Enabling Multi-Stack Software on Partitioned Hardware for Exascale Systems

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

Kevin Pedretti
Sandia National Laboratories
Scalable System Software Department
ktpedre@sandia.gov

***Abstract*—Exascale architectures will exhibit a much greater degree of heterogeneity both at the hardware level as well as in the software that is executed on each compute node. Additionally, more functionality will need to be localized on each compute node due to power and performance constraints. As a result, exascale system software will be tasked with handling a much larger number of co-located and in-situ tasks while at the same time delivering the same level of performance and isolation achievable on a dedicated system. In this paper, we claim that no single system software stack is capable of addressing all of the requirements for every workload executing on a local exascale node. Therefore we believe that future exascale systems should embrace a partitioned multi-stack approach to system software, wherein multiple specialized system software stacks execute in parallel and directly manage disjoint sets of hardware resources. Our approach is based on a multi-stack system software architecture that uses a version of the Kitten lightweight kernel modified to run alongside a Linux based environment on the same local machine. In this architecture, fully independent Kitten (co-)kernels, running concurrently on the same local node, directly manage local hardware resources that have been dynamically allocated into isolated partitions. Furthermore, in combination with the Palacios lightweight virtual machine monitor we demonstrate that we can reduce cross workload interference that results from the co-location of multiple workloads.**

I. Introduction

In this work we present an exascale Operating System/Runtime (OS/R) architecture based on *lightweight co-kernels*. In our architecture, multiple heterogeneous operating system 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 environment that does not rely on any other OS instance for system level services, thus avoiding cross workload contention on OS resources. Each co-kernel is capable of providing fully isolated OS/Rs, or *enclaves*, to local workloads. This approach allows a set of coupled workloads to dynamically compose independent enclaves from arbitrary sets of local hardware resources at runtime based on the applications' resource and isolation requirements. Furthermore, our architecture allows runtime creation and destruction of enclave instances, as well as dynamic resource assignment based on runtime requirements.

Our OS/R architecture targets future exascale systems that, due to the high cost of data movement, will be required to co-locate multiple application components within each compute node, and where each application component will require a different OS/R stack to run most effectively. The expectation is that such systems will include a large number of heterogeneous hardware components, while also supporting much more complex application workloads consisting of multiple distinct application components composed into a larger overall workflow designed to minimize data movement [1]. This presents two new complications for supercomputing OS/R environments: (1) managing the large and diverse collection of computing resources will add additional complexity and state to the OS/R that has heretofore been able to optimize itself around a constrained platform definition, and (2) workloads that used to run on specialized and dedicated systems at petascale (e.g. analytics clusters) will now need to share a single node and OS/R environment as they begin to be deployed as in-situ compositions. Co-kernels address both these problems by distributing the responsibility of managing an exascale compute node between multiple OS/Rs each optimized for the specific hardware and/or application being executed.

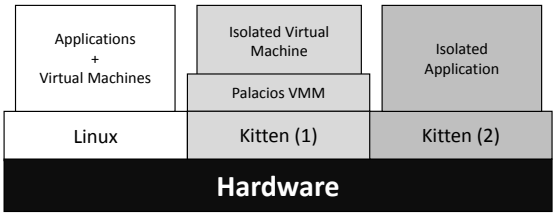


Figure 1: **Co-Kernel Architecture**

Our co-kernel architecture is designed to support this target environment by enabling the dynamic deployment of isolated OS/R instances on a single local compute node. In our scenario enclaves consist of partitions of hardware resources, including CPUs, memory, and I/O devices. These resources can be dynamically allocated and assigned to separate OS/Rs to allow independent management of a single physical machine by multiple concurrent OS environments. A high level overview of our envisioned system architecture is shown in Figure 1. In this environment a single full featured Linux environment provides a management interface to a local node, while one co-kernel executes a simulation workload and a second co-kernel executes an

analysis application in a virtual machine.

