings, these algorithms show comparable performance: pmt-basic adds some overhead compared with ticket spinlocks, while the two optimized implementations (pmt-cpt and pmt-cpt-st) slightly improve the average performance. The results indicate that in the 1VM case, when vCPU

preemptions are not common, preemptable ticket spinlocks have comparable performance with the ticket spinlock. The overhead of pmt-basic comes from maintaining the lock availability variable, since it requires two extra lock operations for each lock and unlock operations compared with the ticket spinlock. Meanwhile, for the compact implementations (pmt-cpt and pmt-cpt-st), only one extra atomic operations is required. Consequently, they show overhead. Though adding some overhead, pmt-cpt and pmt-cpt-st outperform the baseline by addressing the lock waiter preemption problem.

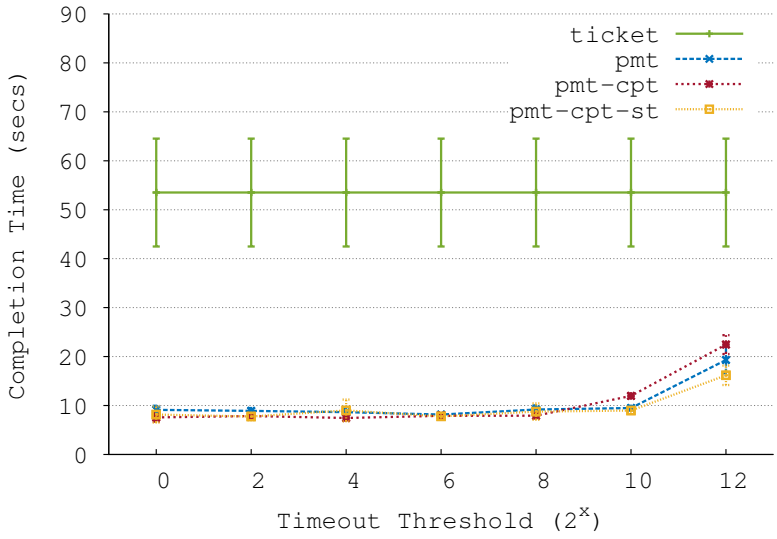


Figure 12: Hackbench Performance v.s. Unit Timeout Threshold (2VM)

Figure 12 shows the hackbench performance when two VMs are deployed on the hardware and competing for the same set of CPU resource (2VM). Performance of one VM running hackbench is reported, while the other VM was running a CPU hog with the sysbench [Sysbench, 2016] benchmark. In the 2VM case, vCPU preemption happens much more often, thus the likelihood of lock waiter preemption increases dramatically. Consequently, preemptable ticket spinlock variants show significant advantages over the baseline. Under most threshold values (e.g. $\leq 2^8$), over 5 times speedup can be achieved, while the performance variances are also much less. Meanwhile, all preemptable ticket spinlock variants show comparable performances, because in 2VM case the lock waiter preemption problem is the dominant factor. It is also worth noticing that when the threshold is set to greater or

equal than 2^{10}, preemptable ticket spinlock performances start to decrease. This is because with larger thresholds, preemptable ticket spinlock behaves more like a ticket spinlock with stricter fairness, which makes it more vulnerable to the lock waiter preemption problem.

According to these experimental results, we choose 2^{10} as the default unit timeout threshold in order to achieve good performance while preserving fairness.

3.3.4.2 Lock Size Next, we compared the kernel size to demonstrate the space effectiveness of the compact preemptable ticket spinlock optimization.

Size (Bytes)	ticket	pmt-basic	pmt-cpt	pmt-cpt-st
Lock	16	32	16	16
Kernel	5,700,016	5,719,088	5,701,008	5,701,904
Δ(Kernel Size)		**+19,372**	**+992**	**+1,888**

Table 2: Preemptable Ticket Spinlock Size and Kernel Size

Table 2 shows the spinlock size and the Linux kernel size under different spinlock implementations. The last row shows the change of kernel size compared with the baseline. The compact implementations (pmt-cpt and pmt-cpt-st) have half of the lock size compared with pmt-basic, while they has the same lock size with the baseline. Also, it turns out that the change of the spinlock size can result in an aggregated effect on the total kernel size, because spinlocks widely exist in many kernel data structures. With the compact optimization, the increase of the kernel size is an order of magnitude smaller than pmt-basic as shown in the last row.

3.3.4.3 Optimizations We then evaluated the two proposed optimizations using the PARSEC benchmark suite [Bienia, 2011], which includes a set of multi-threaded applications with various characteristics. We choose this benchmark suite for its diversity, plus it is previously used by several other related works [Kim et al., 2013; Ding et al., 2014].

Figure 13 shows the execution time of various benchmarks normalized against the ticket spinlock with single VM deployed. In this case, vCPU preemption is not often. Therefore

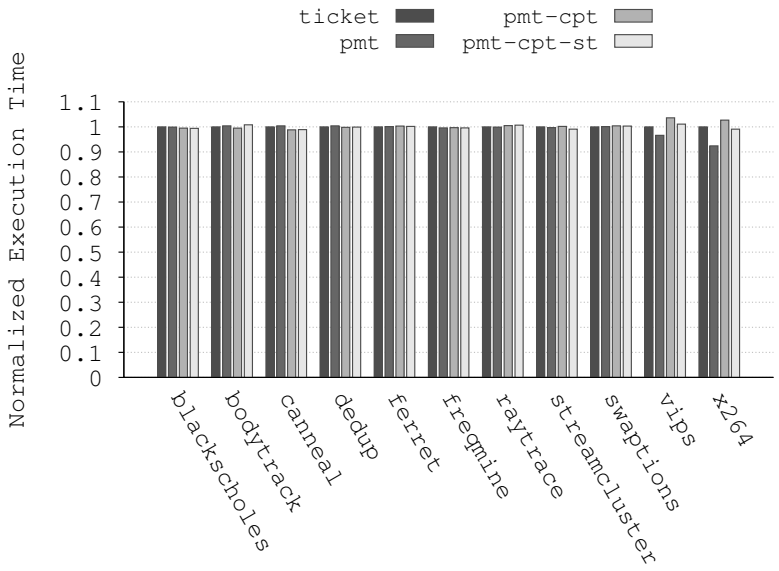


Figure 13: PARSEC Performance with Preemptable Ticket Spinlock Variants (1VM)

for most benchmarks except the last two, the difference between the baseline and the proposed scheme is negligible. It indicates that in a under-committed virtual environment the performance overhead of preemptable ticket spinlocks is acceptable for most applications.

Figure 14 shows the results when two competing VMs are deployed and sharing the same set of CPUs. The preemptable ticket spinlock variants show significant performance improvements over the baseline, indicating that the lock waiter preemption problem is effectively addressed. Besides, the performance of the preemptable ticket spinlock variants are comparable.

Considering that the compact optimization (cpt) reduces lock size, and the static timeout threshold optimization optimization (st) simplifies implementation, while they all achieve comparable performance and fairness, we choose pmt-cpt-st as our default preemptable ticket spinlock implementation, and simply denote it as pmt.

3.3.4.4 Scalability We also compared the performance of preemptable ticket spinlock and ticket spinlocks under different number of cores, in order to study the scalability of

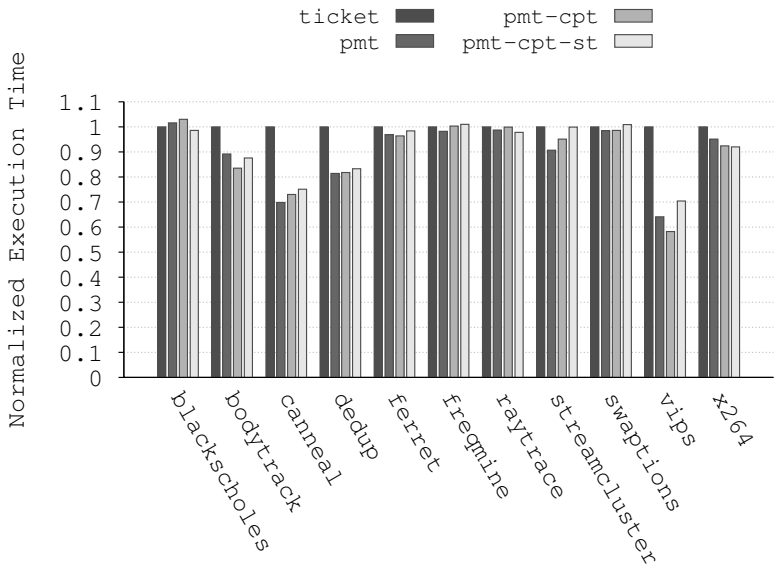


Figure 14: PARSEC Performance with Preemptable Ticket Spinlock Variants (2VM)

our proposed scheme. Figure 15 shows the speedup of preemptable ticket spinlock with both optimizations over ticket spinlock on hackbench, in which x-axis shows the number of vCPUs configured to the VM. A clear trend is that with more vCPUs, more performance improvement can be gained from the preemptable ticket spinlock. Because the average queue length of ticket spinlock increases with the number vCPUs competing for the lock, the longer the queue length, the more likely lock waiters would be preempted. Therefore, increasing vCPU numbers effectively raises the likelihood of lock waiter preemption. However, the speedup does not scale above 10 vCPUs. We believe the reason is that with more vCPUs, the lock contention on cache coherence and the atomic operations would become the performance bottleneck [Boyd-Wickizer et al., 2012], which limits the speedup can be achieved by preemptable ticket spinlocks.

3.3.4.5 Summary So far we have studied many aspects of preemptable ticket spinlocks, our findings can be summarized as following: firstly, smaller unit timeout thresholds make preemptable ticket spinlocks to have less fairness but higher performance, especially under

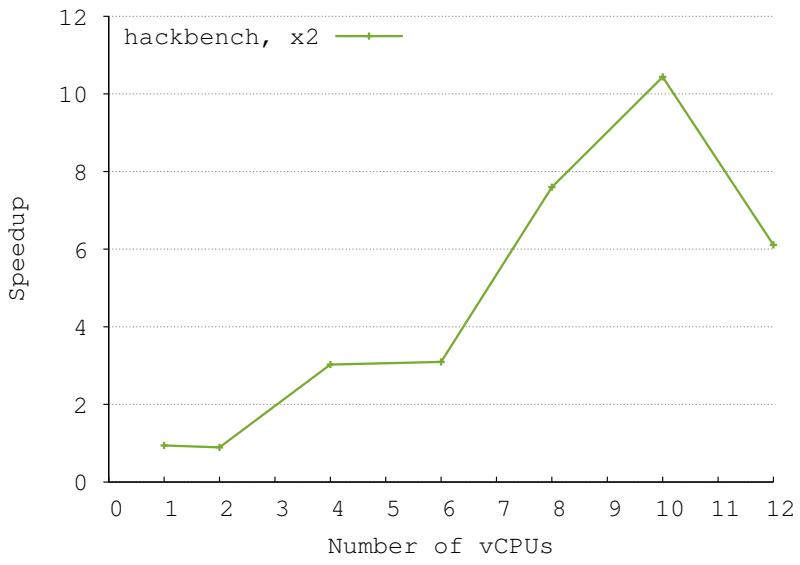


Figure 15: Hackbench Speedup v.s. Number of vCPUs (pmt-cpt-st over ticket, 2VM)

heavy preemptions; while larger thresholds enforce stronger fairness but are more vulnerable to preemptions. We use 2^{10} as the default unit timeout threshold according to our parameter analysis results. secondly, the compact optimization (cpt) reduces the lock size as well as the locking overhead without sacrificing fairness and performance. thirdly, the static unit timeout threshold optimization (st) shows comparable performance and fairness with other preemptable ticket spinlock variants. Besides, it simplifies implementation. lastly, the implementation with both optimizations (pmt-cpt-st) achieves the best design trade-off, therefore we use it as the default implementation and denote it as pmt in the rest of the work.

3.4 SHOOT4U: A PARAVIRTUAL TLB SHOOTDOWN SCHEME

A large body of work has documented the detrimental effects virtual CPU preemption can have on multicore virtual machine performance [Uhlig et al., 2004; Friebel, 2008; Ouyang and

Lange, 2013; Kim et al., 2013; Ousterhout, 1982; Weng et al., 2011; Sukwong and Kim, 2011; Ding et al., 2014; Zhang et al., 2012]. The majority of this work has focused on the impact of spinlock behaviors, due to the direct effects spinlock delays can have on performance critical code paths. However, relatively little attention has been paid to other sources of local delays caused by preemptions of remote CPU cores. In this section, we focus on the issue of performance overhead caused by TLB operations in the presence of preempted virtual CPU cores (vCPUs).

Cross core TLB operations act as a low level synchronization point in modern Operating Systems in order to maintain consistent application memory mappings. The majority of these operations consist of various cache flushing methods that must be invoked on every CPU in the system. For each TLB flush operation the invoking CPU must wait until the operation has been completed on all other cores before continuing, typically by polling a memory region with kernel preemption disabled. This invocation is achieved by issuing Interprocessor Interrupts (IPIs) to each target CPU, the handlers of which directly invoke a local flush operation. In native environments, these operations have very low latency since at most they only need to wait for a target CPU to exit an atomic region before the IPI is handled. In virtual environments these assumptions no longer hold due to the potential for a target vCPU to be preempted by the underlying host scheduler. This can result in the latencies of TLB flush operations increasing by orders of magnitude depending on the scheduling state of the target vCPUs. We refer to this issue as the *TLB shootdown preemption* problem.

To address the TLB shootdown preemption problem we propose Shoot4U, a virtual TLB management mechanism for paravirtualized multicore VMs. Shoot4U eliminates the dependencies on vCPU scheduling states for TLB flush operations and is therefore able to ensure that TLB operations exhibit consistently low latencies. Shoot4U accomplishes this by intercepting cross vCPU TLB flush operations at the VMM layer, and performing the invalidations directly in the VMM instead of requiring that they be handled inside a guest environment. This optimization allows Shoot4U to avoid any delays caused by a preempted vCPU, and to ensure consistent performance of TLB operations. The Shoot4U mechanism provides a better match for the TLB operation semantics, since at the lowest level it shares

the same IPI based signalling behavior as the native versions. This not only allows lower latencies in general, but also eliminates preemption based delays that cause a dramatic increase in the latency variance.

3.4.1 Design of Shoot4U

Shoot4U uses a VMM-assist technique to optimize TLB shootdowns by performing the invalidations inside the VMM itself without the need to invoke or signal the guest OS. It relies on hardware instructions available as part of the virtualization extensions on modern x86 based CPU architectures. These instructions allow targeted invalidation of TLB entries that belong to a specific VM environment.

Before explaining how Shoot4U works, it is necessary to understand how a conventional TLB shootdown operation works in a virtual environment. To initiate a TLB shootdown operation, the invoking vCPU sends an IPI with a specific vector number to a set of target vCPUs. The invoker then enters a polling loop until all receiver vCPUs have processed and acknowledged the requests by setting a flag located in shared memory. The transmission of the IPI by the vCPU causes the hardware to trap into the underlying VMM where it can be emulated, ultimately resulting in the VMM generating a new IPI that is actually transmitted to the VMM on the physical CPUs hosting the targeted vCPUs. Typically an IPI delivery by the hardware will indirectly cause any running VM to trap, so that the underlying host system software can handle it. In this case the IPI is handed off to the VMM, which completes the IPI emulation by delivering an IPI to the targeted vCPU via the injection of a virtual interrupt. After the virtual interrupt has been injected, it will be handled as soon as the target vCPU resumes execution.

