Minimal-overhead Virtualization of a Large Scale Supercomputer

John R. Lange
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
jacklange@cs.pitt.edu

Kevin Pedretti
Scalable System Software Department
Sandia National Laboratories
Albuquerque, NM 87123
ktpedre@sandia.gov

Peter Dinda Chang Bae
Department of Electrical Engineering and Computer Science
Northwestern University
Evanston, IL 60208
{pdinda,changb}@northwestern.edu

Patrick G. Bridges Philip Soltero
Department of Computer Science
University of New Mexico
Albuquerque, NM 87131
{bridges,psoltero}@cs.unm.edu

Alexander Merritt
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
merritt.alex@gatech.edu

Abstract

Virtualization has the potential to dramatically increase the usability and reliability of high performance computing (HPC) systems. However, this potential will remain unrealized unless overheads can be minimized. This is particularly challenging on large scale machines that run carefully crafted HPC OSes supporting tightly-coupled, parallel applications. In this paper, we show how careful use of hardware and VMM features enables the virtualization of a large-scale HPC system, specifically a Cray XT4 machine, with $\leq$5% overhead on key HPC applications, microbenchmarks, and guests at scales of up to 4096 nodes. We describe three techniques essential for achieving such low overhead: passthrough I/O, workload-sensitive selection of paging mechanisms, and carefully controlled preemption. These techniques are forms of symbiotic virtualization, an approach on which we elaborate.

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

General Terms Design, Experimentation, Measurement, Performance

This project is made possible by support from the National Science Foundation (NSF) via grants CNS-0709168 and CNS-0707365, and the Department of Energy (DOE) via grant DE-SC0005343. Patrick Bridges was partially supported by a faculty sabbatical appointment at Sandia National Labs. Alexander Merritt's work on this project was conducted as part of a student internship at Sandia National Laboratories. Sandia is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Keywords virtual machine monitors, parallel computing, high performance computing

1. Introduction

Virtualization has the potential to dramatically increase the usability and reliability of high performance computing (HPC) systems by maximizing system flexibility and utility to a wide range of users [11, 13, 23, 24]. Many of the motivations for virtualization in data centers apply also equally to HPC systems, for example allowing users to customize their OS environment (e.g. between full-featured OSes and lightweight OSes). Additionally, virtualization allows multiplexing of less-demanding users when appropriate. Finally, virtualization is relevant to a number of research areas for current petascale and future exascale systems, including reliability, fault-tolerance, and hardware-software co-design.

The adoption of virtualization in HPC systems can only occur, however, if it has minimal performance impact in the most demanding uses of the machines, specifically running the capability applications that motivate the acquisition of petascale and exascale systems. Virtualization cannot succeed in HPC systems unless the performance overheads are truly minimal and, importantly, that those overheads that do exist do not compound as the system and its applications scale up.

This challenge is amplified on high-end machines for several reasons. First, these machines frequently run carefully crafted custom HPC OSes that already minimize overheads and asynchronous OS interference (OS noise) [9, 25], as well as make the capabilities of the raw hardware readily available to the application developer. Second, the applications on these machines are intended to run at extremely large scales, involving thousands or tens of thousands of nodes. Finally, the applications are typically tightly coupled and communication intensive, making them very sensitive to performance overheads, particularly unpredictable overheads. For this reason, they often rely on the deterministic behavior of the HPC OSes on which they run.

In this paper, we show how scalable virtualization with $\leq$5% overhead for key HPC applications and guests can be achieved in a high-end message-passing parallel supercomputer, in this case a Cray XT4 supercomputer [2] at scales in excess of 4096 nodes. For guests, we examined the behavior of both the custom Catamount HPC OS [17] and the Cray CNL guest [16], an HPC OS derived from the Linux operating system. Our performance overheads are measured using three application benchmarks and a range of microbenchmarks.

The virtual machine monitor that we employ is Palacios, an open source, publicly available VMM designed to support the virtualization of HPC systems and other platforms. We have previously reported on the design, implementation, and evaluation of Palacios [20]. The evaluation included limited performance studies on 32–48 nodes of a Cray XT system. In addition to considering much larger scales, this paper focuses on the essential techniques needed to achieve scalable virtualization at that scale and how a range of different VMM and hardware virtualization techniques impact the scalability of virtualization.

The essential techniques needed to achieve low overhead virtualization at these scales are passthrough I/O, workload-sensitive selection of paging mechanisms, and carefully controlled preemption. Passthrough I/O provides direct guest / application access to the specialized communication hardware of the machine. This in turn enables not only high bandwidth communication, but also preserves the extremely low latency properties of this hardware, which is essential in scalable collective communication.

The second technique we have determined to be essential to low overhead virtualization at scale is the workload-sensitive selection of the paging mechanisms used to implement the guest physical to host physical address translation. Palacios supports a range of approaches, from those with significant hardware assistance (e.g. nested paging, which has several implementations across Intel and AMD hardware), and those that do not (e.g., shadow paging, which has numerous variants). There is no single best paging mechanism; the choice is workload dependent, primarily on guest context switching behavior and the memory reference pattern.

The final technique we found to be essential to low overhead virtualization at scale is carefully controlled preemption within the VMM. By preemption, we mean both interrupt handling and thread scheduling, specifically carefully controlling when interrupts are handled, and using cooperative threading in the VMM. This control mostly avoids introducing timing variation in the environment that the guest OS sees, in turn meaning that carefully tuned collective communication behavior in the application remains effective.

What these techniques effectively accomplish is to keep the virtual machine as true to the physical machine as possible in terms of its communication and timing properties. This in turn allows the guest OS's and the application's assumptions about the physical machine it is designed for to continue to apply to the virtual machine environment. In the virtualization of a commodity machine, such authenticity is not needed. However, if a machine is part of a scalable computer, disparities between guest OS and application assumptions and the behavior of the actual virtual environment can lead to performance impacts that grow with scale.

We generalize beyond the three specific techniques described above to argue that to truly provide scalable performance for virtualized HPC environments, the black box approach of commodity VMMs should be abandoned in favor of a symbiotic virtualization model. In the symbiotic virtualization model, the guest OS and VMM function cooperatively in order to function in a way that optimizes performance. Our specific techniques are examples of symbiotic techniques, and are, in fact, built on the SymSpy passive symbiotic information interface in Palacios.

Beyond supercomputers, our experiences with these symbiotic techniques are increasingly relevant to system software for general-purpose and enterprise computing systems. For example, the increasing scale of multicore desktop and enterprise systems has led OS designers to consider treating multicore systems like tightly-coupled distributed systems. As these systems continue to scale up toward hundreds or thousands of cores with distributed memory hierarchies and substantial inter-core communication delays, lessons learned in designing scalable system software for tightly-coupled distributed memory supercomputers will be increasingly relevant to them.

Our contributions are as follows:

- We demonstrate that it is possible to virtualize a high-end supercomputer at large scales (4096 nodes) with minimal performance overhead ($\leq$5%). As far as we are aware, our results represent the largest scale virtualization study to date.
- We describe the three techniques essential for achieving such low overheads at scale: passthrough I/O, workload-sensitive selection of paging mechanisms, and carefully controlled preemption.
- We generalize from the mechanisms to the concept of symbiotic virtualization, which we describe and argue will become of increasing importance as scalable systems become ubiquitous.

2. Virtualization system overview

Our contributions are made in the context of the Palacios VMM and Kitten lightweight kernel. For our experiments in this paper, Palacios is embedded into Kitten, making possible a system call for instantiating a VM from a guest OS image. A detailed description of these systems and their interaction is available elsewhere [20]. We now summarize these systems.

2.1 Palacios

Palacios is a publicly available, open source, OS-independent VMM designed and implemented as part of the V3VEE project (`http://v3vee.org`). developed from scratch that targets the x86 and x86_64 architectures (hosts and guests) with either AMD SVM [3] or Intel VT [14] extensions. It is designed to be embeddable into diverse host OSes, and we presently have embedded it into Kitten, GeekOS, Minix 3, and Linux. When embedded into Kitten, the combination acts as a type-I VMM—guest OSes do not require any modification to run. Palacios can run on generic PC hardware, in addition to specialized hardware such as Cray XT supercomputer systems.