As a foundation for this work, we have leveraged our experiences with the Kitten Lightweight Kernel and the Palacios Virtual Machine Monitor [2]. 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 [3], as well as the potential of properly configured virtualized environments to outperform native environments for certain workloads [4]. In this paper we present a novel architecture using both Kitten and Palacios deployed using a co-kernel architecture to provide lightweight environments on systems running full featured OS/Rs.

While previous work has explored the co-kernel concept [5], [6], we claim that our approach is novel in the following ways:

- In our system, hardware resources are *dynamically* partitioned and assigned to specialized OS/Rs running on the same physical machine.
- Our co-kernel architecture is based on 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.
- We leverage the Palacios VMM coupled with a Kitten co-kernel to provide fully isolated environments to other arbitrary OS/Rs not implemented as a co-kernel.

A. High Level Approach

At the heart of our approach is the ability to dynamically partition a subset of a node's hardware resources into a single execution environment capable of supporting a fully independent OS environment. A hardware resource subset is referred as an enclave, which is dynamically constructed based on the runtime requirements of an application. An isolated software stack can be deployed on an enclave and directly manages the hardware resources assigned to this enclave. Furthermore, enclaves also allow dynamic resizing based on changing performance needs. To achieve this, our co-kernel architecture provides the capability to assign and dynamically re-assign hardware resources based on runtime policy decisions.

To fully ensure performance isolation to a given application it is necessary that 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. Our approach does not preclude such architectures, but rather does not require them as a fundamental component of the system. For environments where such shared access is necessary, we intend to rely on user level I/O services similar to microkernel based approaches. Instead we focus primarily on providing hosted workloads with direct access to the underlying hardware devices, relying on the ability to partition and isolate them from the different enclaves in the system.

The ability to dynamically compose collections of hardware resources provides significant flexibility for the system management service. 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. Reclamation requires cooperation between both the co-kernel and management enclave, however an entire enclave can be destroyed and reclaimed unilaterally if need be.

Figure 2 shows an example configuration of a co-kernel based system. In this example case, a full featured Linux kernel 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 directly controlled by a Kitten co-kernel. In addition, the Kitten co-kernel has direct control over one of the network interfaces connected through the PCI bus.

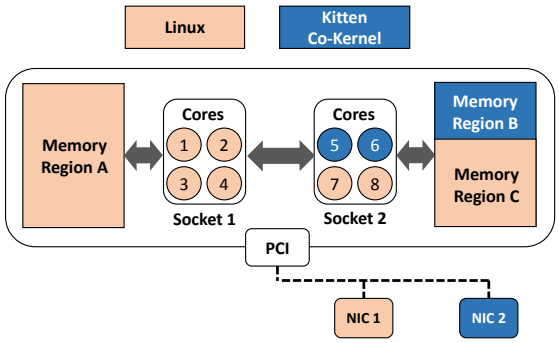


Figure 2: **Resource Assignment to an Isolated Enclave**

Note that decomposing the hardware resources in the manner described above is possible only if the hardware itself supports isolated operation both in terms of performance as well as management. The degree to which we can decompose a local set of resources is largely system and architecturally dependent, and relies on the capabilities of the underlying hardware.

While our primary goal is to provide fully isolated enclaves on a single machine, it is still necessary to provide some form of communication channel across enclaves for management purposes. Our model of operation assumes the existence of a single managerial enclave running a full featured OS/R (in our case Linux). All system management processes execute in the Linux environment, which is responsible for coordinating resource and workload assignments between itself and each locally hosted co-kernel. The only communication necessary for the management tasks is the initial allocation and eventual revocation of resources as well as the initialization and termination of workloads. Each

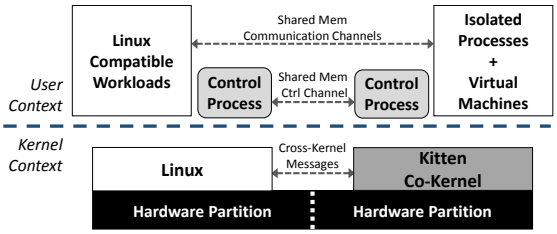


Figure 3: **Cross Enclave Communication**

of these tasks is considered to be a heavy weight operation, that neither requires low latency or a preemptive communication mechanism. Instead all communication is achieved via shared memory regions mapped into process address spaces. Communication with an enclave is bootstrapped by a local control process running on each co-kernel that is responsible for the internal management of the enclave. This control process creates a single shared memory channel with the global management service in the commodity enclave, through which control messages are sent and received and additional shared memory regions are created or attached to.