Shoot4U is based on the observation that modern hardware allows the underlying VMM to perform the invalidation operation internally, thus removing the need to inject a virtual interrupt into the target vCPU. Current x86 processors from both Intel and AMD support the use of Virtual Processor IDs or VPIDs (Intel) and Address Space IDs or ASIDs (AMD) to tag TLB entries with a given ID assigned to a VM context. Our implementation targets the Intel architecture (using KVM/Linux 3.16), but there is nothing preventing the same approach

from being used on an AMD based system. Along with the ability to tag TLB entries with an associated VPID/ASID these CPUs support a new set of invalidation instructions (e.g. *invpid*) that selectively flush TLB entries based on a given ID tag. These instructions can be executed by the VMM itself, without any involvement of the VM's guest OS. Therefore, instead of relying on IPI injection as described above, Shoot4U enables the VMM to process the TLB invalidation request immediately by invalidating guest TLB entries itself.

3.4.2 Shoot4U Implementation

Our implementation of Shoot4U introduces a paravirtual hypercall interface that replaces the existing IPI based TLB shootdown mechanism. In Shoot4U, the invoking vCPU issues a hypercall down to the underlying VMM with the target vCPUs and address range being invalidated specified as parameters. Upon trapping into the hypercall handler, the VMM determines the set of physical CPUs that are currently hosting the set of vCPUs, and issues a physical IPI to each of them. These IPIs are handled by the VMM itself, which then executes the appropriate set of invalidation operations internally without any interaction with the VM context. While the VMM handles the invalidations for the target vCPUs, the VMM on the invoking CPU polls for completion in a busy wait loop. Once the operations complete the VMM then returns from the hypercall and the VM resumes operation. While superficially it might appear that we have just moved the polling loop from the guest into the VMM, it should be noted that operation completion is no longer dependent on host scheduler behaviors since it does not have to wait for a vCPU to be running in order to complete.

Code 3.3 shows the paravirtual hypercall interface provided by a KVM host with Shoot4U support. To utilize this interface, the guest VM needs to specify the hypercall ID, a bitmap of targeted vCPUs, and the address range being invalidated. Our current implementation of Shoot4U supports up to 64 vCPUs due to the size of the bitmap. However, we can easily support more vCPUs by mapping the bitmap into memory.

Code 3.3: The Shoot4U API

```
1 kvm_hypercall3(unsigned long KVM_HC_SHOOT4U,
```

```
2    unsigned long vcpu_bitmap,
3    unsigned long start,
4    unsigned long end);
```

3.4.3 Evaluation

In this section, we will show the evaluation results of the Shoot4U TLB shootdown scheme. The experiment environment and system configurations are the same as what we used in Section 3.3.4. We used the Linux default TLB shootdown scheme with as the baseline, and compared it with Shoot4U as well as the state-of-the-art TLB shootdown optimization implemented for KVM, denoted kvmtlb [Dadhania, 2012].

		baseline	kvmtlb	shoot4u
1VM	Mean	166	122	28
	Max	24,428	9,953	453
2VM	Mean	9,048	5,401	22
	Max	194,108	126,923	15,034

Table 3: TLB Shootdown Latency (usec)

The first experiment used ktap [Ktap, 2016] to measure the completion time of TLB shootdown requests in the guest, running the dedup benchmark from PARSEC. The results are shown in Table 3, including the average and maximum completion time both with and without a 2nd VM sharing a physical CPU. It shows that both kvmtlb and Shoot4U significantly improve TLB shootdown performance in both cases. However, Shoot4U outperforms the other schemes: it is 4.3 and 245.5 times faster than kvmtlb on average for the 1-VM and 2-VMs cases respectively. Its worse case performance is also order of magnitude better than others.

Figure 16 shows the cumulative distribution function (CDF) of TLB shootdown latencies from the same experiment. Shoot4U not only provides better overall performance, but also exhibits much less variance than other approaches. In a non-overcommitted configuration

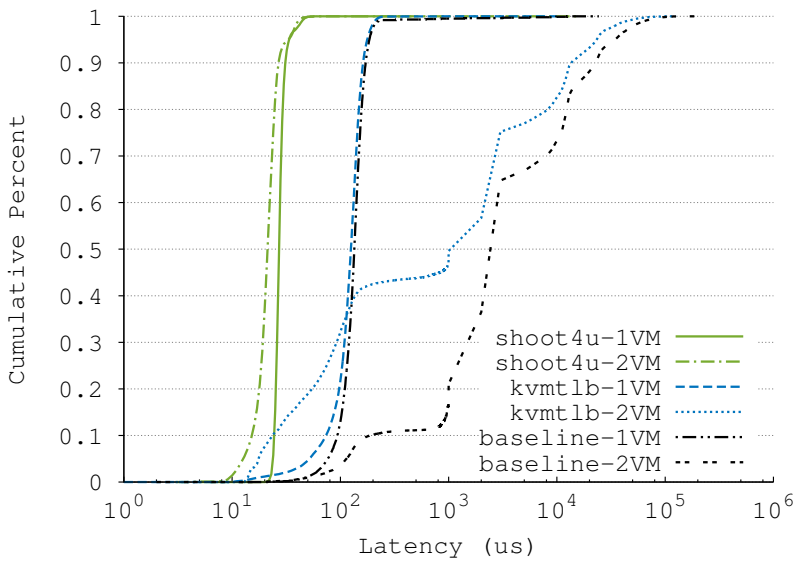


Figure 16: CDF of TLB Shootdown Latency with Shoot4U

Shoot4U provides superior performance by reducing the overheads of a TLB shootdown by reducing the number of world switches needed for IPI propagation. Moreover, Shoot4U is able to maintain consistent performance in an overcommitted configuration, while the other solutions experience slowdowns due to vCPU preemptions, which Shoot4U is immune to.

Our next experiment evaluated the performance with both our spinlock and TLB shootdown schemes applied, using multi-threaded benchmarks from the PARSEC [Bienia, 2011] benchmark suite. Each configuration was evaluated 3 times, and the average is reported. This allowed us to compare the performance impact of spinlock based operations versus TLB operations. We also studied the impact of Pause-Loop Exiting (PLE), a hardware assisted spinning detection and optimization feature supported by KVM and recent Intel processors.

Figure 17 and 18 show the normalized execution time of each benchmark using a sweep of various configurations. In the 1-VM case in Figure 17, performance of various schemes are comparable as the preemption rate is low when the system is not over-committed. However, our scheme still achieves about 20% performance improvement on dedup, which is the most TLB shootdown intensive workload. It is also notable that enabling PLE introduces about

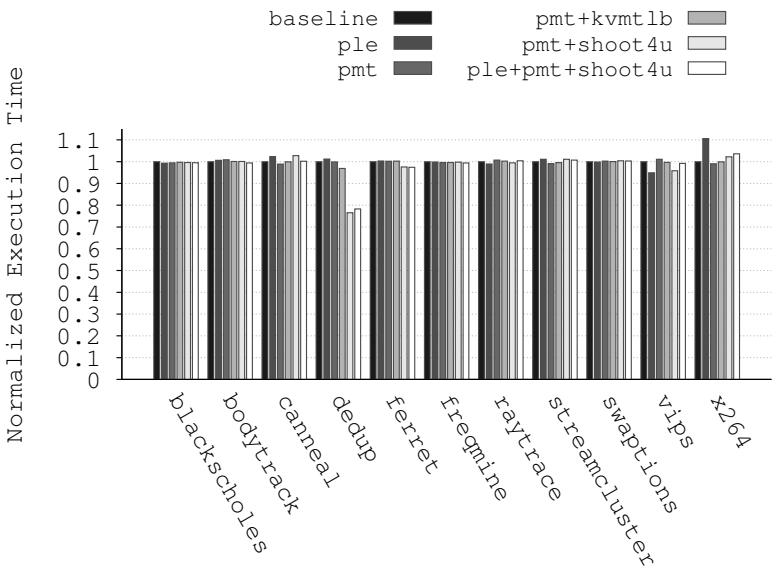


Figure 17: PARSEC Performance with Optimizations (1VM)

10% overhead on x264 in this case. In the 2-VMs case in Figure 18, Shoot4U outperforms kvmtlb by more than 10% on 4 benchmarks, and in the best cases, it is 85% faster than the baseline on dedup, and 44% faster than kvmtlb on ferret. It can also be observed that PLE yields pretty good performance improvements on many benchmarks; moreover, further improvements can be achieved when PLE is combined with preemptable ticket spinlock and Shoot4U.

Finally we reran the profiling experiments in order to compare the reduction of synchronization overheads possible using Shoot4U and preemptable ticket spinlocks. Figure 19 compares the slowdown of both the baseline and optimized configurations in the 2-VMs scenario. Significant performance improvement is observed on 6 out of the 11 benchmarks. For dedup and vips in particular, the slowdown decreases from 70.6 to 4.8 and from 10.1 to 2.9 respectively.

Figure 20 provides a profile of the sources of overheads for the two configurations. There are significant reductions of kernel based overhead for all kernel intensive benchmarks, explaining the overall performance improvements for those benchmarks. Furthermore, for

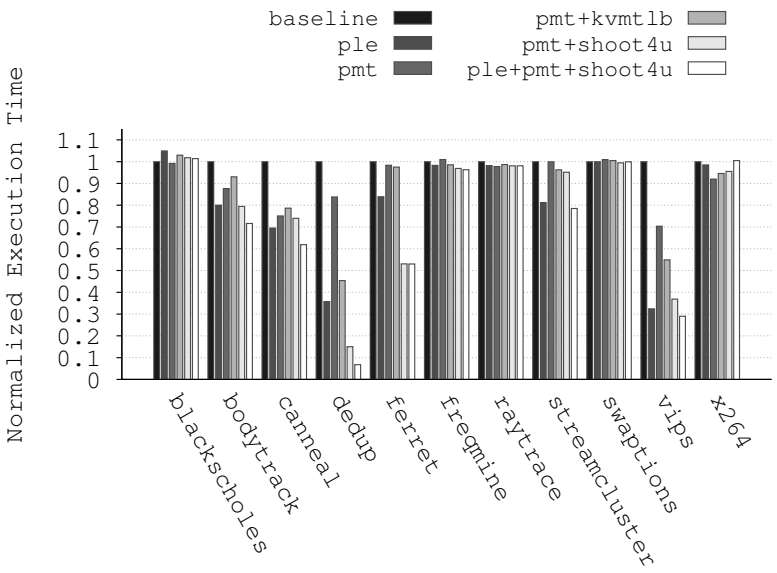


Figure 18: PARSEC Performance with Optimizations (2VM)

nearly every benchmark the time spent in TLB related functions is almost eliminated, with the exception of dedup which is still greatly reduced.

3.5 SUMMARY

In this chapter, we discussed the performance overhead of busy waiting based synchronization operations in virtual environments. We conducted performance analysis that pin-points spinlocks and TLB shootdown operations as the kernel performance bottleneck. To address these problems, we presented the preemptable ticket spinlock, a VMM independent spinlock algorithm that dynamically adjust lock fairness and performance, and Shoot4U an optimization for TLB shootdown operations that internalizes the operation in the VMM and so no longer requires the involvement of a guest's vCPUs. Our evaluation demonstrates the effectiveness of our approach, and illustrates how under certain workloads our approach is dramatically better than current state of the art techniques.

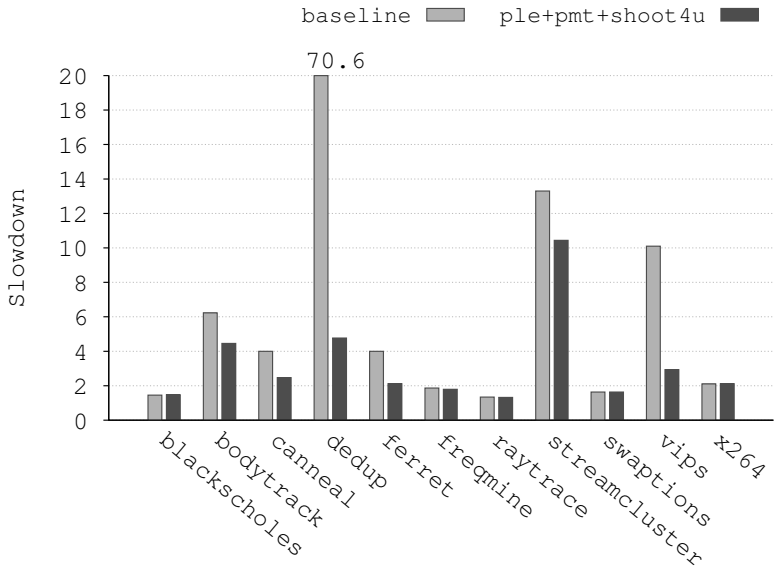


Figure 19: Performance Slowdown of CPU Overcommitment with Optimizations

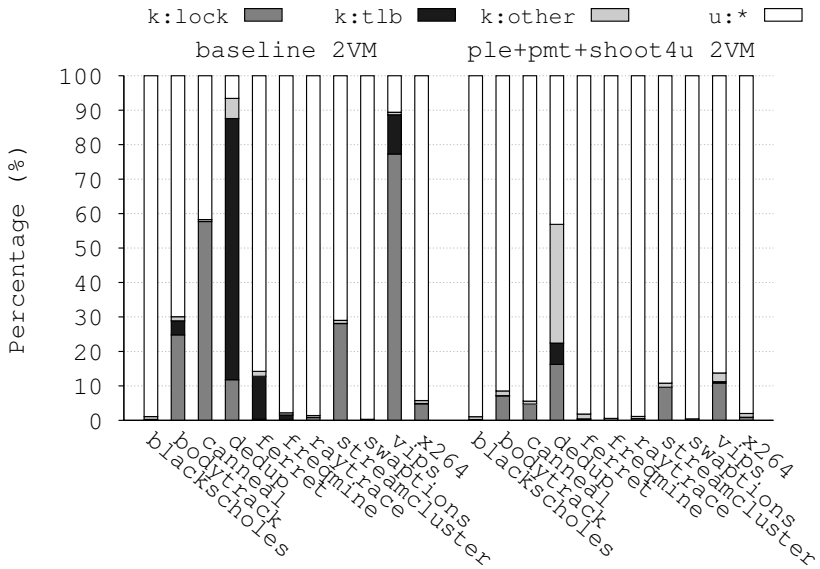


Figure 20: CPU Usage Profiling with Optimizations

4.0 THE PISCES LIGHTWEIGHT CO-KERNEL ARCHITECTURE

Performance isolation is emerging as a requirement for High Performance Computing (HPC) applications, particularly as HPC architectures turn to *in situ* data processing [Ma et al., 2007; Zheng et al., 2013] and application composition techniques to increase system throughput. These approaches require the co-location of disparate workloads on the same compute node, each with different resource and runtime requirements.

