Covirt: Lightweight Fault Isolation and Resource Protection for Co-Kernels

Nicholas Gordon and John R. Lange
Department of Computer Science
University of Pittsburgh
Pittsburgh, US
{nick.gordon, jacklange}@cs.pitt.edu

***Abstract*—The challenges of the exascale era have generated a number of advancements in HPC systems software, with co-kernel architectures emerging as one such novel approach for HPC operating system and runtime (OS/R) design. Co-kernels function by running multiple specialized, lightweight OS kernels natively on the same host as a general purpose OS/R. These specialized kernels are able to provide optimized OS/R environments for HPC applications while still retaining access to the full feature set of the co-running general purpose OS/R. While co-kernels are able to effectively optimize for performance, they generally lack effective mechanisms for cross OS/R fault isolation and resource protection. In this paper we present *Covirt*, a lightweight OS/R protection layer that leverages the hardware virtualization features found on modern CPUs. Covirt interposes a minimal hypervisor layer between a co-kernel OS/R and hardware to prevent OS level faults from impacting other OS/Rs running on the same system. Covirt is different from other virtualization-based approaches due to the level of integration necessary between the co-kernel instances, requiring the support of higher level semantic interfaces between the different OS/Rs. Covirt features a split architecture consisting of a hypervisor and controller module that continuously monitors changes to the underlying resource partitioning and translates those events to hypervisor configuration changes. We have implemented a prototype of Covirt in the context of the Hobbes exascale OS/R stack, specifically targeting the Pisces co-kernel framework and Kitten Lightweight Kernel. Our evaluation shows that Covirt is able to add fault isolation for memory and interrupt processing with minimal performance overheads.**

***Index Terms*—virtualization, hardware virtualization, co-kernels**

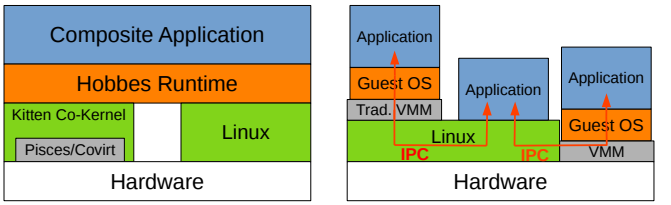
I. Introduction

As we enter into the exascale era, significant effort has been put into exploring the use of specialized Operating System/Runtime (OS/R) environments for HPC and supercomputing platforms. The result of this effort is a new approach for coupling specialized OS/Rs with more general purpose OS/R environments that supports the native execution of both OS/R instances concurrently on the same physical system. Multiple efforts have explored this multistack or co-kernel approach as a means of providing optimized environments for complex HPC applications [1]–[3]. In a co-kernel architecture the local hardware resources are partitioned into isolated *enclaves*, each of which is managed by an independent OS/R instance. Typically, these co-kernels execute a Lightweight Kernel (LWK)-based OS/R to provide maximal performance with minimal OS overheads. The key advantage of a co-kernel architecture is the ability to have specialized and general kernels co-exist on a system in order to obtain the benefits of both. This is achieved by allowing applications running in a LWK co-kernel to offload heavy-weight operations to the general purpose OS/R, while performance-critical operations can run directly in the LWK. This requires that co-kernel environments support high-level semantic interfaces (such as system calls) to bridge the OS/R boundaries, which in turn requires both OS/Rs to share access to a subset of the process's state (such as its memory map).

The ability to tightly integrate processes running in separate co-kernel OS/Rs has enabled support for emerging complex application compositions [4]–[7]. Co-kernels allow the decomposition of these applications at the process and thread levels across multiple OS/R environments, while still supporting the higher level IPC mechanisms necessary to support the integrated operation of the application as a whole. An example system configuration based on the Hobbes OS/R environment [4] is shown in Figure 1a. Using Hobbes, complex applications are able to span multiple enclaves seamlessly by leveraging the IPC mechanisms provided by the underlying Hobbes runtime environment. The benefit of this approach is a consistent high-level API for composing applications that can automatically adapt to arbitrary enclave topologies. However it does require a significant amount of state sharing between enclaves that must be synchronized and kept consistent by the underlying co-kernel architecture.

While co-kernel approaches have been shown to provide performance benefits, they do so at the cost of expanding the system's trusted computing base (TCB). This is because each co-kernel instance runs as a native OS with full access to the underlying hardware resources. This leads to a situation where errors and bugs in a single co-kernel instance can escape the originating enclave and produce additional errors in other enclaves or crash the entire system. This issue is exacerbated by the need for tightly coordinated resource sharing to support cross-enclave runtimes, where state inconsistencies between enclaves can lead to resource assignment conflicts. While

This material is based upon work supported by the National Science Foundation under Grant No. 1718287



(a) The Hobbes OS/R.

(b) Other application composition approaches.

Fig. 1: Co-kernel applications vs typical applications.

traditional virtualization techniques are capable of addressing this problem (by running each co-kernel in a dedicated virtual machine), they have so far been rejected due to the perceived overhead cost of enabling virtualization features.

In this paper we present *Covirt*, a virtualization-based approach for co-kernel fault isolation and resource protection that minimizes performance overheads. The goal of Covirt is to quantify the baseline overheads introduced through the use of hardware virtualization. While Covirt leverages hardware virtualization features, it does so in a way that is fully transparent to each co-kernel OS/R and does not impact how the OS/R interacts with the underlying hardware or other system software components. Covirt is different from other low overhead, no-abstraction VMM architectures due to its ability to dynamically track configuration changes that require updating the underlying virtualization configuration. This means that the existing cross-enclave resource sharing interfaces are supported directly, and do not require the addition of abstracted APIs implemented through virtual devices or paravirtual interfaces. Furthermore, configuration updates are done asynchronously with respect to the hypervisor's execution and impose minimal overheads due to update latencies.

Covirt consists of two components: (1) a lightweight hypervisor layer that adds resource protection features using hardware virtualization extensions, and (2) a controller module that integrates with the co-kernel management framework to monitor changes in the resource assignment and sharing configuration. Configuration changes are detected as they happen and any change to the hypervisor's state is done remotely via the controller module. The hypervisor is notified of configuration changes only when it is necessary to reload the virtual hardware state or flush local caches. Covirt is implemented inside the Pisces co-kernel framework and specifically targets Intel VMX virtualization extensions. However, we believe the overall approach is generalizable to other co-kernel architectures such as IHK/McKernel [2] and MOS [3].

