SANDIA REPORT

SAND2010-6232
Unlimited Release
Printed September, 2010

LDRD Final Report: A Lightweight Operating System for Multi-core Capability Class Supercomputers

Kevin T. Pedretti

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2010-6232
Unlimited Release
Printed September, 2010

LDRD Final Report: A Lightweight Operating System for Multi-core Capability Class Supercomputers

Kevin T. Pedretti (PI)
Scalable System Software Department (1423)
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1319
ktpedre@sandia.gov

Michael Levenhagen (1422)
Kurt Ferreira (1423)
Ron Brightwell (1423)
Suzanne Kelly (PM, 1423)
Patrick Bridges (UNM, sabbatical at SNL)
Trammell Hudson (OS Research, SNL contractor)

Abstract

The two primary objectives of this LDRD project were to create a lightweight kernel (LWK) operating system(OS) designed to take maximum advantage of multi-core processors, and to leverage the virtualization capabilities in modern multi-core processors to create a more flexible and adaptable LWK environment. The most significant technical accomplishments of this project were the development of the Kitten lightweight kernel, the co-development of the SMARTMAP intra-node memory mapping technique, and the development and demonstration of a scalable virtualization environment for HPC. Each of these topics is presented in this report by the inclusion of a published or submitted research paper. The results of this project are being leveraged by several ongoing and new research projects.

Acknowledgment

We wish to acknowledge collaborators Peter Dinda (Northwestern University), John Lange (Univ. Pittsburgh), and Patrick Bridges (University of New Mexico) for their leadership in developing the Palacios virtual machine monitor, contributions to the Kitten lightweight kernel, and many helpful discussions.

We also wish to acknowledge collaborators Fredrik Unger (NEC HPC Europe), Erich Focht (NEC HPC Europe), and Jaka Močnik (XLAB Research) for their contributions to the Kitten lightweight kernel, particularly their substantial contributions to the Infiniband networking stack.

We thank the many people who have influenced the "lightweight" design philosophy for scalable, HPC-focused operating systems, and the developers of previous lightweight kernels (SUNMOS, Cougar, Puma, Catamount). This work would not have been possible without their input.

Likewise, we thank the Linux kernel community for their continued development of the Linux kernel. The Kitten lightweight kernel makes significant use of Linux kernel code, such as the x86 bootstrap code and user-space ABI.

We thank Rolf Riesen for contributing the LaTeX template for this report.

Contents

Summary **11**

1 SMARTMAP: Operating System Support for Efficient Data Sharing Among Processes on a Multi-Core Processor **13**

1.1 Introduction 14

1.2 Background 15

1.2.1 Intra-Node MPI 16

1.2.2 Intra-Node Communication on the Cray XT 17

1.3 SMARTMAP Implementation 18

1.3.1 Catamount 18

1.3.2 Limitations 20

1.4 Using SMARTMAP 20

1.4.1 Cray SHMEM 20

1.4.2 MPI Point-to-Point Communication 21

1.4.3 MPI Collective Communication 23

1.5 Performance Evaluation 24

1.5.1 Test Environment 24

1.5.2 SHMEM 24

1.5.3 MPI Point-to-Point 25

1.5.4 MPI Collectives 28

1.6 Conclusion 30

1.7 Future Work 30

1.8 Acknowledgments 31

2 Palacios and Kitten: New High Performance Operating Systems For Scalable Virtualized and Native Supercomputing 33

2.1 Introduction 35

2.2 Motivation 36

2.3 Palacios 38

2.3.1 Architecture 38

2.3.2 Palacios as a HPC VMM 41

2.4 Kitten 42

2.4.1 Architecture 42

2.4.2 Memory Management 44

2.4.3 Task Scheduling 44

2.5 Integrating Palacios and Kitten 44

2.6 Performance 46

2.6.1 Testbed 46

2.6.2 Guests 47

2.6.3 HPCCG Benchmark Results 47

2.6.4 CTH Application Benchmark 49

2.6.5 Intel MPI Benchmarks 50

2.6.6 Infiniband microbenchmarks 51

2.6.7 Comparison with KVM 52

2.7 Future Work 53

2.8 Related Work 53

2.9 Conclusion 54

3 Minimal-overhead Virtualization of a Large Scale Supercomputer 55

3.1 Introduction 56

3.2 Virtualization system overview 57

3.2.1 Palacios 58
3.2.2 Kitten host OS 58
3.3 Virtualization at scale 59
3.3.1 Hardware platform 59
3.3.2 Software environment 59
3.3.3 MPI microbenchmarks 61
Point-to-point performance 61
Collective performance 62
3.3.4 HPCCG application 63
3.3.5 CTH application 64
3.3.6 SAGE application 65
3.4 Passthrough I/O 66
3.4.1 Passthrough I/O implementation 66
3.4.2 Current implementations 68
3.4.3 Infiniband passthrough 69
3.4.4 Future extensions 70
3.5 Workload-sensitive paging mechanisms 70
3.5.1 Scaling analysis 70
3.5.2 Memory virtualization optimizations 71
2 MB nested page tables 71
Shadow paging optimizations 73
3.5.3 Summary 74
3.6 Controlled preemption 75
3.6.1 Future extensions 76
3.7 A symbiotic approach to virtualization 76
3.8 Conclusion 77

References **78**

Appendix

A External Impact **85**

A.1 Peer-reviewed Published Papers 85

A.2 Submitted Papers 85

A.3 Other Publications 85

A.4 Invited Talks 86

A.5 Service to Professional Societies 86

A.6 Patent Applications 86

A.7 Awards 86

A.8 New Ideas for R&D 87

List of Figures

1.1 SMARTMAP kernel code 19

1.2 User function for converting a local virtual address to a remote virtual address 19

1.3 SHMEM Put Function 21

1.4 SHMEM Put Performance 25

1.5 IMB MPI Point-to-Point Results 26

1.6 MPI Message Rate 27

1.7 MPI Halo Exchange Performance 28

1.8 IMB MPI Collective Performance 29

2.1 Lines of code in Palacios and Kitten as measured with the SLOCCount tool. 39

2.2 Palacios architecture. 39

2.3 Kitten architecture. 43

2.4 HPCCG benchmark comparing scaling for virtualization with shadow paging, virtualization with nested paging, and no virtualization. Palacios/Kitten can provide scaling to 48 nodes with less than 5% performance degradation. 48

2.5 CTH application benchmark comparing scaling for virtualization with shadow paging, virtualization with nested paging, and no virtualization. Palacios/Kitten can provide scaling to 32 nodes with less than 5% performance degradation. 49

2.6 IMB PingPong Bandwidth in MB/sec as a function of message size 50

2.7 IMB Allreduce 16 byte latency in μsec as a function of nodes up to 48 nodes . 51

2.8 Bandwidth and latency of node-to-node Infiniband on Kitten, comparing native performance with guest performance. Linux numbers are provided for reference. 52

2.9 Comparison of Palacios to KVM for HPCCG benchmark. 52

3.1 MPI PingPong microbenchmark measuring (a) latency and (b) bandwidth . . 61

3.2 MPI barrier scaling microbenchmark results measuring the latency of a full barrier. 62

3.3 MPI all-reduce scaling microbenchmark results measuring the latency of a 16 byte all-reduce operation. 63

3.4 MPI all-to-all scaling microbenchmark results measuring the latency of a 16 byte all-to-all operation. 63

3.5 HPCCG application benchmark performance. Weak scaling is measured. Virtualized performance is within 5% of native. 64

3.6 CTH application benchmark performance. Weak scaling is measured. Virtualized performance is within 5% of native. 65

3.7 Sage application benchmark performance. Weak scaling is measured. Virtualized performance is within 5% of native. 65

3.8 Infiniband bandwidth at message sizes from 1 byte to 4 megabytes averaged over 10000 iteration per sample. 1-byte round-trip latency both native and virtualized was identical at6.46μsec, with peak bandwidth for 4MB messages at 12.49 Gb/s on Linux virtualized with Palacios compared to 12.51 Gb/s for native Linux. 69

3.9 Weak 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. 71

3.10 Weak 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. 72

3.11 Performance of 2MB Nested Paging running HPC Challenge HPL, STREAM Triad, and RandomAccess benchmarks on a Cray XT4 node with a Catamount guest OS. 73

3.12 Performance of HPC Challenge benchmark suite in Palacios for different memory virtualization approaches. 74

Summary

This report summarizes the R&D activities of the FY08–FY10 LDRD project "A Lightweight Operating System for Multi-core Capability Class Supercomputers," which was funded at a level of approximately 1.25 FTE per year. The two primary objectives of this project were to create a lightweight kernel (LWK) operating system (OS) designed to take maximum advantage of multi-core processors, and to leverage the virtualization capabilities in modern multi-core processors to create a more flexible and adaptable LWK environment. The most significant technical accomplishments of this project were:

1. Development of the Kitten LWK OS. Kitten is a modern, open-source LWK platform that supports many-core processors (100's of cores), advanced intra-node data movement (via SMARTMAP), current multi-threaded programming models (via Linux user-space compatibility), commodity HPC networking (Infiniband), and full-featured guest operating systems (via the Palacios virtual machine monitor). The Kitten LWK has been downloaded 100's of times, and there are several external researchers actively working to extend its capabilities. Several projects are leveraging the platform and we are positioning it as an ideal platform for conducting exascale supercomputer hardware/software co-design research. Kitten can be downloaded from `http://software.sandia.gov/trac/kitten`.

2. Co-development of SMARTMAP (Chapter 1). SMARTMAP is a virtual memory mapping technique that significantly reduces intra-node memory bandwidth requirements, which is a bottleneck on multi-core processors. For example, SMARTMAP allows intra-node MPI point-to-point messages to be copied directly to the target process's address space with a single copy, compared to multiple copies with traditional approaches. This reduces memory bandwidth requirements by a factor of two or more. SMARTMAP also allows many MPI collective operations to be implemented in-place without any data copies within a node, leading to substantial performance advantages with multi-core compute nodes. The SMARTMAP technique applies more broadly than MPI and can be used to optimize SHMEM and PGAS communication and data sharing. SMARTMAP was an important component of the 2009 "Catamount N-way Kernel" R&D100 award and we have applied for a patent.

3. Created a scalable virtualization environment for HPC (Chapters 2 and 3). We teamed with separately funded researchers at Northwestern University and the University of New Mexico to incorporate their embeddable virtual machine monitor (VMM), called Palacios, into the Kitten LWK, and customize it for HPC workloads. Our solution is unique compared to other cloud/utility computing virtualization layers in that it is focused on HPC and takes advantage of the unique characteristics of the LWK environment to reduce virtualization overhead. We used the Kitten and Palacios combination

to perform large-scale testing on up to $6,240$ nodes of Red Storm Cray XT4 system using HPC micro-benchmarks and several real applications. The observed virtualization overhead of less than 5% demonstrated that it is feasible to run real, communication-intensive HPC applications in a virtualized environment at large scale.

This work is targeted to have impact on exascale supercomputers, which are expected in the 2018–2020 time frame. Several projects are building upon the outcomes of this project and are using Kitten and Palacios as tools for conducting research. In the near term, we are pushing to get virtualization technology incorporated into the production supercomputer system software stack so that end-users and system software researchers can take advantage of the new capabilities that it provides. This, and other topics, will be pursued in the context of a FY11-13 DOE ASCR "X-Stack Software Research" project that we proposed and that was selected for funding. This effort is a consortium of researchers from University of New Mexico, Northwestern University, Sandia National Laboratories, and Oak Ridge National Laboratory.

The remainder of this report is a compilation of three selected research papers resulting from this project. The first two papers are peer-reviewed conference papers and the third paper has been submitted for consideration to a peer-reviewed conference. Each paper is prefaced with a short paragraph describing the research history leading up to the paper. Appendix A includes a full listing of the publications, invited talks, follow-on funded projects, and other forms of external impact derived from this project.

Chapter 1

SMARTMAP: Operating System Support for Efficient Data Sharing Among Processes on a Multi-Core Processor

The content of this chapter was originally published in a paper by the same name at the 2008 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'08). The proper citation is [10]. The ideas leading to SMARTMAP sprung from observations made in an SC'07 poster [54] and discussions with Ron Brightwell. SMARTMAP was originally implemented for the Catamount LWK in FY08, and later implemented for the Kitten LWK in FY09. SMARTMAP (patent pending) was an important component of the 2009 "Catamount N-Way Lightweight Kernel" R&D100 award winning submission.

Abstract

This paper describes SMARTMAP, an operating system technique that implements fixed offset virtual memory addressing. SMARTMAP allows the application processes on a multi-core processor to directly access each other's memory without the overhead of kernel involvement. When used to implement MPI, SMARTMAP eliminates all extraneous memory-to-memory copies imposed by UNIX-based shared memory strategies. In addition, SMARTMAP can easily support operations that UNIX-based shared memory cannot, such as direct, in-place MPI reduction operations and one-sided get/put operations. We have implemented SMARTMAP in the Catamount lightweight kernel for the Cray XT and modified MPI and Cray SHMEM libraries to use it. Micro-benchmark performance results show that SMARTMAP allows for significant improvements in latency, bandwidth, and small message rate on a quad-core processor.