We claim that these workloads cannot be effectively managed by a single Operating System/Runtime (OS/R). This is because commodity systems, such as Linux, are designed to maximize a set of design goals that conflict with those required to provide complete isolation. Specifically, commodity systems are almost always designed to maximize resource utilization, ensure fairness, and most importantly, gracefully degrade in the face of increasing loads. These goals often result in software level interference that has a significant impact on HPC application performance as these workloads are susceptible to system noise and overheads [Petrini et al., 2003; Ferreira et al., 2008; Hoefler et al., 2010].

Therefore, we present *Pisces*, a system software architecture that enables the co-existence of multiple independent and fully isolated OS/Rs, or *enclaves*, that can be customized to address the disparate requirements of next generation HPC workloads. Each enclave consists of a specialized lightweight OS co-kernel and runtime, which is capable of independently managing partitions of dynamically assigned hardware resources. Contrary to other co-kernel approaches, in this work we consider performance isolation to be a primary requirement and present a novel co-kernel architecture to achieve this goal. We further present a set of design requirements necessary to ensure performance isolation, including: (1) elimination of cross OS dependencies, (2) internalized management of I/O, (3) limiting cross enclave communication to explicit shared memory channels, and (4) using virtualization techniques

to provide missing OS features. The implementation of the Pisces co-kernel architecture is based on the Kitten Lightweight Kernel and Palacios Virtual Machine Monitor, two system software architectures designed specifically for HPC systems. Finally we will show that lightweight isolated co-kernels can provide better performance for HPC applications, and that isolated virtual machines are even capable of outperforming native environments in the presence of competing workloads.

4.1 INTRODUCTION

Performance isolation has become a significant issue for both cloud and High Performance Computing (HPC) environments [Dean and Barroso, 2013; Phull et al., 2012; Dorier et al., 2014]. This is particularly true as modern applications increasingly turn to composition and *in situ* data processing [Ma et al., 2007; Zheng et al., 2013] as substrates for reducing data movement [Kogge et al., 2008] and utilizing the abundance of computational resources available locally on each node. While these techniques have the potential to improve I/O performance and increase scalability, composing disparate application workloads in this way can negatively impact performance by introducing cross workload interference between each application component that shares a compute node's hardware and Operating System/Runtime (OS/R) environment. These effects are especially problematic when combined with traditional Bulk Synchronous Parallel (BSP) HPC applications, which are particularly prone to interference resulting from noise and other system-level overheads across the nodes of large scale deployments [Petrini et al., 2003; Ferreira et al., 2008; Hoefler et al., 2010]. While previous work has identified shared hardware resources as a source of interference, we claim that workload interference can also result from shared resources residing inside a node's *system software*. Therefore, to fully prevent interference from affecting a given workload on a system, it is necessary to provide isolation features both at the hardware and system software layers.

In the last decade, HPC systems have converged to use Linux as the preferred node operating system. This has led Linux to emerge as the dominant environment for many modern

HPC systems [Yoshii et al., 2009; Kaplan, 2007] due to its support of extensive feature sets, ease of programmability, familiarity to application developers, and general ubiquity. While Linux environments provide tangible benefits to both usability and maintainability, they contain fundamental limitations when it comes to providing effective performance isolation. This is because commodity systems, such as Linux, are designed to maximize a set of design goals that conflict with those required to provide complete isolation. Specifically, commodity systems are almost always designed to maximize resource utilization, ensure fairness, and most importantly, gracefully degrade in the face of increasing loads. These goals often result in software level interference that has a significant impact on HPC application performance as these workloads are susceptible to system noise and overheads.

To address these issues we present *Pisces*, an OS/R architecture designed primarily to provide full system isolation for HPC environments through the use of *lightweight co-kernels*. [1] In our architecture, multiple heterogeneous OS/R instances co-exist on a single HPC compute node and directly manage independent sets of hardware resources. Each co-kernel executes as a fully independent OS/R environment that does not rely on any other instance for system level services, thus avoiding cross workload contention on system software resources and ensuring that a single OS/R cannot impact the performance of the entire node. Each co-kernel is capable of providing fully isolated OS/Rs, or *enclaves*, to local workloads. This approach allows a user to dynamically compose independent enclaves from arbitrary sets of local hardware resources at runtime based on a coupled applications' resource and isolation requirements.

While others have explored the concept of lightweight co-kernels coupled with Linux [Park et al., 2012; Wisniewski et al., 2014; Tomita et al., 2014], our approach is novel in that we consider performance isolation to be the primary design goal. In contrast to other systems, where some or all system services are delegated to remote OS/Rs to achieve application compatibility, we explicitly eliminate cross OS dependencies for external services. Instead, each co-kernel must provide a self contained set of system services that are implemented internally. Second, we require that each co-kernel implement its own I/O layers and device drivers that internalize the management of hardware and I/O devices. Third, we restrict

[1] http://www.prognosticlab.org/pisces

cross enclave communication to user space via explicit shared memory channels [Kocoloski and Lange, 2015], and do not provide any user space access to in-kernel message passing interfaces. Finally, we support applications that require unavailable features through the use of fully isolated virtual machines hosted by the lightweight co-kernel. Taken together, these design requirements ensure that our system architecture can provide full isolation at both the hardware and software levels for existing and future HPC applications.

As a foundation for this work, we have leveraged our experience with the Kitten Lightweight Kernel (LWK) and the Palacios Virtual Machine Monitor (VMM) [Lange et al., 2010]. Previous work has shown the benefits of using both Palacios and Kitten to provide scalable and flexible lightweight system software to large scale supercomputing environments [Lange et al., 2011], as well as the potential of properly configured virtualized environments to outperform native environments for certain workloads [Kocoloski and Lange, 2012]. In this work we present a novel approach to achieving workload isolation by leveraging both Kitten and Palacios, deployed using the Pisces co-kernel framework, to provide lightweight isolation environments on systems running full featured OS/Rs.

We claim that our approach is novel in the following ways:

- Pisces emphasizes isolation as the primary design goal and so provides fully isolated OS instances, each of which has direct control over its assigned hardware resources (including I/O devices) and furthermore contains no dependencies on an external OS for core functionality.
- With Pisces, hardware resources are *dynamically* partitioned and assigned to specialized OS/Rs running on the same physical machine. In turn, enclaves can be created and destroyed at runtime based on application requirements.
- We leverage the Palacios VMM coupled with a Kitten co-kernel to provide fully isolated environments to arbitrary OS/Rs.

4.1.1 High Level Approach

At the heart of our approach is the ability to dynamically decompose a node's hardware resources into multiple partitions, each capable of supporting a fully independent and iso-

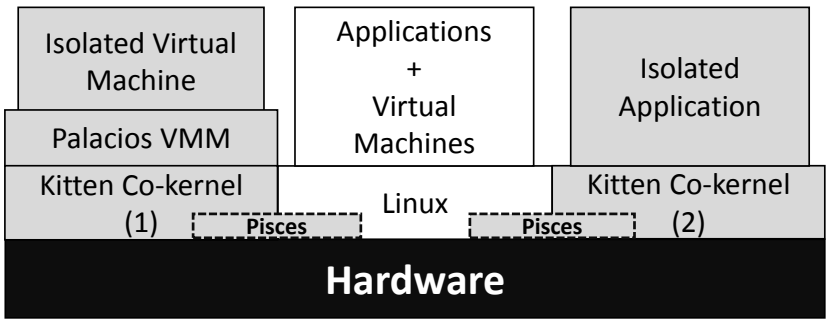


Figure 21: The Pisces Co-Kernel Architecture

lated OS environment. Each OS instance is referred to as an enclave, which is dynamically constructed based on the runtime requirements of an application. A high level overview of the Pisces system architecture is shown in Figure 21. In this environment a single Linux environment has dynamically created two separate enclaves to host a composed application, consisting of a traditional HPC simulation running natively on a co-kernel, and a coupled data visualization/analytic application running inside an isolated VM. Each enclave OS/R directly manages the hardware resources assigned to it, while also allowing dynamic resource assignment based on changing performance needs.

The ability to dynamically compose collections of hardware resources provides significant flexibility for system management. This also enables lightweight enclaves to be brought up quickly and cheaply since they can be initialized with a very limited set of resources, for example a single core and 128 MB of memory, and then dynamically expanded based on the needs of a given application. Furthermore, to fully ensure performance isolation for a given application, each enclave has direct control of the I/O devices that it has been assigned. This is in contrast to many existing OS/hypervisor architectures that incorporate the concept of a driver domain or I/O service domain to mediate access to shared I/O resources. Instead, we provide hosted workloads with direct access to the underlying hardware devices, relying on the hardware's ability to partition and isolate them from the different enclaves in the system.

Figure 22 shows an example configuration of a co-kernel running on a subset of hardware resources. In this example case, a Linux environment is managing the majority of system

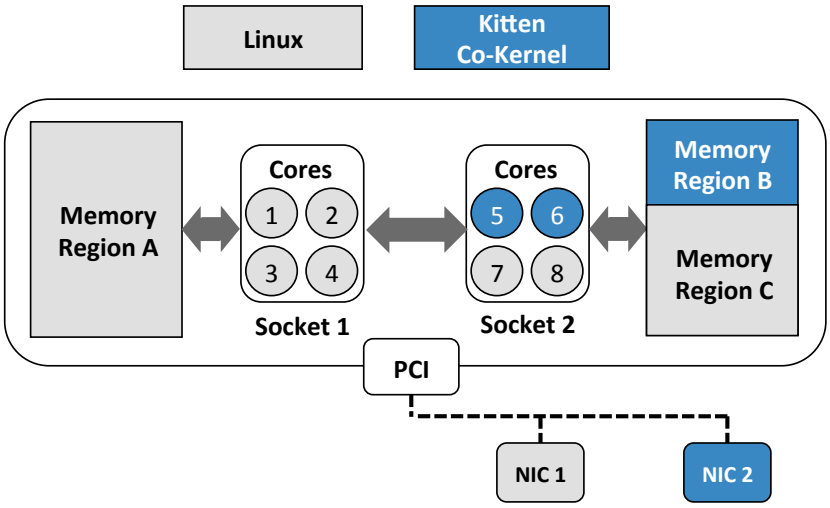


Figure 22: Example Hardware Configuration with one Pisces Co-Kernel

resources with the exception of 2 CPU cores and half the memory in the 2nd NUMA domain, which are assigned to the co-kernel. In addition, the co-kernel has direct control over one of the network interfaces connected through the PCI bus. It is important to note that partitioning the hardware resources in the manner presented here is possible only if the hardware itself supports isolated operation both in terms of performance as well as management. Therefore, the degree to which we can partition a local set of resources is largely system and architecture dependent, and relies on the capabilities of the underlying hardware.

4.1.2 Background

Kitten Lightweight Kernel The Kitten Lightweight Kernel [Lange et al., 2010] is a special-purpose OS kernel designed to provide an efficient environment for executing highly-scalable HPC applications at full-system scales (10's of thousands of compute nodes). [2] Kitten is similar in design to previous LWKs, such as SUNMOS [Maccabe et al., 1994], Puma/-Cougar [Wheat et al., 1994], and Catamount [Kelly and Brightwell, 2005], that have been deployed on Department of Energy supercomputers. Some of Kitten's unique characteristics are its modern code base that is partially derived from the Linux kernel, its improved Linux

[2] https://software.sandia.gov/trac/kitten

API and ABI compatibility that allows it to fit in better with standard HPC toolchains, and its use of virtualization to provide full-featured OS support when needed.

The basic design philosophy underlying Kitten is to constrain OS functionality to the bare essentials needed to support highly scalable HPC applications and to cover the rest through virtualization. Kitten therefore augments the traditional LWK design with a hypervisor capability, allowing full-featured OS instances to be launched on-demand in virtual machines running on top of Kitten. This allows the core Kitten kernel to remain small and focused, and to use the most appropriate resource management policies for the target workload rather than one-size-fits-all policies.

Palacios Virtual Machine Monitor Palacios [Lange et al., 2010, 2011] is a publicly available, open source, OS-independent VMM that targets the x86 and x86_64 architectures (hosts and guests) with either AMD SVM or Intel VT extensions. It is designed to be embeddable into diverse host OSes, and is currently fully supported in both Linux and Kitten based environments. When embedded into Kitten, the combination acts as a lightweight hypervisor supporting full system virtualization. Palacios can run on generic PC hardware, in addition to specialized hardware such as Cray supercomputer systems. In combination with Kitten, Palacios has been shown to provide near native performance when deploying tightly coupled HPC applications at large scale (4096 nodes on a Cray XT3).

4.2 PISCES CO-KERNEL ARCHITECTURE

The core design goal of Pisces is to provide isolated heterogeneous runtime environments on the same node in order to fulfill the requirements of complex applications with disparate OS/R requirements. Co-kernel instances provide isolated enclaves with specialized system software for tightly coupled HPC applications sensitive to performance interference, while a traditional Linux based environment is available for applications with larger feature requirements such as data analytics, visualization, and management workloads. Our work assumes that hardware level isolation is achieved explicitly through well established allocation techniques (memory/CPU pinning, large pages, NUMA binding, etc.), and instead we focus our

work on extending isolation to the software layers as well.

The architecture of our system is specifically designed to provide as much isolation as possible so as to avoid interference from the system software. In order to ensure that isolation is maintained we made several explicit decisions while designing and implementing the Pisces architecture:

- Each enclave must implement its own complete set of supported system calls. System call forwarding is not supported in order to avoid contention inside another OS instance.
- Each enclave must provide its own I/O device drivers and manage its hardware resources directly. Driver domains are not actively supported, as they can be a source of contention and overhead with I/O heavy workloads.
- Cross enclave communication is not a kernel provided feature. All cross enclave communication is explicitly initialized and managed by userspace applications using shared memory.
- For applications with larger feature requirements than provided by the native co-kernel, we use a co-kernel based virtual machine monitor to provide isolated VM instances.

4.2.1 Cross Kernel Dependencies

A key claim we make in this work is that cross workload interference is the result of both hardware resource contention and *system software* behavior. It should be noted that software level interference is not necessarily the result of contention on shared software resources, but rather fundamental behaviors of the underlying OS/R. This means that even with the considerable amount of work that has gone into increasing the *scalability* of Linux [Boyd-Wickizer et al., 2010] there still remain a set of fundamental issues that introduce interference as the utilization of the system increases.

A common source of system software level interference are the system calls invoked by an application. Many of the control paths taken by common system calls contain edge cases in which considerably longer execution paths can be invoked under certain conditions.

Our approach is in direct contrast to other co-kernel approaches that have been proposed [Park et al., 2012; Tomita et al., 2014], and which make extensive use of system call

forwarding and other inter-kernel communication. Our approach avoids not only the overhead associated with cross enclave messaging, but also ensures that interference cannot be caused by additional workloads in a separate OS/R instance.

