Achieving Performance Isolation with Lightweight Co-Kernels

Jiannan Ouyang, Brian Kocoloski, John Lange
Department of Computer Science
University of Pittsburgh
{ouyang, briankoco, jacklange}@cs.pitt.edu

Kevin Pedretti
Center for Computing Research
Sandia National Laboratories
ktpedre@sandia.gov

ABSTRACT

Performance isolation is emerging as a requirement for High Performance Computing (HPC) applications, particularly as HPC architectures turn to *in situ* data processing 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. In this paper we claim that these workloads cannot be effectively managed by a single Operating System/Runtime (OS/R). 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.

HPDC'15, June 15–20, 2015, Portland, Oregon, USA.

http://dx.doi.org/10.1145/2749246.2749273.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design

Keywords

Operating Systems, Virtualization, Exascale

1. INTRODUCTION

Performance isolation has become a significant issue for both cloud and High Performance Computing (HPC) environments [6, 25, 9]. This is particularly true as modern applications increasingly turn to composition and *in situ* data processing [21, 32] as substrates for reducing data movement [18] 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 [24, 11, 13]. 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 [31, 14] due to its support of exten-

This project is made possible by support from the National Science Foundation (NSF) via grant CNS-1421585, and by the 2013 Exascale Operating and Runtime Systems Program under the Office of Advanced Scientific Computing Research in the DOE Office of Science. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

sive 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 [23, 30, 28], 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 cross enclave communication to user space via explicit shared memory channels [17], 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) [19]. 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 [20], as well as the potential of properly configured virtualized environments to outperform native environments for certain workloads [16]. In this paper 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.

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

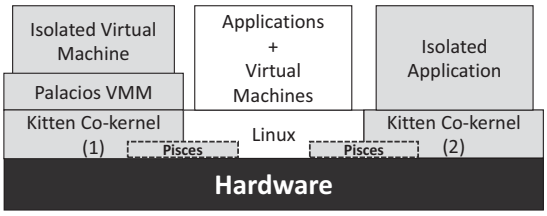


Figure 1: **The Pisces Co-Kernel Architecture**

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.

2. 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 isolated 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 1. 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.

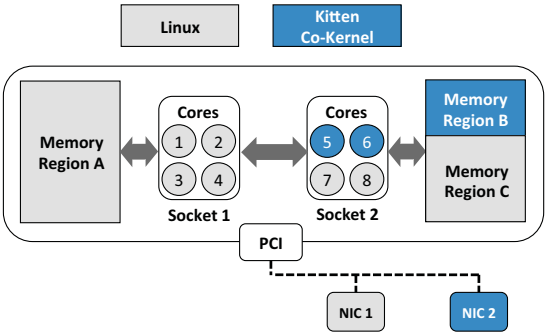


Figure 2: **Example Hardware Configuration with one Pisces Co-Kernel**

Figure 2 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 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.

2.1 Background

Kitten Lightweight Kernel. The Kitten Lightweight Kernel [19] 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 [22], Puma/Cougar [29], and Catamount [15], 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 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 [26] 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 [19, 20] 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).

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

3. 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.

3.1 Cross Kernel Dependencies

A key claim we make in this paper 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 [5] 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. As an example, any system call that allocates memory from a slab allocator will with a certain (though small) probability be forced to undertake additional memory management operations based on the current state of the slab cache. The probability of these edge cases occurring increases along with the system utilization. As more workloads attempt to allocate memory from a slab cache, the probability of it being empty (and so requiring a more heavy weight allocation) increases. Operations such as this directly add overhead to an application, and must be avoided to ensure full isolation. It is for this reason that our architecture explicitly forbids reliance on another external OS/R for system call handling.

Our approach is in direct contrast to other co-kernel approaches that have been proposed [23, 28], 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.

	solitary workload (usecs)	w/ other workloads (usecs)
Linux	3.05	3.48
co-kernel fwd	6.12	14.00
co-kernel	0.39	0.36

Table 1: **Execution time of getpid() (in us).**

To demonstrate how system call forwarding can impact isolation, we conducted an experiment that measured the execution time of several implementations of the getpid() system call under different load conditions. These results, included in Table 1, show how additional workloads (parallel Linux kernel compilations) impact the performance of even simple system calls when they are executed in the same OS/R. For this experiment we evaluated the performance of getpid() natively on Linux as well as handled natively inside a co-kernel. In addition, we implemented a simple system call forwarding interface to measure the overheads associated with forwarding the system call to Linux (co-kernel fwd). For each of these cases we measured the execution time with and without a second workload. Each workload was isolated as much as possible at the hardware layer by pinning CPUs and memory to separate NUMA domains. Our results show that a native Linux environment adds an additional 14% overhead to the execution time of getpid() when running with additional workloads on the system. In contrast our isolated co-kernel shows no additional overhead, and in fact performs slightly better even when the other OS/R is heavily utilized. Finally, our system call forwarding implementation shows a drastic slowdown of 128% when handled by a highly utilized Linux environment. While our system call forwarding implementation is likely not as efficient as it could be, it does demonstrate the effect that cross workload interference can have on an application in a separate OS/R if it is not properly isolated.