1.1 Introduction

As the core count on processors used for high-performance computing continues to increase, the performance of the underlying memory subsystem becomes significantly more important. In order to make effective use of the available compute power, applications will likely have to become much more sensitive to the way in which they access memory. Applications that are memory bandwidth bound will need to avoid any extraneous memory-to-memory copies. For many applications, the memory bandwidth limitation is compounded by the fact that the most popular and effective parallel programming model, MPI, mandates copying of data between processes. MPI implementors have worked to make use of shared memory for communication between processes on the same node. Unfortunately, the current schemes for using shared memory for MPI can require either excessive memory-to-memory copies or potentially large overheads inflicted by the operating system (OS).

In order to avoid the memory copy overhead of MPI altogether, more and more applications are exploring mixed-mode programming models where threads and/or compiler directives are used on-node and MPI is used off-node. Unfortunately, the complexity of shared memory programming using threads has hindered both the development of applications as well as the development of thread-safe and thread-aware MPI implementations. The initial attractiveness of mixed-mode programming was tempered by the additional complexity induced by finding multi-level parallelism and by initial disappointing performance results [20, 34, 22]. Recently, however, unpublished data on mixed-mode applications suggest more encouraging results on multi-core processors.

In this paper, we introduce a scheme for using fixed-offset virtual address mappings for the parallel processes within a node to enable efficient direct access shared memory. This scheme, called Simple Mapping of Address Region Tables for Multi-core Aware Programming, or SMARTMAP, achieves a significant performance increase for on-node MPI communications and eliminates all of the extraneous memory-to-memory copies that shared memory MPI implementations incur. SMARTMAP can also be used for more than MPI. It maps very well to the partitioned global address space (PGAS) programming model and can be used to implement one-sided get/put operations, such as those available in the Cray SHMEM model. This strategy can also be used directly by applications to eliminate the need for on-node memory-to-memory copying altogether.

The main contributions of this paper are:

- an OS virtual memory mapping strategy that allows direct access shared memory between processes on a multi-core processor
- a description of how this strategy can be used for on-node data movement between processes on a multi-core processor
- a detailed analysis of the performance impacts of using this strategy for MPI peer communication, MPI collective communication, and Cray SHMEM data movement operations

The rest of this paper is organized as follows. The next section provides background on the current approaches to using shared memory for intra-node data movement. In Section 1.3, we describe the implementation of the SMARTMAP and its advantages over other approaches. Section 1.4 provides a detailed description of the enhancements that we have made to MPI and SHMEM implementations on the Cray XT to use it. Section 1.5 presents performance results using several micro-benchmarks. Relevant conclusions of this paper are summarized in Section 3.8, and we close by discussing possible avenues of future work in Section 1.7.

1.2 Background

POSIX-based operating systems generally support shared memory capability through two fundamental mechanisms: threads and memory mapping. Unlike processes, which allow for a single execution context inside an address space, threads allow for multiple execution contexts inside a single address space. When one thread updates a memory location, all of the threads sharing the same address space also see the update. A major drawback of threads is that great care must be taken to ensure that common library routines are reentrant, meaning that multiple threads could be executing the same piece of code simultaneously. For non-reentrant functions, some form of locking must be used to ensure atomic execution. The same is true for data accessed by multiple threads – updates must be atomic with respect to one another or else difficult to debug race conditions will occur. Race conditions and fundamentally non-deterministic behavior make threads difficult to use correctly.

In memory mapping, cooperating processes request a shared region of memory from the operating system and then map it into their private address space, possibly at a different virtual address in each process. Once initialized, a process may access the shared memory region in exactly the same way as any other memory in its private address space. As with threads, updates to shared data structures in this region must be atomic.

Explicit message passing is an alternative to shared memory for intra-node data sharing. In message passing, processes pass messages carrying data between one another. No data is shared directly, but rather is copied between processes on an as necessary basis. This eliminates the need for re-entrant coding practices and careful updates of shared data, since no data is shared. The main downside to this approach is the extra overhead involved in copying data between processes.

In order to accelerate message passing, memory mapping is often used as a high-performance mechanism for moving messages between processes [23]. Unfortunately, such approaches to using page remapping are not sufficient to support MPI semantics, and general-purpose operating systems lack the appropriate mechanisms. The sender must copy the message into a shared memory region and the receiver must copy it out – a minimum of two copies must occur. It would be ideal if messages could be moved directly between the two processes with a single copy. This would be possible if all processes operated entirely out of the shared memory region, but this would amount to the processes essentially becoming threads, with

all of their inherit problems. Furthermore, message passing APIs such as MPI allow message buffers to be located anywhere in an address space, including the process's data, heap and stack.

As of MPI 2.0, MPI applications may make use of both threads and memory mapping, although few MPI implementations provide full support for threads. More commonly, MPI implementations utilize memory mapping internally to provide efficient intra-node communication. During MPI initialization, the processes on a node elect one process to create the shared memory region and then the elected process broadcasts the information about the region to the other processes on the node (e.g., via a file or the sockets API). The other processes on the node then "attach" to the shared memory region, by requesting that the OS map it into their respective address spaces.

Note that the approach of using shared memory for intra-node MPI messages only works for the point-to-point operations, collective communication operations, and a subset of the MPI-2 remote memory access operations. Copying mandates active participation of the two processes involved in the transfer. Single-sided put/get operations, such as those in the Cray SHMEM programming interface, cannot be implemented using POSIX shared memory.

1.2.1 Intra-Node MPI

There are several limitations in using regions of shared memory to support intra-node MPI [18, 17, 16]. First, the MPI model doesn't allow applications to allocate memory out of this special shared region, so messages must first be copied into shared memory by the sender and then copied out of the shared region by the receiver. This copy overhead can be a significant performance issue. Typically there is a limitation on the amount of shared memory that a process can allocate, so the MPI implementation must make decisions about how to most effectively use this memory in terms of how many per-process messages to support relative to the size of the contents of each message. The overhead of copying messages using shared memory has led researchers to explore alternative single-copy strategies for intra-node MPI message passing.

One such strategy is to use the operating system to perform the copy between separate address spaces [40]. In this method, the kernel maps the user buffer into kernel space and does a single memory copy between user space and kernel space. The drawback of this approach is that the overhead of trapping to the kernel and manipulating memory maps can be expensive. Another limitation is that all transfers must be serialized through the operating system. As the number of cores on a node increases, serialization and management of shared kernel data structures for mapping is likely to be a significant performance limitation. Another important drawback of this approach is that there are two MPI receive queues – one in the MPI library and one in the kernel. When the application posts a non-specific receive using `MPI_ANY_SOURCE`, great care must be taken to insure that the atomicity and ordering semantics of MPI are preserved. There is a potential race for a non-specific receive request to be satisfied by both the MPI library and the operating system. Managing atomicity between

events in kernel space and user space is non-trivial.

Another strategy for optimizing intra-node transfers is to use hardware assistance beyond the host processors. The most common approach is to use an intelligent or programmable network interface to perform the transfer. Rather than sending a local message out to the network and back, the network interface can simply use its DMA engines to do a single copy between the communicating processes. The major drawback of this approach is serialization through the network interface, which is typically much slower than the host processor(s). Also, large coherent shared memory machines typically have hardware support for creating a global shared memory environment. This hardware can also be used when running distributed memory programs to map arbitrary regions of memory to provide direct shared memory access between processes. SGI's NUMAlink hardware is one such example [26]. The obvious drawback of this approach is the additional cost of this hardware.

A comprehensive analysis of the different approaches for intra-node MPI communication was presented in [15]. More recently, a two-level protocol approach that uses shared memory regions for small messages and OS support for page remapping individual buffers for large messages was proposed and evaluated [21]. There has also been some recent work on optimizing MPI collective operations using shared memory for multi-core systems [33].

1.2.2 Intra-Node Communication on the Cray XT

All communication between processes on the Cray XT use the Portals [11] data movement layer. Two implementations of Portals are available for the SeaStar [12] network. The default implementation is interrupt driven and all Portals data structures are contained inside the operating system. When a message arrives at the SeaStar, it interrupts the Opteron host processor, which then inspects the message header, traverses the Portals data structures and programs the DMA engines on the SeaStar to deliver the message to the appropriate location in the application process' memory. This implementation is referred to as "Generic Portals" (GP) because it works for both Catamount on compute nodes and in Linux on service and I/O nodes. The other implementation supports a complete offload of Portals processes and uses no interrupts. When a message arrives at the SeaStar, all of the Portals processing occurs on the SeaStar itself. This implementation is known as "Accelerated Portals" (AP) and is available only on Catamount, largely due to the simplified address translation that Catamount offers.

For intra-node transfers, the Generic Portals implementation takes advantage of the fact that Portals structures for both the source and destination are in kernel space. The kernel is able to traverse the structures and perform a single memory copy to move data between processes, since all of user space is also mapped into kernel space. At large message sizes, it becomes more efficient for the kernel to use the DMA engines on the SeaStar to perform the copy, so there is a crossover point where it switches to using this approach. For the Accelerated Portals implementation, all Portals data structures are in SeaStar memory, so it must traverse these structures in the same way it does for incoming network messages,

so there is little advantage to intra-node transfers. In fact, intra-node transfers are slower going through the SeaStar rather then the operating system, due to the speed of the host processor (2+ GHz) relative to the network processor (500 MHz).

1.3 SMARTMAP Implementation

SMARTMAP is a virtual memory mapping technique that allows for direct access shared memory between the processes running on a multi-core processor. This technique leverages many of the characteristics of a lightweight compute node kernel to achieve shared memory capability without the limitations of POSIX shared memory mapping or the complexity of multi-threading. SMARTMAP preserves the idea of running a single execution context within a separate address space, but also provides the ability to easily access the address space of the other execution contexts within the same parallel job on the same node. The following provides a description the implementation of SMARTMAP and its advantages over existing approaches for intra-node data movement.

1.3.1 Catamount

The Catamount lightweight kernel [43] is a third-generation compute node operating system developed by Sandia National Laboratories along with Cray, Inc., as part of the Red Storm project [19]. Red Storm is the prototype for what has become the commercially successful Cray XT line of massively parallel processing systems. Catamount has several unique features that are designed to optimize performance and scalability specifically for a distributed memory message passing-based parallel computing platform.

One such important feature is memory management. Unlike traditional UNIX-based operating systems, Catamount does not support demand-paged virtual memory and uses a linear mapping from virtual addresses to physical pages of memory. This approach can potentially have several advantages. For instance, there is no need to register memory or "lock" memory pages involved in network transfers to prevent the operating system from unmapping or remapping pages. The mapping in Catamount is done at process creation time and is never changed. This greatly simplifies translation and validation of virtual address for the network interface. Virtual address validation is a simple bounds check and translating virtual addresses to physical addresses is a simple offset calculation.

The SMARTMAP approach for direct access shared memory takes advantage of Catamount's simple memory management model, specifically the fact that Catamount only uses a single entry in the top-level page table mapping structure (PML4) on each X86-64 (AMD Opteron or Intel EM64T) core. Each PML4 slot covers 39 bits of address space, or 512 GB of memory. Normally, Catamount only uses the first entry covering physical addresses in the range `0x0` to `0x007FFFFFFFFF`. The X86-64 architecture supports a 48-bit address space, so there are 512 entries in the PML4.

```
1  static void initialize_shared_memory( void )
2  {
3    extern VA_PML4T_ENTRY *KN_pml4_table_cpu[];
4    int cpu;
5    for( cpu=0 ; cpu < MAX_NUM_CPUS ;cpu++ ) {
6      VA_PML4T_ENTRY * pml4 = KN_pml4_table_cpu[ cpu ];
7      if( !pml4 )
8        continue;
9      KERNEL_PCB_TYPE * kpcb = (KERNEL_PCB_TYPE*) KN_cur_kpcb_ptr[cpu];
10     if( !kpcb ) continue;
11     VA_PML4T_ENTRY dirbase_ptr = (VA_PML4T_ENTRY)
12       (KVTOP( (size_t) kpcb->kpcb_dirbase ) | PDE_P | PDE_W | PDE_U );
13     int other;
14     for( other=0 ; other<MAX_NUM_CPUS ; other++ ) {
15       VA_PML4T_ENTRY * other_pml4 = KN_pml4_table_cpu[other];
16       if( !other_pml4 ) continue;
17       other_pml4[ cpu+1 ] = dirbase_ptr;
18     }
19   }
20 }
```

Figure 1.1: SMARTMAP kernel code

```
1  static inline void * remote_address( unsigned core,
2                                       volatile void * vaddr)
3  {
4    uintptr_t addr = (uintptr_t) vaddr;
5    addr |= ((uintptr_t) (core+1)) << 39;
6    return (void*) addr;
7  }
```

Figure 1.2: User function for converting a local virtual address to a remote virtual address

Each core writes the pointer to its PML4 table into an array at core 0 when a new parallel job is started. Each time the kernel enters the routine to run the user-level process, it copies all of the PML4 entries from each core into the local core. This allows every core on a node to see every other core's view of the virtual memory across the node, at a fixed offset into its own virtual address space. Figure 1.1 shows the 20 lines of kernel code that implement direct access shared memory in Catamount.