Palacios creates a PC-compatible virtual environment for guest OSes by handling *exits* that are raised by the hardware on guest operations that the VMM defines as requiring interception. This is a common structure for a VMM, often referred to as "trap-and-emulate". For example, VM exits frequently occur on interrupts, reads and writes to I/O ports and specific areas of memory, and use of particular hardware instructions and registers (e.g. CPU control registers). These exits allow the VMM to intervene on key hardware operations when necessary, emulating or changing requested hardware behavior as needed. Because exit handling incurs overhead, carefully controlling what operations exit and what is done on each exit is essential to providing scalability and performance.

2.2 Kitten host OS

Kitten is a publicly available, GPL-licensed, open source OS designed specifically for high performance computing. The general philosophy being used to develop Kitten is to borrow heavily from the Linux kernel when doing so does not compromise scalability or performance (e.g., adapting the Linux bootstrap code). Performance critical subsystems, such as memory management and task scheduling, are replaced with code written from scratch.

Kitten's focus on HPC scalability makes it an ideal host OS for a VMM on HPC systems, and Palacios's design made it easy to embed it into Kitten. In particular, host OS/VMM integration was accomplished with a single interface file of less than 300 lines of code. The integration includes no internal changes in either the VMM or host OS, and the interface code is encapsulated together with the VMM library in an optional compile time module for the host OS.

The Kitten host OS exposes VMM control functions via a system call interface available from user space. This allows user level tasks to instantiate VM images directly. The result is that VMs can be loaded and controlled via processes received from the job loader. A VM image can thus be linked into a standard job that includes loading and control functionality.

3. Virtualization at scale

In our initial experiments, we conducted a detailed performance study of virtualizing a Cray XT 4 supercomputer. The study included both application and microbenchmarks, and was run at the largest scales possible on the machine (at least 4096 nodes, sometimes 6240 nodes). The upshot of our results is that it is possible to virtualize a large scale supercomputer with $\leq 5\%$ performance penalties in important HPC use-cases, even when running communication-intensive, tightly-coupled applications. In the subsequent sections, we explain how and present additional studies that provide insight into how different architectural and OS approaches to virtualization impact the performance of HPC applications and micro-benchmarks.

3.1 Hardware platform

Testing was performed during an eight hour window of dedicated system time on Red Storm, a Cray XT4 supercomputer made up of 12,960 single-socket compute nodes, each containing either a dual-core or quad-core processor. Because Palacios requires virtualization support not present in the older dual-core processors, testing was limited to the system's 6,240 quad-core nodes. These nodes each consist of a 2.2 GHz AMD Opteron Barcelona quad-core processor, 8 GB of DDR2 memory, and a Cray SeaStar 2.1 network interface. The nodes are arranged in a 13x20x24 3-D mesh topology with wrap-around connections in the Z dimension (i.e. the system is a torus in the Z-dimension only).

Red Storm was jointly developed by Sandia and Cray, and was the basis for Cray's successful line of Cray XT supercomputers. There are many Cray XT systems in operation throughout the world, the largest of which currently being the 18,688 node, 2.3 PetaFLOP peak "Jaguar" XT5-HE system at Oak Ridge National Laboratory. The experiments and results described in this paper are relevant to these systems and could be repeated on systems with quad-core or newer processors. We are in the process of negotiating time to repeat them on Jaguar.

3.2 Software environment

Each test was performed in at least three different system software configurations: native, guest with nested paging, and guest with shadow paging. In the native configuration, the test application or micro-benchmark is run using the Catamount HPC operating system [17] running on the bare hardware. This is the same environment that users normally use on Red Storm. Some tests were also run, at much smaller scales, using Cray's Linux-derived CNL [16] operating system.

The environment labeled "Guest, Nested Paging" in the figures consists of the VMM running on the bare hardware, managing an instance of Catamount running as a guest operating system in a virtual machine environment. In this mode, the AMD processor's nested paging memory management hardware is used to implement the guest physical address to host physical address mapping that is chosen by Palacios. The guest's page tables and a second set of page tables managed by the VMM are used for translation. Palacios does not need to track guest page table manipulations in this case; however, every virtual address in the guest is translated using a "two dimensional" page walk involving both sets of page tables [6]. This expensive process is sped up through the use of a range of hardware-level TLB and page walk caching structures.

In contrast, the "Guest, Shadow Paging" mode uses software-based memory management which disables the processor's nested paging hardware. Shadow paging avoids the need for a two dimensional page walk, but requires that the VMM track guest page tables. Every update to the guest's page tables causes an exit to the VMM, which must then validate the request and commit it to a set of protected "shadow" page tables, which are the actual page tables used by the hardware. We elaborate on the choice of paging mechanism later in the paper.

Virtualizing I/O devices is critical to VM performance, and, here, the critical device is the SeaStar communications interface [7]. Palacios provides guest access to the SeaStar using passthrough I/O, an approach we elaborate on later. We consider two ways of using the SeaStar, the default way, which is unnamed in our figures, and an alternative approach called "Accelerated Portals." The default approach uses interrupt-driven I/O and host-based message matching[1], while accelerated portals performs message matching on the NIC and does not generally require interrupt delivery.

In the version of AMD SVM available on the Cray XT4, intercepting any interrupt requires that all interrupts be intercepted. Because a variety of non-SeaStar interrupts must be intercepted by the VMM, this adds a VM exit cost to SeaStar interrupts. Essentially, when the VMM detects an exit has occurred due to a SeaStar interrupt, it immediately re-enters the guest, re-injecting the SeaStar interrupt as a software interrupt. This process requires O(1000) cycles, resulting in interrupt-driven SeaStar performance having a higher latency under virtualization than natively. Because accelerated portals uses user-level polling instead, the interrupt exit cost described above does not occur when the guest is virtualized. As a result, virtualized accelerated portals performance is nearly identical to native accelerated portals performance.

It is important to point out that if future versions of AMD's SVM hardware (and of Intel's VT hardware) supported *selective* interrupt exiting, we would be able to use it to avoid exiting on SeaStar interrupts, which should make interrupt-driven SeaStar performance under virtualization identical to that without virtualization.

The guest Catamount OS image we used was based on the same Cray XT 2.0.62 Catamount image used for the native experiments. Minor changes were required to port Catamount to the PC-compatible virtual machine environment provided by Palacios (the native Cray XT environment is not fully PC-compatible). Additionally, the SeaStar portals driver was updated to allow passthrough operation as described in Section 4.

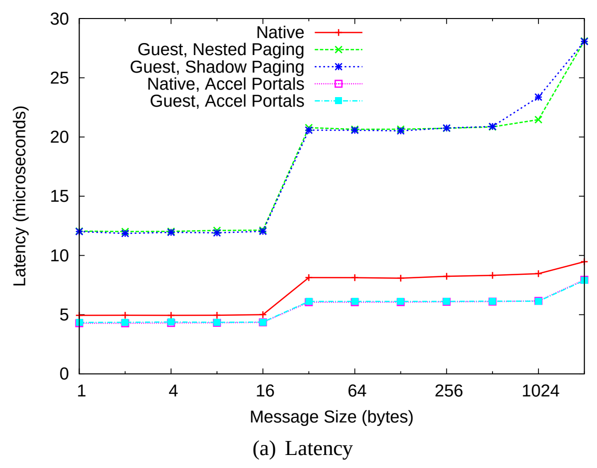
3.3 MPI microbenchmarks

The Intel MPI Benchmark Suite version 3.0 [15] was used to evaluate point-to-point messaging performance and scalability of collective operations.

3.3.1 Point-to-point performance

Figure 1 shows the results of a ping-pong test between two adjacent nodes. Small message latency, shown in Figure 1(a), is approxi-

[1] Many high-performance messaging systems *match* incoming large messages with pre-posted user buffers into which the data is directly received, avoiding unnecessary data copies.