3.2 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 [7]) 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 2: **Execution time of sequential reads from a block device**

Table 2 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.

3.3 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

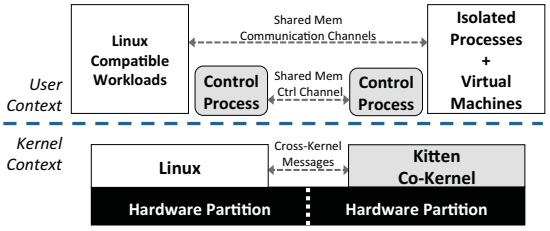


Figure 3: **Cross Enclave Communication in Pisces**

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 [17]. 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 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 3 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.

3.4 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. 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.

4.1 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.

4.2 Communicating with the co-kernel

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 3: **Latency of various Pisces operations.**

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 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 3 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 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.

CPU Cores.

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.

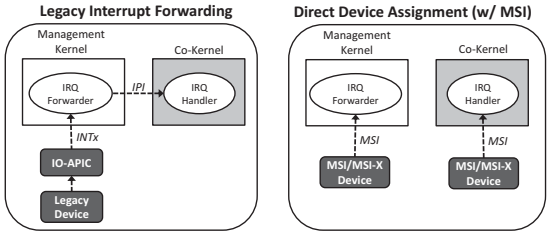


Figure 4: **Interrupt Routing Between Pisces Enclaves**

Memory.

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.

PCI devices.

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-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 4. 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 4.

4.4 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 [23], 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.

5. 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.

5.1 Noise analysis

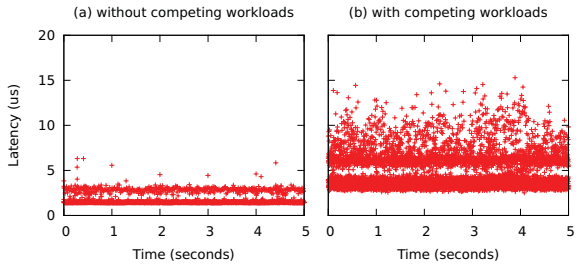


Figure 5: **Native Linux**

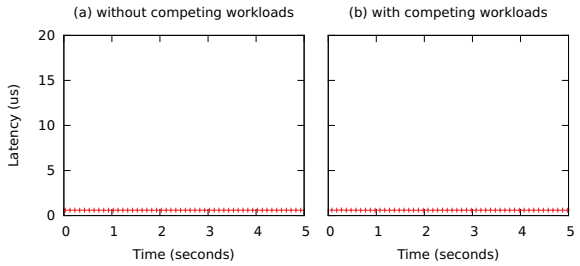


Figure 6: **Native Kitten (Pisces Co-Kernel)**

Our first experiments measured the impact that co-located workloads had on the noise profile of a given environment, collected using the Selfish Detour benchmark [4] from Argonne National Lab. 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. The results of these experiments are shown in Figures 5 and 6. 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.

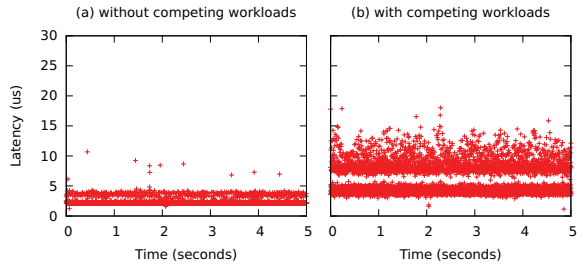


Figure 7: **Kitten Guest (KVM)**

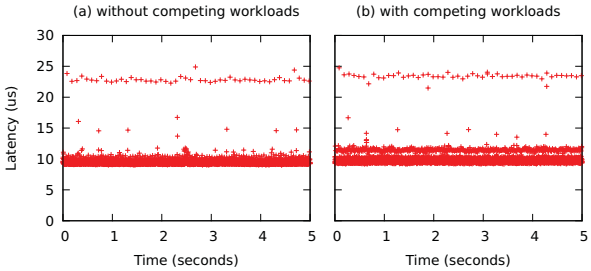


Figure 8: **Kitten Guest (Palacios/Linux)**

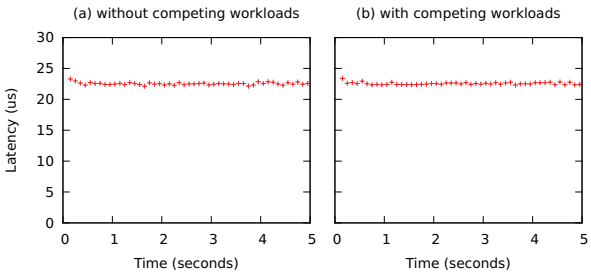


Figure 9: **Kitten Guest (Pisces Co-VMM)**

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), and Palacios integrated with a Kitten co-kernel (Palacios/Kitten). The results are shown in Figure 7, 8 and 9 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 8 and 9), 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 [24, 11, 13]. Thus the 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.