Another feature of Catamount is that the mapping of virtual addresses for the same executable image is identical across all of the processes on all of the nodes. The starting address of the data, stack, and heap is the same. This means that the virtual address of variables with global scope is the same everywhere. The Cray SHMEM environment refers to such addresses as *symmetric* addresses, whereas other addresses, such as those allocated off of the stack as the application is running, are termed to be *non-symmetric*. Figure 1.2 shows the user-level function for converting a *local* virtual address into a *remote* virtual address for a process on a different core. Symmetric addresses combined with this simple remote address translation function make it extremely easy for one process to read or write the corresponding data structure in another process' address space running on a different core of the same processor.

Catamount's memory management design is much simpler than a general-purpose OS like Linux. Linux memory management is based on the principle that processes execute in different address spaces and threads execute in the same address space. Most architecture

ports, x86 included, maintain a unique set of address translation structures (e.g., a page table tree on x86) for each process and a single set for each group of threads. SMARTMAP operates differently in that a process's address space and associated translation structures are neither fully-unique or fully-shared. For example, SMARTMAP on the x86 architecture maintains a unique top-level page table (the PML4) for each process; however, all processes share a common set of leaves linked from this top-level table. Linux memory management does not support this form of page-table sharing, so each process must be given a replicated copy of each shareable leaf. This results in more memory being wasted on page tables (2 MB per GB of address space on x86) and a larger cache footprint than necessary. Modifications to Linux to support sharing a single page table entry for shared memory mapped regions has been proposed, but the changes have not been accepted in the mainline kernel.

1.3.2 Limitations

SMARTMAP is currently limited to what the top-level X86-64 page table supports – 511 processes (one slot is needed for the local process) and 512 GB of memory per process. However, this will likely be sufficient for a typical compute node for the foreseeable future. Since Catamount only runs on X86-64 processors, SMARTMAP is currently limited to this processor family as well. However, the concepts are generally applicable to other architectures that support virtual memory. For example, even though the PowerPC uses an inverted page table scheme that is very different from x86-64, the hardware's support for segmentation can be used to implement SMARTMAP just as efficiently. On other architectures with software-based virtual memory support (i.e., a software managed translation look-aside buffer), SMARTMAP is straightforward to implement.

1.4 Using SMARTMAP

We have used the SMARTMAP capability in Catamount to optimize intra-node data movement for the Cray SHMEM one-sided operations, as well as for MPI point-to-point and collective operations. This section describes the modifications to these libraries.

1.4.1 Cray SHMEM

The Cray SHMEM library was first available on the Cray T3 series of machine circa 1994. It supports a variety of one-sided get/put data movement functions as well as collective reduction functions and remote atomic memory operations, such as atomic-swap and fetch-and-increment.

The existing implementation for Catamount on the Cray XT uses Portals for all data movement operations. Similar to MPI's profiling interface, Cray has implemented an alter-

```
1  void shmem_putmem( void *target, void *source, size_t length, int pe )
2  {
3    int core;
4
5    if ( (core = smap_pe_is_local(pe)) != -1 ) {
6      void *target_r = (void *)remote_address( core, target );
7      memcpy( target_r, source, length );
8    } else {
9      pshmem_putmem( target, source, length, pe );
10   }
11 }
```

Figure 1.3: SHMEM Put Function

native library interface to all SHMEM functions to support user-level redefinition of library routines. All functions are defined as weak symbols with a set of shadow functions whose names are prefaced by a 'p'. For example, the library defines `shmem_put()` as a weak symbol and defines `pshmem_put()` as the actual function. This makes it possible for an application to define it's own version of the function that in turn calls the underlying library function. This mechanism makes it easy to extend the implementation to use SMARTMAP for intra-node transfers.

At library initialization time, we determine which destination ranks are on the local node. We do this using information from the Catamount runtime system that conveys the rank, node id, and core of each process in the job. We actually use the SMARTMAP capability for each process on a node to determine a global rank to core rank mapping. Once this mapping is determined, we simply add logic to each function to determine whether the destination process is on-node or off-node. For on-node communications, we use the virtual address conversion function to determine the remote virtual address to use and then perform the appropriate operation. If the destination rank is off-node, we fall through to the actual function. Figure 1.3 shows the implementation of the `shmem_putmem()` routine using SMARTMAP. We have done this for the basic put and get operations in order to measure the performance gain from SMARTMAP. Implementations of the strided get/put operations as well as the atomic memory operations would be similarly straightforward. Changes to the internal implementation of the collective operations would be needed to differentiate between on-node and off-node data movement.

1.4.2 MPI Point-to-Point Communication

We have modified the Open MPI implementation to make use of SMARTMAP. We chose Open MPI because it is the only open-source implementation that supports shared memory that already has support for the Cray XT. Recently, Cray has released an implementation of MPI for their compute node Linux environment that supports shared memory. However, this implementation is encumbered with SGI contributions and is not available outside of Cray. Cray is continuing to maintain a completely separate MPI implementation for Catamount, which is also not available as open source. The modular component-based architecture of Open MPI also simplifies the introduction of a new transport layer.

There are two different paths that Open MPI can use for MPI point-to-point communications using Portals. The default path is to use a PML module that implements MPI matching semantics inside the MPI library and uses the underlying Byte Transport Layer (BTL) to simply move bytes. This layer can make use of several BTL modules at one time, including shared memory or the network as appropriate for the destination. The second path is for the PML to use a Matching Transport Layer (MTL). This path assumes that the underlying module is responsible for implementing MPI matching semantics. Unlike the BTL, there can only be one of these modules in use at any given time. An important distinction between these two paths is the location of the MPI receive queue. For the BTLs, the MPI receive queue is inside the library, but for an MTL, the receive queue is managed outside of the MPI library.

We modified both the shared memory BTL in Open MPI as well as the Portals MTL to use SMARTMAP. This approach allows us to better quantify the advantage of avoiding an extra copy in the shared memory BTL.

Relatively few changes were necessary to allow the shared memory BTL to use SMARTMAP. Rather than having the individual processes use `mmap()` to map the same block of shared memory, the core 0 process on a node simply publishes the location of the block of memory that it has allocated from its local heap. Using SMARTMAP, the other processes read this location from core 0's memory and convert it to the appropriate remote address.

More extensive changes were required to enable the Portals MTL to use SMARTMAP. A detailed description of a prototype of this implementation can be found in [9]. The prototype only had support for intra-node transfers, but it has since been extended to support both on-node and off-node communication. This implementation has two posted receive queues – one inside Portals for off-node transfers and one inside the MPI library for on-node transfers – so it is subject to the same complexity that other such implementations are. In particular, non-specific receives are not currently fully supported. If a receive request using `MPI_ANY_SOURCE` cannot be immediately completed, a failure is returned. We are currently extending the implementation to handle this situation. We do not expect this extra logic to have a significant impact on performance, especially for communication micro-benchmarks and codes that do not employ a large number of wildcard receive requests.

A key difference between the BTL and MTL implementations is that the BTL is able to copy user data along with the MPI envelope information, allowing for short send operations to complete before the data has actually been transfered to the receiver's buffer. Given that the focus of SMARTMAP is to decrease the number of memory-to-memory copies, we chose not to employ this optimization for the MTL. Therefore, short messages using SMARTMAP are synchronous – the data is only copied when the matching receive buffer has been posted.

1.4.3 MPI Collective Communication

We have also created an Open MPI collective communication module that uses SMARTMAP to implement the barrier, broadcast, reduce, allreduce, and alltoall collective operations. We briefly describe the implementation here.

The SMARTMAP collective module uses a structure containing the following information:

- `counter`
- `context`
- `address`
- `turn`
- `finished`

This structure is globally-scoped so that it is at the same memory location in all of the processes on a node. The first two items, `counter` and `context`, are specific to the MPI communicator involved in the collective operation. Since MPI collective operations are blocking, a process can be participating in at most one collective at a time. The communicator's `counter` is incremented each time a collective operation is started and the `context` is used to identify the specific communicator that is being used. This prevents sub-communicators in overlapping collective operations from interfering with each other.

When a process enters a collective operation, it first determines whether it is the root of the collective operation, For non-rooted operations, the root defaults to rank 0 within the communicator. Once the root is determined, the process determines the core on which the root process is running. If a process is not the root, it gets the remote address of the collective structure in the root process' address space and waits for the `counter` and `context` values to indicate that the root has entered the same collective operation.

For the barrier operation, the root process initializes the `finished` value to 1, sets the `counter` and `context` values appropriately, and then spins on this value waiting for it to be equal to the size of the communicator. Once the non-root processes enter the collective operation, they increment the `finished` value in the root's address space using an assembly language atomic increment operation. As with the root, they also spin waiting for this value to be equal to the size of the communicator.

For the broadcast operation, the root process again initializes the `finished` value to 1 and sets the `address` value to the location of the user buffer. It then waits for the other processes to increment the `finished` value. Once the non-root processes enter the collective operation, they read the `address` value in the other process' address space, convert this value to a remote address, and then copy the data from the source buffer directly to the

destination buffer in its address space. When the copy is complete, the process atomically increments the `finished` value.

For the reduce operation, the root process first copies the send buffer to the receive buffer (provided the MPI_IN_PLACE flag is not used), initializes the `finished` value to 1, and initializes the `turn` value to 0. It sets the `context` and `counter` values, and then proceeds as the non-root processes do. Once in the collective operation, a non-root process recognizes the destination address and converts it to a remote address in the root core's address space. It then waits for the `turn` value to be equal to its rank. Once this occurs, the process performs the reduce operation with the its buffer and the root's buffer. When the reduce operation is complete, it atomically increments the `turn` value to let the next rank proceed, and atomically increments the `finished` value to indicate that it is done. When the root process' turn is up, it simply increments the counter to let the following rank proceed. As with the other SMARTMAP collectives, the root waits for the `finished` value to reach the size of the communicator.

Currently, the allreduce operation is implemented as a reduce followed by a broadcast. The alltoall operation is implemented as a broadcast with each process taking turns being the root. The current implementation of alltoall is cache friendly, since all cores are copying the same buffer at the same time. An alternative implementation could allow for each process to copy its chunk of data to the other processes.

1.5 Performance Evaluation

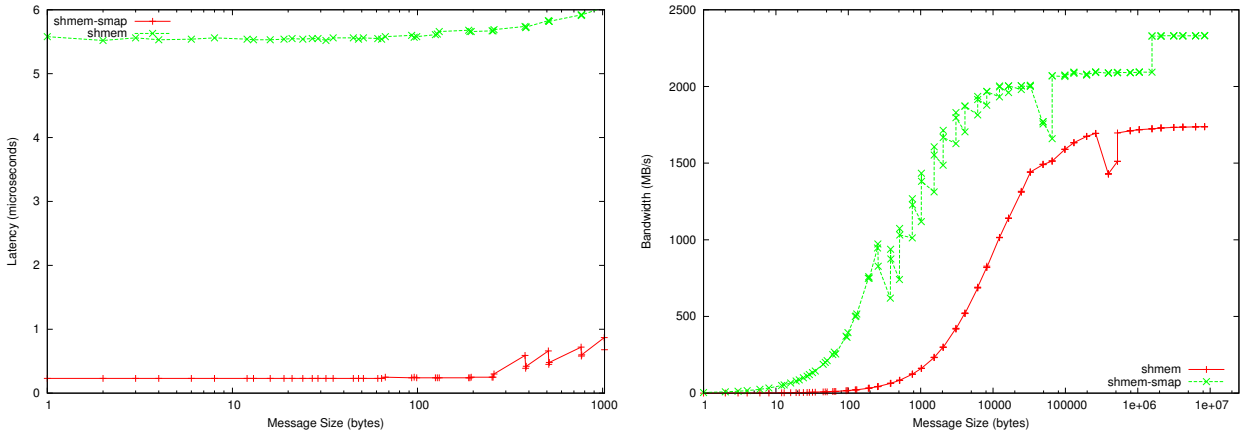
1.5.1 Test Environment

The platform used to gather our performance results is a Red Storm development system that contains four 2.2 GHz quad-core Opterons. We have added SMARTMAP capability to the Catamount N-Way (CNW) kernel version 2.0.41. Our changes to Open MPI were performed on the head of the development tree.

For intra-node results, we limited our results to the interrupt-driven version of Portals because it is more efficient at intra-node transfers. The ability to have the operating system perform a copy between processes outperforms having the SeaStar adapter do the copy. Due to limitations of the SeaStar, send operations must go through the OS, so in addition to serializing requests through a slower network interface, requests must also be serialized through the OS.