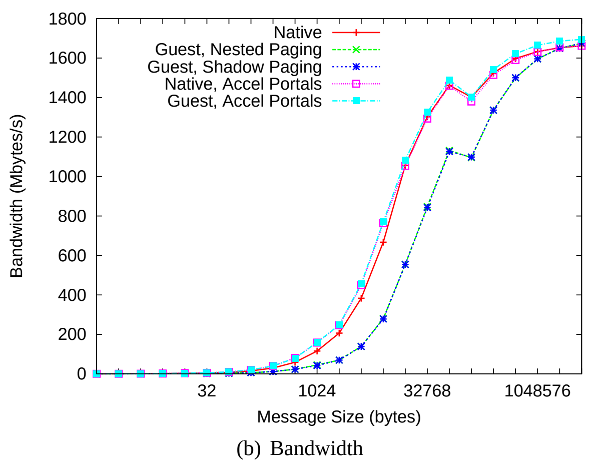


Figure 1. MPI PingPong microbenchmark (a) latency and (b) bandwidth for native and virtualized with both interrupt-driven message delivery and message processing offloaded to the Cray XT SeaStar NIC (accelerated portals). Note that with accelerated portals, guest and native performance are nearly identical due to the removal of interrupt virtualization overhead, resulting in overlapping lines on both graphs.

mately 2.5 times worse with nested or shadow guest environments compared to native, though choice of paging virtualization mode does not effect messaging latency. This is a result of the larger interrupt overhead in the virtualized environment. However, note that in absolute terms, for the smallest messages, the latency for the virtualized case is already a relatively low 12 μs, compared to the native 5 μs. Eliminating this virtualized interrupt overhead, as is the case with accelerated portals and would be the case with more recent AMD SVM hardware implementations, results in virtually identical performance in native and guest environments.

Figure 1(b) plots the same data but extends the domain of the x-axis to show the full bandwidth curves. The nested and shadow guest environments show essentially identical degraded performance for mid-range messages compared to native, but eventually reach the same asymptotic bandwidth once the higher interrupt cost is fully amortized. Bandwidth approaches 1.7 GByte/s. Avoiding the interrupt virtualization cost with accelerated portals results again in similar native and guest performance.

3.3.2 Collective performance

Figures 2, 3, 4, and 5 show the performance of the MPI Barrier, Allreduce, Broadcast, and Alltoall operations, respectively. The operations that have data associated with them, Allreduce and Alltoall, are plotted for the 16-byte message size since a common usage pattern in HPC applications is to perform an operation on a single double-precision number (8 bytes) or a complex double precision number (16 bytes).

Both Barrier. Allreduce, and Broadcast scale logarithmically with node count, with Allreduce having slightly higher latency at all points. In contrast, Alltoall scales quadratically and is therefore plotted with a log y-axis. In all cases, the choice of nested vs. shadow paging is not significant. What does matter, however, is the use of interrupt-driven versus polling-based communication in the guest environment. Similarly to what was observed in the point-to-point benchmarks, eliminating network interrupts by using the polling-based accelerated portals network stack results in near native performance. As noted previously, more recent AMD SVM implementations support selective interrupt exiting, which would make the virtualized interrupt-driven performance identical to the native or virtualized accelerated portals numbers. Still, even with this limitation, virtualized interrupt-driven communication is quite fast in absolute terms, with a 6240 node barrier or all-reduce taking less than 275 μs to perform.

The Alltoall operation is interesting because the size of the messages exchanged between nodes increases with node count. This causes all of the configurations to converge at high node counts, since the operation becomes bandwidth limited, and the cost of interrupt virtualization is amortized.

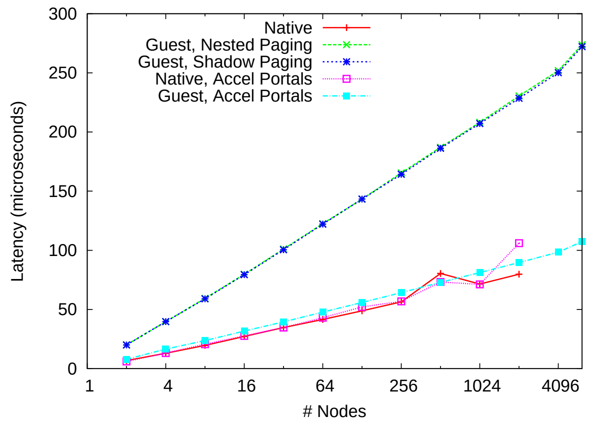


Figure 2. MPI barrier scaling microbenchmark results measuring the latency of a full barrier.

3.4 HPCCG application

HPCCG [12] is a simple conjugate gradient solver that is intended to mimic the characteristics of a broad class of HPC applications while at the same time is simple to understand and run. A large portion of its runtime is spent performing sparse matrix-vector multiplies, a memory bandwidth intensive operation.

HPCCG was used in weak-scaling mode with a “100x100x100” sub-problem on each node, using approximately 380 MB of memory per node. This configuration is representative of typical usage, and results in relatively few and relatively large messages being communicated between neighboring nodes. Every iteration of the CG algorithm performs an 8-byte Allreduce, and there are 149 iterations during the test problem’s approximately 30 second runtime.

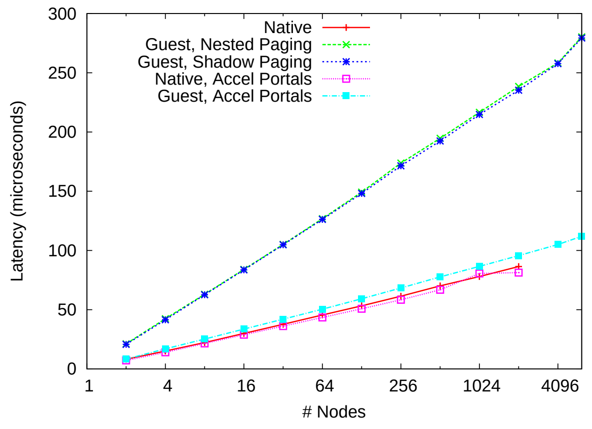


Figure 3. MPI all-reduce scaling microbenchmark results measuring the latency of a 16 byte all-reduce operation.

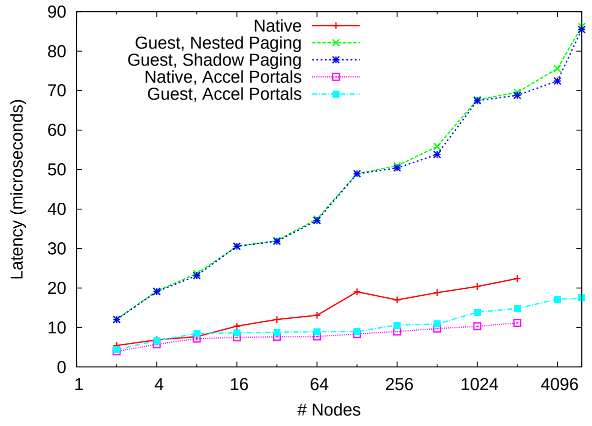


Figure 4. MPI broadcast scaling microbenchmark results measuring the latency of a broadcast of 16 bytes.

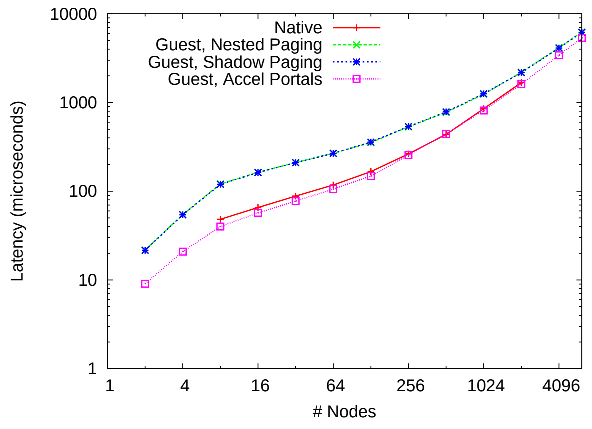


Figure 5. MPI all-to-all scaling microbenchmark results measuring the latency of a 16 byte all-to-all operation.

