# XEMEM: Efficient Shared Memory for Composed Applications on Multi-OS/R Exascale Systems

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

## ABSTRACT

Current trends in exascale systems research indicate that heterogeneity will abound in both the hardware and software layers on future HPC systems. It is our position that exascale environments are likely to be constructed from independent partitions of hardware and system software called *enclaves*, with multiple enclaves co-located on the same physical nodes and each executing an optimized operating system and runtime (OS/R) to support a particular application behavior. Fully utilizing these systems will require the ability to execute composed workloads, such as *in situ* applications, whereby HPC simulations execute synchronously with co-located analytic packages that in turn process simulation output via shared memory. In this work, we present the design and implementation of XEMEM, a shared memory system that can efficiently construct memory mappings across enclave OSes to support composed workloads while allowing diverse application components to execute in strictly isolated enclaves. By utilizing modifications to the Kitten lightweight kernel and Palacios lightweight virtual machine monitor, as well as leveraging our recent work on lightweight "co-kernels," we demonstrate that our approach can support a diverse range of native and virtualized environments likely to be deployed on future exascale systems. Finally, we demonstrate that a multi-enclave system can reduce cross-workload contention and improve performance for a sample composed benchmark compared to a single OS approach.

## Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Shared Memory; C.5.1 [**Computer System Implementation**]: Super (very large) computers; D.4.7 [**Operating Systems**]: Organization and Design

This project is made possible by support from the National Science Foundation (NSF) via grant CNS-1421585, and by the 2013 Exascale Operating and Runtime Systems Program under the Office of Advanced Scientific Computing Research in the DOE Office of Science.


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

http://dx.doi.org/10.1145/2749246.2749274.

## Keywords

operating systems; shared memory; supercomputing; virtualization; exascale; application composition

## 1. INTRODUCTION

Historically, tightly-coupled and massively parallel HPC applications have enjoyed the luxury of single user operating environments devoted entirely to the forward progress of the application. However, as the HPC community ushers in the exascale era, a new and diverse set of challenges are emerging that will increase the amount of workload consolidation on exascale nodes, limiting the utility of single application environments. As systems become increasingly heterogeneous, new parallel programming paradigms and communication models [8] are likely to lead to the development of applications highly specialized for particular hardware and software environments which cannot be universally supported by a single system configuration [3, 1]. Furthermore, consolidating workloads to restrict data movement and communication will likely be needed to address power and performance constraints of exascale applications [12].

A key example of this type of workload consolidation can be seen in the emergence of composed workloads, such as *in situ* applications. Petascale-class and earlier generations of HPC applications generally employed models of computation whereby processing of simulation output was performed as a post-processing task on a separate I/O node or visualization cluster and as part of an entirely separate process [6]. However, this type of model becomes less feasible as the amount of data produced by simulations continues to increase, as it creates bottlenecks in networks and I/O subsystems, introduces contention on those resources for separate workloads, and unnecessarily consumes nontrivial amounts of power and memory to move data [23, 17]. These issues have driven the emergence of composed in situ models whereby the compute/memory intensive HPC simulation communicates via shared memory with a more general purpose, I/O intensive analytics package that executes on the same physical node.

While co-locating HPC simulation and analytics workloads provides the potential to increase the throughput of an exascale system, the individual workload components present different resource management and isolation requirements to the operating and runtime management systems (OS/R). In order to effectively consolidate exascale applications in such a system, it is necessary to support both the performance characteristics and strict isolation requirements of traditional HPC applications, while at the same time pro-

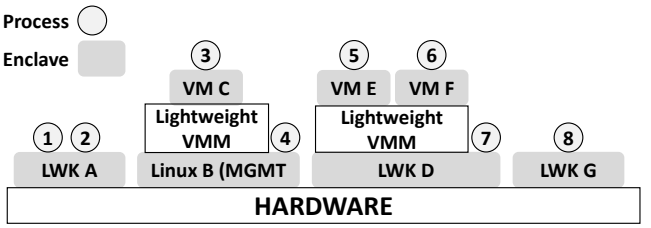



Figure 1: Exascale Enclave Partitioning

viding the full-featured environments required by analytic and visualization applications. Given these considerations, it is our position that exascale OS/Rs will be composed of a wide range of heterogeneous hardware and software environments, called *enclaves*, where each enclave is specialized in supporting a particular class of exascale workload, but still remains strictly isolated from the other enclaves on the system. Composed workloads executing in multi-enclave environments will be afforded high performance and consistent execution experience required by large-scale HPC applications, while at the same time having access to feature-rich runtimes and operating environments.

While the concept of a multi-enclave environment has been established for future exascale systems [3, 1], many details pertaining to enclave compositions are largely unresolved to this point. For example, previous work has shown the benefits of virtualization for high performance computing [13], and it is reasonable to consider virtualization to be a key component of an exascale system. Other work has identified techniques by which multiple operating systems can be deployed on partitioned hardware components without virtualization [16, 18, 20]. Our position, illustrated in Figure 1, is that many different software and hardware configurations, both native and virtualized, will be commonly deployed as exascale enclaves.

As Figure 1 illustrates, the individual OS/Rs of each enclave are capable of executing local processes, which by default are isolated from each other at both the system software and hardware layers. However, supporting composed workloads in these types of multi-enclave environments requires the ability to allow processes to communicate efficiently with other processes executing in separate enclaves. Thus, in this work, our key contribution is the design, implementation, and evaluation of XEMEM (Cross Enclave Memory), a system that can efficiently and dynamically provide inter-process shared memory mappings between separate OS/Rs to support the execution of composed workloads.

This paper makes the following main contributions:

- We present the design and implementation of XEMEM, a shared memory mechanism to enable composed workloads in a diverse set of multi-OS/R environments expected in exascale systems.

- We provide compatibility with existing applications by providing an implementation whose API is backwards compatible with the API exported by XPMEM [21], a shared memory implementation for large scale supercomputers. This allows unmodified applications to be deployed, without any knowledge of enclave topology or cross-enclave communication mechanisms.

- We demonstrate the scalability of our implementation, with respect to both the sizes of shared memory regions, as well as the number of enclaves concurrently executing on the system.

- By utilizing extensions to the Kitten lightweight kernel, Palacios virtual machine monitor, and a lightweight co-kernel architecture, we show that a multi-enclave system using shared memory allows an in situ workload to achieve superior and more consistent performance when compared to single OS/R configurations. Our results demonstrate the benefits of multi-OS/R configurations both on a single node as well as multiple nodes.

## 2. RELATED WORK

Prior research in shared memory for HPC workloads has focused on optimized memory mappings for single OS/R environments. SMARTMAP [4] allows sharing of coarse-grained regions through shared top-level tables between processes. However, utilizing this approach in a multi-enclave exascale environment would prove challenging, as different system software environments will likely utilize conflicting address space management routines that may not be easily negotiated in the same set of page tables. KNEM [10] provides single-copy shared memory designed to optimize intra-node MPI communication between processes as well as various communication devices, but again is designed to operate in a single OS/R environment and would require significant modifications to support a multi-enclave configuration.

Other research has focused on optimizing communication between co-located VMs. Fido [5] demonstrated that providing read-only inter-VM address space access can yield a high degree of performance suitable for many enterprise applications. More general inter-VM message passing techniques have been implemented via shared memory [19, 22]. However, exascale systems will require broader support for heterogeneous environments than these techniques support. For one, some exascale (co)processors may not support virtualization capabilities. Even those that do may still be required to support workload configurations requiring shared memory mappings between native processes and VM processes executing on different host OS/R environments.