4.2.2 I/O and Device Drivers

In addition to system call handling, device I/O represents another potential source of interference between workloads. The same issues discussed with system calls apply to the I/O paths as well, in which higher levels of utilization can trigger longer execution paths inside the I/O handling layers. Therefore, in order to further avoid software level interference, we require that each enclave independently manage its own hardware resources, including I/O devices. This requires that each independent OS/R contain its own set of device drivers to allow hosted applications access to the devices that have been explicitly assigned to that enclave. This prevents interference caused by other application behaviors, as well as eliminates contention on I/O resources that could be caused by sharing I/O devices or by routing I/O requests to a single driver domain. This approach does require that I/O devices either support partitioning in some way (e.g., SRIOV [Dong et al., 2008]) or that they be allocated entirely to one enclave. While this would appear to be inefficient, it matches our space sharing philosophy and furthermore it represents the same requirements placed on passthrough I/O devices in virtualized systems.

	solitary workload (ms)	w/ other workloads (ms)
Linux	231.69	312.66
co-kernel	212.75	212.38

Table 4: Execution Time of Sequential Reads from a Block Device

Table 4 shows the benefits of isolated I/O in the case of a SATA disk. In this case we measured the performance differences between a local device driver implementation both on Linux and inside a co-kernel. For this experiment we evaluated the performance of

sequential reads from a secondary SATA disk exported as a raw block device bypassing any file system layers. As with the first experiment additional workloads took the form of parallel kernel compilations occurring on the primary SATA disk hosting the main Linux file system. Accessing the SATA disk from Kitten required a custom SATA driver and block device layer which we implemented from scratch to provide zero-copy block access for our Kitten applications while also sharing a single SATA controller with other enclave OS/Rs. The results show that the optimized Kitten driver is able to outperform the Linux storage layer in each case, however more importantly the co-kernel is able to maintain isolation without meaningful performance degradation even in the face of competing workloads contending on the same SATA controller. Linux, on the other hand, demonstrates significant performance degradation when a competing workload is introduced even though it is accessing separate hardware resources.

4.2.3 Cross Enclave Communication

While full isolation is the primary design goal of our system, it is still necessary to provide communication mechanisms in order to support *in situ* and composite application architectures. Cross enclave interactions are also necessary to support system administration and management tasks. To support these requirements while also enforcing the isolation properties we have so far discussed, we chose to restrict communication between processes on separate enclaves to explicitly created shared memory regions mapped into a process' address space by using the XEMEM shared memory system [Kocoloski and Lange, 2015]. While this arrangement does force user space applications to implement their own communication operations on top of raw shared memory, it should be noted that this is not a new problem for many HPC applications. Moreover this decision allows our architecture to remove cross enclave IPC services from the OS/R entirely, thus further ensuring performance isolation at the system software layer.

Due to the fact that communication is only allowed between user space applications, management operations must therefore be accomplished via user space. Each enclave is required to bootstrap a local control process that is responsible for the internal management

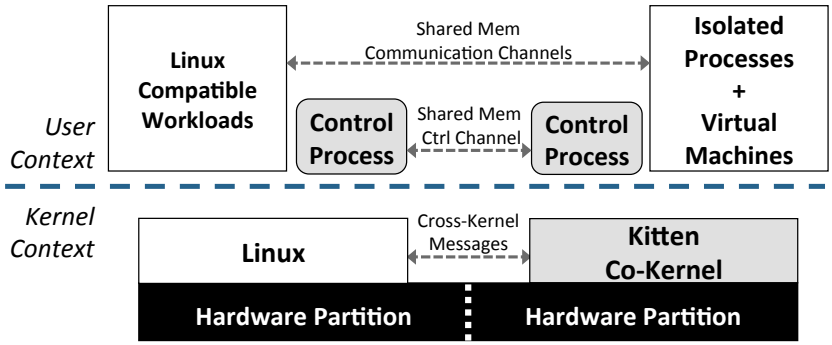


Figure 23: Cross Enclave Communication in Pisces

of the enclave. Administrative operations are assumed to originate from a single management enclave running a full featured OS/R (in our case Linux), which is responsible for coordinating resource and workload assignments between itself and each locally hosted co-kernel. Administrative operations are therefore accomplished through control messages that are sent and received across a shared memory channel between the enclave control process and a global administrative service in the management enclave.

Figure 23 illustrates the model for communication between enclaves managed by a Linux environment and a Kitten co-kernel. While the purpose of our approach is to avoid unpredictable noise in the form of inter-core interrupts (IPIs) and processing delays that would necessarily accompany a kernel-level message-oriented approach, it is nevertheless necessary to allow some level of inter-kernel communication in some situations, specifically in the bootstrap phase and when dealing with legacy I/O devices. For these purposes we have implemented a small inter-kernel message passing interface that permits a limited set of operations. However this communication is limited to only the kernel systems where it is necessary, and is not accessible to any user space process.

4.2.4 Isolated Virtual Machines

The main limitation of lightweight kernels is the fact that the features they remove in order to ensure performance consistency are often necessary to support general purpose applications and runtimes. This is acceptable when the application set is tightly constrained, but for more

general HPC environments and applications a larger OS/R feature set is required. For these applications we claim that we can still provide the isolation benefits of our co-kernel approach *as well as* their required feature set through the use of isolated virtual machine instances. For these applications our co-kernel architecture is capable of providing a full featured OS/R inside a lightweight virtual environment hosted on the Palacios VMM coupled with a Kitten co-kernel, which we denote as a co-VMM. While past work has shown that Palacios is capable of virtualizing large scale HPC systems with little to no overhead, the isolation properties of Palacios (deployed as a co-VMM) actually provide *performance benefits* to applications. As we will show, a full featured Linux VM deployed on a co-VMM is capable of *outperforming* a native Linux environment when other workloads are concurrently executing on the same node.

4.3 PISCES IMPLEMENTATION

The Pisces architecture extends the Kitten Lightweight Kernel to allow multiple instances of it to run concurrently with a Linux environment. Each Kitten co-kernel instance is given direct control over a subset of the local hardware resources, and is able to manage its resources directly without any coordination with other kernel instances running on the same machine. Pisces includes the following components:

1. A Linux kernel module that allows the initialization and management of co-kernel instances.
2. Modifications to the Kitten architecture to support dynamic resource assignment as well as sparse (non-contiguous) sets of hardware resources.
3. Modifications to the Palacios VMM to support dynamic resource assignment and remote loading of VM images from the Linux environment.

As part of the implementation we made a significant effort to avoid any code changes to Linux itself, in order to ensure wide compatibility across multiple environments. As a result our co-kernel architecture is compatible with a wide range of unmodified Linux kernels (2.6.3x

- 3.x.y). The Linux side components consist of the Pisces kernel module that provides boot loader services and a set of user-level management tools implemented as Linux command line utilities.

The co-kernel used in this work is a highly modified version of the Kitten lightweight kernel as previously described. The majority of the modifications to Kitten centered around removing assumptions that it had full control over the entire set of system resources. Instead we modified its operation to only manage resources that it was explicitly granted access to, either at initialization or dynamically during runtime. Specifically, we removed the default resource discovery mechanisms and replaced them with explicit assignment interfaces called by a user-space management process. Other modifications included a small inter-kernel message passing interface and augmented support for I/O device assignment. In total our modifications required ∼9,000 lines of code. The modifications to Palacios consisted of a command forwarding interface from the Linux management enclave to the VMM running in a Kitten instance, as well as changes to allow dynamic resource assignments forwarded from Kitten. Together these changes consisted of ∼5,000 lines of code.

In order to avoid modifications to the host Linux environment, our approach relies on the ability to *offline* resources in modern Linux kernels. The offline functionality allows a system administrator to remove a given resource from Linux's allocators and subsystems while still leaving the resource physically accessible. In this way we are able to dynamically remove resources such as CPU cores, memory blocks and PCI devices from a running Linux kernel. Once a resource has been offlined, a running co-kernel is allowed to assume direct control over it. In this way, even though both Linux and a co-kernel have full access to the complete set of hardware resources they are able to only assume control of a discontiguous set of resources assigned to them.

4.3.1 Booting a Co-kernel

Initializing a Kitten co-kernel is done by invoking a set of Pisces commands from the Linux management enclave. First a single CPU core and memory block (typically 128 MB) is taken offline, and removed from Linux's control. The Pisces boot loader then loads a Kitten

kernel image and init task into memory and then initializes the boot environment. The boot environment is then instantiated at the start of the offlined memory block, and contains information needed to initialize the co-kernel and a set of special memory regions used for cross enclave communication and console I/O. The Kitten kernel image and init task are then copied into the memory block below the boot parameters. Pisces then replaces the host kernel's trampoline (CPU initialization) code with a modified version that initializes the CPU into long (64 bit) mode and then jumps to a specified address at the start of the boot parameters, which contains a set of assembly instructions that jump immediately into the Kitten kernel itself. Once the trampoline is configured, the Pisces boot loader issues a special INIT IPI (Inter-Processor Interrupt) to the offlined CPU to force it to reinitialize using the modified trampoline.

Once the target CPU for the co-kernel has been initialized, execution will vector into the Kitten co-kernel and begin the kernel initialization process. Kitten will proceed to initialize the local CPU core as well as the local APIC and eventually launch the loaded init task. The main difference is that instead of scanning for I/O devices and other external resources, the co-kernel instead queries a resource map provided inside the boot parameters. This resource map specifies the hardware that the co-kernel is allowed to access (offlined inside the Linux environment). Finally, the co-kernel activates a flag notifying the Pisces boot loader that initialization is complete, at which point Pisces reverts the trampoline back to the original Linux version. This reversion is necessary to support the CPU online operation in Linux, which is used to return the CPU to Linux after the co-kernel enclave has been destroyed.

4.3.2 Communicating with the co-kernel

To allow control of a Pisces co-kernel, the init task that is launched after boot contains a small control process that is able to communicate back to the Linux environment. This allows a Pisces management process running inside Linux to issue a set of commands to control the operation of the co-kernel enclave. These commands allow the dynamic assignment and revocation of additional hardware resources, as well as loading and launching VMs and processes inside the co-kernel. The communication mechanism is built on top of a

Operation	Latency (ms)
Booting a Co-kernel	265.98
Adding a single CPU core	33.74
Adding a 128MB memory block	82.66
Adding an Ethernet NIC	118.98

Table 5: Pisces Operation Latency

shared memory region included in the initial memory block assigned at boot time. This shared memory region implements a simple message passing protocol, and is used entirely for communication with the control process in the co-kernel. Table 5 reports the latency for booting and dynamically assigning various hardware resources to a co-kernel.

In addition to the enclave control channel, an additional communication channel exists to allow the co-kernel to issue requests back to the Linux environment. The use of this channel is minimized to only routines that are strictly necessary and cannot be avoided, including the loading of VMs and applications from the Linux file system, configuration of the global IOMMU, and IRQ forwarding for legacy devices. Based on our design goals, we tried to limit the uses of this channel as much as possible to prevent the co-kernel from relying on Linux features. In particular, we did not want to rely on this channel as a means of doing system call forwarding, as that would break the isolation properties we were trying to achieve. For this reason, the channel is only accessible from inside kernel context and is hidden behind a set of constrained and limited APIs. It should be noted that this restriction limits the allowed functionality of the applications hosted by Kitten, but as we will demonstrate later, more full featured applications can still benefit from the isolation of a co-kernel through the use of virtualization.

4.3.3 Assigning hardware resources

The initial co-kernel environment consists of a single CPU and a single memory block. In order to support large scale applications, Pisces provides mechanisms for dynamically

expanding an enclave after it has booted. As before, we rely on the ability to dynamically offline hardware resources in Linux. We have also implemented dynamic resource assignment in the Kitten kernel itself to handle hardware changes at runtime. Currently Pisces supports dynamic assignment of CPUs, memory, and PCI devices.

Adding a CPU core to a Kitten co-kernel is achieved in essentially the same way as the boot process. A CPU core is offlined in Linux and the trampoline is again replaced with a modified version. At this point Pisces issues a command to the control process running in the co-kernel, informing it that a new CPU is being assigned to the enclave. The control process receives the command (which includes the CPU and APIC identifiers for the new CPU) and then issues a system call into the Kitten kernel. Kitten then allocates the necessary data structures and issues a request back the Linux boot loader for an INIT IPI to be delivered to the target core. The CPU is then initialized and activated inside the Kitten environment. Reclaiming a CPU is done in a similar manner, with the complication that local tasks need to be migrated to other active CPUs before reclaiming a CPU.

Adding memory to Kitten is handled in much the same way as CPUs. A set of memory blocks are offlined and removed from Linux, and a command containing their physical address ranges is issued to the co-kernel. The control process receives the command and forwards it via a system call to the Kitten kernel. Kitten then steps through the new regions and adds each one to its internal memory map, and ensures that identity mapped page tables are created to allow kernel level access to the new memory regions. Once the memory has been mapped in, it is added to the kernel allocator and is available to be assigned to any running processes. Removing memory simply requires inverting the previous steps, with the complication that if the memory region is currently allocated then it cannot be removed. While this allows the co-kernel to potentially prevent reclamation, memory can always be forcefully reclaimed by destroying the enclave and returning all resources back to Linux.

Due to the goal of full isolation between enclaves, we expect that I/O is handled on a per enclave basis. Our approach is based on the mechanisms currently used to provide direct passthrough access to I/O devices for virtual machines. To add a PCI device to a co-kernel it is first detached from the assigned Linux device driver and then offlined from the system. The IOMMU (if available) is then configured by Pisces without involvement from the co-

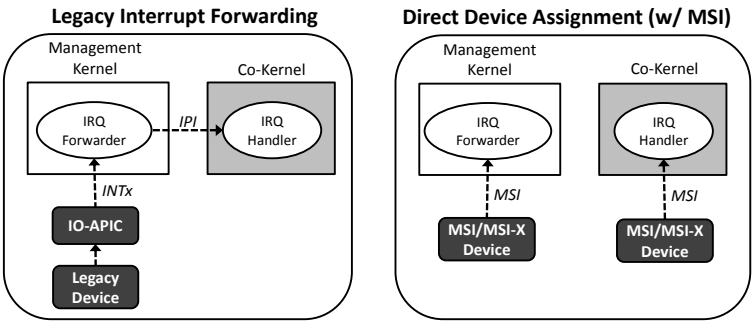


Figure 24: Interrupt Routing Between Pisces Enclaves

kernel itself. This is possible because Pisces tracks the memory regions assigned to each enclave, and so can update the IOMMU with identity mappings for only those regions that have been assigned. This requires dynamically updating the IOMMU as memory is assigned and removed from an enclave, which is accomplished by notifying the module whenever a resource allocation has been changed. Once an IOMMU mapping has been created, the co-kernel is notified that a new device has been added, at which point the co-kernel initializes its own internal device driver.

Unfortunately, interrupt processing poses a potential challenge for assigning I/O devices to an enclave. PCI based devices are all required to support a legacy interrupt mode that delivers all device interrupts to an IO-APIC, that in turn forwards the interrupt as a specified vector to a specified processor. Furthermore, since legacy interrupts are potentially shared among multiple devices a single vector cannot be directly associated with a single device. In this case, it is not possible for Pisces to simply request the delivery of all device interrupts to a single co-kernel since it is possible that it is not the only recipient. To address this issue, Pisces implements an IRQ forwarding service in Linux. When a device is assigned to an enclave any legacy interrupts originating from that device (or any other device sharing that IRQ line) are sent directly to Linux, which then forwards the interrupt via IPI to any enclave which has been assigned a device associated with that IRQ. This approach is shown in the left half of Figure 24. Fortunately, most modern devices support more advanced interrupt routing mechanisms via MSI/MSI-X, wherein each device can be independently configured to generate an IRQ that can be delivered to any CPU. For these devices Pisces is able to

simply configure the device to deliver interrupts directly to a CPU assigned to the co-kernel, as shown in the right half of Figure 24.