In this paper we make the following contributions:

- We describe the design and implementation of Covirt, a lightweight fault isolation and resource protection service for co-kernel architectures, targeting common state inconsistency errors.
- Using Covirt, we quantify the incremental overhead costs of different hardware protection features provided by the virtualization extensions on a modern Intel system.
- We demonstrate that hardware virtualization features are a viable approach to providing inter-OS/R resource protection by evaluating the overheads imposed by Covirt on a set of representative HPC application benchmarks.

II. Background

A. Co-kernels

Co-kernels are an OS/R paradigm that divides system resources into hardware partitions, or *enclaves*, where each enclave runs an OS/R that has full, unrestricted access to the resources assigned to that enclave. There is no restriction on what may run in these enclaves, as each enclave is fully independent. This approach allows arbitrary sets of hardware resources to be flexibly assigned to enclaves, at runtime, and then those enclaves to be composed to suit application and performance requirements. The degree and method of cross OS/R integration between enclaves varies per approach, ranging from source code-level integration to explicitly defined and portable APIs, but each approach employs a co-operative paradigm for managing system resources. Co-kernels have been extensively studied in the context of HPC, where they have enabled the continued use of Lightweight Kernels (LWKs) even as most (if not all) original equipment manufacturers (OEMs) move to general purpose Linux based OS/R environments. Existing efforts have shown that pairing an LWK co-kernel with a general purpose Linux kernel allows HPC systems to gain the performance advantages of an LWK without sacrificing the usability and environmental familiarity that comes with a Linux-based environment [1]. Furthermore, a co-kernel approach allows applications running in an LWK to retain full access to the Linux device driver ecosystem [8], eliminating an engineering hurdle that has long been a significant challenge for LWK architectures.

B. The Hobbes OS/R

In this paper we specifically focus on the Hobbes OS/R environment that supports application composition across arbitrary co-kernel topologies. Hobbes [4] is an HPC-focused OS/R that emphasizes application composition as a primary design goal, and contains a number of different architectural components, including the Pisces co-kernel framework [1], the Kitten LWK [9], the Palacios Virtual Machine Monitor [9], the XEMEM shared memory library system [10], and several integrated communication/IO frameworks such as ADIOS [11] and TCASM [12]. Hobbes enables composite applications that are agnostic to the kernel(s) they are running on. As shown in Figure 1a, the Hobbes runtime spans multiple enclave boundaries, enabling a single application to run transparently with individual components executing on separate, specialized OS environments.

Pisces: The Pisces framework [1] serves as the foundation for the Hobbes OS/R. Pisces is a lightweight co-kernel framework that allows the partitioning of a system's resources into separate enclaves, in which an independent OS/R can execute. The running OS fully manages its own resources,

including cores, memory, and devices. Pisces enclaves are dynamically allocated and initialized at runtime and can be created and destroyed in response to changing workload requirements. Pisces itself is run as a Linux kernel module on an otherwise-unmodified Linux host OS. While capable of hosting arbitrary co-kernel OS/R architectures, Pisces was specifically designed to target the Kitten Lightweight Kernel.

Kitten: Kitten [9] is a special-purpose OS kernel that provides a simple, lightweight, and POSIX-like environment specifically suited to HPC applications. Kitten is similar to previous LWKs, such as Catamount [13], which have been deployed on supercomputers in the past. One of Kitten's main goals is to execute workloads with high performance and high repeatability. This is achieved by simple resource management policies, such as contiguous physical memory and direct network hardware access. Moreover, Kitten is partially derived from Linux, giving it a POSIX-like, modern code base and improved Linux API and ABI compatibility that makes utilizing existing HPC toolchains easier. When combined with Pisces, Kitten is able to execute in a co-kernel mode, allowing it to manage a partition of local resources using low-overhead and low-noise subsystems.

XEMEM: Hobbes uses the XEMEM shared memory system to handle cross enclave communication. XEMEM is a shared memory architecture that supports inter-enclave application-level shared memory regions. XEMEM is an expansion of, and is backward compatible with, SGI/Cray's XPMEM system which allows processes to share arbitrary portions of their address spaces with each other. XEMEM provides a global view of shared memory through the use of XPMEM segment IDs managed across the entire system by a node-local name service. XEMEM allows applications based on the XPMEM library to be easily ported to Hobbes, through slight modifications to the XPMEM API calls. In many cases these changes are fully transparent to the application, as they are internalized into higher level communication libraries, such as Adios [11] or TCASM [12]. XEMEM serves as the basis for all inter-enclave application communication as well as OS services such as system call forwarding.

III. Related Work

A. Co-kernels/Multi-kernels

The co-kernel paradigm has been studied considerably already within the context of HPC and for most projects the overall goals are the same; provide a lightweight execution context for performance-critical applications and delegate the rest to a general purpose OS. These projects can be sorted based on two characteristics: to what degree are the co-kernels integrated, and which kernel is the host OS. Kernel integration is granular and includes a variety of system-level decisions that affect the degree that OS and process state is exposed between kernels, as well as what work can be delegated. The choice of host OS is comparatively narrow by comparison, but affects resource isolation and fault isolation.

IHK/McKernel [2] takes a similar approach in many ways to Hobbes, except the degree of integration between the co-kernel and host OS, Linux, is substantially higher. IHK/McKernel incorporates a "proxy process" on the host OS that requires address space replication in order to support system call delegation. HermitCore [14] is similar to Hobbes and IHK/McKernel but uses unikernels/library OSes for its co-kernel, providing a similar co-kernel framework that reduces the TCB in exchange for requiring application compatibility with their unikernel framework. mOS [3] departs from the previous approaches and sits at the extreme end of the integration axis and squarely in the middle of the host OS choice axis, fully embedding the LWK code into Linux so that the LWK code runs on cores picked at boot-time, so that state sharing between the two OSes is high and LWK processes are nearly indistinguishable from Linux processes. The FFMK/L4 [15] approach runs the L4 microkernel as the host OS and paravirtualizes Linux to provide delegation for LWK processes, requiring all syscalls to be delegated to the paravirtualized Linux and providing weak notions of resource partitioning to maximize performance gains. FusedOS [16] assumes a heterogeneous hardware platform and runs the LWK as a process on Linux, indicating a high degree of system integration that requests specialized hardware resources to run application code on top of a small state monitor that forwards exceptions and interrupts back to the LWK process via Linux.