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.

II. 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 kernel (LWK) 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) [7], 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 [8], [9]. Alternatively, examples using non-Linux based OS deployment can be seen in IBMs Compute Node Kernel (CNK) [10] and several projects being led by Sandia National Laboratories, including the Kitten [2] 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 environment that is somewhat Linux-compatible and can execute a variety of applications built for Linux.

The most relevant efforts to our approach are FusedOS from IBM [5] and the McKernel "hetero operating system" designed by the University of Tokyo and RIKEN AICS [6]. 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 environment in McKernel supports 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.

Co-kernels have a number of advantages over these approaches concerning the construction of multi-enclave exascale system environments. 1) Co-kernels are based on fully isolated OS instances that provide standalone core OS services. Co-kernels handle system calls and interrupts locally without any external dependencies; 2) Co-kernels directly manage their assigned hardware resources including cores, memory, and I/O devices, compared to the the approach of only partitioning CPU cores and relying on Linux resource management for memory and I/O devices; 3) Co-kernels support dynamic enclave resource allocation and revocation, with the ability to dynamically grow and shrink enclaves at runtime; and 4) Co-kernels leverage the Palacios VMM to provide fully isolated environments to other arbitrary OS/Rs that have not yet been modified to operate as co-kernels.

III. Background

Kitten Lightweight Kernel: The Kitten lightweight kernel (LWK) 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). Kitten is similar in design to previous LWKs, such as SUNMOS [11], Puma/Cougar [12], and Catamount [13], 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. Kitten is open-source software released with a GPLv2 license. [1]

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 [14] 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-sized-fits-all policies. For example, Kitten uses a simple virtual memory management scheme that maps large chunks of physically contiguous memory to application address spaces up-front at application launch time. This has a number of advantages, including making memory layout predictable across nodes, minimizing memory performance variability, enabling straightforward use of large virtual memory page sizes, and simplifying network stack address translation. Kitten's simple structure was a good match for developing the co-kernel approach described in this paper, both in terms of enabling rapid development and achieving good runtime performance, as shown in Section V.

Palacios VMM: Palacios 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], [3].

IV. Kitten Co-Kernel Architecture

Our co-kernel architecture was implemented by extending 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. Our co-kernel implementation includes the following components:

1) A custom boot loader implemented as 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.

During our implementation effort 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 co-kernel components are contained to a single kernel module that provides boot loader services and a set of user-level management tools implemented as Linux command line utilities.

The co-kernel itself is a highly modified version of the Kitten lightweight kernel as previously described. [2] 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. The majority of the modifications focused on removing the default resource discovery mechanisms and replacing them with explicit assignment interfaces called by a user-space co-kernel 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, we are able to allow a running co-kernel to assume direct control over it. In this way, even though both Linux and our 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.

A. Booting a Co-kernel

Initializing a Kitten co-kernel is done by invoking a set of commands from the Linux management enclave. First a

[1] Kitten source code: git clone https://software.sandia.gov/git/kitten

[2] All Kitten changes have been committed to the public git repository.

single CPU core and memory block (typically 128 MB) is taken offline, and removed from Linux's control. A Linux module is then activated which 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 kernel image and init task are then copied into the memory block below the boot parameters. Once the boot environment is initialized the boot loader 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 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 our modified boot loader that initialization is complete, at which point the boot loader 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.