5.2 Single Node Co-Kernel Performance

Figure 10 shows the results from a collection of single node performance experiments from the Mantevo HPC Benchmark Suite [1]. 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 10 demonstrates the performance of several Mantevo mini-applications. 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 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 10 demonstrates memory microbenchmark 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.

5.3 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 [8] 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.

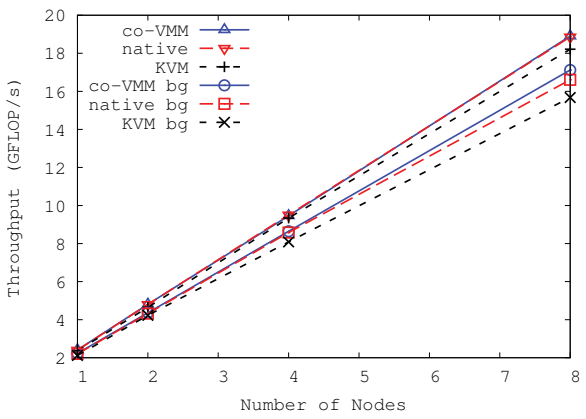


Figure 11: **HPCG Benchmark Performance.** Comparison between native Linux, Linux on KVM and Linux on Pisces co-VMM.

Figure 11 shows the results of these experiments. Without competing workloads, the co-VMM configuration achieves near native performance, while KVM consistently performs worse than both of the other configurations. 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.

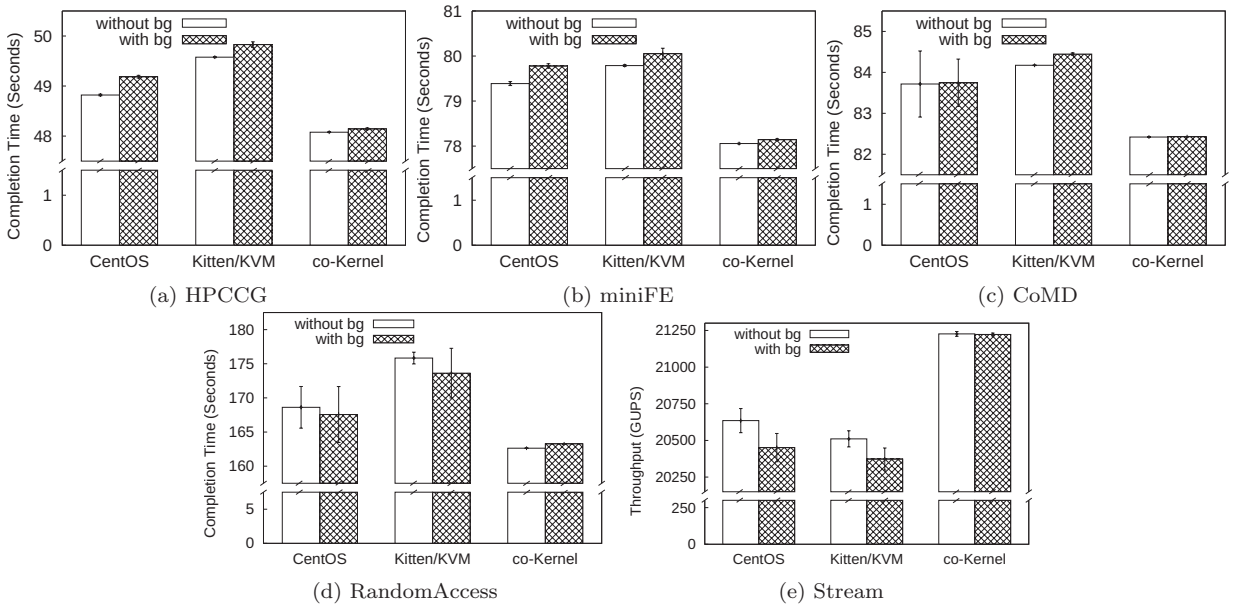


Figure 10: **Single Node Performance (Pisces Co-Kernel)**

5.4 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 [6]. 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.