While each of these co-kernels represent a unique point in the design space, they are all similar in their high level of dependence on co-kernel correctness. Since each of these approaches requires native execution of the co-kernel OS/R, they are limited in their ability to provide fault protection and resource isolation. As such, Covirt represents a unique capability that could be adapted to suit the full range of co-kernel approaches.

B. Hardware Virtualization Features

Virtualization has been studied extensively in the cloud environment, where it is used to manage execution, partition resources, and provide protection. However, cloud computing has requirements and restrictions very different to HPC, so generally cloud research cannot be directly applied to the co-kernel context. Broadly, there are three main obstacles faced by existing hardware virtualization approaches that Covirt overcomes: static resource assignment, virtualization performance overheads, and poor IPC support.

Two systems that explicitly target low overhead and near-native performance capabilities using minimal hypervisor frameworks are DirectVisor [17] and OSV [18]. DirectVisor improves cloud performance while retaining manageability by primarily utilizing hardware virtualization, but temporarily falling back to paravirtualization to continue supporting critical manageability features. Their approach enforces statically-configured DirectVMs that are managed by a central monitor based on QEMU/KVM. In a similar vein, OSV provides VMs that use hardware virtualization, together with a privileged OS that owns and exposes a virtual interface for devices like NICs. The privileged OS requires handling and forwarding of

interrupts, and VMs have limited communication to each other through shared memory.

What separates these systems from Covirt is that they lack the capabilities needed to fully support a co-kernel approach. Co-kernel architectures rely heavily on cross-OS/R IPC mechanisms that are not directly supported by previous work. Instead, IPC support is handled through either virtual hardware or paravirtual interfaces through the hypervisor. We illustrate three systems in figure 1b: full virtualization, direct application execution, and a direct-hardware approach. In each case, IPC interfaces are mediated by the underlying virtualization layer, requiring added overhead for any communication spanning an OS/R boundary. In contrast, Covirt is designed to provide zero overhead IPC mechanisms that do not require any invocation of the virtualization layer, instead relying on synchronization of shared hardware resource mappings to support direct communication.

IV. Covirt

Covirt is a virtualization-based approach providing protection from crash-faults, hardware access violations, and state corruption caused by bugs or compromised behaviors in co-kernel OS/Rs. Existing co-kernel architectures rely on the full cooperation and good-behavior of each co-kernel OS/R to ensure that the system is able to operate without errors. This is due to the fact that each co-kernel instance has full access to the entirety of the underlying hardware platform, and nothing fundamentally prevents a co-kernel OS/R from accessing arbitrary memory regions, sensitive hardware registers, or device I/O ports. Instead, co-kernel architectures assume that each co-kernel will constrain itself to the hardware resources that have been explicitly assigned to it. For example, memory resources are protected by each co-kernel OS/R configuring its memory map to only include those regions it has been assigned or granted shared access to. Yet, nothing prevents a co-kernel from misconfiguring its memory map, unintentionally or not, and accessing memory addresses belonging to other OS/Rs or devices' memory mapped I/O regions. These voluntary restrictions extend beyond memory to other hardware resources such as MSRs, I/O ports, and interrupt handling.

While this resource protection problem is alleviated somewhat by the relatively small size of current LWK designs, it is exacerbated by the fact that co-kernels require coordination and state synchronization across OS/R boundaries to support the high-level interfaces needed by these environments to support complex application compositions. This can lead to errors when a co-kernel's view of its hardware resource assignment gets out of sync with other OS/Rs. In this case, even if a co-kernel is operating correctly based on its own view of the current system configuration, it might in fact be accessing hardware it should not. This requires that dynamic changes to the hardware assignment configuration are universally applied in order to maintain system correctness. The dynamically-changing assignments of hardware resources at fine granularity pose a significant challenge for addressing this issue with traditional virtualization-based approaches, since they typically assume coarse-grain resource assignment, relatively static configurations, and limited resource sharing.

Covirt addresses these challenges through the use of a lightweight hypervisor that is managed by a controller module embedded in the co-kernel resource management framework. Covirt is a novel approach in that it is designed to efficiently modify the underlying hypervisor configuration in response to dynamic changes to resource assignment. In addition, it supports modular protection features that allow users to select the protection features enabled during runtime. Should a particular feature impose too much performance overhead, it can simply be disabled during initialization. We now discuss the design methodology and implementation details of Covirt in the context of the Hobbes OS/R using a modern Intel processor.

A. Design

Covirt was designed with three primary goals based on the requirements of HPC co-kernel systems:

- Covirt should not implement a virtual hardware abstraction layer and instead make all of the underlying hardware details directly visible to the co-kernel OS/R.
- Covirt should integrate seamlessly with existing co-kernel architectures due to the need to support existing cross OS/R dependencies and interfaces.
- Covirt's protection mechanisms should be modular, allowing a system operator to determine the suitable performance/protection trade-off based on workload requirements.

Removing unnecessary abstractions is a key motivation for the use of LWK environments, and one that is necessary for Covirt to support. Most virtualization environments rely on resource abstraction to simplify the view of the underlying hardware and hide details that do not match typical environmental assumptions. Examples of this include remapping the address space to appear as a single contiguous region, hiding NUMA topology information, and masking out processor specific features. While increasing portability and simplifying guest OS/R operation, these abstraction layers are common sources of performance problems, since VM-level assumptions about the hardware are often incorrect. Covirt explicitly provides no abstraction of hardware resources, and presents the guest an unfiltered view of the underlying hardware configuration. This allows the co-kernel OS/R to optimize its behavior around the physical hardware environment without interference from the underlying VMM.