B. Communicating with the co-kernel

To allow management of the co-kernel enclave, the init task that is launched during boot contains a small control process that is able to communicate back to the Linux environment. This allows a 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 that allows only a single message at a time, and is used entirely for communication with the control process in the co-kernel. Message pending notifications are achieved by a dedicated IPI vector inside the 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. These requests are only used to allow the loading of VMs and applications from the Linux file system. 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. The reason for this is that the co-kernel's performance could be adversely impacted if there was contention in the Linux environment that prevented the requests from being completed in a timely manner. 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, however, as we will describe later, more full featured applications can still benefit from the isolation of a co-kernel through the use of virtualization.

C. 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, we have implemented 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 our co-kernel architecture 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 our modified version. At this point a command is issued from our boot loader module 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 target core. The CPU is then initialized and activated inside the Kitten environment.

Reclaiming a CPU is done in a similar manner. A reclamation command is issued to the co-kernel's control process, which then issues a corresponding system call into the Kitten kernel. Kitten suspends the core's CPU scheduler and migrates any local tasks to other active CPUs. The CPU is then marked inactive and removed from Kitten's internal state. At this point control is returned to the management

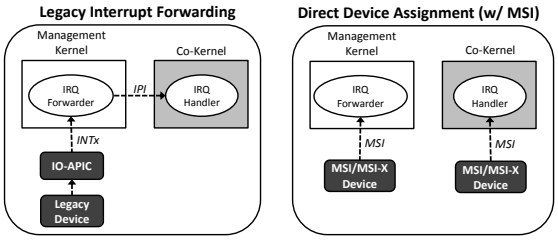


Figure 4: **Interrupt Routing Between Enclaves**

enclave and the CPU is brought back online and returned to the Linux environment.

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 our 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 we first detach it from the assigned Linux device driver and then offline it from the system. While we could simply pass control of the device over the to co-kernel at this point, we instead use an IOMMU (if available) to restrict the device from accessing memory outside the enclave. The IOMMU is configured by our Linux kernel module without involvement from the co-kernel itself. This is possible because our system 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 us to dynamically update the IOMMU as memory is assigned and removed from an enclave, but this is trivial as all control paths cross through the same kernel module. 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 enclave-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 the co-kernel to simply request the delivery of all device interrupts since it is possible that it is not the only recipient. To address this issue, we incorporate an IRQ forwarding service in the Linux enclave. 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 our co-kernel 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.

D. Integration with 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 a significant set of applications from gaining the isolation and performance benefits that come from our multi-stack environment. While other work has addressed this problem by offloading unsupported features to the Linux OS [5], we have taken a different approach due to the possibility of performance interference caused by overheads and contention in the Linux environment. We have instead chosen to leverage our work with the Palacios VMM to allow unmodified Linux applications to execute inside an isolated Linux environment running as a VM on top of a co-kernel. This approach allows us to provide the full set of features available to the native Linux environment while also providing isolation from contention inside that environment. As we will show later, our approach 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 our co-kernel environment. Primarily, we had to add support for the dynamic resource assignment behavior of the underlying co-kernel. These modifications mostly 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 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.

V. Evaluation

Our evaluation is designed to demonstrate the ability of the co-kernel architecture to provide a consistent level of performance to HPC workloads, particularly in the face of competing workloads on the same physical machine.

A. System Configuration

Our experiments were conducted on an eight node cluster of Dell R450 servers 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 installed with CentOS 7 running Linux Kernel version 3.16, augmented with our multi-stack tools. We take advantage of each server's two NUMA nodes to enforce performance isolation between enclaves. In "CentOS" mode, the boot CentOS 7 OS/R runs on both NUMA sockets and the target workload is evaluated running in this non-multi-stack configuration. In the "Kitten Co-Kernel" and "Co-VMM" modes, our multi-stack tools are used to dynamically create a Kitten/Palacios enclave on the second NUMA node of each server.