Finally, projects such as FusedOS [16], McKernel [18], and mOS [20] have investigated the deployment of multi-enclave environments similar to the types of environments we envision in exascale systems. These approaches allow multiple system software configurations to exist on the same nodes by running lightweight environments on a partition of the system's cores and running fullweight environments like Linux on another partition. Threads executing in the lightweight environment request services, such as system calls and device management operations, by delegating these operations to threads running in the Linux environment. To support such a configuration, these systems use a memory mapping approach by which the Linux threads map the same memory as the lightweight threads, and thus can efficiently perform service operations without requiring memory copies. However, we envision that exascale systems will have greater needs for general purpose shared memory operations than are allowed by these systems, such as the sharing of address regions between virtual machines and native processes.

# 3. MULTI-OS/R SHARED MEMORY

To support the types of environments likely to be seen in exascale systems requires that our system, XEMEM, support arbitrary enclave topologies while at the same time provide scalability as the number of co-located enclave OS-/Rs increases. Furthermore, it is critical that the system scale to arbitrarily large shared memory regions as required by composed applications.

One of the key tenets of our approach is that, while meeting these requirements will require significant implementation effort in cross-enclave communication mechanisms and protocols, application programming for shared memory on an exascale system should not be more difficult than it is on current systems. Ideally, applications written for single OS/R systems should not need to be re-written or required to change at all to run in an multi-enclave environment. This section discusses how our system is built to maintain compatibility with existing shared memory applications while at the same time meeting the goals required for scalability and efficiency on future exascale architectures.

## 3.1 Common Global Name Space

Our approach to maintaining the simplicity of shared memory application programming centers around the ability to provision a single common global name space for shared memory registrations that provides two key features: unique naming and discoverability. In a single OS/R environment, shared memory regions can be readily named and tagged by a variety of basic mechanisms, such as using process IDs and virtual address ranges, which afford a simple way to maintain the unique addressability of each region. Furthermore, the OS/R has access to a plethora of shared IPC constructs, such as filesystems, to provide discoverability to processes.

In a multi-enclave environment, however, these operations are considerably more challenging. One approach to provide naming would be to eschew the requirement that the OS/R provide global uniqueness of identifiers, and instead force user applications to add an extra dimension to shared memory addresses by providing some form of unique enclave identifier. However, such an approach directly conflicts with our goal of maintaining application simplicity, as it would require applications to have knowledge of enclave configurations. Instead, our approach is to administer a common global name space by providing a centralized name server responsible for the allocation of segment identifiers for all shared address regions. This approach guarantees the uniqueness of all registered memory regions without requiring local OS/R environments to negotiate the availability of process IDs and virtual address regions, or adding complexity to application programming. This approach also allows our system to provide discoverability, as the name server can be queried for information regarding the existence and names of shared memory regions.

## 3.2 Arbitrary Enclave Topologies

One of the key requirements for an exascale shared memory system is the ability to support the construction of arbitrary enclave topologies. It is our vision that not only will exascale environments incorporate a variety of different enclave architectures, but also that an individual node's partitions are likely to be dynamic and will change in response to the node's workload characteristics. At any point in time, we refer to an enclave's architectural partitioning, including the hardware-supported inter-enclave communication interfaces, as the *enclave topology*. Figure 2 shows the topology for the example enclave partitions shown in Figure 1, for the configuration in which the name server discussed in the previous section is configured to run in the single native Linux enclave. As the figure demonstrates, our system assumes that enclave topologies will be organized in a hierarchical fashion such that communication channels between enclaves can be restricted. This means that enclaves will not necessarily have the ability to communicate directly with all other enclaves in the system, but rather will be required to communicate via an alternative mechanism that supports the routing of information based on the hierarchical configuration.

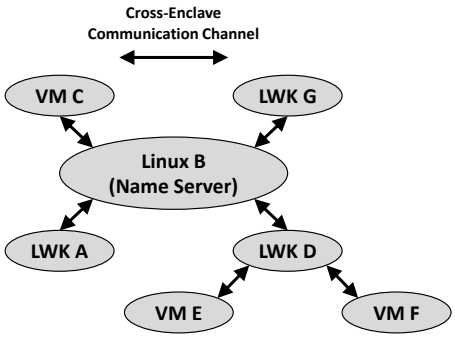



Figure 2: Exascale Enclave Topology

Our system supports communication in any arbitrary topology by utilizing a hierarchical routing algorithm that associates each enclave in the system with a unique identifier called an *enclave ID*. In order to route messages through the system, the algorithm requires each enclave to perform three main operations: (1) determine the local communication channel through which it can communicate with the name server, (2) request an enclave ID via this channel, and (3) maintain a mapping of enclave IDs to local communication channels. Initially, an enclave queries the location of the name server by broadcasting a message on each of its communication channels. When another enclave receives a broadcast message, it creates a response if it knows a path to the name server through one of its own channels. Once a response is received, the enclave remembers the communication channel through which the response came, and then sends a request through the link to allocate an enclave ID, which gets forwarded to the name server.

To maintain a mapping of enclave IDs to communication channels, each enclave is required to keep track of enclave IDs as they are allocated by the name server and forwarded through the system. For example, again referring to Figure 2, assuming *VM F* has determined it can reach the name server through *LWK D*, it sends a request through *LWK D* to allocate an enclave ID. *LWK D*, which has previously identified the name server location and allocated an enclave ID for itself, forwards the request to the name server and remembers that the request came from *VM F*. The name server receives the request, allocates an enclave ID, and updates its enclave map to map the new enclave ID to *LWK D*. Upon receiving the new enclave ID, LWK D queries its outstanding request list and finds that a request previously came from *VM F*. Thus, it forwards the enclave ID to *VM F* and updates its internal map accordingly.

By maintaining enclave ID mappings in this way, enclaves are able to make routing decisions for shared memory registration messages by using a simple algorithm. When an enclave receives a message, it searches its map for the destination enclave ID. If it finds the enclave ID, it forwards the message along the associated communication channel for that enclave. Otherwise, it forwards the message through the channel used to reach the name server. With this hierarchical architecture, our system is able to support any arbitrary enclave topology. It should be noted that the name server can be deployed in any enclave on the system, though we envision that most exascale systems will likely place the name server in a management enclave.

## 3.3 Dynamic Sharing of Memory Regions

Much of the previous work in high performance shared memory [4] has focused on efficiently creating large shared memory mappings between processes executing in single OS/R environments through the use of shared top-level page table entries. This approach is suitable for single-user environments such as the lightweight kernels it is currently deployed in.

Unfortunately, this approach is unsuitable in a multi-enclave environment for a variety of reasons. First, the sharing of page table entries across multiple heterogeneous processors may simply not be possible based on the hardware configurations in some exascale systems. Furthermore, heterogeneous software environments in different enclaves are likely to employ different address space management routines and operations. For example, full-featured environments such as Linux will require mechanisms such as page protections and page faulting semantics in order to support higher-level application behavior. Coalescing this type of behavior with a static, distributed shared memory approach is likely to be difficult to implement efficiently and correctly.

Our approach is to dynamically support more fine-grained memory mapping requests according to the individual memory sharing requirements of composed application processes. As such, our system provides shared memory mappings as they are created/requested by processes executing in different enclave environments. While this approach adds a small amount of overhead in the creation and attachment of shared memory mappings, it allows for a more efficient use of virtual and physical address space, and increases the number of heterogeneous enclave environments likely to be supported by our system.