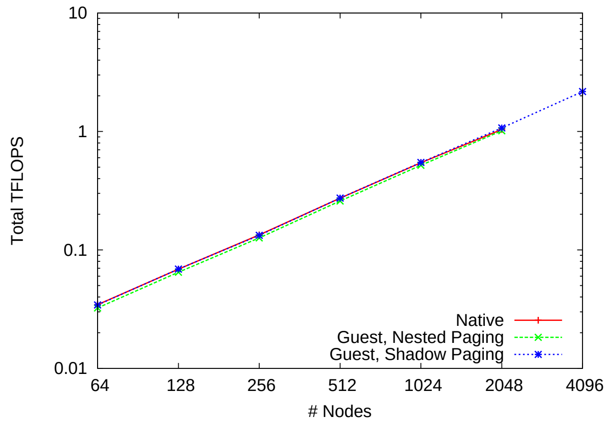


Figure 6. HPCCG application benchmark performance. Weak scaling is measured. Virtualized performance is within 5% of native.

The portion of runtime consumed by communication is reported by the benchmark to be less than 5% in all cases. Interrupt-driven communication was used for this and other application benchmarks. Recall that the microbenchmarks show virtualized interrupt-driven communication is the slower of the two options we considered.

As shown in Figure 6, HPCCG scales extremely well in both guest and native environments. Performance with shadow paging is essentially identical to native performance, while performance with nested paging is 2.5% worse at 2048 nodes.

3.5 CTH application

CTH [8] is a multi-material, large deformation, strong shock wave, solid mechanics code used for studying armor/anti-armor interactions, warhead design, high explosive initiation physics, and weapons safety issues. A shaped charge test problem was used to perform a weak scaling study in both native and guest environments. As reported in [9], which used the same test problem, at 512 nodes approximately 40% of the application's runtime is due to MPI communication, 30% of which is due to `MPI_Allreduce` operations with an average size of 32 bytes. The application performs significant point-to-point communication with nearest neighbors using large messages.

Figure 7 shows the results of the scaling study for native and guest environments. At 2048 nodes, the guest environment with shadow paging is 3% slower than native, while the nested paging configuration is 5.5% slower. Since network performance is virtually identical with either shadow or nested paging, the performance advantage of shadow paging is likely due to the faster TLB miss processing that it provides.

3.6 SAGE application

SAGE (SAIC's Adaptive Grid Eulerian hydrocode) is a multidimensional hydrodynamics code with adaptive mesh refinement [18]. The timing_c input deck was used to perform a weak scaling study. As reported in [9], which used the same test problem, at 512 nodes approximately 45% of the application's runtime is due to MPI communication, of which roughly 50% is due to `MPI_Allreduce` operations with an average size of 8 bytes.

Figure 8 shows the results of executing the scaling study in the native and virtualized environments. At 2048 nodes, shadow paging is 2.4% slower compared to native while nested paging is 3.5% slower. As with CTH, the slightly better performance of shadow paging is due to its faster TLB miss processing.

4. Passthrough I/O

One of the principle goals in designing Palacios was to allow a large amount of configurability in order to target multiple diverse

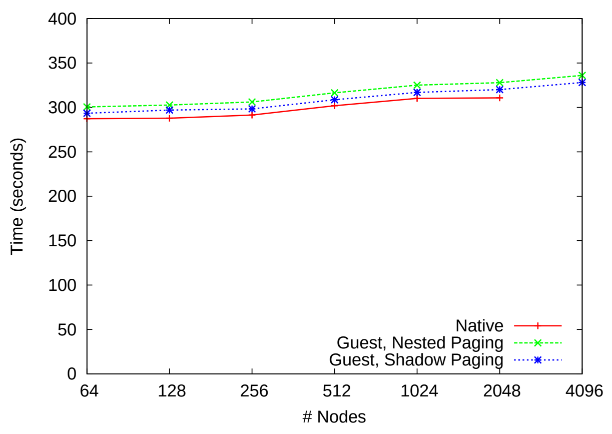


Figure 7. CTH application benchmark performance. Weak scaling is measured. Virtualized performance is within 5% of native.

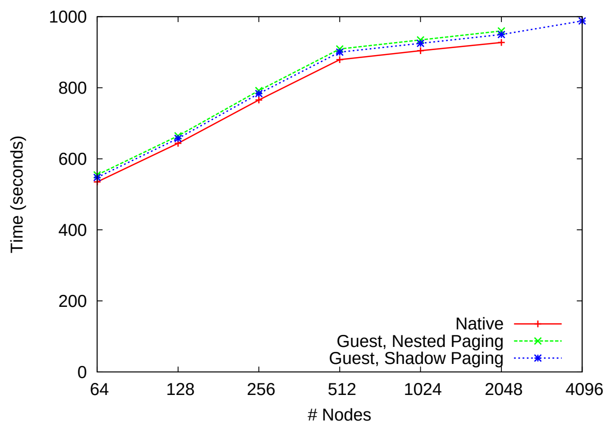


Figure 8. Sage application benchmark performance. Weak scaling is measured. Virtualized performance is within 5% of native.

environments. This allows us to use a number of configuration options specific to HPC environments to minimize virtualization overheads and maximize performance. The special HPC configuration of Palacios makes a number of fundamental choices in order to provide guest access to hardware devices with as little overhead as possible. These choices were reflected both in the architecture of Palacios as configured for HPC, as well as two assumptions about the environment Palacios executes in.

The first assumption we make for HPC environments is that only a single guest will be running on a node at any given time. Restricting each partition to run a single guest environment ensures that there is no resource contention between multiple VMs. This is the common case for large-scale supercomputers as each application requires dedicated access to the entirety of the system resources, and is also the common case for many smaller space-shared high performance systems. The restriction vastly simplifies device management because Palacios does not need to support sharing of physical devices between competing guests; Palacios can directly map an I/O device into a guest domain without having to manage the device itself.

The second assumption we make for HPC environments is that we can place considerable trust in the guest OS because HPC system operators typically have full control over the entire software stack. Under this assumption, the guest OS is unlikely to attempt to compromise the VMM intentionally, and may even be designed to help protect the VMM from any errors.

4.1 Passthrough I/O implementation

In Palacios, passthrough I/O is based on a virtualized PCI bus. The virtual bus is implemented as an emulation layer inside Palacios, and has the capability of providing access to both virtual as well as physical (passthrough) PCI devices. When a guest is configured to use a passthrough device directly, Palacios scans the physical PCI bus searching for the appropriate device and then attaches a virtual instance of that device to the virtual PCI bus. Any changes that a guest makes to the device's configuration space are applied only to the virtualized version. These changes are exposed to the physical device via reconfigurations of the guest environment to map the virtual configuration space onto the physical one.

As an example, consider a PCI Base Address Register (BAR) that contains a memory region that is used for memory-mapped access to the device. Whenever a guest tries to change this setting by overwriting the BAR's contents, instead of updating the physical device's BAR, Palacios updates the virtual device's BAR and reconfigures the guest's physical memory layout so that the relevant guest physical memory addresses are redirected to the host physical memory addresses mapped by the real BAR register. In this way, Palacios virtualizes configuration operations but not the actual data transfer.

Most devices do not rely on the PCI BAR registers to define DMA regions for I/O. Instead the BAR registers typically point to additional, non-standard configuration spaces, that themselves contain locations of DMA descriptors. Palacios makes no attempt to virtualize these regions, and instead relies on the guest OS to supply valid DMA addresses for its own physical address space. While this requires that Palacios trust the guest OS to use correct DMA addresses as they appear in the host, it is designed such that there is a a high assurance that the DMA addresses used by the guest are valid.