B. Noise analysis

Our first experiments compare the noise profiles of Linux and an isolated co-kernel with and without the presence of competing workloads. For this evaluation we used the Selfish Detour benchmark [15] from Argonne National Lab. For each experiment we ran the benchmark for a period of 10 seconds, first with no competing workloads and then in combination with a parallelized Linux kernel compilation running on Linux. In this configuration, the Selfish Detour benchmark is pinned to the second NUMA zone, and the kernel compilation is pinned to the other zone. The results of these experiments are shown in Figures 5 and 6. As can be seen, the co-kernel provides a dramatically lower noise profile. 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 added. It is important to note that the same hardware resources were assigned to both the Kitten co-kernel and native Linux Selfish benchmarks, meaning that the additional noise is a direct result of system software contention in the Linux kernel.

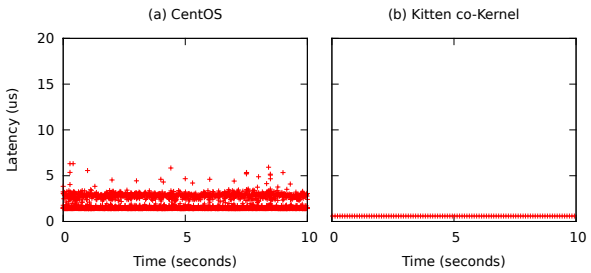


Figure 5: **Noise without Competing Workloads**

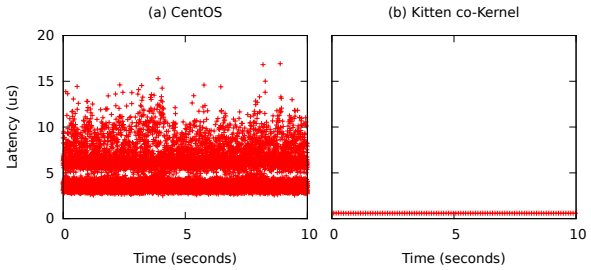
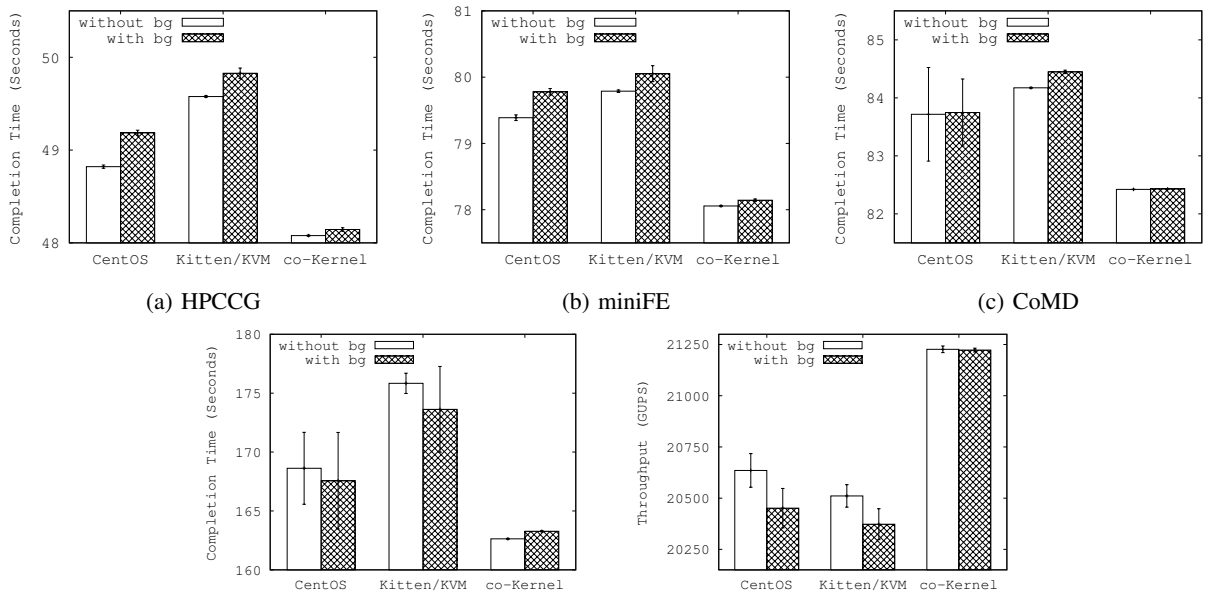