4.3.4 Integration with the Palacios VMM

While our co-kernel architecture is designed to support native applications, the portability of our approach is limited due to the restricted feature set resident in Kitten's lightweight design. This prevents some applications from gaining the isolation and performance benefits provided by Pisces. While other work has addressed this problem by offloading unsupported features to Linux [Park et al., 2012], we have taken a different approach in order to avoid dependencies (and associated interference sources) on Linux. Instead we have leveraged our work with the Palacios VMM to allow unmodified applications to execute inside an isolated Linux guest environment running as a VM on top of a co-VMM. This approach allows Pisces to provide the full set of features available to the native Linux environment while also providing isolation from other co-located workloads. As we will show later, Pisces actually allows a virtualized Linux image to *outperform* a native Linux environment in the face of competing workloads.

While Palacios had already been integrated with the Kitten LWK, in this work we implemented a set of changes to allow it to effectively operate in the Pisces environment. Primarily, we added support for the dynamic resource assignment operations of the underlying co-kernel. These modifications entailed ensuring that the proper virtualization features were enabled and disabled appropriately as resources were dynamically assigned and removed. We also added checks to ensure Palacios never accessed stale resources or resources that were not assigned to the enclave. In addition we integrated support for loading, controlling, and interacting with co-kernel hosted VMs from the external Linux environment. This entailed forwarding VM commands and setting up additional shared memory channels between the Linux and co-kernel enclaves. Finally, we extended Kitten to fully support passthrough I/O for devices assigned and allocated for a VM. This device support was built on top of the PCI assignment mechanisms discussed earlier, but also included the ability to dynamically update the IOMMU mappings in Linux based on the memory map assigned to the VM

guest. Finally, we implemented a simple file access protocol to allow Palacios to load a large (multi-gigabyte) VM disk image from the Linux file system.

4.4 EVALUATION

We evaluated Pisces on an experimental 8 node research cluster at the University of Pittsburgh. Each cluster node consists of a Dell R450 server connected via QDR Infiniband. Each server was configured with two six-core Intel "Ivy-Bridge" Xeon processors (12 cores total) and 24 GB of RAM split across two NUMA domains. Each server was running CentOS 7 (Linux Kernel version 3.16). Performance isolation at the hardware level was achieved by pinning each workload to a dedicated NUMA domain. Our experiments used several different software configurations. The standard "CentOS" configuration consisted of running a single Linux environment across the entire machine and using the Linux resource binding APIs to enforce hardware level resource isolation. The KVM configuration consisted of assigning control of one NUMA domain to Linux, while the other was controlled by a KVM VM. Similarly, the "co-kernel" configuration consisted of one Linux managed NUMA domain while the other was managed by a Kitten co-kernel. Finally the "co-VMM" configuration consisted of a Linux guest environment running as a VM on Palacios integrated with a Kitten co-kernel.

4.4.1 Noise analysis

Using the Selfish Detour benchmark [Beckman et al., 2008] from Argonne National Lab, our first experiments measured the impact that co-located workloads had on the noise profile of a given environment, Selfish is designed to detect interruptions in an application's execution by repeatedly sampling the CPU's cycle count in a tight loop. For each experiment we ran the benchmark for a period of 5 seconds, first with no competing workloads and then in combination with a parallelized Linux kernel compilation running on Linux. For each configuration the Selfish benchmark was pinned to the second NUMA domain while the kernel compilation was pinned to the first domain. Therefore changes to the noise profile are

almost certainly the result of software level interference events, and not simply contention on hardware resources.

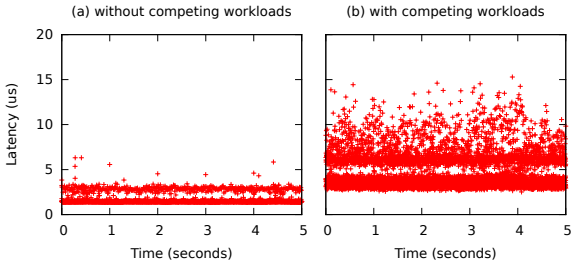


Figure 25: Noise on Native Linux

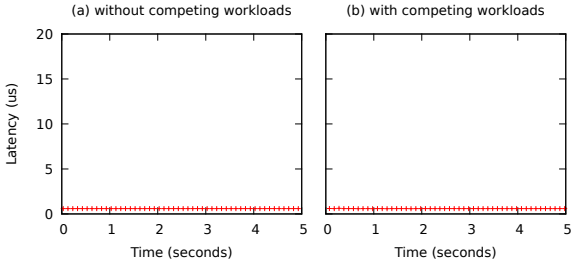


Figure 26: Noise on Native Kitten (Pisces Co-Kernel)

The results of these experiments are shown in Figures 25 and 26. Each interruption (above a threshold) is plotted, and the length of the interruption is reported as the latency. As can be seen, the co-kernel predictably provides a dramatically lower noise profile, while the native Linux environment also exhibits a fairly low level of noise when no competing workloads are present. However, the native Linux configuration exhibits a significant increase in the number and duration of detour events once the competing workload is introduced.

Next we used the same benchmark to evaluate the isolation capabilities of various virtualization architectures. The goal of these experiments were to demonstrate the isolation capabilities of our co-VMM architecture. For these experiments the Selfish benchmark was executed inside a VM running the Kitten LWK. The same VM image was used on 3 separate VMM architectures: KVM on Linux, Palacios integrated with Linux (Palacios/Linux),

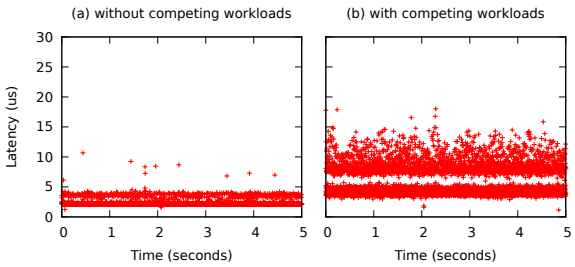


Figure 27: Noise on Kitten Guest (KVM)

and Palacios integrated with a Kitten co-kernel (Palacios/Kitten). The results are shown in Figure 27, 28 and 29 respectively. While each of these configurations result in different noise profiles without competing workloads, in general the co-VMM environment shows considerably less noise events than either of the Linux based configurations. However when a second workload is added to the system KVM shows a marked increase in noise events, while Palacios/Linux shows a slight but noticeable increase in the amount of noise. Conversely, the Palacios/Kitten co-VMM environment shows no noticeable change to the noise profile as additional workloads are added to the system.

We note that the Palacios based environments do experience some longer latency noise events around the 23 microsecond mark (Figures 28 and 29), which are caused by Kitten's 10 Hz guest timer interrupts. The longer latency is a result of the fact that Palacios does not try to overly optimize common code paths, but instead is designed to prioritize consistency. For common events such as timer interrupts, this leads to slightly higher overhead and lower average case performance than demonstrated by KVM. However, as these figures demonstrate, the Palacios configurations provide more consistent performance, particularly as competing workloads are added to the system.

Taken together, these results demonstrate the effectiveness of a lightweight co-kernels in eliminating sources of interference caused by the presence of other co-located workloads running on the same local node. These results are important in analyzing the potential scalability of these OS/R configurations due to the noise sensitivity exhibited by many of our target applications [Petrini et al., 2003; Ferreira et al., 2008; Hoefler et al., 2010]. Thus the

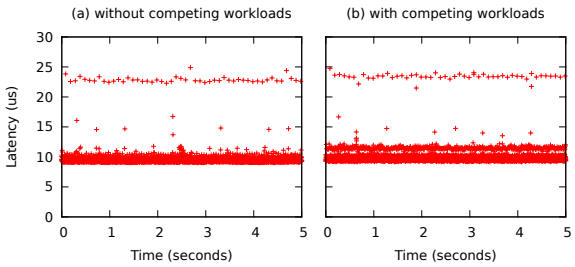


Figure 28: Noise on Kitten Guest (Palacios/Linux)

ability of the Pisces architecture to reduce the noise effects caused by competing workloads indicates that it will provide better scalability than less isolatable OS/R environments.

4.4.2 Single Node Co-Kernel Performance

Figure 30 shows the results from a collection of single node performance experiments from the Mantevo HPC Benchmark Suite [Mantevo, 2016]. In order to evaluate the local performance characteristics of Pisces we conducted a set of experiments using both micro and macro benchmarks. Each benchmark was executed 10 times using 6 OpenMP threads across 6 cores on a single NUMA node. The competing workload we selected was again a parallel compilation of the Linux kernel, this time executing with 6 ranks on the other NUMA node to avoid overcommitting hardware cores. To eliminate hardware-level interference as much as possible, the CPUs and memory used by the benchmark application and background workload were constrained to separate NUMA domains. The NUMA configuration was selected based on the capabilities of the OS/R being evaluated: process control policies on Linux and assigned resources for the co-kernel. For these experiments we evaluated two different OS/R configurations: a single shared Linux environment, a Kitten environment running in a KVM guest, and a Kitten co-kernel environment.

The top row of Figure 30 demonstrates the performance of several Mantevo benchmarks. In all cases, the Kitten co-kernel exhibits better overall performance. In addition the co-kernel environment also exhibits much less variance than the other system configurations. This can be seen especially with the CoMD benchmark, that has a large degree of variance

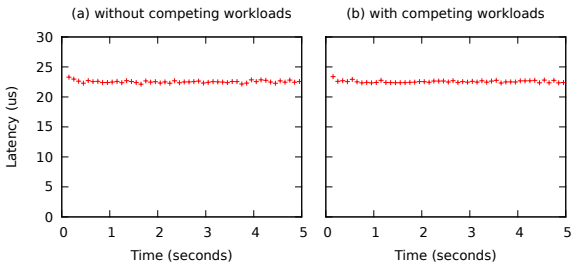


Figure 29: Noise on Kitten Guest (Pisces Co-VMM)

when running on a native Linux environment. Collectively, these results suggest that Pisces is likely to exhibit better scaling behavior to larger node counts than either alternate system configuration.

The bottom row of Figure 30 demonstrates memory micro-benchmark performance with and without competing workloads in the different system configurations. The Stream results demonstrate that a Kitten co-kernel provides consistently better memory performance than either of the other system configurations, averaging 3% performance improvement over the other configurations, with noticeably less variance. Furthermore, the addition of a competing workload has a negligible effect on performance, whereas both of the other configurations show measurable degradation.

4.4.3 Co-Kernel Scalability

Next we evaluated whether the single node performance improvements would translate to a multi-node environment. For this experiment we deployed the HPCG [Dongarra and Heroux, 2013] benchmark from Sandia National Labs across the 8 nodes of our experimental cluster. Because Kitten does not currently support Infiniband hardware, these experiments use a Linux environment running natively or as a VM hosted on either the Pisces co-VMM architecture or KVM. As in the previous sections, the workload configurations consist of an isolated configuration running only the HPCG benchmark, as well as a configuration with competing workloads consisting of parallel kernel compilations configured to run on all 6 cores of a single NUMA socket. We run each experiment for 10 times and report the average

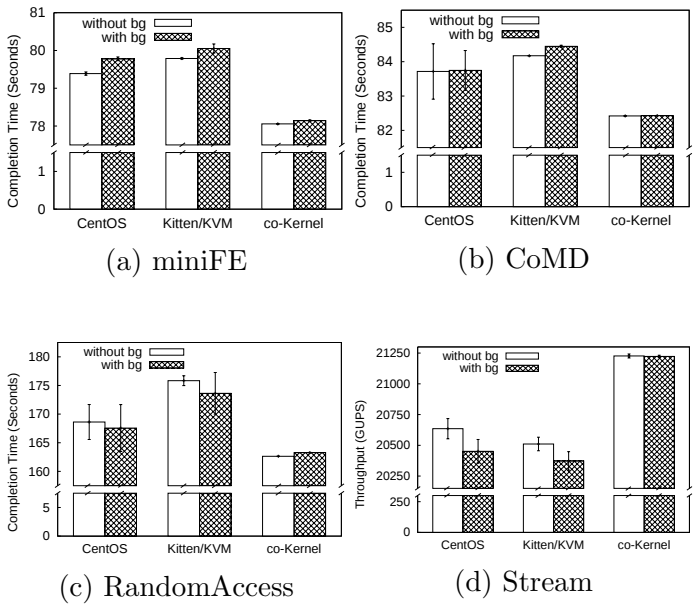


Figure 30: Kitten Co-Kernel Single Node Performance

completion time and standard deviation in Figure 31.

We can see that without competing workloads, the co-VMM configuration achieves near native performance, while KVM consistently performs worse than both of the other configurations. This demonstrates the benefit of lightweight virtualization, compared with virtualization based on commodity software. The performance gain is due to the lower overhead of lightweight virtualization, as well as the consist performance ensured by a lightweight VMM. Consist performance is especially important when the workload is tightly-coupled and running at a large scale.

When a background workload is introduced, all of the configurations perform slightly worse. However, as the node count increases, the co-VMM configuration begins to actually *outperform* both the KVM instance as well as the native environment. These results demonstrate the benefits of performance isolation to an HPC class application, while also showing how cross workload interference can manifest itself inside system software and not just at the hardware level. Specifically, even though lightweight virtualization introduces some performance overhead on a single node compared with native environments, because

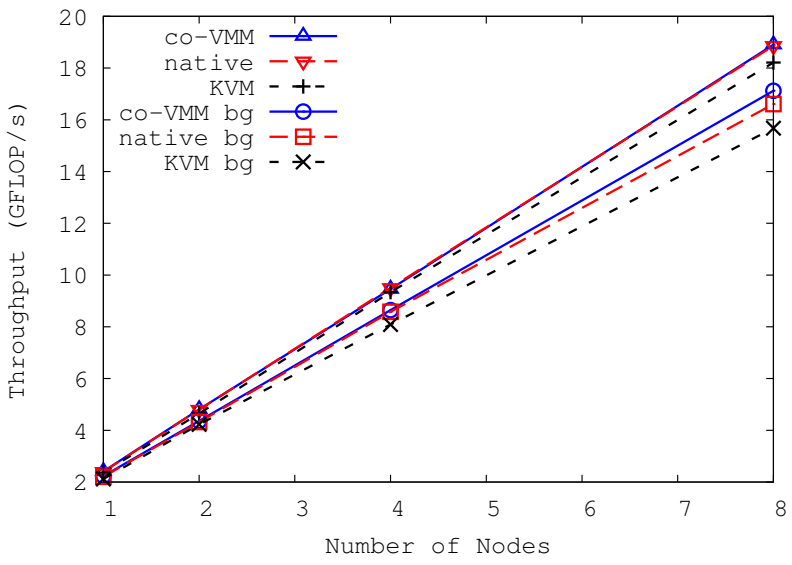


Figure 31: HPCG Benchmark Performance (Up to 8 Nodes)

of the performance isolation ensured by our co-kernel architecture, a co-VMM configuration actually tend to outperform native environment as we scale up the number of server nodes. Given the sensitivity of tightly-coupled HPC applications to performance variances [Petrini et al., 2003; Ferreira et al., 2008; Hoefler et al., 2010], we believe that more performance improvement can be observed at a larger scale.