The key design choice that provides high assurance of secure DMA address translation from the guest physical addresses to the host physical addresses is the shape of the guest's physical address space. A Palacios guest is initially configured with a physically contiguous block of memory that maps into the contiguous portion of the guest's physical address space that contains memory. This allows the guest to compute a host physical address from a guest physical address by simply adding an offset value. This means that a passthrough DMA address can be immediately calculated as long as the guest knows what offset the memory in its physical address space begins at. Furthermore, the guest can know definitively if the address is within the bounds of its memory by checking that it does not exceed the range of guest physical addresses that contain memory, information that is readily available to the guest via the e820 map and other standard mechanisms. Because guest physical to host physical address translation for actual physical memory is so simple, DMA addresses can be calculated and used with a high degree of certainty that they are correct and will not compromise the host or VMM.

It is also important to point out that as long as the guest uses physical addresses valid with respect to its memory map, it cannot affect the VMM or other passthrough or virtual devices with a DMA request on a passthrough device.

To allow the guest to determine when a DMA address needs to be translated (by offsetting) for passthrough access, Palacios uses a shared memory region to advertise which PCI devices are in fact configured as passthrough. Each PCI bus location tuple (bus ID, device ID, and function number) is combined to form an index into a bitmap. If a device is configured as passthrough the bit at its given index will be set by the VMM and read by the guest OS. This bitmap allows the guest OS to selectively offset DMA addresses, allowing for compatibility with both passthrough devices (which require offsetting) and virtual devices (which do not). Furthermore, when the guest is run without the VMM in place, this mechanism naturally turns off offsetting for all devices.

Comparison with other approaches to high performance virtualized I/O: Due to both the increased trust and control over the guest environments as well as the simplified mechanism for DMA address translation, Palacios can rely on the guest to correctly interact with the passthrough devices. The passthrough I/O technique allows direct interaction with hardware devices with as little overhead as possible. In contrast, other approaches designed to provide passthrough I/O access must add additional overhead. For example, VMM-Bypass [21], as designed for the Xen Hypervisor, does not provide the same guarantees in terms of address space contiguity. Furthermore, its usage model assumes that the guest environments are not fully trusted entities. The result is that the implementation complexity is much higher for VMM-Bypass, and further overheads are added due to the need for the VMM to validate the device configurations. Furthermore, this technique is highly device specific (specifically Infiniband) whereas our passthrough architecture is capable of working with any unmodified PCI device driver.

Self-Virtualization [26] is a technique to allow device sharing without the need for a separate virtual driver domain. While self virtualization permits direct guest interaction with hardware devices, it uses a simplified virtual interface which limits the usable capabilities of the device. It also requires specially architected hardware, while our passthrough implementation supports any existing PCI device.

Finally, recent work on assuring device driver safety in traditional operating systems [29] could also be used to supplement passthrough device virtualization. In particular, these techniques could be used to validate safety-critical guest device manipulations in virtual machines. This would enable the high performance of passthrough I/O while providing additional guest isolation in environments that where guest OSes are less trusted than in HPC environments.

4.2 Current implementations

We have currently implemented passthrough I/O for both a collection of HPC OSes, such as Catamount and Kitten, as well as for commodity Linux kernels. The Catamount OS specifically targets the Cray SeaStar as its only supported I/O device, so Catamount did not require a general passthrough framework. However, Kitten and Linux are designed for more diverse environments so we have implemented the full passthrough architecture in each of them. In each case, the implementation is approximately 300 lines of C and assembler built on the SymSpy guest implementation (Section 7). The actual DMA address offsetting and bounds checking implementation is about 20 lines of C.

Both Kitten and Linux include the concept of a DMA address space that is conceptually separate from the address space of core memory. This allows a large degree of compatibility between different architectures that might implement a separate DMA address space. The environment exposed by Palacios is such an architecture. Every time a device driver intends to perform a DMA operation it must first transform a memory address into a DMA address via a DMA mapping service. Our guest versions of both Linux and Kitten include a modified mapping service that selectively adds the address offset to each DMA address if the device requesting the DMA translation is configured for passthrough. Our modifications also sanity check the calculated DMA address, thus protecting the VMM from any malformed DMA operations. These modifications are small, easy to understand, and all-encompassing, meaning that the VMM can have a high degree of confidence that even a complicated OS such as Linux will not compromise the VMM via malformed DMA operations.

4.3 Infiniband passthrough

To verify that Palacios's passthrough I/O approach also resulted in low-overhead communication on commodity NICs in addition to specialized hardware like the Cray SeaStar, we examined its performance on a small Linux cluster system built around the commodity Infiniband network interface. Specifically, we examined the performance both a low-level Infiniband communication microbenchmark (the OpenFabrics `ibv_rc_pingpong` test) and the HPCCG benchmark described earlier. Tests were run on a 4-node 2.4GHz AMD Barcelona cluster communicating over 64-bit PCI Express Mellanox MLX4 cards configured for passthrough in Linux. For ping-pong tests, the client system which performed the timings ran native Fedora 11 with Linux kernel 2.6.30, and the client machine ran a diskless Linux BusyBox image that also used Linux kernel 2.6.30 with symbiotic extensions either natively or virtualized in Palacios. For HPCCG tests, all nodes ran the Linux BusyBox image, and timings were taken using the underlying hardware cycle counter to guarantee accuracy.

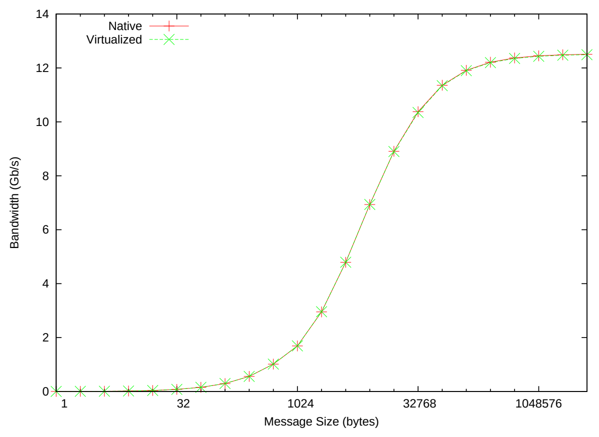


Figure 9. Infiniband bandwidth at message sizes from 1 byte to 4 megabytes averaged over 10000 iteration per sample. 1-byte round-trip latency for both native and virtualized environments was identical at 6.46 μsec, with peak bandwidth for 4 MB messages at 12.49 Gb/s on Linux virtualized with Palacios compared to 12.51 Gb/s for native Linux.

As Figure 9 shows, Palacios's pass-through virtualization imposes almost no overhead on Infiniband message passing. In particular, Palacios's passthrough PCI support enables virtualized Linux to almost perfectly match the bandwidth of native Linux on Infiniband, and because Infiniband does not use interrupts for high-speed message passing with reliable-connected channels, the 1-byte message latencies with and without virtualization are identical. Similarly, HPCCG ran an average of only 4% slower (43.1 seconds versus 41.4 seconds averaged over 5 runs) when virtualized using passthrough I/O and nested paging.

4.4 Future extensions

Future advances in hardware virtualization support may obviate the need for the passthrough techniques described above. For example, AMD's IOMMU adds hardware support for guest DMA translations. However, we should note that our approach includes a very minimal amount of overhead and it is not clear that hardware techniques will necessarily perform better. An IOMMU would introduce additional performance overhead in the form of page table lookups, something which our approach completely avoids. As we will show in the next section and as others have demonstrated [1], software approaches can often operate with demonstrably less overhead than hardware approaches.

5. Workload-sensitive paging mechanisms

In our scaling evaluations, we focused on the two standard techniques for virtualizing the paging hardware: shadow paging and nested paging as described in Section 3.2. These results demonstrate that while memory virtualization can scale, making it do so is non-trivial; we discuss the implications of these results in this section. Based on these results, we also present the results of several additional experiments that examine how more sophisticated architectural and VMM support for memory virtualization impacts HPC benchmark performance.