1.5.2 SHMEM

Figure 1.4 shows the ping-pong latency and bandwidth performance for a Cray SHMEM put operation using the default implementation and the SMARTMAP-enabled implemen-



(a) Latency (b) Bandwidth

Figure 1.4: SHMEM Put Performance

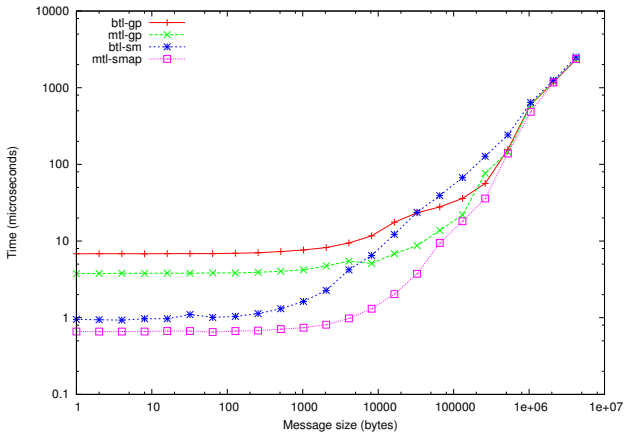
tation as measured with the NetPIPE [63] benchmark. Single-byte latency for the default implementation is more than 5 μs, while the SMARTMAP latency is 230 ns. Bandwidth performance for the SMARTMAP-enabled SHMEM also significantly outperforms the default implementation, having a much steeper curve and achieving much higher asymptotic performance. The erratic nature of the bandwidth curve for the SMARTMAP-enabled SHMEM is due to the sensitivity of the memory sub-system to misalignment as a result of the various transfer lengths that NetPIPE uses. The dip at 32 KB is repeatable and is also likely due to the memory hierarchy, since this is half of the size of the first-level cache. We used a memory copy routine with non-temporal stores, which we believe is responsible for the jump in bandwidth performance at 2 MB. For SHMEM over Portals, the crossover point from using shared memory to using the network is clearly visible at 512 KB.

1.5.3 MPI Point-to-Point

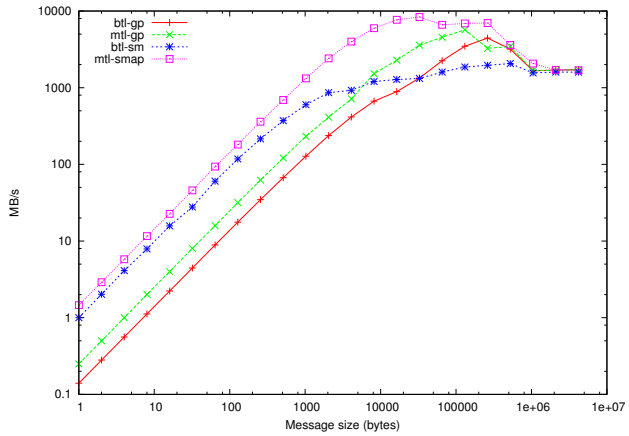
Figure 1.5 shows the performance of widely-used Intel MPI Benchmark suite version 2.3 for the point-to-point operations. We compare the default Portals BTL (btl-gp) and MTL (mtl-gp) with the shared memory BTL (btl-sm) using SMARTMAP and the SMARTMAP Portals MTL (mtl-smap).

Ping-pong latency performance is shown in Figure 1.5(a). The Portals MTL with SMARTMAP is able to achieve a zero-byte latency of 630 ns, with the shared memory BTL using SMARTMAP slightly higher at 830 ns. This is a significant improvement over the 3 μsPortals MTL, where the OS performs the memory copy between the processes. The difference in performance between the MTL and BTL is likely due to the additional memory operations needed by the BTL to enqueue a request in a shared data structure. For the MTL, each process has exclusive access to the data structures necessary for enqueing a request.

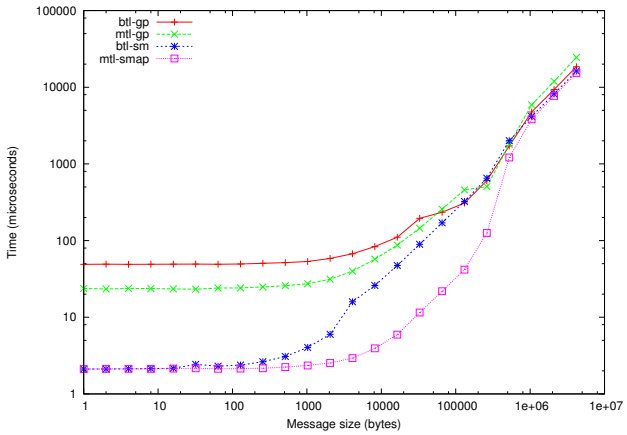
Ping-pong bandwidth performance is shown in Figure 1.5(b). Here again we see that the



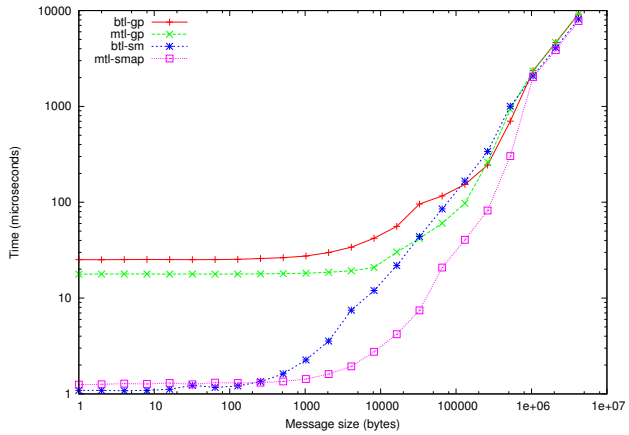
(a) MPI PingPong Latency



(b) MPI PingPong Bandwidth

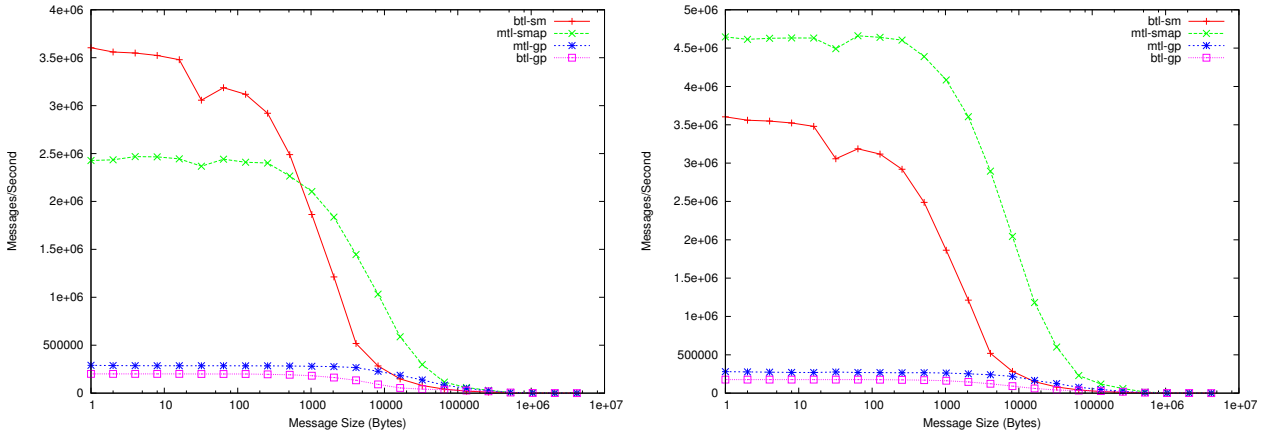


(c) MPI Exchange



(d) MPI Sendrecv

Figure 1.5: IMB MPI Point-to-Point Results



(a) One pair (b) Two pairs

Figure 1.6: MPI Message Rate

SMARTMAP-enabled MTL is able to outperform the others, peaking at 9.4 GB/s. This is significantly higher than the peak 5.7 GB/s of the Portals MTL without SMARTMAP. We can also see that the performance of the shared memory BTL starts to be affected by doing two memory copies rather than one. Unlike the previous SHMEM bandwidth test that uses NetPIPE, the IMB bandwidth test does not actually read the receive buffer, so the improved performance of MPI over SHMEM is due to cache effects.

Figures 1.5(c) and 1.5(d) show performance for the IMB Sendrecv and Exchange benchmarks. We chose these benchmarks to illustrate the capability of SMARTMAP to allow for simultaneous communications within a node. The Sendrecv benchmark measures performance between pairs of processes communicating with the `MPI_Sendrecv()` function, while the Exchange benchmark measures the performance of exchanging data with a pair of neighbor processes. The Portals BTL and MTL are limited by serialization through the OS, while with the shared memory based transports, the processes are able to communicate without any serialization. We can also see the penalty that the two-copy shared memory strategy has for these operations as well.

Another important measurement of MPI point-to-point performance is message rate. We used the PathScale (now QLogic) MPI message rate benchmark, which is a modified version of an MPI bandwidth benchmark from Ohio State University. The original benchmark was enhanced to support reporting message rate as well as bandwidth, to calculate and report the $N^{1/2}$ message size and rate, and to allow for running multiple processes per node to calculate aggregate performance. Figures 1.6(a) and 1.6(a) show the message rate for one pair of communicating processes and two pairs of processes respectively. For one pair, the shared memory BTL is able to achieve more than 3.5 million messages per second, while the Portals MTL with SMARTMAP achieves about 2.4 million message per second. The non-SMARTMAP layers achieve less than 300 thousand messages per second. The memory copies in the shared memory BTL allow for decoupling the sender and the receiver. For short messages, the BTL is able to copy the message into shared memory and, from the MPI

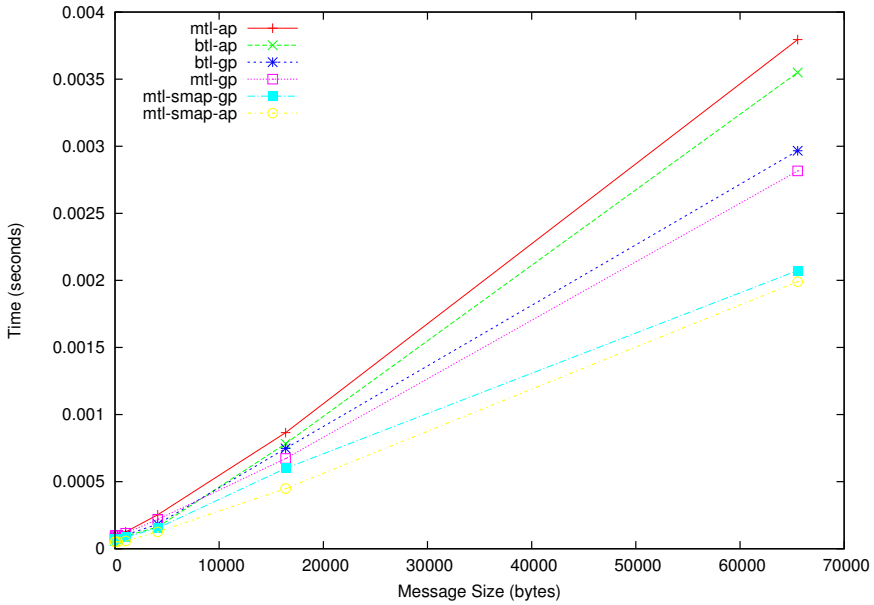


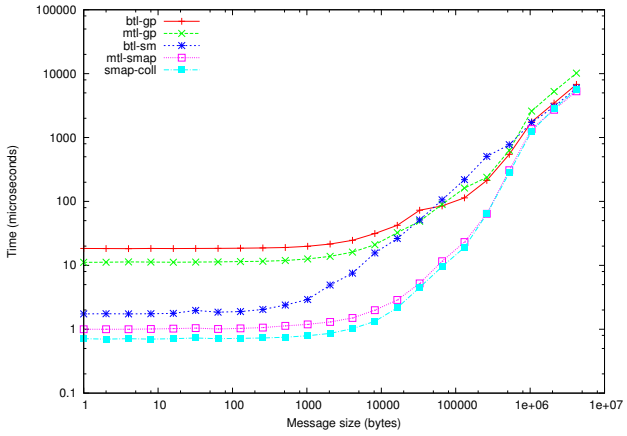
Figure 1.7: MPI Halo Exchange Performance

perspective, the send is complete. However, since the SMARTMAP MTL is synchronous, it does not perform the memory copy until the receiver has posted a receive request. The overhead of this synchronization degrades message rate performance, but the single-copy ability of SMARTMAP eventually catches up at larger message sizes. For two pairs of processes, message rate for the MTL scales nearly linearly, almost doubling to 4.6 million messages per second, while the BTL rate remains constant. The message rate actually decreases slightly for the Portals-based transports.