Covirt is designed to seamlessly integrate with existing co-kernel frameworks. This is a challenge due to the degree of integration required across OS/R boundaries, which requires Covirt to be able to efficiently reconfigure the hypervisor in response to resource allocation and sharing operations. In addition, Covirt must transparently support the underlying communication channels that are necessary for coordination and state synchronization between the OS/Rs. This task is greatly simplified by the lack of abstraction, since Covirt is not

required to remap virtual resource addresses to physical ones. Instead, the challenge consists of monitoring for configuration changes and efficiently updating the hypervisor configuration.

Finally, Covirt implements a configurable and modular approach to resource protection that allows runtime configuration of hypervisor protection features. The use of virtualization features will always introduce some measure of overhead to co-kernel performance, and in many cases these overheads will be dependent on application workloads. Co-kernel architectures implicitly prioritize performance over safety (otherwise they would leverage traditional virtualization), and so Covirt is designed to allow users to selectively enable protection features only when the performance impact is within acceptable bounds.

B. Covirt Architecture

In order to meet the above requirements we designed Covirt as two separate components: a controller module integrated with the master control process that provides resource management coordination services for the enclaves, and a lightweight hypervisor that handles protection faults, synchronizes hardware state with configuration updates, and handles the small set of operations that require emulation. Figure 2 shows a high level overview of the architectural components.

Hypervisor: The Covirt hypervisor is responsible for all enforcement operations implemented by the system. In practice, the hypervisor itself does very little, and is primarily responsible for ensuring the hardware level virtualization features are correctly configured on the local CPU. This consists of initializing the CPU's virtualization features, serializing the virtualization configuration to the CPU's internal context, and updating local caches when changes are made to the enclave's configuration. In addition, the hypervisor is responsible for trapping any access violations in the enclave's OS/R and handling them appropriately. For the most part this means terminating the enclave, notifying the master control process, and safely halting the CPU. The master control process in turn is responsible for reclaiming the enclave's resources and notifying any other components that had dependencies on the failed enclave.

The degree of resource protection Covirt is able to provide is directly dependent on the degree of virtualization support in the hardware. While Covirt could feasibly extend protection capabilities beyond what the hardware provides, doing so would require significant virtualization overheads which would violate a primary design goal. Therefore, the range of Covirt's capabilities is tightly tied to the feature set provided by Intel's VMX virtualization extensions. Even so, Covirt is capable of providing protection for the following operations:

- Memory accesses
- MSR accesses
- IPI transmissions
- I/O operations
- Handling of abort exceptions such as Double Faults

While Covirt tries to push protection enforcement into hardware as much as possible, there are still operations that

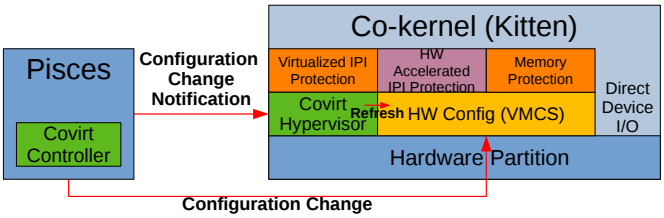


Fig. 2: Covirt's structure.

must be trapped and emulated. In the simplest case, these consist of single instructions (i.e. *cpuid* and *xsetbv*) that can be directly executed by the VMM with either minor or no modifications. In other cases more involved emulation is required, such as interrupt handling and injection. Where emulation is required, Covirt takes a minimalist approach in order to reduce overheads as much as possible.

The Covirt hypervisor is managed via a simple command queue between itself and the controller module. Commands are fixed-size messages containing update notifications directing the hypervisor to synchronize part of its local state. Commands are designed to be lightweight synchronization events that do not require extensive processing in the hypervisor. This means that the actual configuration updates are handled in the controller module, with the hypervisor only being responsible for activating new configurations and invalidating stale state. The purpose of this approach is to reduce virtualization overheads associated with re-configurations, as configuration updates are handled asynchronously with respect to the enclave and do not require the guest to pause while the hypervisor computes a new configuration. The hypervisor is only invoked after the configuration change has been made, so that it can ensure that the change is activated. Synchronization and consistency issues are managed by the controller module and higher level interfaces, which we describe next.

Controller: Each Covirt hypervisor is managed by the controller module integrated with the master control process, which is responsible for managing inter-enclave coordination, resource assignment, and resource sharing. Thus, the controller module can hook into the control paths that manage the system-wide hardware configuration. The Covirt controller intercepts resource management events and translates them into policy modifications that need to be applied for every affected enclave. These policy updates are then delivered to a kernel-level component that translates them into hardware configurations.

Configuration modifications are performed by the controller by directly modifying the hardware-level data structures associated with the co-kernel's virtualization context. Once a configuration change has been made, the hypervisor is notified in case it needs to reload the virtualization context or flush any caches, such as the TLB. The controller is responsible for maintaining consistency between the co-kernel OS/R's view of usable resources and the active hypervisor configuration. This is done by blocking any change to the actual OS/R resource assignment until the resources have been either mapped or

unmapped in the hypervisor context.

C. Implementation

The Covirt control module is implemented as an extension to the Hobbes OS/R framework. When enabled, it places a series of callback routines into various locations within the Hobbes infrastructure in order to capture notifications when resource management operations are performed. In addition, the control module includes a kernel-level component that is embedded into the Pisces kernel module. The userspace control module piggy-backs on the Pisces kernel ABI by adding a new set of ioctl commands that can be used to pass configuration update information into the kernel. Upon receiving these events, the Covirt kernel module identifies the specific hardware resources that are being operated on and updates the co-kernels hypervisor configuration. In addition, the kernel-level extensions are responsible for handling activation of Covirt during enclave initialization through a modification of the enclave boot procedure.

The Covirt hypervisor is written in a combination of C and x86 assembly, and operates using a minimal execution environment. Ideally the Covirt hypervisor would only initialize the local CPU virtualization context, jump into the co-kernel initialization routines, and never run again. However, there are situations that require invocation of the hypervisor, so Covirt contains enough of an execution context to support simple exit and event handling. The context is as lightweight as possible and provides the bare minimum of capabilities. For example, there is no support for dynamic memory allocation and Covirt relies on a small, 8KB stack that is preallocated by the control module. In addition, each hypervisor context only supports a single CPU core and is unaware of other hypervisor instances managing other enclave CPUs. This is achieved by replicating the hypervisor context (code, data, and hardware state) for each CPU core managed by Covirt into isolated memory locations. The hypervisor is responsible for invoking the local CPU's virtualization (VMX) instructions, and managing the hardware data structures containing the guest's Virtual Machine Control Structure (VMCS). This minimizes complexity but does not interfere with topology or other underlying hardware information, appearing transparent to the OS and applications.