Figure 6: **Noise with Competing Workloads**

Next we evaluated the effect of virtualization on noise levels seen in a guest environment. The goals of these experiments were twofold: First, to show that a Linux based virtualization environment (KVM) is not capable of isolating a guest environment from the noise in the host environment, and second, to show that virtualized environments can exhibit low noise levels when deployed on top of lightweight co-kernel. For these experiments we used the same workload deployment as the previous experiment, but instead ran the Selfish benchmark two different virtualized environments. First, we ran the benchmark on Kitten running in a KVM guest environment. As shown in Figure 7, without a competing workload the Kitten VM environment is still able to provide a consistent noise profile though with a slightly higher baseline. However, once a competing kernel compilation is added, the VM environment experiences a significant amount of interference from the underlying host environment resulting in a large increase in the level of noise. Next, we deployed the Selfish benchmark in a Linux VM running on the Palacios/Kitten co-kernel environment. As the results in Figure 8 show, the co-kernel is able to effectively isolate the virtual machine from the effects of competition in the native Linux environment.

C. Single Node Co-Kernel Performance

Next we evaluated the single node performance characteristics of our co-kernel approach. For these experiments we selected two memory micro-benchmarks and a set of



(a) HPCCG (b) miniFE (c) CoMD

(d) RandomAccess (e) Stream

Figure 9: **Single Node Co-Kernel Performance.**

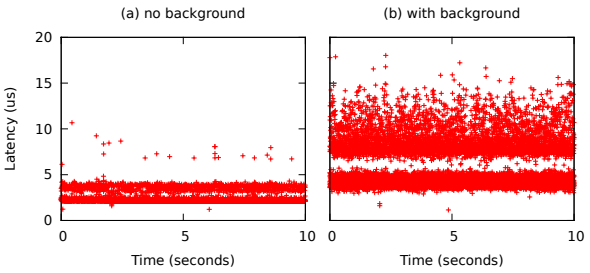


Figure 7: **KVM Kitten Guest Noise**

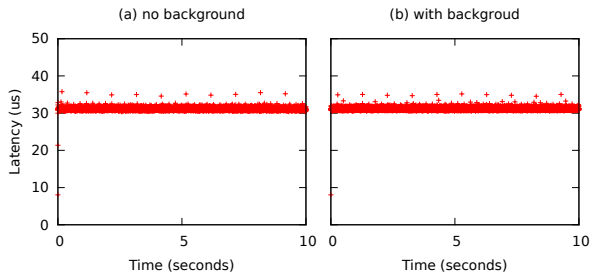


Figure 8: **Co-VMM Linux Guest Noise**

three mini-applications from the Mantevo HPC Benchmark Suite [16] capable of executing in a Kitten environment. We executed 10 runs of each benchmark 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 architectures 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 bottom row of Figure 9 demonstrates the memory micro-benchmark performance with and without competing workloads in the different system configurations. The Stream results demonstrate that Kitten provides consistently superior memory performance to either of the other system configurations, with memory access throughput increased by 2.9% and 3.8% for isolated and co-located workload deployments, respectively. Both additional system configurations demonstrate the performance degradation caused by the competing kernel compilation. Similarly, the RandomAccess results demonstrate similar behavior in terms of average performance improvement, but also demonstrate that the additional workload causes increased variance in the performance of the KVM and Linux configurations.

The top row of Figure 9 demonstrates the performance of the Mantevo mini-applications in these environments. In all cases, the Kitten co-kernel exhibits the best overall performance. In addition the co-kernel environment also exhibits much more consistent performance, than the other system configurations, represented by large standard devi-

ation bars for Linux executing the CoMD benchmark, as well as KVM executing the miniFE benchmark with the competing kernel compilation. These results suggest that the co-kernel architecture is likely to exhibit better scaling behavior to larger node counts than either alternate system configuration.