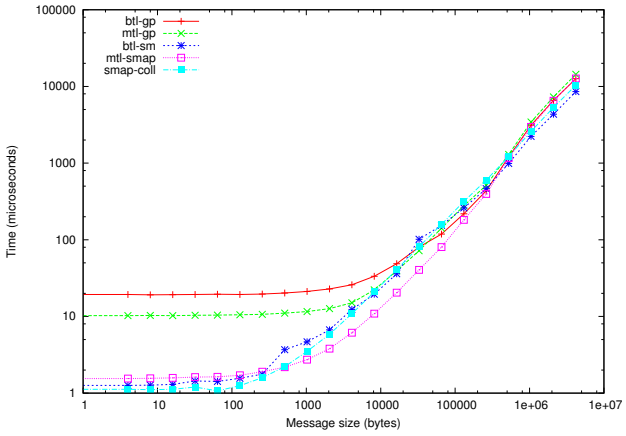
We finish our analysis of MPI point-to-point communication with a halo exchange benchmark from Argonne National Lab. We ran this benchmark across four quad-core nodes using sixteen processes. The results are shown in Figure 1.7. Unlike the intra-node performance results, this benchmark shows the advantage of the AP version of Portals. The Portals MTL with SMARTMAP enabled allows for efficient on-node transfers, while the AP implementation of Portals allows for more efficient off-node transfers.

1.5.4 MPI Collectives

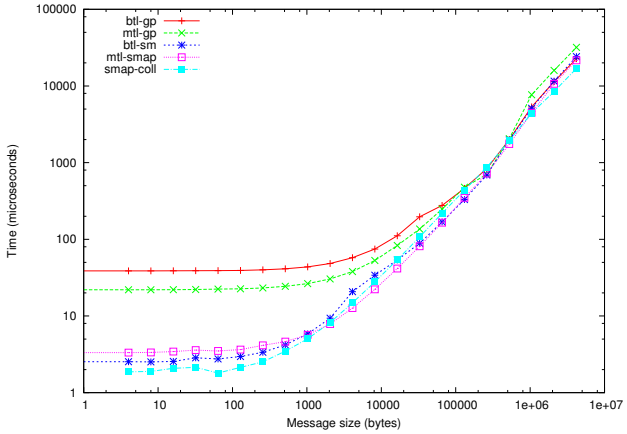
Figure 1.8 shows performance for the broadcast, reduce, allreduce, alltoall, and barrier MPI collective operations on a single quad-core node. This graph also includes performance of the SMARTMAP collective module (smap-coll). As with the point-to-point operations, we can again see the significant performance gain for using SMARTMAP. For the broadcast and alltoall operations in Figures 1.8(a) and 1.8(d) respectively, we can also see the advantage that the single copy approach has for larger message sizes over the two-copy approach of the shared memory BTL. Barrier performance in Figure 1.8(e) demonstrates the advantage of



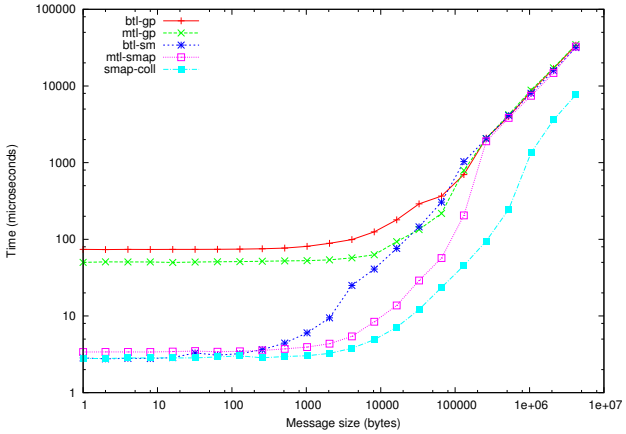
(a) MPI Broadcast



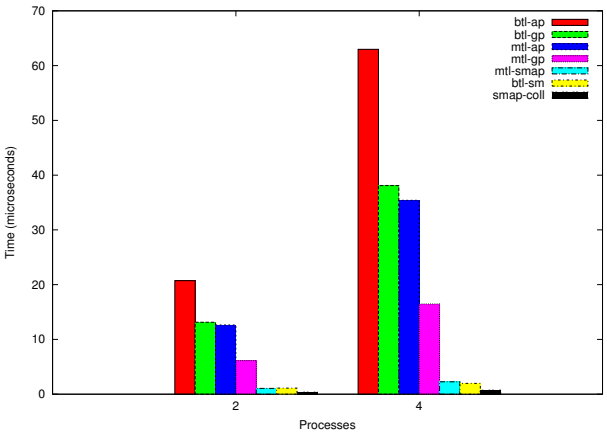
(b) MPI Reduce



(c) MPI Allreduce



(d) MPI Alltoall



(e) MPI Barrier

Figure 1.8: IMB MPI Collective Performance

using a counter in shared memory rather than using message passing in shared memory.

1.6 Conclusion

The SMARTMAP capability in Catamount is able to deliver significant performance improvements for intra-node MPI point-to-point and collective operations. It is able to dramatically outperform the current approaches for intra-node data movement using Portals on the Cray XT. We expect the shared memory BTL performance to be similar to what Cray's Compute Node Linux (CNL) environment could achieve using shared memory in Linux. However, we have also shown that the single-copy ability of SMARTMAP in Catamount is able to significantly outperform the mulitple-copy approach that must be used in a POSIX-based shared memory environment like Linux. Additionally, SMARTMAP can support operations that Linux shared memory cannot. First, SMARTMAP can eliminate *all* extraneous memory-to-memory copies for intra-node MPI communications. This is a significant advantage in light of the growing memory bandwidth limitation of multi-core processors. SMARTMAP can also support true one-sided get/put operations and extremely efficient collective operations, including the ability to perform reduction operations directly on the destination buffer.

1.7 Future Work

There is more work left to do to fully utilize the SMARTMAP capability for MPI. First, because the Portals data movement layer encapsulates the MPI posted receive queue, the complexity of handling `MPI_ANY_SOURCE` receives is significantly increased. The current implementation does not fully support non-specific receives, but we do not expect the logic needed to support them to significantly impact performance. We would also like to implement single-copy non-contiguous data transfers and MPI-2 remote memory access operations.

We are currently working on additional collective operations for the SMARTMAP collective module, specifically the gather operations. We would like to do an in-depth analysis of collective performance using Open MPI's hierarchical collective module, where on-node collectives would use the SMARTMAP module in combination with a network-based collective module.

With the recent release of a Cray implementation of MPI for CNL that supports shared memory transfers, we would like to do an in-depth analysis of on-node MPI communication performance between Catamount and CNL.

Once we have complete point-to-point and collective layers, we would also like to perform and in-depth analysis of application performance. Our current 4-node quad-core environment is not sufficient to analyze application performance and scalability. The center section of the Red Storm system, approximately six thousand nodes, will soon be upgraded to quad-core

processors, and we expect to perform an exhaustive analysis of applications as part of the upgrade. It would also be interesting to measure SMARTMAP performance on larger core counts, such as a dual-socket quad-core Cray XT5 node.

SMARTMAP is also a natural fit for implementation of the Partitioned Global Address Space (PGAS) Model. The implementations of Unified Parallel C, Co-Array Fortran, and Global Arrays could be enhanced to leverage SMARTMAP capabilities.

We are also exploring ways for applications to use the SMARTMAP capability directly, through library interfaces that allow processes to do direct remote loads and stores. We currently have MPI applications that are conducive to recoding pieces of them to use shared-memory style communications. The advantage of SMARTMAP for this is that we can avoid the memory copy overhead of using MPI and also avoid the complexity of mixing MPI with threads or OpenMP compiler directives.

Finally, we are also considering exposing the topology of the underlying machine to applications using MPI communicators. We can easily create communicators to be used for on-node or off-node communications (e.g, `MPI_COMM_NODE` and `MPI_COMM_NET`). Some applications may be able to decompose communication into two levels to better leverage the advantages of intra-node communication performance.

1.8 Acknowledgments

The implementation of SMARTMAP would not have been possible without the efforts of John Van Dyke, who is responsible for implementing virtual node mode support in Catamount. Kurt Ferreira also provided many useful discussions regarding virtual memory mapping and lightweight kernel memory management. The authors also gratefully acknowledge the assistance of Sue Kelly and the Cray support staff at Sandia with the Red Storm development systems. We would also like to thank the anonymous reviewers for their helpful comments and suggestions.

Chapter 2

Palacios and Kitten: New High Performance Operating Systems For Scalable Virtualized and Native Supercomputing

The content of this chapter was originally published in a paper by the same name at the 2010 International Parallel and Distributed Processing Symposium (IPDPS'10). The proper citation is [45]. This LDRD project began focusing on adding a virtualization layer to the Kitten LWK in FY09, and teamed up with separately funded researchers from Northwestern University and the University of New Mexico who were developing an embeddable virtual machine monitor called Palacios. Together, we embedded Palacios in Kitten, tuned the combination for HPC workloads, and got it running on a 48 node Cray XT4 test system. This paper describes Palacios and Kitten and includes performance results obtained on the test system. A subsequent paper, listed in Chapter 3, describes the optimizations for HPC in more depth and includes results obtained on up to $6,240$ *nodes of the Red Storm Cray XT4 supercomputer.*

The authors of this paper are John Lange (Univ. Pittsburgh, formerly Northwestern Univ.), Kevin Pedretti (SNL), Trammell Hudson (OS Research), Peter Dinda (Northwestern Univ.), Zheng Cui (Univ. New Mexico), Lei Xia (Northwestern Univ.), P. Bridges (Univ. New Mexico), Andy Gocke (Northwestern Univ.), Steven Jaconette (Northwestern Univ.), Michael Levenhagen (SNL), and Ron Brightwell (SNL).

Abstract

Palacios is a new open-source VMM under development at Northwestern University and the University of New Mexico that enables applications executing in a virtualized environment to achieve scalable high performance on large machines. Palacios functions as a modularized extension to Kitten, a high performance operating system being developed at Sandia National Laboratories to support large-scale supercomputing applications. Together, Palacios and Kitten provide a thin layer over the hardware to support full-featured virtualized

environments alongside Kitten's lightweight native environment. Palacios supports existing, unmodified applications and operating systems by using the hardware virtualization technologies in recent AMD and Intel processors. Additionally, Palacios leverages Kitten's simple memory management scheme to enable low-overhead pass-through of native devices to a virtualized environment. We describe the design, implementation, and integration of Palacios and Kitten. Our benchmarks show that Palacios provides near native (within 5%), scalable performance for virtualized environments running important parallel applications. This new architecture provides an incremental path for applications to use supercomputers, running specialized lightweight host operating systems, that is not significantly performance-compromised.[1]

[1]This project is made possible by support from the National Science Foundation (NSF) via grants CNS-0709168, CNS-0707365, and the Department of Energy (DOE) via a subcontract from Oak Ridge National Laboratory on grant DE-AC05-00OR22725. John Lange was partially supported by a Symantec Research Labs Fellowship. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

2.1 Introduction

This paper introduces Palacios, a new high performance virtual machine monitor (VMM) architecture, that has been embedded into Kitten, a high performance supercomputing operating system (OS). Together, Palacios and Kitten provide a flexible, high performance virtualized system software platform for HPC systems. This platform broadens the applicability and usability of HPC systems by:

- providing access to advanced virtualization features such as migration, full system checkpointing, and debugging;
- allowing system owners to support a wider range of applications and to more easily support legacy applications and programming models when changing the underlying hardware platform;
- enabling system users to incrementally port their codes from small-scale development systems to large-scale supercomputer systems while carefully balancing their performance and system software service requirements with application porting effort; and
- providing system hardware and software architects with a platform for exploring hardware and system software enhancements without disrupting other applications.

Palacios is a "type-I" pure VMM [32] under development at Northwestern University and the University of New Mexico that provides the ability to virtualize existing, unmodified applications and their operating systems with no porting. Palacios is designed to be embeddable into other operating systems, and has been embedded in two so far, including Kitten. Palacios makes extensive, non-optional use of hardware virtualization technologies and thus can scale with improved implementations of those technologies.

Kitten is an OS being developed at Sandia National Laboratories that is being used to investigate system software techniques for better leveraging multicore processors and hardware virtualization in the context of capability supercomputers. Kitten is designed in the spirit of *lightweight kernels* [58], such as Sandia's Catamount [42] and IBM's CNK [60], that are well known to perform better than commodity kernels for HPC. The simple framework provided by Kitten and other lightweight kernels facilitates experimentation, has led to novel techniques for reducing the memory bandwidth requirements of intra-node message passing [10], and is being used to explore system-level options for improving resiliency to hardware faults.

Kitten and Palacios together provide a scalable, flexible HPC system software platform that addresses the challenges laid out earlier and by others [49]. Applications ported to Kitten will be able to achieve maximum performance on a given machine. Furthermore, Kitten is itself portable and open, propagating the benefits of such porting efforts to multiple machines. Palacios provides the ability to run existing, unmodified applications and their operating systems, requiring no porting. Furthermore, as Palacios has quite low overhead,

it could potentially be used to manage a machine, allowing a mixture of workloads running on commodity and more specialized OSes, and could even run ported applications on more generic hardware.

Palacios and Kitten can be used separately or together, and run on a variety of machines ranging from commodity clusters and servers to large scale parallel machines at Sandia. Both Palacios and Kitten are open source tools that are currently available to download and use.