Initializing Covirt: Covirt repurposes Pisces' existing enclave initialization procedure by interposing its hypervisor into an enclave's CPU boot process. Instead of booting directly into a co-kernel OS/R, Pisces instead boots into the Covirt hypervisor, which handles the virtualization hardware setup before directly invoking the actual co-kernel. It should be noted that this process is entirely transparent to the co-kernel OS/R, because Covirt configures the virtualization context to mirror the hardware state that would have resulted if the co-kernel had been booted normally by Pisces. Specifically, Covirt sets up the VMX guest configuration to launch at the co-kernel's start address as if it was coming from the Pisces trampoline code. This includes launching the VM directly into 64-bit long mode with pre-configured, identity mapped page tables and segmentation tables. The VMX hardware configuration is written by the controller module before the CPU is booted, allowing the hypervisor to simply load the pre-configured VMCS onto the local CPU core and perform a VM launch operation.

In Pisces, initial enclave configuration information is passed to a co-kernel via a boot parameter structure stored in memory. The address of this structure is passed to the co-kernel by the trampoline code, and contains the assigned hardware configuration, as well as several communication channels used for coordination between the co-kernel OS/R and master control process. Covirt replaces the standard boot parameter structure with a new, specialized structure used by the hypervisor. The Covirt boot parameters contain the VM configuration information, a minimal communication channel used as a command queue, and a pointer to the unmodified Pisces boot parameter structure used by the co-kernel. At VM launch, the address of the original Pisces boot parameters are passed to the co-kernel OS/R via register.

Hypervisor Coordination: As stated previously, certain resource configuration changes require that the hypervisor update the local virtualization context to keep it consistent. These update notifications are implemented using a shared memory command queue included inside the Covirt boot parameter structure. Commands are synchronous and are always invoked directly by the controller module, with pending commands signaled using NMI IPIs. NMIs are used in order to avoid IPI number conflicts that would in turn require virtualization of the interrupt vector space in the hypervisor. By using NMIs, Covirt is able to provide direct IRQ vector mappings between the hardware and the co-kernel OS/R. Notably, not all configuration changes require hypervisor coordination. This is because the controller module retains access to the data structures of the co-kernel's virtualization context, and in many cases can update those data structures directly. We will discuss this in more detail when we cover Covirt's memory protection features, but the benefit is that only state that might be cached by the local co-kernel CPU needs to be synchronized via the command queue.

Covirt's approach allows configuration changes to be made asynchronous with respect to the enclave's execution. In other words, Covirt can update the underlying virtual hardware configuration while the enclave continues to run in the virtual context. This is safe because the co-kernel OS/R is always explicitly notified whenever a configuration change is made, which follows directly from the zero-abstraction approach we have taken. To maintain consistency between the enclave OS/R's internal hardware mappings and the underlying virtual hardware configuration, Covirt orders the operations such that accesses to hardware is only done after the virtual mappings are fully synchronized. For resource assignment, this means that the enclave OS/R is only notified of new hardware resources after those resources have been mapped into the virtualization context, and conversely resource reclamation only occurs after those resources have been fully unmapped by the hypervisor. As a high level example, this means that if one co-kernel process is blocked waiting on a shared memory

mapping request, other processes can continue to run while the memory is mapped into the virtualization context in the background, and the blocked process will only return once the virtualization configuration has been successfully completed.

Memory Mapping: Memory is the primary resource Covirt is designed to protect, as (based on our experience) memory mapping and access bugs are the cause of the majority of co-kernel errors. Covirt provides memory protection through the use of nested page tables (or EPTs in the Intel nomenclature). When memory protection is enabled, the EPTs are created at enclave initialization time with an identity map of the memory regions allocated for that enclave. All EPT entries are mapped with full access permissions, and access violations occur only when an enclave tries to evaluate a memory address that is outside of its assigned memory regions. All EPT access violations are considered abort class errors and result in the termination of the co-kernel by the Covirt hypervisor. For further optimization, contiguous memory pages are coalesced into large (2MB) and giant (1GB) EPT page mappings whenever possible. Finally, it should be noted that while enabling memory protection features is optional in Covirt, many other protection features rely on it.

In co-kernel architectures, memory tends to be a very dynamic resource both at the OS/R and application levels. At the OS/R level, shared memory regions are used extensively to communicate and coordinate with other OS/Rs on the system. At the application layer, shared memory and shared address spaces that span OS/R boundaries are used to enable such things as application composition, low overhead data exchange, and remote system call invocation via proxy processes. As a result, shared memory regions are created and destroyed with a significant amount of frequency during an enclave's execution. Existing co-kernel architectures provide a large degree of support for cross OS/R memory sharing, often relying on multiple APIs to address specific needs. In order to provide seamless integration with co-kernel architectures Covirt must support each one. In the context of the Hobbes OS/R environment this support boils down to integrating with two control paths: the Hobbes memory management service and the XEMEM shared memory system.

While the mechanics of both Hobbes and XEMEM are described elsewhere [10], [19], they both result in the need to transmit memory lists of page frame information to a co-kernel OS/R to establish/destroy the memory mappings inside the co-kernel's context. The Covirt control module monitors these operations and intercepts the page information before it is transmitted to the co-kernel OS/R, at which point the Covirt control module updates the enclave's EPT mappings to reflect the updated memory configuration. On mapping operations (when the co-kernels visible memory is being expanded), the control module returns immediately after modifying the EPT mappings, allowing the Pisces framework to complete the transmission of the page frame list to the co-kernel. The co-kernel is then able to update its internal memory map, at which point the memory will be accessible via a nested page table walk. For unmapping operations (shrinking the co-kernel's memory map), the operations proceeds in a similar fashion, with Pisces transmitting a list of page frames that need to be removed from the co-kernels memory map. In this case, Covirt intercepts the operation after the page frame list has been transmitted and acknowledged by the co-kernel, but before a completion notification is passed to the Hobbes management layer. At this point the Covirt control module removes the requisite pages from the EPT mapping. However, instead of returning immediately, in this case the control module issues a memory update command to the Covirt hypervisor indicating that a change event occurred. This causes the enclave's CPU cores to trap into the hypervisor context so that they can flush their local TLBs. After the command has been completed on each CPU of the enclave, Covirt returns and continues the unmapping procedure by notifying the Hobbes resource manager that the memory has been released.