5.1 Scaling analysis

The basic scaling results presented earlier in Section 3 demonstrate that the best performing technique is dependent on the application workload as well as the architecture of the guest OS. As an example, Catamount performs a minimal number of page table operations, and never fully flushes the TLB or switches between different page tables. This means that very few operations are required to emulate the guest page tables with shadow paging. Because the overhead of shadow paging is so small, shadow paging performs better than nested paging due to the better use of the hardware TLB. In contrast, Compute Node Linux (CNL), another HPC OS, uses multiple sets of page tables to handle multitasking and so frequently flushes the TLB. For this OS, there is a great deal more overhead in emulating the page table operations and any improvement in TLB performance is masked by the frequent flush operations. As a result, nested paging is the superior choice in this case.

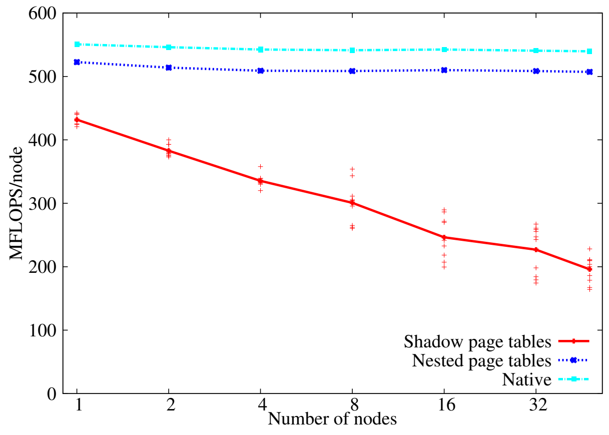


Figure 10. Strong scaling of HPCCG running on CNL. Nested paging is preferable, and the overhead of shadow paging compounds as we scale up. This is due to the relatively high context switch rate in CNL.

As these results demonstrate, behavior of the guest OS and applications have a critical impact on the performance of the virtualized paging implementation. We have found this to be true in the broader server consolidation context [5] as well as the HPC context we discuss here. Figures 10 and 11 (previously published elsewhere [20]) illustrate this point for HPC. Figure 10 shows the results of the HPCCG benchmark being run with a CNL guest environment as we scale from 1 to 48 nodes of a Cray XT. As the results

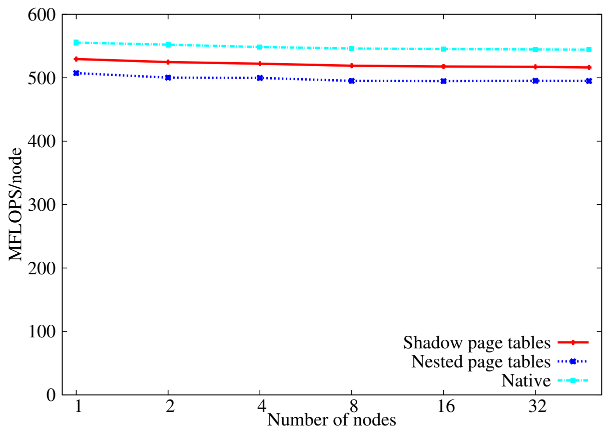


Figure 11. Strong scaling of HPCCG running on Catamount. Here shadow paging is preferable. This is due to the relatively low context switch rate in Catamount revealing shadow paging's better TLB behavior.

show, the overhead introduced with shadow paging is large enough to dramatically degrade scalability, while the nested paging configuration is able to still perform well as it scales up. Figure 11 shows the same benchmark run on the Catamount guest OS. Here, the situation is reversed. Shadow paging clearly performs better than nested paging due to the improved TLB behavior and lower overhead from page table manipulations.

5.2 Memory virtualization optimizations

In light of these results, we also examined the performance of key optimizations to nested and shadow paging. In particular, we studied the aggressive use of large pages in nested page tables and two different optimizations to shadow paging. For these evaluations, we used HPC Challenge 1.3 [22, 27], an HPC-oriented benchmark suite that tests system elements critical to HPC application performance.

5.2.1 2 MB nested page tables

Aggressively using large pages in nested page tables is an optimization that could dramatically improve the performance of nested paging on applications and benchmarks that are TLB-intensive. For example, using 2MB nested page tables on the x86_64 architecture reduces the length of full page table walks from 24 steps to 14 steps. Note that using large pages in shadow paging is also possible, but, like using large pages in a traditional operating system, can be quite challenging as the guest may make permission and mapping requests at smaller page granularities that could require the VMM to split large pages or merge smaller pages.

To evaluate the potential impact of using 2MB nested page tables on HPC applications, we implemented support for large-page nested page tables in Palacios. We then evaluated its performance when running the Catamount guest operating system, the guest on which nested paging performed comparatively worse in our scaling study. Because Catamount can make aggressive use of large pages, this also allowed us to study the impact of these different paging choices on guests that used 4KB pages versus guests like Catamount that make aggressive use of 2MB pages.

Our evaluation focused on the HPL, STREAM Triad, and RandomAccess benchmarks from HPC Challenge. HPL is a compute-intensive HPC benchmark commonly used to benchmark HPC systems [28], STREAM Triad is a memory bandwidth intensive benchmark, and RandomAccess is a simulated large-scale data analytics benchmark that randomly updates an array approximately the size of physical memory, resulting in a very high TLB miss rate.

Figure 12 shows the relative performance of nested paging with different nested and main page table sizes, with shadow paging and native paging numbers included for comparison. HPL performance shows little variability due to its regular memory access patterns in these tests, though 2MB nested page tables does improve nested paging performance to essentially native levels. Using large pages with nested paging makes a dramatic difference on the TLB miss-intensive RandomAccess benchmark. In particular, using large pages in the nested page tables reduces the penalty of nested paging from 64% to 31% for guests that use 4KB pages and from 68% to 19% for guests that use 2MB pages.

The RandomAccess results also show that nested paging is better able to support guests that aggressively use large pages compared to shadow paging. While nested paging performance is 19% worse than native, it is significantly better than shadow paging performance, which is limited by the performance of its underlying 4KB page-based page tables. With guests that use only 4KB pages, however, shadow paging achieves native-level performance while nested paging with 2MB pages is 30% slower than native.

5.2.2 Shadow paging optimizations

With stable guest page tables, shadow paging has the benefit of having shorter page walks on a TLB miss than nested paging. However, context switches in the guest ameliorate this advantage in a basic shadow paging implementation because they force a flush of the "virtual TLB" (the shadow page tables). Subsequent to this, a stream of exits occurs as page faults are used to rebuild the shadow page tables. We have considered two techniques in Palacios to reduce this cost: shadow page table caching and shadow page table prefetching.

In contrast to a basic shadow paging implementation, both caching and prefetching introduce a new overhead as they must monitor the guest page tables for changes so that the corresponding cached or prefetched shadow page table entries may be updated or flushed. While conceptually simple, preserving x86_64 page table consistency requirements is quite challenging, leading to considerably higher software complexity in the VMM. In particular, because portions of page tables may be shared across address spaces, page table updates in one address space may affect page tables mapping additional address spaces. Furthermore, a physical page containing a page table is allowed to appear at multiple levels in a page table hierarchy.

In a shadow page table caching implementation, when the guest switches from one context to another and the VMM already has the corresponding shadow context in its cache, the cost of the context switch is dramatically reduced. In the best case, where the guest makes frequent context switches but rarely edits its page tables, a context switch requires only that the VMM load the page table base register and continue. In the worse case, the guest frequently edits its page tables, but rarely performs context switches.

In a shadow page table prefetching implementation, a guest context switch acts as in a basic implementation, flushing the shadow page tables. However, on a single page fault, multiple guest page table entries are visited and reflected into the shadow page table. Our implementation prefetches an entire page's worth of entries on each page fault, so in the best case, where the guest makes frequent context switches but rarely edits its page tables, the overhead of a context switch is reduced by a factor of 512 (PAE) or 1024 (non-PAE). In the worst case, the guest frequently edits page tables but rarely performs context switches. In contrast to shadow page table caching, shadow page table prefetching requires no more space than basic shadow paging.