In the remainder of this paper, we describe the design and implementation of both Palacios and Kitten, and evaluate their performance. The core contributions of this paper are the following:

- We introduce and describe the Palacios VMM.
- We introduce and describe the Kitten HPC OS.
- We show how the combination of Palacios and Kitten can provide an incremental path to using many different kinds of HPC resources for the mutual benefit of users and machine owners.
- We show that an integrated virtualization system combining Palacios and Kitten can provide nearly native performance for existing codes, even when extensive communication is involved.
- We present evaluations of parallel application and benchmark performance and overheads using virtualization on high-end computing resources. The overheads we see, particularly using hardware nested paging, are typically less than 5%.

2.2 Motivation

Palacios and Kitten are parts of larger projects that have numerous motivations. Here we consider their joint motivation in the context of high performance computing, particularly on large scale machines.

Maximizing performance through lightweight kernels Lightweight compute node OSes maximize the resources delivered to applications to maximize their performance. As such, a lightweight kernel does not implement much of the functionality of a traditional operating system; instead, it provides mechanisms that allow system services to be implemented *outside* the OS, for example in a library linked to the application. As a result, they also require that applications be carefully ported to their minimalist interfaces.

Increasing portability and compatibility through commodity interfaces Standardized application interfaces, for example partial or full Linux ABI compatibility, would

make it easier to port parallel applications to a lightweight kernel. However, a lightweight kernel cannot support the full functionality of a commodity kernel without losing the benefits noted above. This means that some applications cannot be run without modification.

Achieving full application and OS compatibility through virtualization Full system virtualization provides full compatibility at the hardware level, allowing existing unmodified applications and OSes to run. The machine is thus immediately available to be used by any application code, increasing system utilization when ported application jobs are not available. The performance of the full system virtualization implementation (the VMM) partially drives the choice of either using the VMM or porting an application to the lightweight kernel. Lowering the overhead of the VMM, particularly in communication, allows more of the workload of the machine to consist of VMMs.

Preserving and enabling investment in ported applications through virtualization A VMM which can run a lightweight kernel provides straightforward portability to applications where the lightweight kernel is not available natively. Virtualization makes it possible to emulate a large scale machine on a small machine, desktop, or cluster. This emulation ability makes commodity hardware useful for developing and debugging applications for lightweight kernels running on large scale machines.

Managing the machine through virtualization Full system virtualization would allow a site to dynamically configure nodes to run a full OS or a lightweight OS without requiring rebooting the whole machine on a per-job basis. Management based on virtualization would also make it possible to backfill work on the machine using loosely-coupled programming jobs [56] or other low priority work. A batch-submission or grid computing system could be run on a collection of nodes where a new OS stack could be dynamically launched; this system could also be brought up and torn down as needed.

Augmenting the machine through virtualization Virtualization offers the option to enhance the underlying machine with new capabilities or better functionality. Virtualized lightweight kernels can be extended at runtime with specific features that would otherwise be too costly to implement. Legacy applications and OSes would be able to use features such as migration that they would otherwise be unable to support. Virtualization also provides new opportunities for fault tolerance, a critical area that is receiving more attention as the mean time between system failures continues to decrease.

Enhancing systems software research in HPC and elsewhere The combination of Palacios and Kitten provides an open source toolset for HPC systems software research that can run existing codes without the need for victim hardware. Palacios and Kitten enable new systems research into areas such as fault-tolerant system software, checkpointing, overlays, multicore parallelism, and the integration of high-end computing and grid computing.

2.3 Palacios

Palacios[2] is an OS independent VMM designed as part of the the V3VEE project (`http://v3vee.org`). Palacios currently targets the x86 and x86_64 architectures (hosts and guests) and is compatible with both the AMD SVM [4] and Intel VT [38] extensions. Palacios supports both 32 and 64 bit host OSes as well as 32 and 64 bit guest OSes[3]. Palacios supports virtual memory using either shadow or nested paging. Palacios implements full hardware virtualization while providing targeted paravirtualized extensions.

Palacios is a fully original VMM architecture developed at Northwestern University. Figure 2.1 shows the scale of Palacios as of the 1.1 release (and the Kitten 1.1.0 release). Note that the Palacios core is quite small. The entire VMM, including the default set of virtual devices is on the order of 28 thousand lines of C and assembly. The combination of Palacios and Kitten is 89 thousand lines of code. In comparison, Xen 3.0.3 consists of almost 580 thousand lines of which the hypervisor core is 50–80 thousand lines, as measured by the `wc` tool. Similarly, Kernel Virtual Machine (KVM) is massive when its Linux kernel dependencies are considered (a performance comparison with KVM is given in Section 2.6.7). Palacios is publicly available from `http://v3vee.org`, with additional documentation about its theory of operation available in a technical report [46]. Palacios is released under a BSD license.

Palacios supports multiple physical host and virtual guest environments. Palacios is compatible with both AMD SVM and Intel VT architectures, and has been evaluated on commodity Ethernet based servers, a high end Infiniband cluster, as well as Red Storm development cages consisting of Cray XT nodes. Palacios also supports the virtualization of a diverse set of guest OS environments, including commodity Linux and other OS distributions, modern Linux kernels, and several lightweight HPC OSes such as CNL [41], Catamount [42], and Kitten itself.

2.3.1 Architecture

Palacios is an OS independent VMM, and as such is designed to be easily portable to diverse host operating systems. Currently Palacios supports multiple operating systems, but specifically supports Kitten for high performance environments. Palacios integrates with a host OS through a minimal and explicitly defined functional interface that the host OS is responsible for supporting. Furthermore, the interface is modularized so that a host environment can decide its own level of support and integration. Less than 300 lines of code needed to be written to embed Palacios into Kitten. Palacios's architecture, shown in Figure 2.2, is designed to be internally modular and extensible and provides common interfaces for registering event handlers for common operations.

[2]Palacios, TX is the "Shrimp Capital of Texas."
[3]64 bit guests are only supported on 64 bit hosts

Component	Lines of Code
Palacios	
Palacios Core (C+Assembly)	15,084
Palacios Virtual Devices (C)	8,708
Total	28,112
Kitten	
Kitten Core (C)	17,995
Kitten Arch Code (C+Assembly)	14,604
Misc. Contrib Code (Kbuild/lwIP)	27,973
Palacios Glue Module (C)	286
Total	60,858
Grand Total	88,970

Figure 2.1: Lines of code in Palacios and Kitten as measured with the SLOCCount tool.

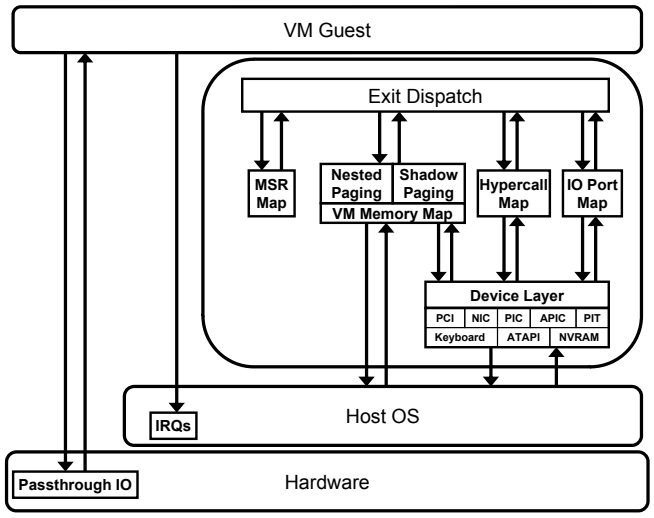


Figure 2.2: Palacios architecture.

Configurability Palacios is designed to be highly modular to support the generation of specialized VMM architectures. The modularity allows VMM features and subsystems to be selected at compile time to generate a VMM that is specific to the target environment. The configuration system is also used to select from the set of available OS interfaces, in order to enable Palacios to run on a large number of OS architectures. The build and compile time configuration system is based on a modified version of KBuild ported from Linux.

Palacios also includes a runtime configuration system that allows guest environments to specifically configure the VMM to suit their environments. Virtual devices are implemented as independent modules that are inserted into a runtime generated hash table that is keyed to a device's ID. The guest configuration also allows a guest to specify core configuration options such as the scheduling quantum and the mechanism used for shadow memory.

The combination of the compile time and runtime configurations make it possible to construct a wide range of guest environments that can be targeted for a large range of host OS and hardware environments.

Resource hooks The Palacios core provides an extensive interface to allow VMM components to register to receive and handle guest and host events. Guest events that can be hooked include accesses to MSRs, IO ports, specific memory pages, and hypercalls. Palacios also includes functionality to receive notifications of host events such as general interrupts, keystrokes and timer ticks.

Palacios interfaces with the host OS through a small set of function hooks that the host OS is required to provide. These functions include methods for allocating and freeing physical memory pages as well as heap memory, address conversion functions for translating physical addresses to the VMMs virtual address space, a function to yield the CPU when a VM is idle, and an interface for interfacing with the host's interrupt handling infrastructure. In addition to this interface, Palacios also includes an optional socket interface that consists of a small set of typical socket functions.

Interrupts Palacios includes two models for hardware interrupts, passthrough interrupts and specific event notifications. Furthermore, Palacios is capable of disabling local and global interrupts in order to have interrupt processing on a core run at times it chooses. The interrupt method used is determined by the virtual device connected to the guest.

For most virtual devices, interrupts are delivered via a host event notification interface. This interface requires the presence of a host OS device driver to handle the interrupt and transfer any data to or from the device. The data from the device operation is then encapsulated inside a host event and delivered to Palacios. The event is then delivered to any virtual devices listening on the notification channel. The virtual device is then responsible for raising virtual interrupts as needed.

For high performance devices, such as network cards, Palacios supports passthrough

operation which allows the guest to interact directly with the hardware. For this mechanism no host OS driver is needed. In this case, Palacios creates a special virtual passthrough device that interfaces with the host to register for a given device's interrupt. The host OS creates a generic interrupt handler that first masks the interrupt pin, acks the interrupt to the hardware interrupt controller, and then raises a virtual interrupt in Palacios. When the guest environment acks the virtual interrupt, Palacios notifies the host, which then unmasks the interrupt pin. This interface allows direct device IO to and from the guest environment with only a small increase to the interrupt latency that is dominated by the hardware's world context switch latency.

2.3.2 Palacios as a HPC VMM

Part of the motivation behind Palacios's design is that it be well suited for high performance computing environments, both on the small scale (e.g., multicores) and large scale parallel machines. Palacios is designed to interfere with the guest as little as possible, allowing it to achieve maximum performance.

Palacios is currently designed for distributed memory parallel computing environments. This naturally maps to conventional cluster and HPC architectures. Multicore CPUs are currently virtualized as a set of independent compute nodes that run separate guest contexts. Support for single image multicore environments (i.e., multicore guests) is currently under development.

Several aspects of Palacios's design are suited for HPC:

- Minimalist interface: Palacios does not require extensive host OS features, which allows it to be easily embedded into even small kernels, such as Kitten and GeekOS [36].
- Full system virtualization: Palacios does not require guest OS changes. This allows it to run existing kernels without any porting, including Linux kernels and whole distributions, and lightweight kernels [58] like Kitten, Catamount, Cray CNL [41] and IBM's CNK [60].
- Contiguous memory preallocation: Palacios preallocates guest memory as a physically contiguous region. This vastly simplifies the virtualized memory implementation, and provides deterministic performance for most memory operations.
- Passthrough resources and resource partitioning: Palacios allows host resources to be easily mapped directly into a guest environment. This allows a guest to use high performance devices, with existing device drivers, with no virtualization overhead.
- Low noise: Palacios minimizes the amount of OS noise [27] injected by the VMM layer. Palacios makes no use of internal timers, nor does it accumulate deferred work.
- Extensive compile time configurability: Palacios can be configured with a minimum set of required features to produce a highly optimized VMM for specific environments.

This allows lightweight kernels to include only the features that are deemed necessary and remove any overhead that is not specifically needed.

2.4 Kitten

Kitten is an open-source OS designed specifically for high performance computing. It employs the same "lightweight" philosophy as its predecessors—SUNMOS, Puma, Cougar, and Catamount[4]—to achieve superior scalability on massively parallel supercomputers while at the same time exposing a more familiar and flexible environment to application developers, addressing one of the primary criticisms of previous lightweight kernels. Kitten provides partial Linux API and ABI compatibility so that standard compiler tool-chains and system libraries (e.g., Glibc) can be used without modification. The resulting ELF executables can be run on either Linux or Kitten unchanged. In cases where Kitten's partial Linux API and ABI compatibility is not sufficient, the combination of Kitten and Palacios enables unmodified guest OSes and applications to be loaded on demand.

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 for Kitten). Performance critical subsystems, such as memory management and task scheduling, are replaced with code written from scratch for Kitten. To avoid potential licensing issues, no code from prior Sandia-developed lightweight kernels is used. Like Linux, the Kitten code base is structured to facilitate portability to new architectures. Currently only the x86_64 architecture is officially supported, but NEC has recently ported Kitten to the NEC SX vector architecture for research purposes[30]. Kitten is publicly available from `http://software.sandia.gov/trac/kitten` and is released under the terms of the GNU Public License (GPL) version 2.