IPI Protection: The other significant resource protected by Covirt is the Inter-Processor Interrupt (IPI) vector space. Co-kernels typically have extensive reliance on IPI transmission as a means of providing direct notifications between both OS/R and application components. In Hobbes, per-core IPI vectors are a globally allocatable application resource, and provide low overhead signaling across OS/R boundaries. Due to their scale of use and dynamic allocatability, bugs due to errant or misconfigured IPI operations are another significant source of errors (again based on our experience). IPI-based errors can manifest either through superfluous notifications indicating events that did not occur or interference with device driver operations by mimicking hardware level interrupts. Covirt provides optional support for protecting co-kernels from IPI errors by virtualizing and masking IPI transmission operations originating from an enclave's OS/R.

IPI protection is implemented using the APIC virtualization (VAPIC) features provided by VMX. This allows the Covirt hypervisor to intercept all IPI transmission operations by trapping write operations to the APIC ICR register. The hypervisor is then able to compare the destination CPU and vector against a whitelist in order to verify that the IPI operation is permitted, and any errant IPIs are simply dropped by the hypervisor. IPI protection is implemented in one of two ways dependent on the virtualization capabilities of the hardware. The first approach relies on fully virtualizing the APIC and trapping and emulating the complete range of IPI operations. One side effect of this approach, however, is that VMX requires that all incoming interrupts trigger VM exits, resulting in increased IPI handling latencies.

The second approach leverages *Posted Interrupt Vector* (PIV) support on modern CPUs. Posted interrupts work by registering an in-memory vector bitmap with a guest's VMCS, which is used to indicate all interrupt vectors that are currently pending. In addition, PIV requires the registration of a single IPI vector with the VMCS that will be used to notify the hardware when a vector has been added to the bitmap. With PIV, IPI delivery is handled entirely by the hardware and no longer requires exits for every incoming IPI. However, while PIV allows exitless IPIs, it still requires exits for all external

interrupt generated by hardware devices. For the most part, this is ameliorated since most co-kernel approaches offload device driver operations to a general purpose OS/R via system call forwarding. However, there are some hardware interrupts that must be handled locally (notably, the local APIC timer), and these still require emulation. But again this is ameliorated by the fact that timer interrupts have long been a target of optimization in LWK architectures and their use is usually minimized.

V. Evaluation

Covirt's protection features were designed based on our previous experiences in developing and running co-kernel frameworks. One of the main sources of errors we have experienced with Pisces comes from consistency issues in the shared state between the co-kernel and host kernel. Sources of these inconsistencies included bugs in rarely used error handling paths, changes to semantics across interface versions, and trivial coding mistakes during the development process that nonetheless were difficult to diagnose due to causing node crashes. Covirt was explicitly designed to catch these errors before they were able to cause system level faults or corrupt other kernel instances on the node, and provide graceful fault handling that preserved the correctness of the host kernel. Anecdotally, Covirt is able to protect against a range of notorious bugs we dealt with during the development of Pisces. In one instance, a bug in an XEMEM cleanup path resulted in stale shared memory regions persisting in the co-kernel state for a short time window after they had been reclaimed by the host OS. This bug caused extremely rare system crashes that were only seen when running Pisces at large scale, and could not be reproduced in local development environments. Covirt would have made diagnosing the error substantially easier, as it would have (1) prevented the bug from taking down the entire node, and (2) provided the ability to collect debugging traces when it did occur.

We have also seen the beneficial capabilities of Covirt more directly during the development of Covirt itself. Developing Covirt resembled the early work on porting the Kitten LWK to Pisces, as it required working with the early hardware initialization process and correctly mapping in partitions of the node's hardware resources. The original Pisces effort required the use of full system virtualization tools (QEMU and VMWare) to deal with the frequency of system crashes. This required a substantial amount of effort later on as we moved to physical hardware to reconcile the differences between emulated and real hardware. With Covirt we were able to move to real hardware almost immediately in the development process, and so avoided all of the problems related to transitioning from virtual to physical systems. The same benefits are present in our efforts to port other kernel architectures (such as the Nautilus Aero-kernel) to the Pisces framework, as Covirt provides the capability to work directly on the target hardware from the start and is able to contain errors from propagating outside of the co-kernel itself. Based on our experiences with Covirt so far, we can anecdotally say that, in general, Covirt is able to reduce complex debugging efforts from weeks to days and porting efforts for new co-kernel architectures from months to weeks.

Benchmark Name	Version	Parameters
Selfish Detour [20]	1.0.7	None
STREAM	5.10	None
RandomAccess_OMP	10/28/04	`25`
HPCG [21]	Revision 3.1	`104 104 104 330`
MiniFE [21]	2.0	`nx 250 ny 250 nz 250`
LAMMPS [22]	3 Mar 2020	None

TABLE I: Benchmark Versions and Parameters

We evaluated our implementation of Covirt on a modern Intel system using a set of microbenchmarks, HPC mini-apps, and the LAMMPS multi-physics application. Parallelism was achieved with the OpenMP library. The specific benchmark details are provided in table I. All benchmarks were run on a system with two Xeon E5-2603 v4 1.70GHz CPUs in a dual-socket configuration, with 64GB of DDR4 2667MHz memory. All benchmarks were run ten times in a co-kernel environment configured with 14GB of memory spread across the two NUMA zones.

A. Microbenchmarks

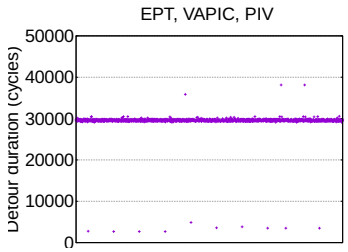


Fig. 3: Selfish-Detour benchmark results

We first evaluated the impact of Covirt on the underlying noise profile of a co-kernel environment. This was done using the Selfish-Detour benchmark that detects sources of interruptions on application execution. This and other microbenchmarks were all run on a single-core hardware configuration. For this experiment we evaluated multiple configurations of Covirt's enabled various protection features. The result for all features enabled is shown in Figure 3. The different configurations show little variation in their noise profiles, indicating that hardware level virtualization does not inherently increase system noise.