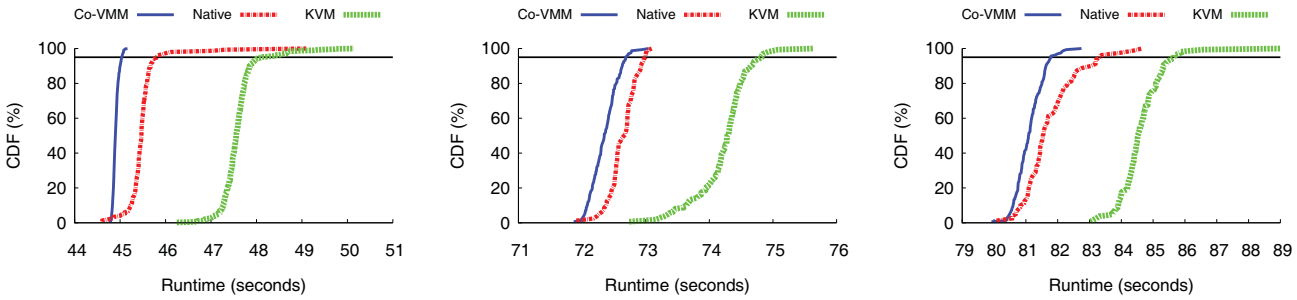
The cloud workload used for these tests was the Mahout machine learning benchmark from the CloudSuite [10] benchmark suite. The HPC benchmarks were taken from the Mantevo benchmark suite, and consisted of HPCCG, CloverLeaf, and miniFE. For these experiments we were focused on detecting performance outliers 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 [24, 11, 13]. 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 12. 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.

6. RELATED WORK

Two separate philosophies have emerged over recent years concerning the development of operating systems 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 OS. 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) [14], 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 [2, 3]. Alternatively, examples using non-Linux based OS deployment can be seen in IBM's Compute Node Kernel (CNK) [12] and several projects being led by Sandia National Laboratories, including the Kitten [19] 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 LWK OSes by providing an environ-



(a) HPCCG (b) CloverLeaf (c) miniFE

Figure 12: **Mantevo Mini-Application CDFs with Hadoop.** Solid horizontal lines shows 95th percentiles.

ment 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 [27], 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.

The most relevant efforts to our approach are FusedOS from IBM [23], mOS from Intel [30], and McKernel from the University of Tokyo and RIKEN AICS [28]. 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, Pisces 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 Pisces 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.

7. CONCLUSION

In this paper 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.

8. REFERENCES

[1] Mantevo Project `https://software.sandia.gov/mantevo`.

[2] ZeptoOS: The Small Linux for Big Computers `http://www.mcs.anl.gov/research/projects/zeptoos/projects/`.

[3] J. Appavoo, V. Uhlig, and A. Waterland. Project Kittyhawk: Building a Global-Scale Computer. *ACM Sigops Operating System Review*, Jan 2008.

[4] P. Beckman, K. Iskra, K. Yoshii, S. Coghlan, and A. Nataraj. Benchmarking the Effects of Operating System Interference on Extreme-scale Parallel Machines. *Cluster Computing*, 11(1):3–16, 2008.

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

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

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

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

[9] M. Dorier, G. Antoniu, R. Ross, D. Kimpe, and S. Ibrahim. CALCioM: Mitigating I/O Interference in HPC Systems Through Cross-Application

Coordination. In *Proc. 28th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2014.

[10] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. 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)*, 2012.

[11] K. B. Ferreira, P. Bridges, and R. Brightwell. 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)*, 2008.

[12] M. Giampapa, T. Gooding, T. Inglett, and R. Wisniewski. Experiences with a Lightweight Supercomputer Kernel: Lessons Learned from Blue Gene's CNK. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–10, Nov 2010.

[13] T. Hoefler, T. Schneider, and A. Lumsdaine. 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)*, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[14] L. Kaplan. Cray CNL. In *FastOS PI Meeting and Workshop*, 2007.

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

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

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

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

[19] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, and R. Brightwell. 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)*, 2010.

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

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

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

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

[24] F. Petrini, D. J. Kerbyson, and S. Pakin. 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)*, New York, NY, USA, 2003. ACM.

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

[26] R. Riesen, R. Brightwell, P. G. Bridges, T. Hudson, A. B. Maccabe, P. M. Widener, and K. Ferreira. Designing and Implementing Lightweight Kernels for Capability Computing. *Concurrency and Computation: Practice and Experience*, 21(6), 2009.

[27] T. Shimosawa and Y. Ishikawa. Inter-kernel Communication between Multiple Kernels on Multicore Machines. *IPSJ Transactions on Advanced Computing Systems*, 2(4):62–82, 2009.

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

[29] S. R. Wheat, A. B. Maccabe, R. Riesen, D. W. van Dresser, and T. M. Stallcup. PUMA : An Operating System for Massively Parallel Systems. *Scientific Programming*, 3:275–288, 1994.

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

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

[32] F. Zheng, H. Yu, C. Hantas, M. Wolf, G. Eisenhauer, K. Schwan, H. Abbasi, and S. Klasky. 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)*, 2013.