D. Co-Kernel Scalability

Next we evaluated whether the performance improvements provided by the co-kernel environment on a single node would translate to a multi-node benchmark configuration. For this experiment we deployed the HPCG [17] benchmark from Sandia National Labs across the 8 nodes of our experimental cluster. Because Kitten does not currently support our particular Infiniband hardware, these experiments compare a co-kernel VMM architecture to KVM and native Linux environments as in the previous experiments. For the guest environments, we used a stripped-down Busybox [18] Linux environment. As in the previous sections, our workload configurations consist of an isolated configuration running only the HPCG benchmark, as well as a configuration with competing workloads consisting of three parallel kernel compilations configured to run on all 6 cores of a single NUMA socket. This workload, which was duplicated on all 8 nodes of the cluster, was designed to introduce contention on each local system's resources.

Figure 10 shows the results of these experiments. When running without competing workloads, the co-kernel VMM configuration achieves near native performance, while KVM shows a trend of performance degradation as the benchmark scales beyond a single node. Furthermore, as the competing workload scales up, the co-kernel VMM configuration begins to show better performance than both the native Linux and KVM configurations. The performance at 8 nodes is further depicted in Table I and shows that the execution environment provided by the co-kernel VMM environment is much more consistent than either the native Linux or KVM configurations. This can be seen by the increase in standard deviation when compared to the same configurations executing without competing workloads. In particular, the coefficient of variation ($stdev/mean$) demonstrates that, although the competing workload does have a minor impact on the co-VMM architecture, the corresponding impact on the other system configurations is much more substantial. Given these results, the co-kernel VMM configuration, which is beginning to exhibit a divergence at the largest scale of the benchmark, is likely to continue to diverge as the scale increases further.

	CentOS	KVM	Co-VMM
Mean	-11.94%	-13.92%	-9.41%
Coefficient of Variation	+313.97%	+646.01%	+4.03%

Table I: **Impact of Background Workloads on the HPCG Benchmark Executing on 8 Nodes.**

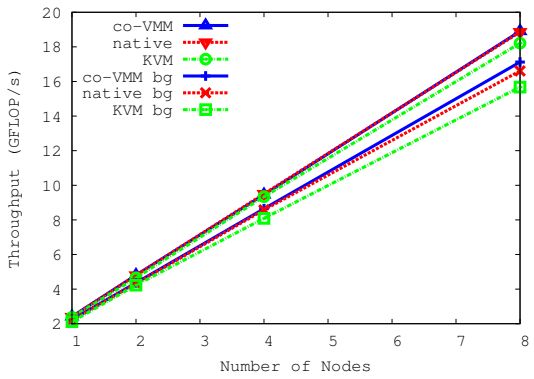
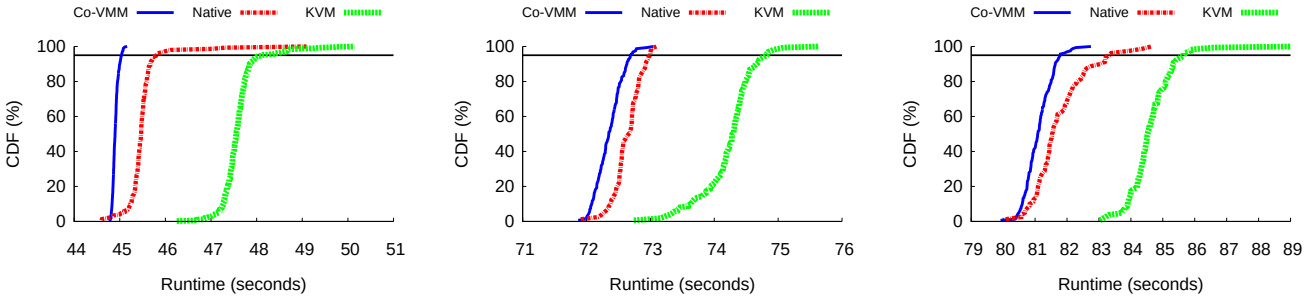


Figure 10: **HPCG Benchmark Performance.** Comparison between native Linux, Linux on KVM and Linux on co-VMM.