We next evaluated the overheads Covirt adds to resource configuration changes by measuring XEMEM attach operations. Operation latencies were measured by sampling the co-kernel's hardware TSC counter immediately before and after an XEMEM attach operation. We evaluated the latencies for various memory region sizes (up to 1024MB) with Covirt both enabled and disabled. The results are shown in Figure 4. Covirt imposes little to no overhead for this range of region sizes. Considering that the code paths between the Covirt and non-Covirt configurations are identical, this suggests the mapping operation is either masked by other operations or is insignificant compared to other operations.

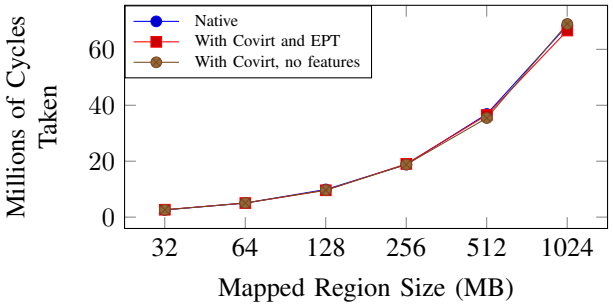


Fig. 4: XEMEM Attach Delay

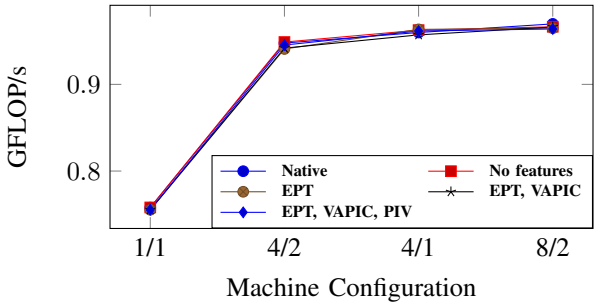


Fig. 7: HPCG scaling over CPU-core/NUMA-zone layouts

Finally, we evaluated the impact of various Covirt configurations on raw memory performance using the STREAM and RandomAccess memory microbenchmarks, shown in Figure 5(a) and 5(b). For STREAM, performance across all configurations of Covirt was comparable to native performance with no noticeable overheads. RandomAccess did exhibit some performance degradation with Covirt almost certainly due to EPT table lookups. However the overhead did not exceed more than 3.1% in the worst case (memory + IPI protection enabled), and adding memory protection only added overhead of 1.8%.

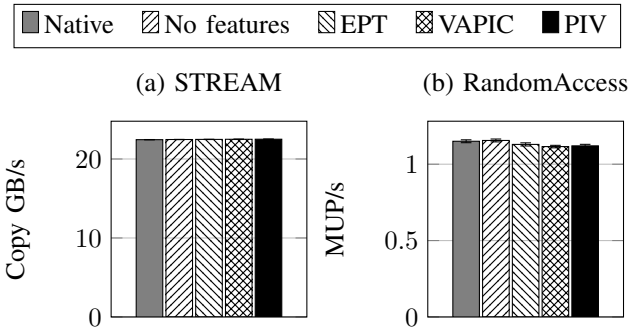


Fig. 5: STREAM (a) and RandomAccess (b) benchmark performance comparison

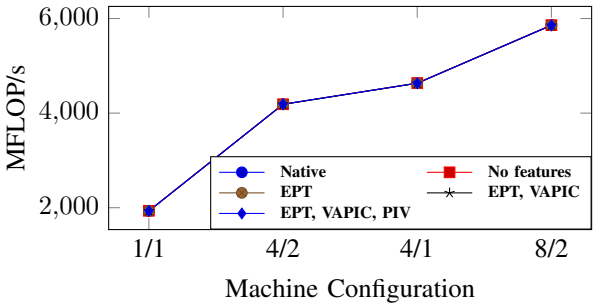


Fig. 6: MiniFE scaling over CPU-core/NUMA-zone layouts

B. Mantevo Mini-apps

We next evaluated Covirt using 2 mini-apps: HPCG and MiniFE from the Mantevo Suite. MiniFE is a proxy app that approximates an unstructured implicit finite element or finite volume application. HPCG is a conjugate gradient solver commonly used for HPC system benchmarking. For these experiments we configured Covirt with various protection features enabled and scaled the enclave environment across several hardware layout configurations. We scaled the enclave's resources from a single core up to 8 cores split evenly between 2 NUMA domains. The amount of memory assigned to the enclave was kept constant at 14GB, but the memory was divided evenly between NUMA zones as the number of cores was increased, also split evenly across domains. The evaluated hardware configurations consisted of (1) a single core enclave running entirely in one NUMA domain, (2) a 4-core enclave split evenly across two NUMA domains, (3) a 4-core enclave running in a single NUMA domain, and (4) an 8-core enclave split evenly across two NUMA domains. The results for MiniFE are shown in Figure 6 and for HPCG in Figure 7.

From our evaluation, we see that Covirt imposes little to no overhead on MiniFE across all configurations. Due to its design, MiniFE does not require significant amounts of inter-process coordination and so the inclusion of IPI protection features does not have a noticeable impact on application performance. Moreover, the addition of memory protection likewise does not introduce noticeable performance degradation. In the case of HPCG, Covirt does impose minor overheads, but they stay consistent across Covirt feature configurations and varying hardware layout configurations. This implies that there is a baseline performance penalty imposed by the introduction of virtualization features, that stays roughly constant regardless of how those features are configured. Regardless, in the worst case, Covirt only degrades HPCG's performance by 1.4%. These results show that not only does Covirt impose minimal overheads for realistic workloads on a single core, but also that its protection features do not impose significant overheads to inter-core communication and coordination operations that impact application performance.