## 3.4 Localized Address Space Management

Given the significant scale and level of heterogeneity that is likely to be found in the hardware and software environments on exascale systems, our system requires the enclave operating systems to perform memory mapping operations locally using the techniques required by the enclave's hardware configuration. These operations include walking page tables to generate physical address regions that map to segment identifiers, as well as performing page table modifications to modify process address spaces. While it may be possible to perform shared memory mappings by modifying enclave address spaces remotely, it would likely be difficult to provide such an implementation correctly, and would at least require complicated and inefficient address space synchronization mechanisms.

| Function | Operation |
|---|---|
| `xpmem_make` | Export address region as shared memory. Returns *segid*. |
| `xpmem_remove` | Remove an exported region associated with a *segid*. |
| `xpmem_get` | Request access to shared memory region associated with a *segid*. Returns permission grant. |
| `xpmem_release` | Release permission grant. |
| `xpmem_attach` | Map a region of shared memory associated with a *segid*. |
| `xpmem_detach` | Unmap a region of shared memory. |

Table 1: The XPMEM User-Level API

# 4. IMPLEMENTATION

The goals of the implementation of XEMEM are twofold. First, we seek to provide transparency to applications so that they are not required to have knowledge about the existence of enclaves or specific cross-enclave communication interfaces. Second, we seek to provide scalability with respect to the number of concurrently executing enclaves, as well as the amount of memory being shared in the system at any point in time.

Our implementation is largely based on our previous research in providing multi-OS/R system configurations. The implementation is based specifically on the Kitten lightweight kernel and the Palacios virtual machine monitor, as well as our recent work with lightweight "co-kernels" that leverage modifications to these systems to support the types of multi-enclave configurations proposed in Figure 1.

### *Kitten Lightweight Kernel.*

Kitten [13] is a special-purpose OS kernel designed to provide an efficient environment for executing massively parallel HPC applications. Some of Kitten's unique characteristics are its modern code base that is partially derived from the Linux kernel and its use of virtualization to provide full-featured guest OS support when needed.

### *Palacios Virtual Machine Monitor.*

Palacios [13] is a publicly available, open source, OS independent VMM that targets the x86 and x86_64 architectures with either ADM SVM or Intel VT extensions. Palacios is designed to be embeddable into multiple host OSes, and is currently supported in both the Linux and Kitten host environments. The combination of Palacios and Kitten acts as a lightweight hypervisor supporting full system virtualization.

### *Pisces Lightweight Co-Kernel Architecture.*

Recently, we have proposed the use of lightweight "co-kernels" through the Pisces co-kernel architecture [15]. Co-kernels enable the decomposition of a node's hardware resources (cores, memory blocks, devices) into partitions that are fully managed by independent system software stacks. Pisces allows a node to deploy multiple co-located Kitten instances as co-kernels executing alongside an unmodified Linux host OS. By leveraging Palacios support, the enclave OSes managing these co-kernels may be virtualized, resulting in customizable OS/R configurations well-suited for the deployment of composed multi-enclave workloads.

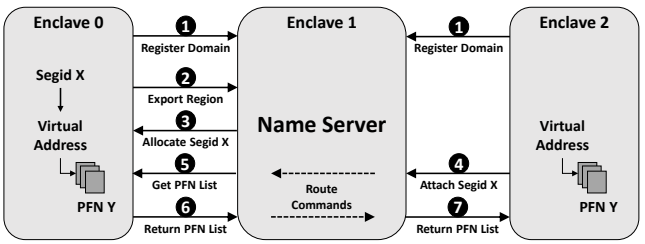



Figure 3: Shared Memory Protocol

## 4.1 XPMEM

To provide transparency to user processes executing in heterogeneous enclave environments, we leveraged an API that is backwards compatible with the XPMEM API for Linux systems developed by SGI/Cray [21]. XPMEM provides zero-copy shared memory to areas of a process' address spaces by allowing processes to selectively export address space regions and associate publicly visible segment identifiers (*segids*) with them. Given a valid *segid*, a process can create shared memory mappings with the source process. The XPMEM user-level API is shown in Table 1.

XPMEM also provides a kernel module responsible for providing shared memory mappings between multiple native Linux processes. We developed a similar XEMEM kernel module which implements the same functionality as XPMEM for Linux systems, but also supports sending memory attachment requests to remote enclaves, as well as serving attachment requests from remote enclaves by performing page table walks to generate page lists. We implemented the name server as a component of our XEMEM kernel module, which is responsible for the creation of unique *segids* for all enclaves in the system. We also implemented the routing protocol discussed in Section 3.2 to support arbitrary enclave topologies. Finally, we implemented the XEMEM service in both the Kitten lightweight kernel and Palacios Virtual Machine Monitor.

## 4.2 Shared Memory Protocol

While user-level processes are not required to have explicit knowledge of enclave topologies, they must still be able to discover exported memory regions created by external enclave processes. Our system provides a mechanism, illustrated in Figure 3, by which requests to create and attach to shared memory regions may be processed by the underlying enclave OS/R environments. As discussed in the previous section, XEMEM administers a common global name space for shared memory regions by utilizing a centralized name server responsible for assigning globally unique *segids* for all exported memory regions. Thus, to create a shared memory region, a local enclave's OS/R first sends a request to the name server to allocate a *segid*. The *segid* is then communicated to the user-level process according to the XEMEM API.

In order to attach to a shared memory segment, a process wishing to map the region must first discover the unique *segid* from the source process in some way. While single OS/R environments would likely resort to the IPC constructs provided by the local OS, our system provides an alternative mechanism for querying the name server using kernel-level inter-enclave messages.

Given a valid *segid*, the local OS/R for the attaching process first determines if the segment was created by a local process or not. If so, the attachment proceeds using the conventions of the local OS shared memory facilities. Otherwise, an attachment request is sent to the name server using the command routing protocol described in Section 3.2. Upon receiving the attachment request, the name server, which maps *segids* to enclaves, then forwards the command to the destination enclave which owns the *segid*. The destination enclave creates a list of the physical page frames that have been allocated to map the *segid*. A response message containing the list of frames is then forwarded back through the name server to the attaching enclave using the same routing mechanisms. Upon receiving the list of page frames, the attaching enclave maps them to user memory using the memory mapping constructs provided by the local OS.

## 4.3 OS Memory Mapping Routines

As discussed in the previous section, our system provides the ability to generate lists of page frames corresponding to *segids*, as well to map lists of page frames into destination enclave address spaces. Page frame lists are generated as enclaves receive remote requests for *segid* attachments. For Linux enclaves, memory must first be pinned in the user process' address space before a list can be generated. For this, we primarily rely on the `get_user_pages`[1] kernel function to allocate and pin physical memory for the process. Once the memory has been pinned, Linux provides a set of page table walking functions that allow the list to be easily generated. For mapping remote enclave page frame lists into Linux address spaces, we use the `vm_mmap` function to allocate an unused portion of virtual address space, and then use `remap_pfn_range` to map the page frames into the region.