E. Co-Kernel Performance Isolation

Finally, we studied the ability of the co-kernel architecture to effectively support larger scale HPC application deployments under realistic workload deployments likely to be seen on an exascale machine. This evaluation focuses on two types of application workloads: HPC simulations and data analytics. We are particularly interested in analyzing the abilities of the co-kernel environments to provide effective isolation for HPC workloads over the course of long-running sustained competition from data analytics workloads. For these experiments we measured whether the environments provided by various system configurations could sustain their level of performance over much longer periods of time than were measured in previous experiments. Thus, in these experiments, each benchmark is executed for a period of multiple hours, allowing us to collect a much more detailed evaluation of the isolating capabilities of the environments in questions. For the data analytics workload, we ran the Mahout machine learning benchmark selected from the CloudSuite benchmark suite [19], which uses the open source Hadoop framework for parallel computation. For the HPC workloads we selected three mini-applications from the Mantevo benchmark suite. The experiments were again conducted on our 8-node cluster with the HPC and data analytics workloads configured to run on isolated hardware partitions, using the workload isolation capabilities of the individual system environments.

The results of our experiments are demonstrated via cumulative distribution functions (CDFs) and are reported in Figures 11. The CDFs show at least three hours of repeated executions of the HPC benchmarks as well as the analytics workloads. As the figure demonstrates, the co-VMM provides the most consistent HPC-capable environment in the face of sustained competition from the analytics workload. The co-kernel VMM configuration exhibits the least variability, particularly around the 95th% percentiles while both the native Linux and KVM configurations exhibit outliers that achieve significantly degraded performance.



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

Figure 11: **Mantevo Mini-Application CDFs with Hadoop.** Solid horizontal lines shows 95th percentiles.

VI. Conclusion

In this paper we have presented a lightweight co-kernel architecture as system software solution for exascale supercomputers. Our approach uses the Kitten lightweight kernel modified to run alongside a Linux based environment on the same local machine. The Kitten co-kernel directly manages local hardware resources that have been dynamically allocated into isolated enclaves. Furthermore, in combination with the Palacios lightweight virtual machine monitor, Kitten co-kernel provides fully isolated virtual environments to arbitrary OS/Rs. Our evaluation demonstrates that the co-kernel architecture achieves better isolation than Linux in the face of locally competing workloads. Moreover, the Kitten co-kernel shows superior performance as well as orders of magnitude decrease in performance variability at scale.

References

[1] P. M. Kogge *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," University of Notre Dame CSE Department Technical Report, TR-2008-13, Tech. Rep., September 2008.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] T. Shimosawa and Y. Ishikawa, "Inter-kernel Communication between Multiple Kernels on Multicore Machines," *IPSJ Transactions on Advanced Computing Systems*, vol. 2, no. 4, pp. 62–82, 2009.

[7] L. S. Kaplan, "Lightweight Linux for High-Performance Computing," Dec 2006. [Online]. Available: http://www.networkworld.com/article/2301501/software/lightweight-linux-for-high-performance-computing.html

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

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

[10] 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)*, Nov 2010, pp. 1–10.

[11] 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.

[12] 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*, vol. 3, pp. 275–288, 1994.

[13] 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.

[14] 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*, vol. 21, no. 6, 2009.

[15] 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*, vol. 11, no. 1, pp. 3–16, 2008.

[16] "Mantevo Project https://software.sandia.gov/mantevo."

[17] J. Dongarra and M. A. Heroux, "Toward a new metric for ranking high performance computing systems," *Sandia Report, SAND2013-4744*, vol. 312, 2013.

[18] "Busybox Project http://www.busybox.net/."

[19] 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 *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, 2012.