Approach	Run-time (s)
Native	15.7
Shadow	798.9
Shadow+Prefetching	1305.6
Shadow+Caching	32.9
Nested (4KB pages)	24.7

Figure 13. Performance of HPC Challenge benchmark suite in Palacios for different memory virtualization approaches.

To further evaluate the potential benefits of caching and prefetching, we studied their overall performance on the HPC Challenge benchmark suite. It is important to point out that this evaluation involved configuring HPC Challenge to emphasize the effects of address translation performance instead of to maximize overall benchmark performance. In particular, we configured the benchmarks to run with four processes per core, resulting in a significant context switch rate. The combination of context switches and memory reference behavior in the benchmark processes interacts differently with the different paging approaches, and represents a particular challenge for shadow paging. HPC Challenge includes seven benchmarks, with two, Random Access and HPL, accounting for almost all the variation among the different paging approaches. The experiments were run on a Dell PowerEdge SC1450 system with an AMD Opteron 2350 "Barcelona" processor with 2GB of RAM. The guest operating system was running Puppy Linux 3.01 (32-bit Linux kernel 2.6.18). The study is further described elsewhere [5].

Figure 13 shows the results. While shadow paging with prefetching is not an effective optimization for this workload, shadow paging with caching brings performance much closer to nested paging performance, although there remains a gap. We also evaluated shadow paging with caching using the more mature implementation in the KVM VMM. There, a run time of 24.3 s was measured, right on par with nested paging. Note that performance remains distant from native due to the Random Access benchmark.

5.3 Summary

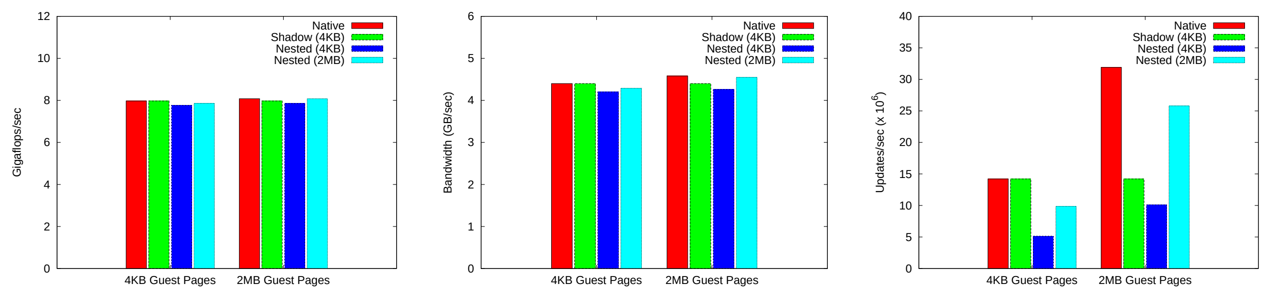
These results show that the choice of virtual paging techniques is critically important to ensuring scalable performance in HPC environments and that the best technique varies across OSes, hardware, and applications. This suggests that an HPC VMM should provide a mechanism for specifying the initial paging technique as well as for switching between techniques during execution. Furthermore, an HPC VMM should provide a range of paging techniques to choose from. Palacios supports this through a modular architecture for paging techniques. New techniques can be created and linked into the VMM in a straightforward manner, with each guest being able to dynamically select among all the available techniques at runtime. We are also currently exploring adaptive runtime modification of guest paging virtualization strategies.

6. Controlled preemption

We now consider the third technique we found to be essential to low overhead virtualization at scale, controlled preemption.

6.1 Overview

It is well understood that background noise can have a serious performance impact on large scale parallel applications. This has led to much work in designing OSes such that the amount of noise they inject into the system is minimized and that the impact of necessary noise on application performance is minimized. Palacios is designed to support such efforts by minimizing the amount of overhead due to virtualization, placing necessary overheads and



(a) HPL (b) STREAM Triad (c) RandomAccess

Figure 12. Performance of 2MB Nested Paging running HPC Challenge HPL, STREAM Triad, and RandomAccess benchmarks on a Cray XT4 node with a Catamount guest OS.

work into deterministic points in time in an effort to minimize the amount of noise added to the system by virtualization, and allowing system deployers control over when guests are preempted.

Palacios runs as a non-preemptable kernel thread in Kitten. Only interrupts and explicit yields by Palacios can change control flow. Palacios controls the global interrupt flag and guest interrupt exiting and uses this control to allow interrupts to happen only at specific points during exit handling. This allows Palacios to provide well-controlled availability of CPU resources to the guest. Background processes and deferred work are only allowed to proceed when their performance impact will be negligible.

When a guest is configured it is allowed to specify its execution quantum which determines the frequency at which it will yield the CPU to the Kitten scheduler. It is important to note that the quantum configured by Palacios is separate from the scheduling quantum used by Kitten for task scheduling. This separation allows each guest to override the host OS scheduler in order to prevent the host OS from introducing additional OS noise. Furthermore this quantum can be overridden at runtime such that a guest can specify critical sections where Palacios should not under any circumstances yield the CPU to another host process.

As a result of this, Palacios adds minimal additional noise to guests and applications that run on top of it, particularly compared to other VMMs not designed for use in HPC systems. To quantify this, we compiled the selfish noise measurement benchmark frequently used to measure OS interference in HPC systems into a minimal kernel with interrupts disabled to measure the amount of interference the VMM presents to guest operating systems. This benchmark spins the CPU while continually reading the timestamp counter, which is configured by the VMM to passthrough to hardware. We ran this guest kernel on a Cray XT4 node on both a minimal Linux/KVM VMM (`init`, `/bin/sh`, and `/sbin/sshd` were the only user-level processes running) and on Palacios.

In these tests, the Linux/KVM virtual machine used almost exactly 10 times as much CPU for management overhead as Palacios (0.22% for Linux/KVM versus 0.022% for Palacios). This is largely due to the increased timer and scheduler frequency of KVM's Linux host OS compared to the Kitten host OS, which can use a much larger scheduling quantum because of its focus on HPC systems. While these overheads both appear small in absolute terms, they are in addition to any overhead that the guest OS imposes on an application. Because past research has shown that even small asynchronous OS overheads can result in application slowdowns of orders of magnitude on large-scale systems [9, 25], minimizing these overheads is essential for virtualization to be viable in HPC systems.

6.2 Future extensions

An extant issue in HPC environments is the overhead induced via timer interrupts. An eventual goal of Kitten is to implement a system with no dependence on periodic interrupts, and instead rely entirely on on-demand one shot timers. However, periodic timers are occasionally necessary when running a guest environment with Palacios, in order to ensure that time advances in the guest OS. Because some guest OSes do require periodic timer interrupts at a specified frequency, the VMM needs to ensure that the interrupts can be delivered to the guest environment at the appropriate rate. We are developing a method in which the guest OS is capable of both enabling/disabling as well as altering the frequency of the host's periodic timer. This would allow a guest OS to specify its time sensitivity[2], which will allow Palacios and Kitten to adapt timer behavior to the current workload.

7. A symbiotic approach to virtualization

While our experiences have shown that it is indeed possible to virtualize large scale HPC systems with minimal overhead, we have found that doing so requires cooperation between the guest and VMM. Each of the three techniques we have described (Sections 4–6) relies on communication and trust across the VMM/guest interface for the mutual benefit of both entities. In other words, the relationship between the VMM and the guest is *symbiotic*. We have been working to generalize the interfaces involved in our techniques into a general purpose *symbiotic interface* that provides VMM$\leftrightarrow$guest information flow that can be leveraged in these and future techniques.

Our symbiotic interface allows for both passive, asynchronous and active, synchronous communication between guest and VMM. The symbiotic interface is *optional* for the guest, and a guest which does use it can also run on non-symbiotic VMMs or raw hardware without any changes. We focus here on the passive interface, *SymSpy*; the active interface, *SymCall*, is described elsewhere [19].