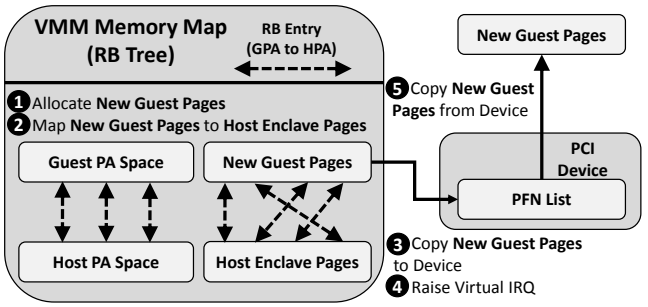
For Kitten LWK enclaves, two existing design protocols complicate the dynamic creation of shared memory regions. First, the original shared memory mechanism in Kitten relies on the SMARTMAP [4] protocol, which uses page table sharing techniques to support local process shared memory in a way that would be difficult to extend to multi-enclave configurations. Furthermore, all virtual address space regions (heap, stack, etc.) for Kitten processes are mapped statically to physical memory as processes are created, and there was originally no support for dynamically expanding these regions. To address these issues, we added support for dynamic heap expansion, by which processes can map remote page frame lists in a way that does not sacrifice the ability to use SMARTMAP for shared memory with local processes, or negate the ability to map all other virtual regions with physical memory during process creation.

To implement page frame list generation for handling shared memory attachments from remote processes, we utilized existing page table walking functions already provided by the kernel.

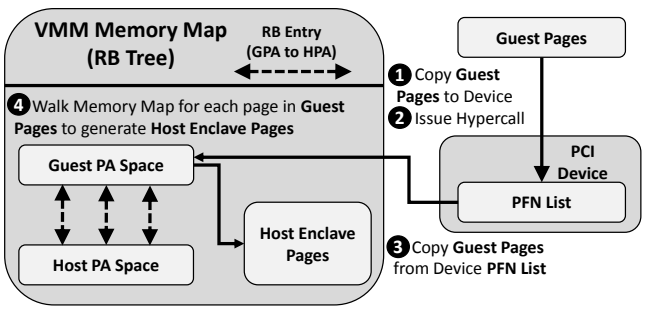
## 4.4 Palacios Host/Guest Memory Sharing

Providing support for shared memory between Palacios virtual machine enclaves and the corresponding host enclave presents two main challenges. First, both the guest and host need to be able to initiate shared memory operations,

[1]Note that `get_user_pages` is a bit of a misnomer. Pages are generally already allocated when the function is invoked, with the main purpose being the pinning of memory to prevent paging out




(a) Guest Attachments to Host Enclave Memory




(b) Host Attachments to Guest Enclave Memory

Figure 4: Palacios Memory Translations for Shared Memory Attachments

so our implementation must support efficient notifications of operations from host to the guest, as well as from the guest to the host. Additionally, Palacios should efficiently perform memory translations between host and guest page frame numbers for page lists that are passed during shared memory attachments.

To address these challenges, our approach, illustrated in Figure 4, consists of a virtual PCI device that allows efficient two-way notifications, as well as modifications to Palacios' internal memory management system to support the required memory translations. Figure 4(a) demonstrates the process by which a guest enclave completes an attachment to memory shared by the host. First, Palacios allocates a completely new region of guest physical address space that is equal in size to the amount of memory being shared. Then, Palacios updates its internal memory map to map this memory to the host page frames specified in the "Host Enclave Pages" list, which is supplied by the source enclave which exported the shared region. Once the memory map has been updated, Palacios copies the page frames corresponding to the new guest memory region to a list accessible through the virtual PCI device, and then issues an interrupt on the device. When the guest enters, it receives the interrupt, reads the page frame list from the device, and maps them into the attaching process' virtual address space.

It is important to note that Palacios' memory map is currently implemented as a red-black tree, where each entry in the tree maps a physically contiguous guest region to a physically contiguous host region. Palacios is usually configured to manage large blocks of physically contiguous memory, and thus is able to map all guest memory with a relatively small number of entries in this tree. However, the host enclave page frames that are mapped as part of XEMEM attachments are not guaranteed to be contiguous, and thus the process of updating the memory map may require a new entry in the red-black tree for each host page frame. We study the performance implications of this process in detail in Section 5.4.

Figure 4(b) illustrates the process in the reverse direction, by which the host enclave can generate a host page frame list representing a memory region exported by the guest. To notify the host of the completion of an attachment operation, the guest copies the page frames to a list accessible through the PCI device, and then issues a hypercall to trigger an exit into the host. Using the pages frames specified by the guest, Palacios walks the memory map and generates the list of host page frames that correspond to the guest memory. The host enclave can then map the host page frames to the attaching process' virtual address space, or forward them according to the routing protocol if the attaching process exists in a separate enclave.

## 4.5 Cross-Enclave Communication Channels

Facilitating the shared memory protocol described in Section 4.2 requires the ability to send messages between enclaves. Messages generally compose one of the commands shown in Table 1, but also exist to support the routing protocol outlined in Section 3.2, such as sending or responding to the broadcasts needed to query the name server location. Our current system provides two separate mechanisms for cross-enclave messages: a channel leveraging the Palacios virtual machine monitor for allowing communication between virtual machines and a native host enclave, and a channel based on the Pisces co-kernel architecture, which allows communication between two native enclaves.

### *Palacios Host/Guest Channel.*

The Palacios communication channel is implemented via a virtual PCI device, which is discussed at length in Section 4.4. In both transfer directions (host-to-guest or guest-to-host), if the message being transfered does not have an associated page frame list component (e.g., any of the operations in Table 1 except `xpmem_attach`), the mechanisms used are similar but simpler, as there is no need to translate page frame lists. For these operations, a simple command header located in the PCI device is set, and then either an interrupt into the guest or a hypercall into the host is used to signal the message notification.

### *Pisces IPI-Based Channel.*

The other cross-enclave communication channel supported by our system is based on the Pisces co-kernel architecture. During the creation of a Kitten co-kernel enclave, the co-kernel creates a small shared memory region through which kernel-level messages can be transferred to/from the Linux management enclave. Both the Linux management enclave and the co-kernel enclave initialize special IPI (Inter-Processor Interrupt) vectors for negotiating access to this memory. To initiate a message transfer, an enclave sends an IPI to the destination enclave CPU using the indicated IPI vector. Upon receiving the IPI, the destination enclave sets a flag in the shared memory region indicating it is ready to receive data. The source enclave then copies the message into the shared memory region and waits for destination enclave to copy the message out into a separate locally allocated memory buffer.

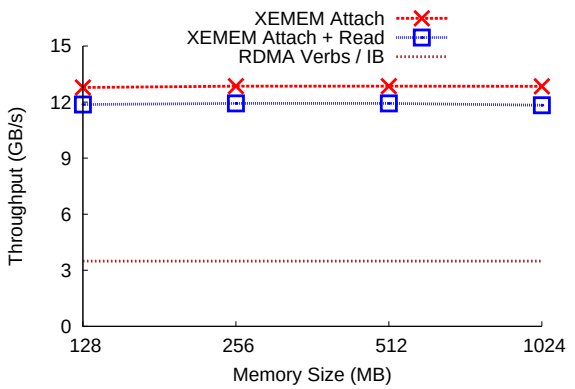



Figure 5: Cross-enclave throughput using shared memory and RDMA Verbs over Infiniband. Each XEMEM data point represents the throughput of 500 attachments for a given size

# 5. SHARED MEMORY EVALUATION

In this section, we present an experimental evaluation of our cross-enclave shared memory implementation. This evaluation will focus on the scalability of our implementation, with respect to the amount of memory shared across enclave partitions, as well as with respect to the number of enclaves running simultaneously on the system. These experiments will focus on performance measurements of shared memory attachment operations, as well as the performance implications for performing memory mappings in different enclave environments. We also compare our approach with an alternative approach to inter-enclave communication using RDMA.