2.4.1 Architecture

Kitten (Figure 2.3) is a monolithic kernel that runs symmetrically on all processors in the system. Straightforward locking techniques are used to protect access to shared data structures. At system boot-up, the kernel enumerates and initializes all hardware resources (processors, memory, and network interfaces) and then launches the initial user-level task, which runs with elevated privilege (the equivalent of root). This process is responsible for interfacing with the outside world to load jobs onto the system, which may either be native Kitten applications or guest operating systems. The Kitten kernel exposes a set of resource management system calls that the initial task uses to create virtual address spaces, allocate physical memory, create additional native Kitten tasks, and launch guest operating systems.

The Kitten kernel supports a subset of the Linux system call API and adheres to the

[4]The name Kitten continues the cat naming theme, but indicates a new beginning.

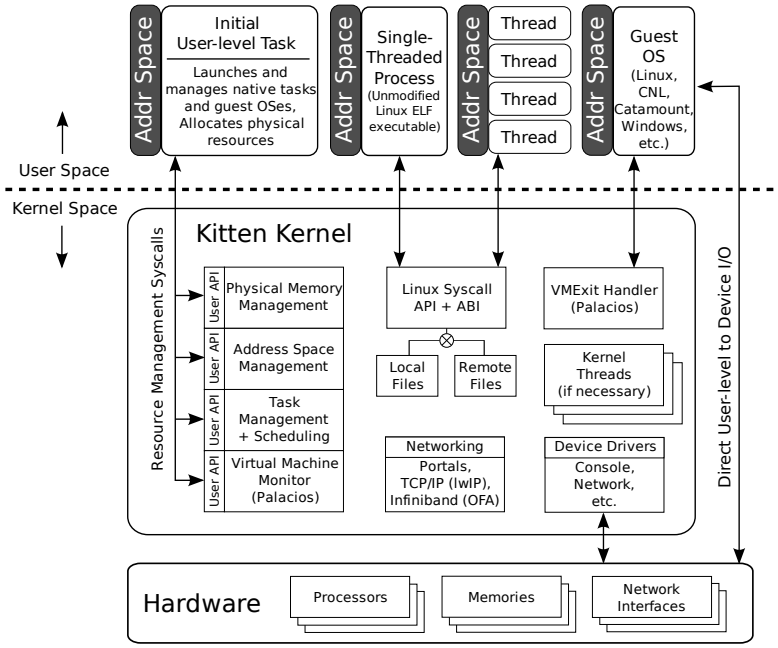


Figure 2.3: Kitten architecture.

Linux ABI to support native user-level tasks. Compatibility includes system call calling conventions, user-level stack and heap layout, thread-local storage conventions, and a variety of standard system calls such as `read()`, `write()`, `mmap()`, `clone()`, and `futex()`. The subset of system calls that Kitten implements natively is intended to support the requirements of existing scalable scientific computing applications in use at Sandia. The subset is also sufficient to support Glibc's NPTL POSIX threads implementation and GCC's OpenMP implementation without modification. Implementing additional system calls is a relatively straightforward process.

The Kitten kernel contains functionality aimed at easing the task of porting of Linux device drivers to Kitten. Many device drivers and user-level interface libraries create or require local files under `/dev`, `/proc`, and `/sys`. Kitten provides limited support for such files. When a device driver is initialized, it can register a set of callback operations to be used for a given file name. The `open()` system call handler then inspects a table of the registered local file names to determine how to handle each open request. Remote files are forwarded to a user-level proxy task for servicing. Kitten also provides support for kernel threads, interrupt registration, and one-shot timers since they are required by many Linux drivers. The Open Fabrics Alliance (OFA) Infiniband stack was recently ported to Kitten without making any significant changes to the OFA code.

2.4.2 Memory Management

Unlike traditional general-purpose kernels, Kitten delegates most virtual and physical memory management to user-space. The initial task allocates memory to new native applications and Palacios virtual machines by making a series of system calls to create an address space, create virtual memory regions, and bind physical memory to those regions. Memory topology information (i.e., NUMA) is provided to the initial-task so it can make intelligent decisions about how memory should be allocated.

Memory is bound to a context of execution before it starts executing and a contiguous linear mapping is used between virtual and physical addresses. The use of a regular mapping greatly simplifies virtual to physical address translation compared to demand-paged schemes, which result in an unpredictable mapping with complex performance implications. Networking hardware and software can take advantage of the simple mapping to increase performance (which is the case on Cray XT) and potentially decrease cost by eliminating the need for translation table memory and table walk hardware on the network interface. The simple mapping also enables straightforward pass-through of physical devices to para-virtualized guest drivers.

2.4.3 Task Scheduling

All contexts of execution on Kitten, including Palacios virtual machines, are represented by a task structure. Tasks that have their own exclusive address space are considered processes and tasks that share an address space are threads. Processes and threads are identical from a scheduling standpoint. Each processor has its own run queue of ready tasks that are preemptively scheduled in a round-robin fashion. Currently Kitten does not automatically migrate tasks to maintain load balance. This is sufficient for the expected common usage model of one MPI task or OpenMP thread per processor.

The privileged initial task that is started at boot time allocates a set of processors to each user application task (process) that it creates. An application task may then spawn additional tasks on its set of processors via the `clone()` system call. By default spawned tasks are spread out to minimize the number of tasks per processor but a Kitten-specific task creation system call can be used to specify the exact processor that a task should be spawned on.

2.5 Integrating Palacios and Kitten

Palacios was designed to be easily integrated with different operating systems. This leads to an extremely simple integration with Kitten consisting of an interface file of less than 300 lines of code. The integration includes no internal changes in either Kitten or Palacios, and the interface code is encapsulated with the Palacios library in an optional compile time

module for Kitten. This makes Palacios a natural virtualization solution for Kitten when considered against existing solutions that target a specific OS with extensive dependencies on internal OS infrastructures.

Kitten exposes the Palacios control functions via a system call interface available from user space. This allows user level tasks to instantiate virtual machine images directly from user memory. This interface allows VMs to 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.

SeaStar Passthrough Support Because Palacios provides support for passthrough I/O, it is possible to support high performance, partitioned access to particular communication devices. We do this for the SeaStar communication hardware on the Red Storm machine. The SeaStar is a high performance network interface that utilizes the AMD HyperTransport Interface and proprietary mesh interconnect for data transfers between Cray XT nodes [14]. At the hardware layer the data transfers take the form of arbitrary physical-addressed DMA operations. To support a virtualized SeaStar the physical DMA addresses must be translated from the guest's address space. However, to ensure high performance the SeaStar's command queue must be directly exposed to the guest. This requires the implementation of a simple high performance translation mechanism. Both Palacios and Kitten include a simple memory model that makes such support straightforward.

The programmable SeaStar architecture provides several possible avenues for optimizing DMA translations. These include a self-virtualizable firmware as well as an explicitly virtualized guest driver. In the performance study we conducted for this paper we chose to modify the SeaStar driver running in the guest to support Palacios's passthrough I/O. This allows the guest to have exclusive and direct access to the SeaStar device. Palacios uses the large contiguous physical memory allocations supported by Kitten to map contiguous guest memory at a known offset. The SeaStar driver has a tiny modification that incorporates this offset into the DMA commands sent to the SeaStar. This allows the SeaStar to execute actual memory operations with no performance loss due to virtualization overhead. Because each Cray XT node contains a single SeaStar device, the passthrough configuration means that only a single guest is capable of operating the SeaStar at any given time.

Besides memory-mapped I/O, the SeaStar also directly uses an APIC interrupt line to notify the host of transfer completions as well as message arrivals. Currently, Palacios exits from the guest on all interrupts. For SeaStar interrupts, we immediately inject such interrupts into the guest and resume. While this introduces an VM exit/entry cost to each SeaStar interrupt, in practice this only results in a small increase in latency. We also note that the SeaStar interrupts are relatively synchronized, which does not result in a significant increase in noise. We are investigating the use of next generation SVM hardware that supports selective interrupt exiting to eliminate this already small cost.

While implicitly trusting guest environments to directly control DMA operations is not possible in normal environments, the HPC context allows for such trust.

Infiniband and Ethernet Passthrough Support Our integration of Palacios and Kitten also includes an implementation of passthrough I/O for Mellanox Infiniband NICs and Intel E1000 Ethernet NICs. These are similar to the SeaStar implementation.

2.6 Performance

We conducted a careful performance evaluation of the combination of Palacios and Kitten on diverse hardware, and at scales up to 48 nodes. We focus the presentation of our evaluation on the Red Storm machine and widely recognized applications/benchmarks considered critical to its success. As far as we are aware, ours is the largest scale evaluation of parallel applications/benchmarks in virtualization to date, particularly for those with significant communication. It also appears to be the first evaluation on petaflop-capable hardware. Finally, we show performance numbers for native lightweight kernels, which create a very high bar for the performance of virtualization. The main takeaways from our evaluation are the following.

1. The combination of Palacios and Kitten is generally able to provide near-native performance. This is the case even with large amounts of complex communication, and even when running guest OSes that themselves use lightweight kernels to maximize performance.

2. It is generally preferable for a VMM to use nested paging (a hardware feature of AMD SVM and Intel VT) over shadow paging (a software approach) for guest physical memory virtualization. However, for guest OSes that use simple, high performance address space management, such as lightweight kernels, shadow paging can sometimes be preferable due to its being more TLB-friendly.

The typical overhead for virtualization is $\leq 5\%$.

2.6.1 Testbed

We evaluated the performance and scaling of Palacios running on Kitten on the development system *rsqual*, part of the Red Storm machine at Sandia National Laboratories. Each XT4 node on this machine contains a quad-core AMD Budapest processor running at 2.2 GHz with 4 GB of RAM. The nodes are interconnected with a Cray SeaStar 2.2 mesh network [14]. Each node can simultaneously send and receive at a rate of 2.1 GB/s via MPI. The measured node to node MPI-level latency ranges from 4.8 μsec (using the Catamount [42] operating system) to 7.0 μsec (using the native CNL [41] operating system). As we stated earlier, even though we can run multiple guests on a multicore Cray XT node by instantiating them on separate cores, our current implementation only allows the SeaStar to be exposed to a single

guest context. Due to this limitation, our performance evaluation is restricted to a single guest per Cray XT node.

In addition, we used two dual-socket quad-core 2.3 GHz AMD Shanghai systems with 32GB of memory for communication benchmark testing on commodity HPC hardware. Nodes in this system are connected with Mellanox ConnectX QDR Infiniband NICs and a Mellanox Infiniscale-IV 24 port switch. When not running Kitten, these systems run Linux 2.6.27 and the OpenFabrics 1.4 Infiniband stack.

All benchmark timing in this paper is done using the AMD cycle counter. When virtualization is used, the cycle counter is direct mapped to the guest and not virtualized. Every benchmark receives the same accurate view of the passage of real time regardless of whether virtualization is in use or not.

2.6.2 Guests

We evaluated Palacios running on Kitten with two guest environments:

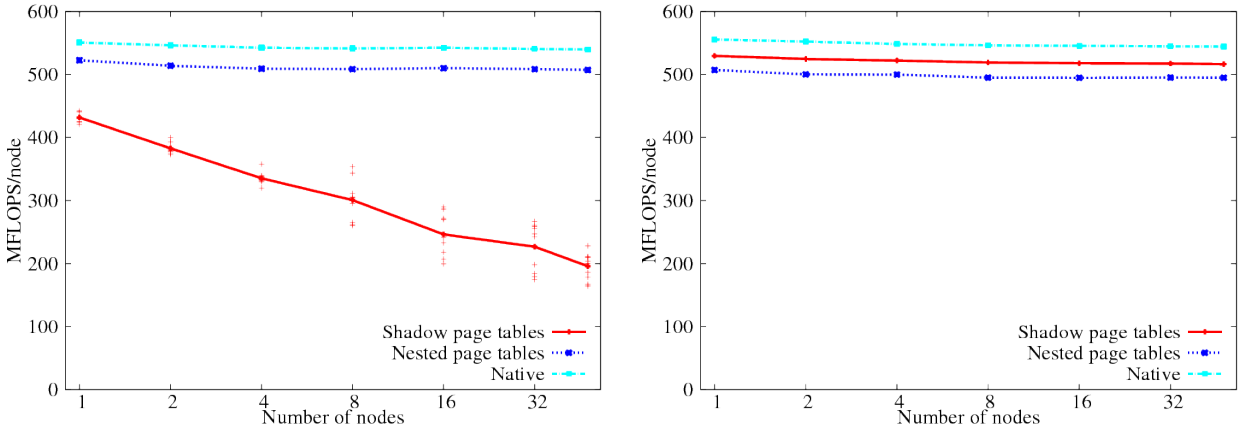
- Cray Compute Node Linux (CNL). This is Cray's stripped down Linux operating system customized for Cray XT hardware. CNL is a minimized Linux (2.6 kernel) that leverages BusyBox [68] and other embedded OS tools/mechanism. This OS is also known as Unicos/LC and the Cray Linux Environment (CLE).