SymSpy builds on the widely used technique of a shared memory region that is accessible by both the VMM and guest. This shared memory is used by both the VMM and guest to expose semantically rich state information to each other, as well as to provide asynchronous communication channels. The data contained in the memory region is well structured and semantically rich, allowing it to be used for most general purpose cross layer communication. Each of the three techniques we have given in this paper are implemented on top of SymSpy. We have implemented SymSpy support in Catamount, Kitten, and in non-HPC guest OSes such as Linux.

[2] You can think of this as being loosely correlated to the guest's timer frequency setting

SymSpy is designed to be enabled and configured at run time without requiring any major structural changes to the guest OS. The discovery protocol is implemented using existing hardware features, such as CPUID values and Model Specific Registers (MSRs). When run on a symbiotic VMM, CPUID and MSR access is trapped and emulated, allowing the VMM to provide extended results. Through this, a guest can detect a SymSpy interface at boot time and selectively enable specific symbiotic features that it supports. Due to this hardware-like model, the discovery protocol will also work correctly if no symbiotic VMM is being used; the guest will simply not find a symbiotic interface.

After the guest has detected the presence of SymSpy it chooses an available guest physical memory address that is not currently in use for the shared memory region and notifies the VMM of this address through a write to a virtualized MSR. The precise semantics and layout of the data on the shared memory region depends on the symbiotic services that are discovered to be jointly available in the guest and the VMM. The structured data types and layout are enumerated during discovery. During normal operation, the guest can read and write this shared memory without causing an exit. The VMM can also directly access the page during its execution.

8. Conclusion

Our primary contribution has been to demonstrate that it is possible to virtualize the largest parallel supercomputers in the world[3] at very large scales with minimal performance overheads. In particular, tightly-coupled, communication-intensive applications running on specialized lightweight OSes that provide maximum hardware capabilities to them can run in a virtualized environment with $\leq 5\%$ performance overhead at scales in excess of 4096 nodes. In addition, other HPC applications and guest OSes can be supported with minimal overhead given appropriate hardware support.

These results suggest that HPC machines can reap the many benefits of virtualization that have been articulated before (e.g., [10, 13]). Another benefit that other researchers have noted [4] but that has not been widely discussed is that scalable HPC virtualization also opens the range of applications of the machines by making it possible to use commodity OSes on them in capacity modes when they are not needed for capability purposes.

We believe our results represent the largest scale study of HPC virtualization by at least two orders of magnitude, and we have described how such performance is possible. Scalable high performance rests on passthrough I/O, workload sensitive selection of paging mechanisms, and carefully controlled preemption. These techniques are made possible via a symbiotic interface between the VMM and the guest, an interface we have generalized with SymSpy. We are now working to further generalize this and other symbiotic interfaces, and apply them to further enhance virtualized performance of supercomputers, multicore nodes, and other platforms. Our techniques are publicly available from `v3vee.org` as parts of Palacios and Kitten.

[3] The Red Storm machine we used is in the top-20.

References

[1] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2006.

[2] S. R. Alam, J. A. Kuehn, R. F. Barrett, J. M. Larkin, M. R. Fahey, R. Sankaran, and P. H. Worley. Cray XT4: an early evaluation for petascale scientific simulation. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-764-3. doi: http://doi.acm.org/10.1145/1362622.1362675.

[3] AMD Corporation. AMD64 virtualization codenamed "Pacifica" technology: Secure Virtual Machine Architecture reference manual, May 2005.

[4] J. Appavoo, V. Uhlig, and A. Waterland. Project kittyhawk: building a global-scale computer: Blue gene/p as a generic computing platform. *SIGOPS Oper. Syst. Rev.*, 42:77–84, January 2008. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/1341312.1341326. URL `http://doi.acm.org/10.1145/1341312.1341326`.

[5] C. Bae, J. Lange, and P. Dinda. Comparing approaches to virtualized page translation in modern VMMs. Technical Report NWU-EECS-10-07, Department of Electrical Engineering and Computer Science, Northwestern University, April 2010.

[6] R. Bhargava, B. Serebrin, F. Spanini, and S. Manne. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2008.

[7] R. Brightwell, T. Hudson, K. T. Pedretti, and K. D. Underwood. SeaStar Interconnect: Balanced bandwidth for scalable performance. *IEEE Micro*, 26(3):41–57, May/June 2006.

[8] J. E.S. Hertel, R. Bell, M. Elrick, A. Farnsworth, G. Kerley, J. McGlaun, S. Petney, S. Silling, P. Taylor, and L. Yarrington. CTH: A Software Family for Multi-Dimensional Shock Physics Analysis. In *19th International Symposium on Shock Waves, held at Marseille, France*, pages 377–382, July 1993.

[9] K. B. Ferreira, R. Brightwell, and P. G. Bridges. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, November 2008.

[10] R. Figueiredo, P. A. Dinda, and J. Fortes. A case for grid computing on virtual machines. In *23rd IEEE Conference on Distributed Computing (ICDCS 2003*, pages 550–559, May 2003.

[11] A. Gavrilovska, S. Kumar, H. Raj, K. Schwan, V. Gupta, R. Nathuji, R. Niranjan, A. Ranadive, and P. Saraiya. High performance hypervisor architectures: Virtualization in HPC systems. In *1st Workshop on System-level Virtualization for High Performance Computing (HPCVirt)*, 2007.

[12] M. Heroux. HPCCG MicroApp. `https://software.sandia.gov/mantevo/downloads/HPCCG-0.5.tar.gz`, July 2007.

[13] W. Huang, J. Liu, B. Abali, and D. K. Panda. A case for high performance computing with virtual machines. In *20th Annual International Conference on Supercomputing (ICS)*, pages 125–134, 2006.

[14] Intel Corporation. Intel virtualization technology specification for the IA-32 Intel architecture, April 2005.

[15] Intel GmbH. Intel MPI benchmarks: Users guide and methodology description, 2004.

[16] L. Kaplan. Cray CNL. In *FastOS PI Meeting and Workshop*, June 2007. URL `http://www.cs.unm.edu/~fastos/07meeting/CNL_FASTOS.pdf`.

[17] S. Kelly and R. Brightwell. Software architecture of the lightweight kernel, Catamount. In *2005 Cray Users' Group Annual Technical Conference*. Cray Users' Group, May 2005.

[18] D. Kerbyson, H. Alme, A. Hoisie, F. Petrini, H. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of ACM/IEEE Supercomputing*, November 2001.

[19] J. Lange and P. Dinda. SymCall: Symbiotic virtualization through VMM-to-guest upcalls. In *Proceedings of the 2011 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2011)*, Newport Beach, CA, March 2011.

[20] 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 *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010)*, April 2010.

[21] J. Liu, W. Huang, B. Abali, and D. Panda. High Performance VMM-Bypass I/O in Virtual Machines. In *Proceedings of the USENIX Annual Technical Conference*, May 2006.

[22] P. Luszczek, J. Dongarra, and J. Kepner. Design and implementation of the HPCC benchmark suite. *CT Watch Quarterly*, 2(4A), Nov. 2006.

[23] M. F. Mergen, V. Uhlig, O. Krieger, and J. Xenidis. Virtualization for high-performance computing. *Operating Systems Review*, 40(2):8–11, 2006.

[24] H. Nishimura, N. Maruyama, and S. Matsuoka. Virtual clusters on the fly - fast, scalable, and flexible installation. In *7th IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 549–556, 2007.

[25] F. Petrini, D. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proceedings of SC'03*, 2003.

[26] H. Raj and K. Schwan. High performance and scalable I/O virtualization via self-virtualized devices. In *16th IEEE International Symposium on High Performance Distributed Computing*, July 2007.

[27] S. Song, R. Ge, X. Feng, and K. W. Cameron. Energy profiling and analysis of the HPC Challenge benchmarks. *International Journal of High Performance Computing Applications*, Vol. 23, No. 3:265–276, 2009.

[28] Top500. Top 500 Supercomputing Sites. URL `http://www.top500.org/`.

[29] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device driver safety through a reference validation mechanism. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI'08)*, 2008.