4.4.4 Performance Isolation with Commodity Workloads

While Pisces is designed to target HPC environments running coupled application workloads, performance isolation has also become a key issue in large scale commodity cloud infrastructures [Dean and Barroso, 2013]. The final set of experiments evaluate the performance isolation provided by the Pisces framework for an example cloud environment with traditional HPC applications co-located with more common cloud workloads on the same local resources. In this experiment, the experimental cluster was setup to co-locate cloud workloads with traditional HPC-class applications in separate Pisces enclaves.

The cloud workload used for these tests was the Mahout machine learning benchmark from the CloudSuite [Ferdman et al., 2012] benchmark suite. The HPC benchmarks were

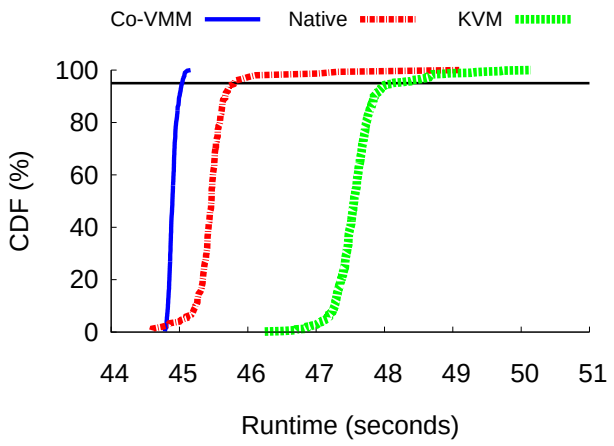
taken from the Mantevo benchmark suite, and consisted of HPCCG, CloverLeaf, and miniFE. For these experiments we were focused on detecting performance outliers at small scale (8 nodes), as these would be indicative of scalability issues, given the tendency for small scale inconsistencies to result in lower average case performance as the node count increases [Petrini et al., 2003; Ferreira et al., 2008; Hoefler et al., 2010]. Therefore, each benchmark was executed for a period of multiple hours, allowing the collection of a large number of total runtimes. The selected configurations consisted of both workloads running natively on Linux, the workloads running in separate KVM VMs, and the workloads running in separate co-VMM environments, where one VM was hosted by a Kitten co-kernel, while the other was hosted by the native Linux OS.

The results are presented using cumulative distribution functions (CDFs) of the benchmark completion times in Figure 32. We chose to present CDFs in order to demonstrate the tail behaviors of each configuration. Based on the results the co-VMM environment is again able to outperform both native and KVM based environments for each of the three benchmarks. In addition the number of outliers (represented by the length of the tails) is generally much smaller for the co-VMM configuration. While the tails only appear above the 95th percentile, it is important to note that as the number of nodes scales up significantly (to the order of thousands), the likelihood of encountering an outlier among any of the application's nodes will increase. Thus, given the tightly synchronized nature of these applications, these outliers are likely to lead be much poorer scalability for both native Linux and KVM.

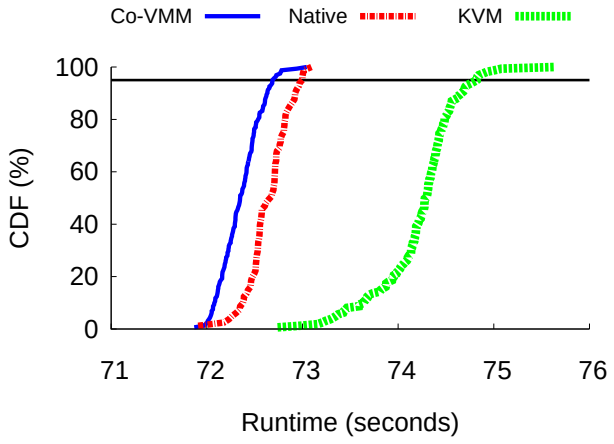
4.5 SUMMARY

In this chapter we presented the Pisces lightweight co-kernel architecture as a means of providing full performance isolation on HPC systems. Pisces enables a modified Kitten Lightweight Kernel to run as a co-kernel alongside a Linux based environment on the same local node. Each co-kernel provides a fully isolated enclave consisting of an independent OS/R environment capable of supporting a wide range of unmodified applications. Furthermore, we have shown that Pisces is capable of achieving better isolation than other

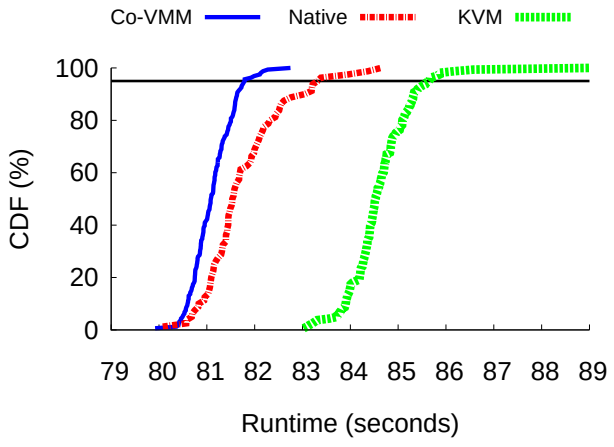
approaches through a set of explicit design goals meant to ensure that isolation properties are maintained in the face of locally competing workloads. By providing superior isolation to applications, we have shown that applications can achieve superior performance as well as significant decrease in performance variability as their scale increases. Finally by utilizing the capabilities of the Palacios Virtual Machine Monitor we have demonstrated that virtual machines can actually outperform native environments in the face of competing or background workloads.



(a) HPCCG



(b) CloverLeaf



(c) miniFE

Figure 32: Mantevo Mini-Application CDFs with Hadoop (8 Nodes)

5.0 HIGH PERFORMANCE I/O ON LIGHTWEIGHT CO-KERNELS

Previously in our multi-node evaluation of Pisces co-kernels, we used lightweight virtualization to deploy a Linux VM on top of the Kitten lightweight co-kernel, because Kitten lacks the support for I/O services, such as device drivers and I/O stacks including the block I/O layer, file systems and the TCP/IP stack. Nevertheless, virtualization allows us to run the full Linux I/O stack on top of a lightweight Kitten co-kernel. Through directly passing-through I/O devices to the Linux VM, we can enable I/O services on an isolated Kitten enclave at the overhead of lightweight virtualization.

We have shown that lightweight virtualization can be used to leverage Linux I/O services on a Kitten enclave, however, the drawback of this approach is that it requires the entire HPC application to be deployed in a Linux VM. While the main advantage of lightweight kernels is consistent performance and low OS noises, introducing the virtualization layer and the Linux software stack enables I/O services at the cost of sacrificing the performance consistency and low OS noise properties of lightweight kernels. Since in this approach, both the computation and I/O components are deployed inside an Linux VM, and both of these components would be impacted by the OS noises and kernel overheads come with Linux and virtualization.

In this chapter, we will rethink the design trade-offs of high performance I/O services on lightweight co-kernels, and try to come up with an approach that can leverage the generality and compatibility of the Linux kernel while preserving the consistent performance and low OS noise properties of the Kitten lightweight kernel. Our core idea is to decouple the computation and I/O components of an application, and run computations natively on lightweight co-kernels while serving I/O requests on an isolated I/O domain. Such that applications can get I/O supports and benefit from lightweight kernels to the largest extend.

5.1 DESIGN GOALS

In our co-kernel architecture, performance isolation is achieved through partitioning at both software and hardware level. Regarding the I/O subsystem, each enclave is assigned with dedicated I/O devices, which are directly managed by local OS/Rs deployed on each enclave. No cross enclave dependency is allowed in order to eliminate cross stack performance interference. Our I/O subsystem on lightweight co-kernels should also align with these principles, which means I/O requests should be handled locally inside an enclave to avoid cross enclave dependencies. As a result, each co-kernel software stack has to provide local I/O supports, including device drivers as well as I/O abstractions such as the block I/O layer, file systems or TCP/IP stacks. Therefore, our main challenge is to *ensure performance isolation and avoid cross enclave dependencies while delegating I/O requests to a Linux instance.* Given these constraints, we summarize the design goals of the I/O subsystem on co-kernels as the following,

- Native lightweight kernel execution. Our first design goal is to execute applications natively on lightweight co-kernels, in order to achieve consistent performance in a low noise environment. Because ensuring performance consistency to achieve scalability is the first order design goal of our co-kernel architecture.

- Isolated I/O services. This design goal inherits the performance isolation property of the co-kernel architecture. The I/O subsystems on co-kernels should ensure consistent I/O performance without introducing cross enclave dependencies.

- High performance I/O services. To achieve high performance, two key mechanisms need to be carefully designed: one is to avoid copying on the data path, the other is an efficiently cross enclave notification mechanism.

- Application transparent I/O Services. An application transparent I/O delegation service allows a executable binary to be directly deployed on a co-kernel without the need to recompile or relink the application. It ensures the usability and compatibility of our I/O service.

5.2 POTENTIAL APPROACHES

Given the design goals discussed above, in this section, we will examine four existing I/O architectures that can be used to enable I/O services on co-kernels and discuss their limitations. Specifically, full virtualization, kernel space I/O stack, userspace I/O stack and I/O service delegation.

Lightweight Virtualization allows unmodified Linux guest VMs to run on top of a co-kernel [Ouyang et al., 2015]. This approach is used in our multi-node evaluation for Pisces co-kernels in chapter 4. Augmented with device passthrough techniques, a Linux VM can directly manage physical I/O devices with little performance overhead. The Linux VM provides POSIX and device compatibilities with no additional engineering cost. Meanwhile, it ensures performance isolation by handling I/O requests locally inside a VM. However, the drawback is that it requires applications to be deployed in a full-fledged Linux guest VM, while it is known that tightly coupled HPC workloads favor lightweight kernels [Lange et al., 2010].

An alternative approach is to implement native kernel space drivers and I/O stacks inside the co-kernels, i.e. kernel space I/O stacks. Most modern monolithic kernels adopt this architecture for good I/O performance, including Linux and Windows. However, adding native device driver support requires huge engineering efforts. Moreover, adding full I/O stacks into a lightweight co-kernel is contradictory to the design goal of lightweight kernels, which could result in higher kernel overheads.

Another approach is to use userspace drivers and I/O stacks. Example userspace I/O stacks include userspace file systems [Fuse, 2016], userspace TCP/IP stacks [Jeong et al., 2014], as well as userspace device drivers [DPDK, 2016]. Userspace I/O stacks are more flexible than kernel space I/O stacks, and allow the co-design of applications, I/O abstractions and device drivers. For instance, the IX kernel [Belay et al., 2014] combines zero-copy I/O, polling based interrupts and batched system call techniques to enable a high performance and low latency data plane operating system. However, a limitation of this approach is that porting userspace device drivers to lightweight co-kernels is non-trivial. Because userspace device drivers are generally developed for Linux, and typically depend on OS features that

is not currently supported by lightweight kernels. Therefore, to support userspace device drivers, the lightweight kernels have to be augmented first to provide kernel level support. Besides, it is difficult to share I/O services between processes using userspace I/O stacks, since the device is owned by a process and Inter-Process Communication (IPC) mechanisms have to be used to share the device among processes.

Finally, I/O delegation based approaches can be used. Depending on how the I/O stack is decomposed between the client and server domains, I/O services can be delegated at three different abstraction levels: virtual device level [LeVasseur et al., 2004; Xen, 2016], device file level [Amiri Sani et al., 2014], and system call level [Nikolaev and Back, 2013]. The Xen driver domain [Xen, 2016] delegates I/O requests at the virtual device level. However, lightweight kernels do not have the native device driver or I/O stack support to use virtual devices. Paradice [Amiri Sani et al., 2014] delegates I/O services at device file level, but it does not support other POSIX I/O interfaces we need, for example Linux sockets. VirtuOS [Nikolaev and Back, 2013] decomposes kernel services into several server VMs and delegates I/O requests to remote VMs at the system call level. However, it introduces an additional data copy on the data path, which is very expensive for data transfer.

In this work, we decided to *delegate I/O requests at the system call level*, while focusing on the performance and isolation of the our I/O services. System call level I/O delegation preserves the I/O interface and allows application transparent I/O delegation. Besides, it reuses the Linux I/O stack to the largest extent, and requires few modifications to the lightweight co-kernel.

5.3 HOBBESIO: HIGH PERFORMANCE CO-KERNEL I/O DELEGATION

In this section, we present *HobbesIO*, a high performance application transparent I/O delegation services on co-kernels. HobbesIO allows the native execution of HPC applications on the lightweight co-kernel, while delegating I/O requests to a Linux instance, denoted as the *I/O domain*, deployed either natively as an enclave or as a virtual machine. HobbesIO allows the native execution of computational tasks and only introduces overheads on I/O

operations, so that the consistent performance and low OS noise properties of the lightweight co-kernel can be preserved. HobbesIO delegates I/O requests at the system call level, and aims to ensure high I/O performance by avoiding data copying on the data path.

In HobbesIO, a stub process is created on a Linux instance (the I/O domain) for each native Kitten process that needs I/O services. Identical memory mappings are setup between the two processes. Besides, a cross enclave command channel is setup between the co-kernel and the Linux I/O domain. I/O system calls from Kitten processes are intercepted and inserted into the command channel. The caller process is then blocked in the Kitten kernel, until its previous system call is handled by the corresponding stub process on Linux and the return value is passed back through the command channel. Busy-waiting based notification is used in HobbesIO. The stub process directly polls the command channel on the Linux side. A dispatching process is also created on Kitten side, which retrieves system call returns and wake up the caller process through ioctl calls. The application process and stub process are paired and identified by a shared stub ID. A stub process only consumes system call requests that match its stub ID, while the dispatching process on Kitten also use the stub ID to wake up the corresponding caller.

In the following, we will provide more details about the design and implementation of the two core building blocks used in HobbesIO: mirrored address spaces on heterogeneous kernels and the system call command channel.

5.3.1 Mirrored Address Spaces on Heterogeneous Kernels

The fundamental technique we used to enable application transparent system call delegation is mirroring the address spaces between an application process on the co-kernel and a stub process on the Linux I/O domain. Because an I/O system call often uses an userspace buffer for data input and output, where the buffer is specified by a pointer pointing to its base address. To allow the stub process to directly issue the system calls on be half of the co-kernel process, the two processes must have identical address spaces, such that the same buffer address in two processes can be mapped to the same copy of physical data. A key advantage of setting up mirrored address spaces is that no extra data copying is introduced on the

data path of I/O delegation. System calls can also be directly executed by the stub process without pre-processing. The I/O delegation mechanism is thus completely transparent to applications with no overhead on the data path. Therefore, the main overhead of HobbesIO would come from transferring system call requests and returns over the command channel, and the overhead of waking up a caller process via ioctls.