## 5.1 System Configuration

This evaluation was carried out on a Dell PowerEdge R420 server configured with dual-socket 6-core Intel Xeon processors running at 2.10 GHz, with hyperthreading enabled for a total of 24 threads of execution. The memory layout consisted of two NUMA sockets with 16 GB of memory each, with memory interleaving disabled for a total of 32 GB of RAM. The system was also configured with a dual-port QDR Mellanox ConnectX-3 Infiniband device with SR/IOV enabled in order to demonstrate the potential for cross-enclave communication using RDMA. The system ran a stock Fedora 19 operating system.

For each of our experiments, the XEMEM name server was configured to run in the native Linux control enclave. Based on the individual details of the experiments, we created a set of enclaves using the Kitten lightweight kernel and Palacios virtual machine monitor to study both native and virtualized environments. In some configurations, the Palacios VMs were configured to run on the native Linux management enclave, while in others the VMs were deployed in isolated co-kernel enclaves managed by Kitten. In either case, all VMs ran a stock Centos 7 guest operating system. Furthermore, in each experiment, enclaves were only allocated memory and CPUs from a single NUMA socket in order to avoid overhead resulting from cross-NUMA domain memory accesses.

## 5.2 Shared Memory Attachment vs. RDMA Throughput

We first ran an experiment to demonstrate the potential throughput of cross-enclave communication using XEMEM compared to an alternative mechanism using RDMA. While these mechanisms are fundamentally different (byte-addressable memory operations versus block transfers over a peripheral bus), our goal was to demonstrate that our implementation does not add significant overhead, to the degree of diminishing throughout to the level of a simpler network-based approach. In this experiment, we created 1 Kitten enclave in addition to the Linux control enclave. A process in the Kitten enclave exported a single memory region of varying sizes, ranging from 128 MB to 1 GB. On the Linux enclave, a process repeatedly attached to the exported memory region, measuring the time it took to complete the attachment, as well as the time to read out the memory contents. Each memory region attachment was repeated 500 times. For the RDMA experiment, we configured the system's dual-port Infiniband device with 2 virtual functions and assigned each to a separate KVM virtual machine. We then performed a simple RDMA write bandwidth test using the recommended MTU supported by the device.

The results of this experiment are shown in Figure 5. As can be seen, for each memory size the shared memory implementation achieves a sustained throughput of around 13 GB/s for attachment operations, with around 12 GB/s when including the time to read out the memory contents. In comparison, the RDMA bandwidth test demonstrated that slightly less than 3.5 GB/s can be transferred across the Infiniband device using RDMA, showing that the overhead of XEMEM operations does not significantly reduce shared memory throughput. The experiment also demonstrates good scalability as XEMEM memory sizes increase, which bodes well for the prospects of applications hoping to make use of large shared memory regions.

## 5.3 Scalability of Multi-OS/R Shared Memory

Our next experiment was designed to demonstrate the ability of our implementation to scale to many enclave partitions running simultaneously on the same system, as well as the ability to support increasingly large shared memory regions in each enclave. In this experiment, we configured our system to create 1, 2, 4, or 8 Kitten enclaves using our co-kernel architecture. Each enclave was configured to run on a single core with 1.5 GB of memory, and to export memory regions ranging from 128 MB to 1 GB in size. Furthermore, for each enclave in the system a separate Linux process was created to attach to a single enclave's memory. In the case of 8 enclaves, this meant that up to 16 of the machine's 24 hardware threads were largely devoted to performing XEMEM operations. Note that although we chose a 1:1 communication model for this experiment, any arbitrary model is supported by our system.

The results of this experiment are illustrated in Figure 6. The figure demonstrates that the system scales well with respect to the amount of memory being shared at any point in time. This is a result of the fact that neither performing page table walks to generates page frame lists, nor subsequently communicating these lists across enclave communication channels, is a limit to the scalability of the memory sizes that can be shared. Furthermore, this figure also

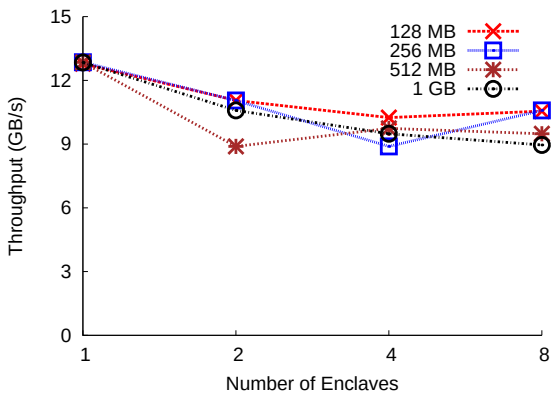



Figure 6: Cross-enclave throughput using shared memory between native Linux processes and native Kitten processes in co-kernel enclaves. Each data point represents the throughput of at least 500 attachments for a given size

demonstrates that, beyond the initial configuration of only 1 enclave and 1 native Linux process, increasing the number of enclaves does not limit the scalability of the system. This indicates that our approach of using a centralized name server for allocating *segids*, as well as using a distributed routing protocol for performing shared memory attachments, are both suitable mechanisms for maintaining performance in the presence of high numbers of contending cores.

On the other hand, the figure does demonstrate slight performance degradation as the system scales from 1 to 2 enclaves. Given the good scaling demonstrated beyond 2 enclaves, we do not attribute this overhead to any scalability bottlenecks in our implementation, but rather attribute it to two main factors that are not fundamental limitations of our system. First, the co-kernel architecture used to perform cross-enclave message transfers (which are required for the transmission of XEMEM operations) currently restricts all IPI-based communication with the Linux management enclave to core 0 of the system, and thus the presence of multiple enclaves may cause contention for interrupt handlers on this core. The result is that even though the separate Linux processes performing XEMEM attachments may be running on separate cores, the low-level messages facilitating their operation must be handled on a single core. Future work in the design of the co-kernel architecture will investigate more intelligent mechanisms for interrupt handling. Additionally, we also attribute this overhead to contention for Linux data structures that are accessed when multiple processes concurrently update memory maps.

## 5.4 Performance of Shared Memory Using Virtual Machines

In addition to measuring the performance achievable in multi-enclave configurations with native OS/Rs, we also measured the throughput achievable when using the Palacios host/guest communication channel and memory translation mechanisms. We ran two separate experiments to measure both the host-to-guest and the guest-to-host performance as discussed in Section 4.4. In the first experiment, we launched a Linux VM running on the Linux management enclave, and we executed a single process in the VM that repeatedly attached to a 1 GB region of memory that was exported by a native Kitten process executing in a co-kernel enclave. In the second experiment, we deployed the same enclave configuration, but instead configured the Linux VM process to export a 1 GB region to be attached by the Kitten process.

| Exporting Enclave | Attaching Enclave | GB/s (w/o rb-tree inserts) |
|---|---|---|
| Kitten | Linux | 12.841 (N/A) |
| Kitten | Linux (VM) | 3.991 (8.79) |
| Linux (VM) | Kitten | 12.606 (N/A) |

Table 2: Cross-enclave throughput using shared memory between a Linux process and a native Kitten process executing in a co-kernel enclave. Each value represents the throughput of at least 500 attachments to a 1 GB memory region

The results of these experiments can be seen in Table 2. The top row of the table shows the 1 GB result reported in Figure 5 for the native co-kernel configuration. The table demonstrates that moving the attaching process from the native Linux enclave to a virtualized Linux enclave causes a roughly 3x decrease in throughput when compared to the native configuration. In order to determine the source of this overhead, we measured the amount of time spent mapping remote enclave memory into the VM (those operations illustrated in Figure 4(a)), and we found that nearly 80% of the time was spent updating the guest's memory map. As discussed in Section 4.4, Palacios maintains a guest's memory map in the form of a red-black tree, and the process of mapping remote enclave memory generally requires as many updates to this tree as there are pages being shared. Thus, as the tree continues to grow, the cost for insertions and re-balancing operations increases, leading to performance degradation. Indeed, when removing the time spent updating the red-black tree, the throughput increases to 8.79 GB/s. In the future we intend to remove this overhead through the use of more intelligent radix tree based data structures that can more appropriately mimic a page table's organization.