C. LAMMPS

Finally, we evaluated the performance of the LAMMPS multi-physics application across multiple configurations of Covirt. For the LAMMPS experiments we used an 8 core enclave split across 2 NUMA domains. The experiments consisted of different workloads, executed using the default run scripts shipped with the application. Figure 8 reports the results as the loop times calculated during LAMMPS execution

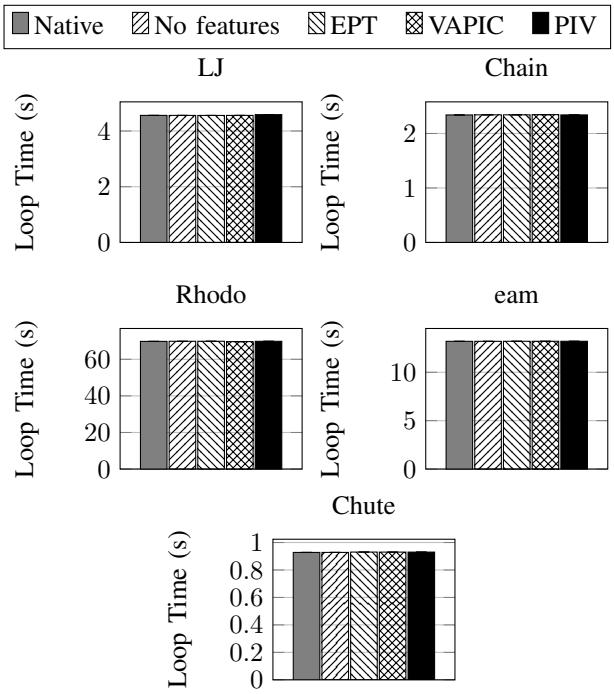


Fig. 8: LAMMPS benchmark performance comparison

(lower is better). The results show that for the LJ, EAM, and chain benchmarks all configurations have similar performance. Chute shows the most sensitivity to the protections being enabled, with the native and no-feature configurations performing the best. Similar to the earlier results, LAMMPS shows that the overheads imposed by Covirt are minimal.

VI. Conclusion

We have presented Covirt, a novel approach to resource protection and fault isolation for co-kernel OS/Rs based on hardware virtualization capabilities. Covirt's split architecture approach allows the use of a minimalistic hypervisor to be interposed underneath a running kernel in an enclave, which enforces hardware access policies and so achieves fault isolation. Our approach allows for seamless integration with existing co-kernel software stacks, and supports dynamic configuration changes needed by tightly integrated cross OS/R applications running on top of co-kernel runtimes. Further, we show these protections introduce only minimal performance overheads, and in many cases Covirt's protection comes at no overhead due to the capabilities of the underlying hardware virtualization features.

References

[1] J. Ouyang, B. Kocoloski, J. R. Lange, and K. Pedretti, "Achieving Performance Isolation with Lightweight Co-Kernels," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, 2015.

[2] B. Gerofi, M. Takagi, and Y. Ishikawa, "IHK/McKernel," in *Operating Systems for Supercomputers and High Performance Computing*, 2019.

[3] R. W. Wisniewski, T. Inglett, P. Keppel, R. Murty, and R. Riesen, "mOS: an architecture for extreme-scale operating systems," in *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers*, 2014.

[4] B. Kocoloski, J. Lange, K. Pedretti, and R. Brightwell, "Hobbes: A Multi-kernel Infrastructure for Application Composition," in *Operating Systems for Supercomputers and High Performance Computing*. Springer, 2019, pp. 241–267.

[5] A. Hori, M. Si, B. Gerofi, M. Takagi, J. Dayal, P. Balaji, and Y. Ishikawa, "Process-in-process: techniques for practical address-space sharing," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, 2018.

[6] B. Gerofi, R. Riesen, R. W. Wisniewski, and Y. Ishikawa, "Toward Full Specialization of the HPC Software Stack: Reconciling Application Containers and Lightweight Multi-Kernels," in *Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers ROSS*, 2017.

[7] A. Dubey, P. H. J. Kelly, B. Mohr, and J. S. Vetter, "Performance Portability in Extreme Scale Computing (Dagstuhl Seminar 17431)," *Dagstuhl Reports*, vol. 7, no. 10, pp. 84–110, 2018.

[8] B. Gerofi, M. Takagi, A. Hori, G. Nakamura, T. Shirasawa, and Y. Ishikawa, "On the Scalability, Performance Isolation and Device Driver Transparency of the IHK/McKernel Hybrid Lightweight Kernel," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016.

[9] 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 *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2010.

[10] B. Kocoloski and J. Lange, "XEMEM: Efficient Shared Memory for Composed Applications on Multi-OS/R Exascale Systems," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, 2015.

[11] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS)," in *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, 2008.

[12] D. Otstott, L. Ionkov, M. Lang, and M. Zhao, "TCASM: An asynchronous shared memory interface for high-performance application composition," *Parallel Computing*, vol. 63, pp. 61–78, 2017.

[13] S. M. Kelly and R. Brightwell, "Software Architecture of the Light Weight Kernel, Catamount," in *Proceedings of the 2005 Cray User Group Annual Technical Conference*, 2005.

[14] S. Lankes, S. Pickartz, and J. Breitbart, "HermitCore: a Unikernel for Extreme Scale Computing," in *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*, 2016.

[15] C. Weinhold, A. Lackorzynski, J. Bierbaum, M. Küttler, M. Planeta, H. Härtig, A. Shiloh, E. Levy, T. Ben-Nun, A. Barak *et al.*, "FFMK: A Fast and Fault-Tolerant Microkernel-Based System for Exascale Computing," in *Software for Exascale Computing-SPPEXA 2013-2015*. Springer, 2016.

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

[17] K. Cheng, S. Doddamani, T.-C. Chiueh, Y. Li, and K. Gopalan, "Directvisor: virtualization for bare-metal cloud," in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2020.

[18] Y. Dai, Y. Qi, J. Ren, Y. Shi, X. Wang, and X. Yu, "A lightweight VMM on many core for high performance computing," in *Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, 2013.

[19] B. Gerofi, Y. Ishikawa, R. Riesen, and R. W. Wisniewski, *Operating Systems for Supercomputers and High Performance Computing*. Springer, 2019, vol. 1.

[20] 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.

[21] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving Performance via Mini-applications," Sandia National Laboratories, Tech. Rep. SAND2009-5574, 2009.

[22] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of computational physics*, vol. 117, no. 1, pp. 1–19.