Given an executable file of a target application, to setup identical memory mapping between the application process and the stub I/O process, the launcher has to parse the executable file at runtime and setup the memory mappings for the stub process. For a Executable and Linkable Format (ELF) binary file, the linker generates sections like *.text*, *.data*, *.rodata*, etc. Those sections are then merged into multiple *segments* in the binary file by the (static) loader. Normally, the text and data segments are loaded into the RAM when executing a executable file. Therefore, we modified our job launcher to parse the base virtual address and size of the text and data segments specified in the ELF file. The job launcher then allocates physical memory for these segments, as well as a stack region and a heap region for the process. The executable file is then loaded to the allocated memory regions. Meanwhile, those memory regions are also exported via XEMEM.

After allocating and exporting the memory regions, information including physical and virtual addresses as well as the size of these memory regions are passed to the stub process via command line parameters. The stub process then attaches to those exported memory regions and map them to the correct virtual addresses using XEMEM. In this way, the stub process can map the physical memory regions at the same virtual addresses with the original application process. Note that the stub process is compiled with the *-fPIC* flag, so that the stub process is position independent, and its text and data segments are allocated at the higher virtual addresses and will not conflict with the virtual addresses of the application process.

Mirroring address spaces under virtualization is more complicated, where we need to not only consider the guest virtual to guest physical memory mappings, but also the guest physical to host physical memory mappings. Such that an guest virtual address in the stub process would be mapped to a host physical address that is mapped to the same virtual address in the co-kernel native process. In other words, both the guest kernel page tables

as well as the VMM page tables need to be carefully managed to setup identical address spaces for a stub process inside a Linux VM. Fortunately, our XEMEM library [Kocoloski and Lange, 2015] handles both native and virtualized cases and hides the difference between native and virtual environments. Consequently, using XEMEM to setup identical address spaces works across native and virtual Linux instances. This feature allows the flexible deployment of the stub process on either a native Linux enclave or a Linux VM on Palacios.

5.3.2 The System Call Command Channel

To enable cross-domain communication so that I/O system calls can be issued to and returned from the I/O domain efficiently, we built a system call command channel based on a *shared ring buffer* between the co-kernel and the I/O domain. We first setup a shared memory region between the co-kernel and the Linux I/O domain using our XEMEM library [Kocoloski and Lange, 2015]. This shared memory region is then used as a command ring buffer between the two domains for the issuing and returning of system calls. We then intercept interested co-kernel I/O system calls by registering customized system call handlers to delegate system calls over the command channel. The caller is then put into a wait queue, until the return is received. The corresponding stub process inside the I/O domain then issues the system call on behalf of the application process. The return value is then return through the ring buffer in a reverse direction.

Besides transferring system calls and returns, a signalling mechanism is needed to notify the receiving of requests and returns. Two approaches can be used for the cross enclave signalling. The first approach is to use an Inter-Processor Interrupt (IPI) based notification. However, this approach may introduce considerable context switch overhead, because every IPI sent to and from the I/O domain involves multiple context switches due to interrupt handling. Another approach is use a polling based approach by creating threads that busy waiting on the ring buffer. This busy-waiting based approach eliminates the context switching overhead at the cost of burning out CPU time. A hybrid approach can also be used, i.e. using interrupt based notification for one direction, while using busy-waiting based notification for the other direction. Which configuration should be used depends on the available

hardware resources, as well as the performance requirement of applications. Our current implementation uses the busy-waiting based approach, aiming for lower latency and higher throughput.

In our implementation, a HobbesIO proxy process is created on the Kitten kernel, which allocates a one page size memory region that is used as a shared ring buffer. This memory region is also exported as a memory segment named "hio-engine-seg" via XEMEM. The buffer address is also passed into the Kitten kernel, so that system calls can be directly inserted into the ring buffer inside the Kitten kernel. We register HobbesIO customized system call handlers for network system calls including socket(), bind(), accept(), listen(), send(), recv(). In HobbesIO system call handlers, the stub ID, system call ID and parameters are encapsulated and inserted into the ring buffer. The caller process is then blocked on a wait queue in Kitten kernel that is identified by its stub ID. The stub process on the Linux side also attaches to "hio-engine-seg", and polls on it for system call requests. Upon receiving a system request, a stub process with the specified stub ID will consume the request and directly issue the system call along with the original parameters on behalf of the application process. Because identical memory mappings are set up between the two processes, the Linux kernel can operate on the same physical data specified by the application. Once the system call returns, the stub process inserts the return value into the ring buffer and then resumes polling for pending system calls. The HobbesIO proxy process on Kitten also polls on the ring buffer for pending return values. Upon receiving a system call return, the proxy process wakes up the corresponding caller through an ioctl call.

For instance, let's look how does the write() system call work in HobbesIO. The write() system call in Linux uses an application buffer and a kernel buffer. Upon a write() system call, the system call handler first copies application data from the application buffer to the kernel buffer, and then write the data from the kernel buffer to a file or a socket. In case of HobbesIO, the write() system call on Kitten is intercepted by our HobbesIO kernel module, and inserted into the command channel. The Linux I/O domain would then retrieve the system call request from the command channel, and wake up the corresponding stub process on Linux. The stub process decodes the system call request and issues the system call on behalf of the Kitten client process, with the original system call number and parameters

without any pre-processing. The Linux kernel then copies the data from the application buffer to its internal kernel buffer, and write them to the target file descriptor. Note that because mirrored address spaces are used, the buffer address that Linux reads from is mapped to the same physical address where the original Kitten application data resides. In order words, the Kitten application process and the Linux stub process are manipulating the same copy of the physical data. Upon completion of the Linux system call, the return value is transferred back to the Kitten side through the command channel. In summary, mirrored address spaces plus the command channel enable our application transparent I/O delegation service on lightweight co-kernels.

5.3.3 Infiniband and RDMA Support

System call delegation enables the POSIX interface that is used by many commodity applications. However, many modern HPC applications use an advanced I/O feature presented in modern supercomputers. In particular, Remote Direct Memory Access (RDMA), which allows the direct memory access between applications reside on different node. RDMA completely bypasses the OS kernel during on the data path after a connection has been setup. Therefore, system call delegation alone is not enough to support RDMA, since data transfers in RDMA do not use system calls.

Different from the POSIX I/O interface, where every data transfer operation traps into the kernel, RDMA separates the data plane from the control plane. The kernel is only responsible for setting up the I/O resources, while the actually data transfer does not involve the kernel. For instance, in case of RDMA with Infiniband [Mellanox, 2003], the TX and RX queues are mapped into the address space of an application, so that the application can directly access device registers without involving the OS kernel. Because network queue operations are device specific, applications typically rely on a vendor provided user-level library that provides device independent programming interfaces, e.g. the verbs RDMA interface.

The key to support RDMA in HobbesIO is to allow the Kitten application to directly access the device memory that is setup by Linux. Therefore, besides the mirrored address

space for RAM we discussed above, we also need to setup identical memory mappings for the Infiniband device memory. The XEMEM library can be used to map the device memory to identical virtual addresses for processes on heterogeneous kernels.

5.4 EVALUATION

We evaluated HobbesIO on a research cluster. Each cluster node consists of a Dell R450 server connected via 1Gbps Ethernet. Each server was configured with two six-core Intel "Ivy-Bridge" Xeon processors (12 cores total) and 24 GB of RAM split across two NUMA domains. Each server was running CentOS 7 (Linux Kernel version 3.16). CPU cores and memory on NUMA node 1 were entirely offlined from Linux and assigned to a Kitten enclave. Network I/O performance was evaluated on the Kitten enclave over HobbesIO, and compared against native Linux performance.

getpid()/μs	Kitten Native	Linux Native	HobbesIO
Mean	0.0534	0.1040	3.9231
Stdev	0.0000	0.0004	0.0023

Table 6: getpid() Latency on HobbesIO

We first measured the latency of our system call delegation service using the getpid() system call. As getpid() simply reads a variable from the kernel, so the majority of the system call time is spent on system call overhead, including context switch overhead and system call forwarding latency in case of HobbesIO.

Table 6 reports the getpid() average latency and standard deviation on under various configurations. HobbesIO shows considerable higher latency compared with native Kitten and Linux, due to the latency of delegating system calls over the command channel and an extra ioctl system call to wake up the caller process. However, it is important to point out that the overhead of system delegation in HobbesIO is acceptable, because our design goal is

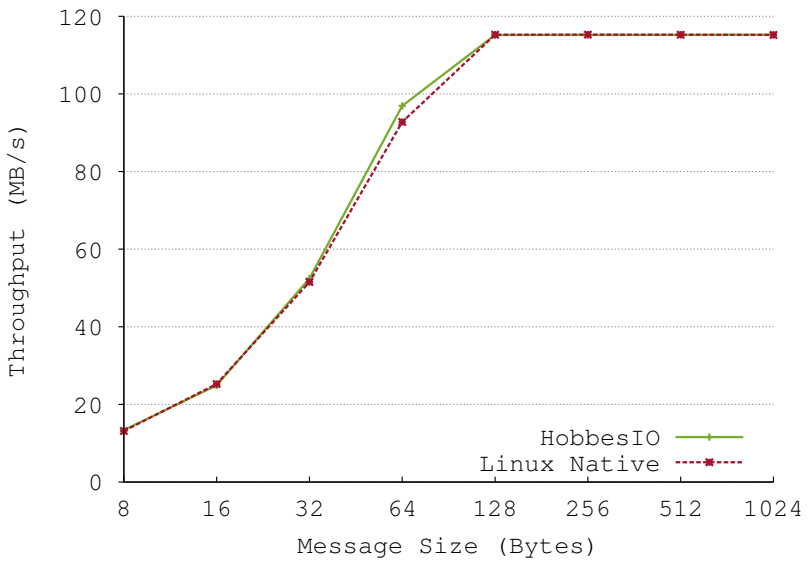


Figure 33: Network Throughput on HobbesIO

to use system call delegation for the control plane and bypass the kernel for the data plane using RDMA. Therefore, system call delegation is not on the critical path, and is mainly used for configuration and resource provision. On the other hand, the absolute latency of system delegation is still low: in the order of microsecond, which is relatively small compared with the round-trip latency of TCP packet transfer. Therefore, the latency of system call delegation could be hidden by the other layers of the software stack, i.e. TCP/IP stack, as we will show next.

Next, we measured the network throughput using the TTCP [TTCP, 2016] benchmark, which transfers certain amount of data over TCP and reports the throughput. We transferred 1GB size of data using different message size between two nodes connected 1Gbps Ethernet. We measured the throughput three times and reported the average throughput in Figure 33. It can be seen that under different message sizes, HobbesIO shows comparable performance compared with native Linux, indicating that the overhead of the system call delegation service is negligible.

We also measured the average and tail latency of the network performance using a simple network server we developed that receives data from multiple clients. Each of our client keeps

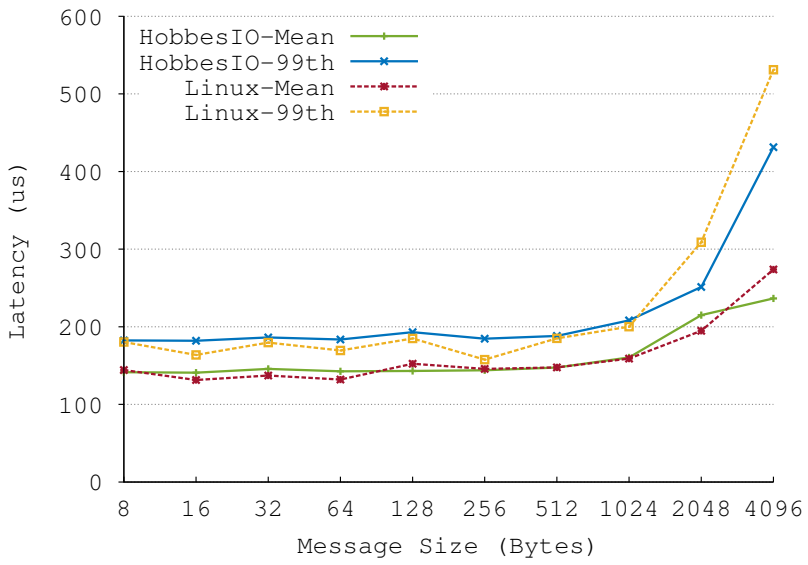


Figure 34: Network Average and Tail Latency on HobbesIO

sending messages over the network for a certain amount of time. In our experiment, we use a 6 threads client sending messages for 10 seconds, and report the average and 99th percentile latency. Figure 34 shows the average and tail latency of HobbesIO and native Linux. It is interesting to see that with smaller message sizes, Linux shows slight lower latencies compared with HobbesIO. However, with message size greater than 1024 bytes, HobbesIO starts to outperform native Linux. We believe this is due to that the amortized overhead of system delegation on large messages is smaller than smaller messages, so that Kitten shows its advantage of consistent performance and low kernel overhead with larger messages.

5.5 SUMMARY

In this chapter, we discussed the design trade-offs of the I/O subsystem on lightweight co-kernels. We presented HobbesIO, a system call delegation service that enables application transparent I/O delegation service on lightweight co-kernels. We built HobbesIO on top of two core techniques, mirrored address spaces on heterogeneous kernels and a shared memory

based command channel. We demonstrated that HobbesIO has comparable network I/O performances compared against native Linux. Our future work includes evaluating RDMA performance and demonstrating the performance isolation capability of HobbesIO.

6.0 CONCLUSIONS

In this dissertation, we looked at the performance isolation problem between co-located applications on the same node. We claim that current *shared homogeneous* kernels based operating system (OS) architectures fall short in ensuring the performance isolation between co-located applications, which impedes the efficiency of large-scale computing infrastructures as well as the scalability of large-scale applications. Therefore, we propose to use an *isolated heterogeneous* kernels based approach to improve the performance isolation between co-located applications on a single server node.

We first studied the performance interference problem between time-shared virtual machines in a environments as seen in the cloud. We identified that busy-waiting based kernel synchronization operations are the root cause of the dramatic performance degradation in shared virtual environments. To address this problem, we designed and implemented two synchronization techniques optimized for virtual environments, the preemptable ticket spinlock (pmtlock) algorithm and the Shoot4U paravirtual TLB shootdown scheme. Our evaluation demonstrates that both of these techniques significantly reduce the performance interference between co-located virtual machines.

Then we looked at the performance isolation problem between in-situ analysis applications in high performance computing (HPC) environments. To ensure the performance isolation between co-located in-situ analysis workloads while providing Linux compatibility, we designed and implemented the Pisces lightweight co-kernel architecture, which allows multiple independent lightweight co-kernels to be deployed side-by-side with Linux on isolated hardware partitions. Each co-kernel can be optimized for the local workload, while the performance isolation between them is enforced by isolating workloads at both the software and hardware level. Our evaluation shows the co-kernel architecture can provide more con-

sistent performance, and achieve better application scalability on multiple nodes compared with native Linux.

Finally, to support high performance I/O on lightweight co-kernels, we investigated HobbesIO, a flexible yet high performance I/O delegation service on lightweight co-kernels. It allows the application transparent I/O delegation from a co-kernel process to an I/O service processes deployed on arbitrary native or virtual Linux enclaves. Our evaluation shows that HobbesIO can achieve comparable performance with native Linux.