Lastly, the bottom row of Table 2 shows that mechanisms for performing guest-to-host memory translations (illustrated in Figure 4(b)) are not nearly as costly, as the Kitten process is able to achieve over 12 GB/s throughput when attaching to the VM Linux process. This result indicates that performing page translations in Palacios does not add significant overhead to shared memory operations in the common case where the size of the Palacios memory map is limited.

## 5.5 Operating System Noise

The final experiment in the first part of our evaluation measured the impact of performing page frame lookups in Kitten enclaves on the Kitten noise profile. OS noise, which has been identified as a significant source of overhead for HPC applications particularly at large scales [9, 14], is largely non-existent in Kitten as a result of its feature-limited design. Thus, while shared memory attachments may not necessarily qualify as noise, given that they are directly enabled by the HPC application and necessary in some sense for the progress of a composed workload, measuring the impact of attachment operations on the Kitten noise profile can provide some insight on the types of syn-

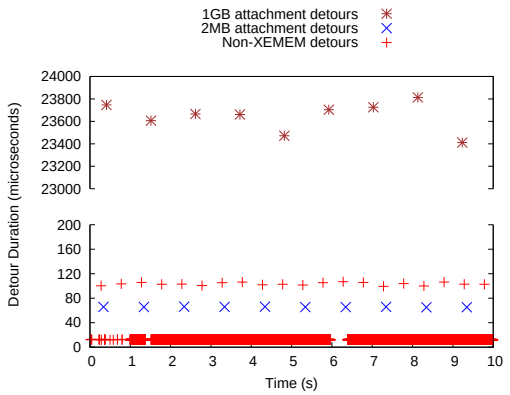



Figure 7: Noise profile of a Kitten enclave serving XEMEM attachment requests on a single core

| Simulation Enclave | Analytics Enclave |
|---|---|
| Native Linux | Native Linux |
| Kitten Co-Kernel | Native Linux |
| Kitten Co-Kernel | Linux VM (Linux Host) |
| Kitten Co-Kernel | Linux VM (Kitten Host) |

Table 3: Enclave configurations for the in situ workload in the single node experiments

chronization that may be needed to prevent attachments from perturbing simulations in future systems.

In this experiment, we configured a Kitten enclave on a single core to export memory regions of sizes 4 KB, 2 MB, and 1 GB, and ran the Selfish Detour benchmark [2] from Argonne National Lab, a noise detection benchmark that is designed to measure the fraction of time the CPU spends executing instructions that are not part of the user's application. We ran a process on the Linux enclave to attach to each region, sleep for one second, and repeat for a period of 10 seconds. Figure 7 presents the results of the experiment. The figure shows that Kitten experiences a baseline level of frequent hardware noise around the 12 microsecond mark, as well as a set of less frequent interruptions likely caused by periodic hardware events such as SMIs around the 100 microsecond mark. Interestingly, detours caused by 4 KB attachments are not noticeable in the figure, as they cause detours similar in length to the frequent noise baseline. Detours caused by 2 MB attachments are more noticeable, but still cause less of a disturbance than the periodic hardware interrupts. Finally, the 1 GB attachments cause much larger detours, creating noise events that are 2 orders of magnitude longer than any other events. For these memory sizes, only the 1 GB attachments seem likely to cause problems for large-scale HPC workloads on Kitten, and thus special care would be needed to synchronize their occurrence with respect to the application workflow.

# 6. SINGLE NODE BENCHMARK EVALUATION

The second part of our evaluation uses a set of HPC benchmark workloads to demonstrate our system's ability to support sample in situ workloads in multi-enclave environments. In this section, our experiments will focus on the single-node performance implications of XEMEM in the presence of a wide range of enclave configurations and in situ workflow models. Then, in Section 7 we will evaluate a multi-node system configuration where each node employs XEMEM for local-node in situ execution, and demonstrate that the performance isolation benefits of multi-enclave configurations can lead to superior scaling behavior.

## 6.1 In Situ Workload

To present the performance characteristics of a composed in situ workload, we modified the HPCCG benchmark from the Mantevo suite [11] as well as the STREAM microbenchmark from the HPC Challenge suite [7] to synchronize their execution flows. Specifically, we modified the applications to use simple stop/go signals implemented on top of variables in shared memory. Throughout the evaluation sections, we refer to these modified benchmark components, respectively, as the HPC simulation and analytics program.

The HPC simulation executes an iterative conjugate gradient algorithm with collective operations in between iterations to gather intermediate results. We modified the benchmark to signal the analytics program at certain intervals during its execution to simulate an in situ workflow. During these intervals, the simulation sends a signal to the analytics program by writing to a variable in shared memory. The simulation then waits for a return signal by polling on another variable in shared memory, which is written by the analytics programs when it determines to resume the simulation. In all, we configured the HPC simulation to execute 600 iterations of the conjugate gradient algorithm, and to communicate with the analytics program every 40 intervals for a total of 15 communication points. Upon receiving a signal from the HPC simulation, the analytics program may or may not attach to a new exported region created by the simulation, a configuration option which we discuss below. In either case, the analytics program executes the STREAM benchmark over a 512 MB region specified by the simulation. The analytics program first copies the shared memory into a private array, and then executes STREAM over the array.

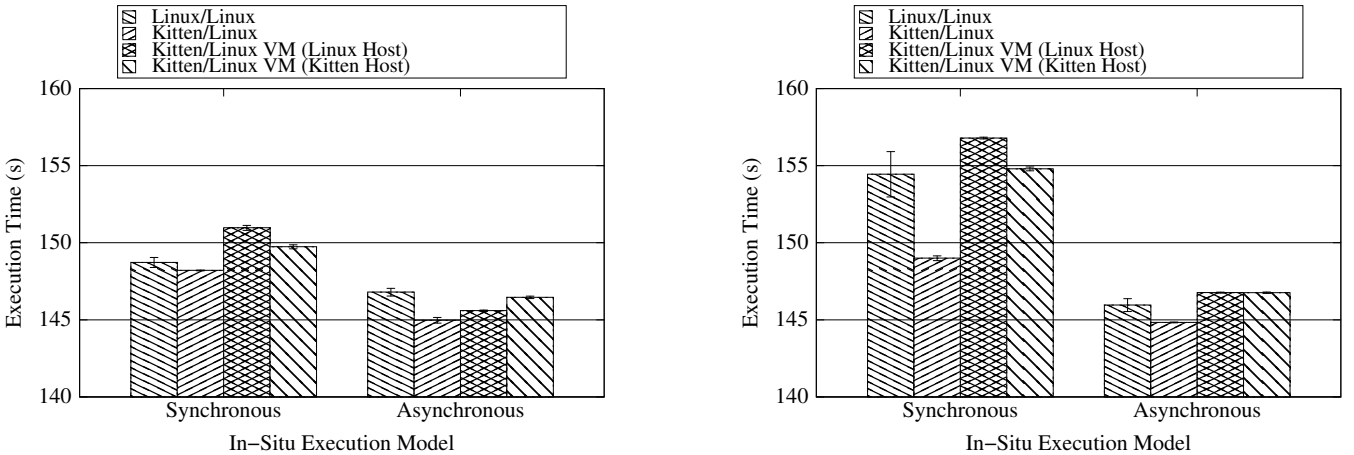
We note that currently the underlying enclave OS/Rs only support application communication through shared memory, and thus operations like event notifications must be supported via ad hoc techniques like polling on variables in memory. We plan to investigate techniques to support additional features in the OS/R environments as requirements of actual composed workflows become more evident.

## 6.2 Execution and Memory Registration Models

The simulation and analytics programs can be configured based on two different parameters to mimic different in situ communication behaviors: synchronous/asynchronous execution models, and one time/recurring shared memory attachments.

### *6.2.1 Synchronous vs. Asynchronous Execution*

When using a synchronous execution model, the simulation and analytics programs do not simultaneously execute. Rather, when the analytics program receives a signal from the simulation, it (optionally) attaches to simulation data via shared memory, and executes the STREAM benchmark. Once the benchmark completes, it sends a signal to the HPC simulation to continue. When using an asynchronous execu-




(a) One time shared memory attachment model (b) Recurring shared memory attachment model

Figure 8: Performance of a sample in situ benchmark on a single node using various communication and execution workflows

tion model, the simulation and analytics programs may execute simultaneously. When the analytics program receives a signal from the simulation, it (optionally) attaches to the simulation data via shared memory, and then immediately signals the HPC simulation to continue. Then, it executes STREAM as the HPC simulation resumes simultaneously.

#### 6.2.2 *One time vs. Recurring Shared Memory Attachments*

When using a one time shared memory attachment model, the HPC simulation exports a single region of memory at the start of the simulation. During communication intervals, the analytics program does not attach to new memory regions, but rather attaches a single time to shared memory region which persists over the benchmark duration. Conversely, when using the recurring attachment model, the simulation exports a new shared memory region during each communication interval, which is subsequently attached by the analytics program. Thus, for the former case, the overhead of setting up shared memory attachments is only experienced once at the beginning of the application; conversely, in the latter case, the overheads are experienced at each communication interval.

## 6.3 System Configuration

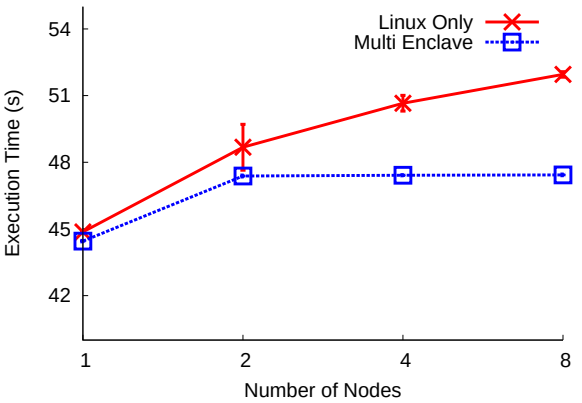
The system used for this evaluation was a Dell OptiPlex server with a single-socket 4-core Intel i7 processor running at 3.40 GHz, with hyperthreading enabled for a total of 8 threads of execution. The memory layout consisted of a single memory zone with 8 GB of RAM. This system ran a stock Centos 7 operating system. As in the previous experiments, for each experiment in this section we configured the XEMEM name server to run in the native Linux control enclave.

We varied the execution environments for the HPC simulation and analytics applications to study a variety of configurations. For a baseline comparison we configured the simulation and analytics applications to execute in the native Linux control enclave, using the XEMEM shared memory facilities provided for single Linux environments. In the remaining configurations, the HPC simulation was configured to execute in a Kitten co-kernel enclave, while the analytics application was deployed in Palacios virtual machines executing on the host Linux control enclave as well as a separate Kitten co-kernel host. The configurations are shown in Table 3.

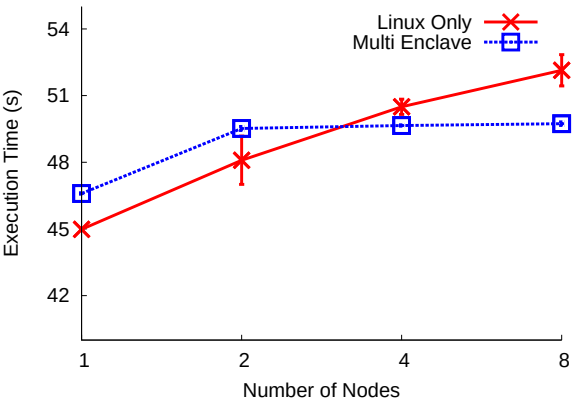
## 6.4 Results

In addition to varying the enclave configurations, we executed the in sample in situ workload in four separate configurations based on the four combinations of the workflow parameters discussed in Section 6.2. The completion times of the HPC simulation using a one time shared memory attachment model are shown in Figure 8(a), where each bar reports the average and standard deviation of 10 runs of the application. As expected, for each environment the HPC simulation completes in less time when executing asynchronously with respect to the analytics program than executing synchronously. Under both execution models, the configuration deploying the HPC simulation in the Kitten enclave and the analytics program natively on the Linux enclave outperforms all other configurations. When using the asynchronous execution model, each environment utilizing the Kitten enclave for the simulation outperforms the Linux-only configuration. Conversely, when using the synchronous execution model, any overheads experienced in processing the analytics directly impact the runtime of the HPC simulation. Given the lack of shared memory communication overhead in these experiments, it is clear that the native analytics program slightly outperforms the same program running virtualized, particularly in the Palacios on Linux case. However, in each multi-enclave configuration the performance is more consistent than exhibited by the Linux-only environment, demonstrating that our approach does not add variable delays to the runtime of the simulation.

The results of the experiments using a recurring shared memory attachment model can be seen in Figure 8(b). Given the overheads from repeated memory map updates identified in Section 5.4, it is clear that performing recurring shared memory attachments in addition to using a synchronous execution model represents the worst case configuration for the virtualized enclaves. However, interestingly the Linux-only environment also suffers significant overhead as well as a marked increase in runtime variance in this case. We attribute this overhead mainly to the page faulting semantics with which single environment XEMEM attachment




(a) One time shared memory attachment model




(b) Recurring shared memory attachment model

Figure 9: Performance of a sample in situ benchmark on a multi-node cluster using an asynchronous execution model. The benchmark is configured in weak-scaling mode

operations are performed in Linux. Indeed, when executing asynchronously, the overheads associated with both the virtualized and native Linux environments largely disappear. As in the previous cases, the performance in each multi-enclave configuration is very consistent, whereas the Linux-only configuration provides a lower degree of workload isolation leading to increased variance.