In this work, we developed kernel optimizations and a novel operating system architecture in order to improve node level performance isolation between co-located applications. We demonstrated that our optimizations of kernel synchronization operations significantly improve performance isolation between time-shared virtual machines. Our proposed co-kernel architecture can provide an execution environment optimized for HPC workloads and a general Linux environment on the same node, while maintaining performance isolation between them. Therefore, we conclude that *isolated heterogeneous* kernels can enable more sustainable and scalable performance isolation towards shared environments in modern and emerging computing platforms.

BIBLIOGRAPHY

AMD (2011). *AMD64 Architecture Programmer's Manual*.

Amiri Sani, A., Boos, K., Qin, S., and Zhong, L. (2014). I/O Paravirtualization at the Device File Boundary. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

Appavoo, J., Uhlig, V., and Waterland, A. (2008). Project Kittyhawk: Building a Global-Scale Computer. *ACM SIGOPS Operating System Review*.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the Art of Virtualization. In *Proc. ACM symposium on Operating Systems Principles (SOSP)*.

Barroso, L. A., Clidaras, J., and Hölzle, U. (2013). The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. *Synthesis Lectures on Computer Architecture*.

Bauman, E., Ayoade, G., and Lin, Z. (2015). A survey on hypervisor based monitoring: Approaches, applications, and evolutions. *ACM Computing Surveys*.

Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., and Singhania, A. (2009). The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proc. 22nd Symposium on Operating Systems Principles (SOSP)*.

Beckman, P., Iskra, K., Yoshii, K., Coghlan, S., and Nataraj, A. (2008). Benchmarking the Effects of Operating System Interference on Extreme-scale Parallel Machines. *Cluster Computing*.

Belay, A., Prekas, G., Klimovic, A., Grossman, S., Kozyrakis, C., and Bugnion, E. (2014). IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proc. 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

Bienia, C. (2011). *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University.

Boyd-Wickizer, S., Chen, H., Chen, R., Mao, Y., Kaashoek, F., Morris, R., Pesterev, A., Stein, L., Wu, M., Dai, Y., Zhang, Y., and Zhang, Z. (2008). Corey: An Operating

System for Many Cores. In *Proc. 8th USENIX conference on Operating Systems Design and Implementation (OSDI)*.

Boyd-Wickizer, S., Clements, A. T., Mao, Y., Pesterev, A., Kaashoek, M. F., Morris, R., and Zeldovich, N. (2010). An Analysis of Linux Scalability to Many Cores. In *Proc. 9th Symposium on Operating Systems Design and Implementation (OSDI)*.

Boyd-Wickizer, S., Kaashoek, M. F., Morris, R., and Zeldovich, N. (2012). Non-Scalable Locks are Dangerous. In *Proceedings of the Linux Symposium*, pages 119–130.

Cgroups (2016). Linux Control Groups (cgroups). https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt.

Chapin, J., Rosenblum, M., Devine, S., Lahiri, T., Teodosiu, D., and Gupta, A. (1995). Hive: Fault Containment for Shared-Memory Multiprocessors. In *Proc. 15th ACM Symposium on Operating Systems Principles (SOSP)*.

Dadhania, N. A. (2012). KVM Paravirt Remote Flush TLB. https://lwn.net/Articles/500188/.

Dean, J. and Barroso, L. A. (2013). The Tail at Scale. *Communications of the ACM*, 56(2).

Ding, X., Gibbons, P. B., Kozuch, M. A., and Shan, J. (2014). Gleaner: Mitigating the Blocked-Waiter Wakeup Problem for Virtualized Multicore Applications. In *Proc. 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC)*, Philadelphia, PA. USENIX Association.

Dong, Y., Yu, Z., and Rose, G. (2008). SR-IOV Networking in Xen: Architecture, Design and Implementation. In *1st Workshop on IO Virtualization (WIOV)*.

Dongarra, J. and Heroux, M. A. (2013). Toward a New Metric for Ranking High Performance Computing Systems. *Sandia Report, SAND2013-4744*, 312.

Dorier, M., Antoniu, G., Ross, R., Kimpe, D., and Ibrahim, S. (2014). CALCioM: Mitigating I/O Interference in HPC Systems Through Cross-Application Coordination. In *Proc. 28th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

DPDK (2016). DPDK: Data Plane Development Kit. http://dpdk.org/.

Ferdman, M., Adileh, A., Kocberber, O., Volos, S., Alisafaee, M., Jevdjic, D., Kaynak, C., Popescu, A. D., Ailamaki, A., and Falsafi, B. (2012). Clearing the Clouds: a Study of Emerging Scale-out Workloads on Modern Hardware. In *Proc. 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

Ferreira, K. B., Bridges, P., and Brightwell, R. (2008). Characterizing Application Sensitivity to OS Interference Using Kernel-level Noise Injection. In *Proc. 21st International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

Friebel, T. (2008). How to Deal with Lock-Holder Preemption. Presented at the Xen Summit North America.

Fu, Y., Zeng, J., and Lin, Z. (2014). Hypershell: A practical hypervisor layer guest os shell for automated in-vm management. In *Proceedings of the 2014 USENIX Annual Technical Conference*, Philadephia, PA.

Fuse (2016). The FUSE Project. http://fuse.sourceforge.net/.

Gamsa, B., Krieger, O., Appavoo, J., and Stumm, M. (1999). Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*.

Gartner (2012). Gartner Says Efficient Data Center Design Can Lead to 300 Percent Capacity Growth in 60 Percent Less Space. http://www.gartner.com/newsroom/id/1472714.

Giampapa, M., Gooding, T., Inglett, T., and Wisniewski, R. (2010). Experiences with a Lightweight Supercomputer Kernel: Lessons Learned from Blue Gene's CNK. In *Proc. 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

Hackbench (2008). Hackbench. http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c/.

Hoefler, T., Schneider, T., and Lumsdaine, A. (2010). Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. In *Proc. 23rd International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

Intel (2016). *Intel 64 and IA-32 Architectures Software Developer Manuals*. Inel Corporation.

Jain, R., Chiu, D.-M., and Hawe, W. (1984). A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Computer Systems. Technical Report TR-301, DEC Research.

Jeong, E., Woo, S., Jamshed, M., Jeong, H., Ihm, S., Han, D., and Park, K. (2014). mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

Kaplan, J., Forrest, W., and Kindler, N. (2008). Revolutionizing Data Center Energy Efficiency. Technical report, McKinsey & Company.

Kaplan, L. (2007). Cray CNL. In *FastOS PI Meeting and Workshop*.

Kelly, S. and Brightwell, R. (2005). Software Architecture of the Lightweight Kernel, Catamount. In *2005 Cray Users' Group Annual Technical Conference*. Cray Users' Group.

Kim, H., Kim, S., Jeong, J., Lee, J., and Maeng, S. (2013). Demand-based Coordinated Scheduling for SMP VMs. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

Kocoloski, B. and Lange, J. (2012). Better Than Native: Using Virtualization to Improve Compute Node Performance. In *Proc. 2nd International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*.

Kocoloski, B. and Lange, J. (2015). XEMEM: Efficient Shared Memory for Composed Applications on Multi-OS/R Exascale Systems. In *Proc. 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*.

Kogge, P. M. et al. (2008). ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems . Technical Report TR-2008-13, University of Notre Dame CSE Department.

Krieger, O., Auslander, M., Rosenburg, B., Wisniewski, R. W., Xenidis, J., Da Silva, D., Ostrowski, M., Appavoo, J., Butrico, M., Mergen, M., Waterland, A., and Uhlig, V. (2006). K42: Building a Complete Operating System. *SIGOPS Operating System Review*.

Ktap (2016). ktap: A lightweight script-based dynamic tracing tool for Linux. http://www.ktap.org/.

Lange, J., Pedretti, K., Hudson, T., Dinda, P., Cui, Z., Xia, L., Bridges, P., Gocke, A., Jaconette, S., Levenhagen, M., and Brightwell, R. (2010). Palacios and Kitten: New High Performance Operating Systems For Scalable Virtualized and Native Supercomputing. In *Proc. 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

Lange, J. R., Pedretti, K., Dinda, P., Bridges, P. G., Bae, C., Soltero, P., and Merritt, A. (2011). Minimal-overhead Virtualization of a Large Scale Supercomputer. In *Proc. 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*.

LeVasseur, J., Uhlig, V., Stoess, J., and Götz, S. (2004). Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proc. 6th Symposium on Operating Systems Design and Implementation (OSDI)*.

Lo, D., Cheng, L., Govindaraju, R., Ranganathan, P., and Kozyrakis, C. (2015). Heracles: Improving Resource Efficiency at Scale. In *Proc. of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, ISCA '15.

Ma, J., Sui, X., Sun, N., Li, Y., Yu, Z., Huang, B., Xu, T., Yao, Z., Chen, Y., Wang, H., Zhang, L., and Bao, Y. (2015). Supporting Differentiated Services in Computers via Programmable Architecture for Resourcing-on-Demand (PARD). In *Proc. 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

Ma, K.-L., Wang, C., Yu, H., and Tikhonova, A. (2007). In-Situ Processing and Visualization for Ultrascale Simulations. In *Journal of Physics: Proceedings of DOE SciDAC 2007 Conference*.

Maccabe, A. B., McCurley, K. S., Riesen, R., and Wheat, S. R. (1994). SUNMOS for the Intel Paragon - A Brief User's Guide. In *Proceedings of the Intel Supercomputer Users' Group*.

Mantevo (2016). Mantevo Project. https://software.sandia.gov/mantevo.

Mellanox (2003). Introduction to InfiniBand. Technical report, Mellanox Technologies Inc.

Nikolaev, R. and Back, G. (2013). VirtuOS: An Operating System with Kernel Virtualization. In *Proc. 24th Symposium on Operating Systems Principles (SOSP)*.

Ousterhout, J. (1982). Scheduling Techniques for Concurrent Systems. In *Proc. 3rd International Conference on Distributed Computing Systems*.

Ouyang, J., Kocoloski, B., Lange, J., and Pedretti, K. (2015). Achieving Performance Isolation with Lightweight Co-Kernels. In *Proc. 24th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*.

Ouyang, J. and Lange, J. R. (2013). Preemptable Ticket Spinlocks: Improving Consolidated Performance in the Cloud. In *Proc. 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*.

Park, Y., Van Hensbergen, E., Hillenbrand, M., Inglett, T., Rosenburg, B., Ryu, K. D., and Wisniewski, R. (2012). FusedOS: Fusing LWK Performance with FWK Functionality in a Heterogeneous Environment. In *Proc. 24th IEEE International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*.

Perf (2016). perf: Linux Profiling with Performance Counters. https://perf.wiki.kernel.org/.

Petrini, F., Kerbyson, D. J., and Pakin, S. (2003). The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *Proc. 16th International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

Phull, R., Li, C.-H., Rao, K., Cadambi, H., and Chakradhar, S. (2012). Interference-driven Resource Management for GPU-based Heterogeneous Clusters. In *Proc. 21st International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*.

Raghavendra, K. and Fitzhardinge, J. (2012). Paravirtualized ticket spinlocks.

Reiss, C., Tumanov, A., Ganger, G. R., Katz, R. H., and Kozuch, M. A. (2012). Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *Proc. 3rd ACM Symposium on Cloud Computing (SoCC)*.

Riel, R. v. (2011). Directed yield for pause loop exiting.

Shimosawa, T. and Ishikawa, Y. (2009). Inter-kernel Communication between Multiple Kernels on Multicore Machines. *IPSJ Transactions on Advanced Computing Systems*.

Soares, L. and Stumm, M. (2010). FlexSC: Flexible System Call Scheduling with Exception-less System Calls. In *Proc. 9th USENIX conference on Operating Systems Design and Implementation (OSDI)*.

Soltesz, S., Pötzl, H., Fiuczynski, M. E., Bavier, A., and Peterson, L. (2007). Container-based Operating System Virtualization: a Scalable, High-Performance Alternative to Hypervisors. In *Proc. 2nd ACM SIGOPS European Conference on Computer Systems (EuroSys)*.

Sukwong, O. and Kim, H. S. (2011). Is Co-scheduling Too Expensive for SMP VMs? In *Proc. 6th European Conference on Computer Systems (EuroSys)*.

Sysbench (2016). Sysbench. https://github.com/akopytov/sysbench.

Tomita, H., Sato, M., and Ishikawa, Y. (2014). Japan Overview Talk. In *Proc. 2nd International Workshop on Big Data and Extreme-scale Computing (BDEC)*.

TTCP (2016). The TTCP Benchmark. ftp://ftp.sgi.com/sgi/src/ttcp/ttcp.c.

Uhlig, V., LeVasseur, J., Skoglund, E., and Dannowski, U. (2004). Towards Scalable Multiprocessor Virtual Machines. In *Proc. 3rd conference on Virtual Machine Research And Technology Symposium*.

Unrau, R., Krieger, O., Gamsa, B., and Stumm, M. (1995). Hierarchical clustering: A structure for scalable multiprocessor operating system design. *The Journal of Supercomputing*.

VMware (2010). Vmware(r) vsphere(tm): The cpu scheduler in vmware esx(r) 4.1. Technical report.

Wells, P. M., Chakraborty, K., and Sohi, G. S. (2006). Hardware Support for Spin Management in Overcommitted Virtual Machines. In *Proc. 15th International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

Weng, C., Liu, Q., Yu, L., and Li, M. (2011). Dynamic Adaptive Scheduling for Virtual Machines. In *Proc. 20th International Symposium on High Performance Parallel and Distributed Computing (HPDC)*.

Wentzlaff, D. and Agarwal, A. (2009). Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores. *Operating System Review*.

Wheat, S. R., Maccabe, A. B., Riesen, R., van Dresser, D. W., and Stallcup, T. M. (1994). PUMA : An Operating System for Massively Parallel Systems. *Scientific Programming*.

Wisniewski, R., Inglett, T., Keppel, P., Murty, R., and Riesen, R. (2014). mOS: An Architecture for Extreme-Scale Operating Systems. In *Proc. 4th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*.

Xen (2016). Xen Driver Domain. http://wiki.xen.org/wiki/Driver_Domain/.

Yoshii, K., Iskra, K., Broekema, P., Naik, H., and Beckman, P. (2009). Characterizing the Performance of Big Memory on Blue Gene Linux. In *Proc. International Conference on Parallel Processing Workshops (ICPPW)*.

ZeptoOS (2016). ZeptoOS: The Small Linux for Big Computers. http://www.mcs.anl.gov/research/projects/zeptoos/projects/.

Zhang, L., Chen, Y., Dong, Y., and Liu, C. (2012). Lock-Visor: An Efficient Transitory Coscheduling for MP Guest. In *Proc. 41st International Conference on Parallel Processing (ICPP)*.

Zheng, F., Yu, H., Hantas, C., Wolf, M., Eisenhauer, G., Schwan, K., Abbasi, H., and Klasky, S. (2013). GoldRush: Resource Efficient In Situ Scientific Data Analytics Using Fine-Grained Interference Aware Execution. In *Proc. 26th International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.