# 7. MULTI-NODE BENCHMARK EVALUATION

The final experiments for our evaluation were designed to demonstrate the benefits multi-enclave configurations provide for composed in situ applications executing on multiple nodes. As the previous section demonstrated, even on a single node, the performance isolation that isolated enclaves provide leads to a more consistent runtime experience for the composed workload. In this section, we show that providing multi-enclave shared memory for in situ components running on each node of a multi-node system leads to superior scalability, demonstrating the value of workload isolation.

## 7.1 Workload and System Configuration

For the experiments in this section, we used a local 8-node experimental research cluster. The nodes were each configured with dual-socket 6-core Intel Xeon processors running at 2.10 GHz, with hyperthreading enabled for a total of 24 threads of execution. The memory layout on each node consisted on two NUMA sockets with 16 GB of memory each, with memory interleaving disabled for a total of 32 GB of RAM. The nodes were interconnected with dual port QDR Mellanox ConnectX-3 Infiniband devices.

In these experiments we deployed two separate enclave environments in the system. In the default configuration, both in situ components were executed in the native Linux enclave with no other enclaves deployed in the system. The other configuration consisted of a Palacios VM enclave running on an isolated Kitten co-kernel host, in addition to the native Linux enclave. For this system composition, we ran the HPC simulation in the Palacios VM, while the analytics program executed in the native Linux enclave. For each individual experiment, every node ran the same enclave configuration.

Finally, we executed the same in situ workload used in the single node experiments, with the exception that the HPC simulation was compiled to use OpenMPI over the Infiniband interconnect for multiple node deployment. The HPC simulation used 8 cores of each node, while the analytics program was parallelized using OpenMP threads to use 8 additional cores on each node. The HPC simulation was configured to execute 300 iterations of the conjugate gradient algorithm, and to communicate with the analytics program every 30 intervals for a total of 10 communication points on each node. The analytics program on each node then executes the STREAM benchmark over a 1 GB region specified by the simulation. As in the previous experiments, all workloads were explicitly pinned to NUMA domains in order to avoid the overheads on cross-domain contention. The benchmark was configured in weak-scaling mode, where the problem size of the HPC simulation scales with the number of nodes.

## 7.2 Results

In order to evaluate the benefits of performance isolation that our multi-enclave system provides, we ran the in situ benchmark using an asynchronous execution workflow, meaning that during the application's communication intervals, the HPC simulation and the analytics program are executing concurrently. This workload deployment is representative of the types of applications that we envision will be present on exascale systems in that it requires high performance local-node communication between its individual components, but at the same time requires that the components be strictly isolated from interference caused by contention for the node's resources.

As in the previous experiments, we ran these experiments with two separate shared memory models: one time and recurring attachments. The results of the experiments can be seen in Figure 9, where each data point reports the average and standard deviation of five runs of the benchmark. As Figure 9(a) demonstrates, the scaling behavior of the multi-enclave configuration is superior to that provided by the baseline Linux only environment for the experiments using a one time memory attachment model. This result is particularly interesting because the HPC simulation is running in a virtualized environment, which demonstrates two key

results. First, it shows that performance isolation is a requirement for these types of applications, to the degree that the same workload running virtualized can outperform itself running natively if it is better isolated. Furthermore, this result demonstrates that the XEMEM implementation yields a consistent execution environment on each node, which can be seen by the low standard deviation and very good scaling behavior.

Finally, the results of the experiments using a recurring shared memory attachment model can be seen in Figure 9(b). As the overhead of shared memory attachments is experienced multiple times during the application, the Linux only configuration is able to outperform the multi-enclave configuration at a single node. However, both system configurations exhibit similar scaling behavior using this shared memory attachment model as they exhibited in Figure 9(a). Namely the lack of performance isolation in the single OS/R configuration leads to steady performance decline as each node has a different runtime experience, while the multi-enclave system shows almost no performance degradation past 2 nodes.

# 8. CONCLUSION

Research trends in the HPC community indicate that exascale systems will be constructed from multiple heterogeneous hardware and system software configurations, called *enclaves*. In this work, we presented the design, implementation, and evaluation of XEMEM, a shared memory system capable of supporting a wide variety of enclave configurations likely to be seen in exascale systems. We demonstrated that XEMEM is able to scale to a high number of enclaves simultaneously executing on a system, even while each enclave creates increasingly large shared memory mappings. We further evaluated our system's ability to support a sample *in situ* workload utilizing a set of different of execution and memory registration models. Finally, we demonstrated XEMEM's ability to support a multi-enclave in situ application, which can outperform the same application executing in a single OS/R environment.

# 9. REFERENCES


[1] P. Beckman et al. Argo: An Exascale Operating System and Runtime Reseach Project. [Online]. Available: `http://www.argo-osr.org`, 2014.

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

[3] R. Brightwell, R. Oldfield, A. Maccabe, and D. Bernholdt. Hobbes: Composition and Virtualization as the Foundations of an Extreme-scale OS/R. In *Proc. 3rd International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, 2013.

[4] R. Brightwell, K. Pedretti, and T. Hudson. SMARTMAP: Operating System Support for Efficient Data Sharing Among Processes on a Multi-Core Processor. In *Proc. 21st. International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2008.

[5] A. Burstev et al. Fido: Fast Inter-Virtual-Machine Communication for Enterprise Appliances. In *Proc. Usenix Annual Technical Conference (ATC)*, 2009.

[6] D. Chen et al. The IBM Blue Gene/Q Interconnection Network and Message Unit. In *Proc. 24th International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[7] J. Dongarra et al. Introduction to the HPCChallenge Benchmark Suite. Technical report, University of Tennessee Knoxville, 2005.

[8] J. Dongarra et al. The International Exascale Software Project Roadmap. *International Journal of High Performance Computing Applications (IJHPCA)*, 25(1):3–60, 2011.

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

[10] B. Goglin and S. Moreaud. KNEM: a Generic and Scalable Kernel-Assisted Intra-node MPI Communication Framework. *Journal of Parallel and Distributed Computing (JPDC)*, 73(2):176–188, 2013.

[11] M. Heroux et al. Improving Performance via Mini-applications. Technical report, Sandia National Laboratories, 2009.

[12] P. Kogge et al. Exascale Computing Study: Technology Challenges in Achieving Exascale Systems. Technical report, University of Notre Dame CSE Department, 2008.

[13] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, M. Levenhagen, R. Brightwell, A. Gocke, and S. Jaconette. Palacios: A New Open Source Virtual Machine Monitor for Scalable High Performance Computing. In *Proc. 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2010.

[14] A. Morari, R. Gioiosa, R. Wisniewski, B. Rosenburg, T. Inglett, and M. Valero. Evaluating the Impact of TLB Misses on Future HPC Systems. In *Proc. 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2012.

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

[16] 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 (SBAC-PAD)*, 2012.

[17] R. Stevens, A. White, et al. Architectures and Technology for Extreme Scale Computing. Technical report, U.S. Department of Energy, 2009.

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

[19] J. Wang, K.-L. Wright, and K. Gopalan. XenLoop: A Transparent High Performance Inter-VM Network Loopback. In *Proc. 17th International Symposium on High Performance Distributed Computing (HPDC)*, 2008.

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

[21] M. Woodacre, D. Robb, D. Roe, and K. Feind. The SGI Altix 3000 Global Shared-Memory Architecture. Technical report, Silicon Graphics International Corporation, 2003.

[22] X. Zhang, S. McIntosh, P. Rohatgi, and J. Griffin. XenSocket: A High Throughput Interdomain Transport for Virtual Machines. In *Proc. ACM/IFIP/USENIX International Conference on Middleware*, 2007.

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