- Catamount. Catamount is a lightweight kernel descended from the SUNMOS and PUMA operating systems developed at Sandia National Labs and the University of New Mexico [62][5]. These OSes, and Catamount, were developed, from-scratch, in reaction to the heavyweight operating systems for parallel computers that began to proliferate in the 1990s. Catamount provides a simple memory model with a physically-contiguous virtual memory layout, parallel job launch, and message passing facilities.

We also use Kitten as a guest for our Infiniband tests. It is important to note that Palacios runs a much wider range of guests than reported in this evaluation. Any modern x86 or x86_64 guest can be booted.

2.6.3 HPCCG Benchmark Results

We used the HPCCG benchmark to evaluate the impact of virtualization on application performance and scaling. HPCCG [35] is a simple conjugate gradient solver that represents an important workload for Sandia. It is commonly used to characterize the performance of new hardware platforms that are under evaluation. The majority of its runtime is spent in a sparse matrix-vector multiply kernel.



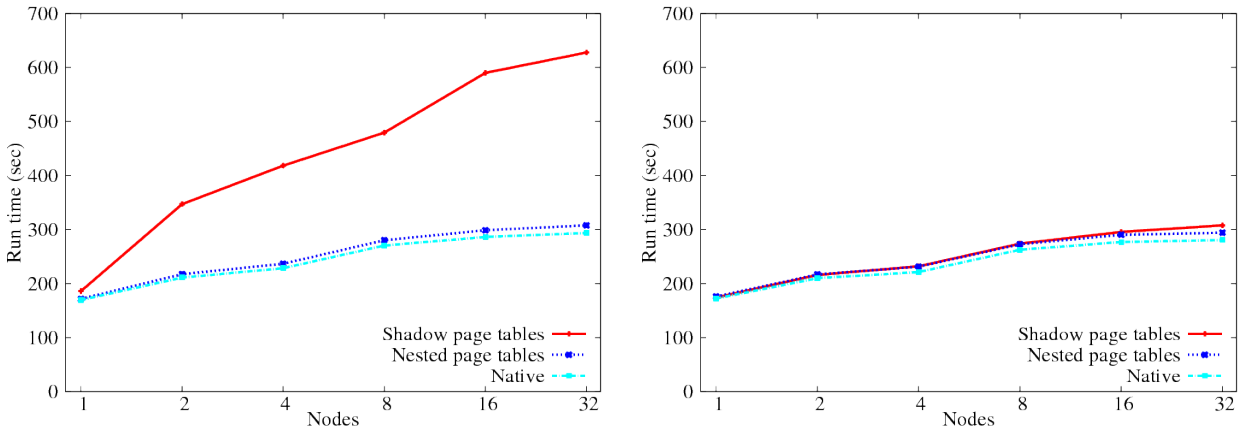
(a) CNL Guest (b) Catamount Guest

Figure 2.4: HPCCG benchmark comparing scaling for virtualization with shadow paging, virtualization with nested paging, and no virtualization. Palacios/Kitten can provide scaling to 48 nodes with less than 5% performance degradation.

We ran HPCCG on top of CNL and Catamount on Red Storm, considering scales from 1 to 48 nodes. A fixed-size problem per node was used to obtain these results. The specific HPCCG input arguments were "100 100 100", requiring approximately 380 MB per node. This software stack was compiled with the Portland Group pgicc compiler version 7, and was run both directly on the machine and on top of Palacios. Both shadow paging and nested paging cases were considered. Communication was done using the passthrough-mapped SeaStar interface, as described earlier.

Figures 2.4a and 2.4b show the results for CNL and Catamount guests. Each graph compares the performance and scaling of the native OS, the virtualized OS with shadow paging, and the virtualized OS with nested paging. The graph shows both the raw measurements of multiple runs and the averages of those runs. The most important result is that the overhead of virtualization is less than 5% and this overhead remains essentially constant at the scales we considered, despite the growing amount of communication. Note further that the variance in performance for both native CNL and virtualized CNL (with nested paging) is minuscule and independent of scale. For Catamount, all variances are tiny and independent, even with shadow paging.

The figure also illustrates the relative effectiveness of Palacios's shadow and nested paging approaches to virtualizing memory. Clearly, nested paging is preferable for this benchmark running on a CNL guest, both for scaling and for low variation in performance. There are two effects at work here. First, shadow paging results in more VM exits than nested paging. On a single node, this overhead results in a 13% performance degradation compared to native performance. The second effect is that the variance in single node performance compounds as we scale, resulting in an increasing performance difference.



(a) CNL Guest (b) Catamount Guest

Figure 2.5: CTH application benchmark comparing scaling for virtualization with shadow paging, virtualization with nested paging, and no virtualization. Palacios/Kitten can provide scaling to 32 nodes with less than 5% performance degradation.

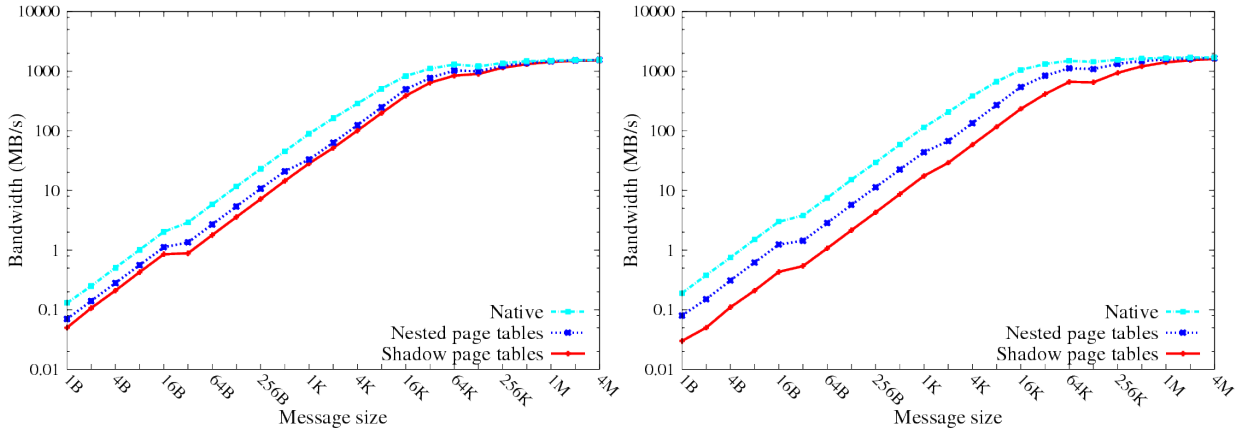
Surprisingly, shadow paging is slightly preferable to nested paging for the benchmark running on the Catamount guest. In Catamount the guest page tables change very infrequently, avoiding the exits for shadow page table refills that happen with CNL. Additionally, instead of the deep nested page walk ($O(nm)$ for n-deep guest and m-deep host page tables) needed on a TLB miss with nested pages, only a regular m-deep host page table walk occurs on a TLB miss with shadow paging. These two effects explain the very different performance of shadow and nested paging with CNL and Catamount guests.

It is important to point out that the version of Palacios's shadow paging implementation we tested only performs on demand updates of the shadow paging state. With optimizations, such as caching, the differences between nested and shadow paging are likely to be smaller.

2.6.4 CTH Application Benchmark

CTH [25] is a multi-material, large deformation, strong shock wave, solid mechanics code developed by Sandia National Laboratories with models for multi-phase, elastic viscoplastic, porous, and explosive materials. CTH supports three-dimensional rectangular meshes; two-dimensional rectangular, and cylindrical meshes; and one-dimensional rectilinear, cylindrical, and spherical meshes, and uses second-order accurate numerical methods to reduce dispersion and dissipation and to produce accurate, efficient results. It is used for studying armor/anti-armor interactions, warhead design, high explosive initiation physics, and weapons safety issues.

Figures 2.5a and 2.5b show the results using the CNL and Catamount guests. We can see that adding virtualization, provided the appropriate choice of shadow or nested paging



(a) CNL Guest (b) Catamount Guest

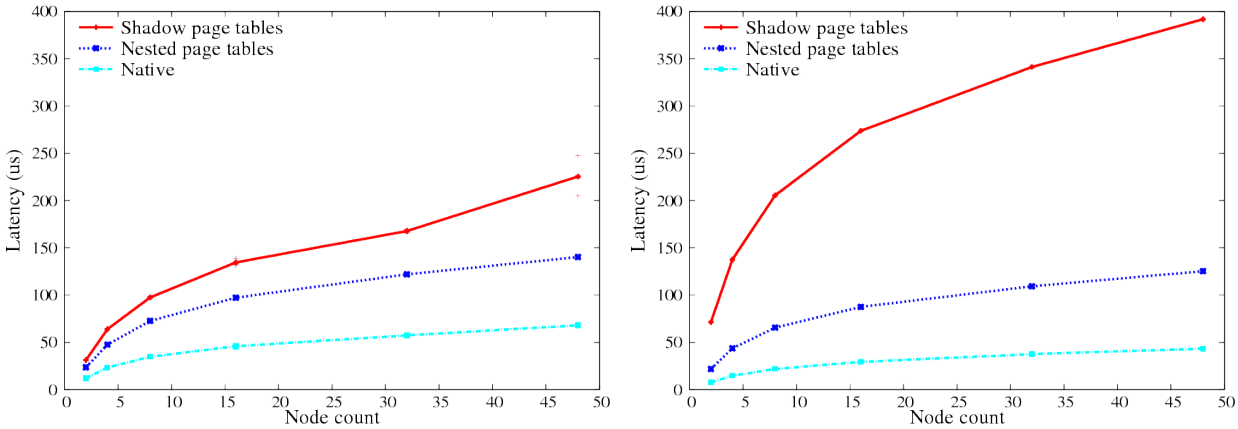
Figure 2.6: IMB PingPong Bandwidth in MB/sec as a function of message size

is made, has virtually no effect on performance or scaling. For this highly communication intensive benchmark, virtualization is essentially free.

2.6.5 Intel MPI Benchmarks

The Intel MPI Benchmarks (IMB) [39], formerly known as PALLAS, are designed to characterize the MPI communication performance of a system. IMB employs a range of MPI primitive and collective communication operations, at a range of message sizes and scales to produce numerous performance characteristics. We ran IMB on top of CNL and Catamount on Red Storm using SeaStar at scales from 2 to 48 nodes. We compared native performance, virtualized performance using shadow paging, and virtualized performance using nested paging. IMB generates large quantities of data. Figures 2.6 through 2.7 illustrate the most salient data on CNL and Catamount.

Figure 2.6 shows the bandwidth of a ping-pong test between two nodes for different message sizes. For large messages, bandwidth performance is identical for virtualized and native operating systems. For small messages where ping-pong bandwidth is latency-bound, the latency costs of virtualization reduce ping-pong bandwidth. We have measured the extra latency introduced by virtualization as either 5 μsec (nested paging) or 11 μsec (shadow paging) for the CNL guest. For the Catamount guest, shadow paging has a higher overhead. Although the SeaStar is accessed via passthrough I/O, interrupts are virtualized. When the SeaStar raises an interrupt, a VM exit is induced. Palacios quickly transforms the hardware interrupt into a virtual interrupt that it injects into the guest on VM entry. The guest will quickly cause another VM exit/entry interaction when it acknowledges the interrupt to its (virtual) APIC. Shadow paging introduces additional overhead because of the need to refill the TLB after these entries/exits. This effect is especially pronounced in Catamount since, other than capacity misses, there is no other reason for TLB refills; in addition, Catamount



(a) CNL Guest (b) Catamount Guest

Figure 2.7: IMB Allreduce 16 byte latency in μsec as a function of nodes up to 48 nodes

has a somewhat more complex interrupt path that causes two additional VM exits per interrupt. Avoiding all of these VM exits via nested paging allows us to measure the raw overhead of the interrupt exiting process.

In Figure 2.7, we fix the message size at 16 bytes and examine the effect on an IMB All-Reduce as we scale from 2 to 48 nodes. We can see that the performance impacts of nested and shadow paging diverges as we add more nodes—nested paging is superior here.

The upshot of these figures and the numerous IMB results which we have excluded for space reasons is that the performance of a passthrough device, such as the SeaStar, in Palacios is in line with the expected hardware overheads due to interrupt virtualization. This overhead is quite small. Virtualized interrupts could be avoided using the AMD SVM interrupt handling features, which we expect would bring IMB performance with nested paging-based virtualization in line with native performance. However, at this point, we expect that doing so would require minor guest changes.

2.6.6 Infiniband microbenchmarks

To quantify the overhead of Palacios virtualization on a commodity NIC, we ported OpenIB MLX4 (ConnectX) drivers to Kitten along with the associated Linux driver. We also implemented passthrough I/O support for these drivers in Palacios. We then measured round-trip latency for 1 byte messages averaged over 100000 round trips and 1 megabyte message round trip bandwidth averaged over 10000 trips using a ported version of the OpenFabrics `ibv_rc_pingpong`. The server system ran Linux 2.6.27, while the client machine ran either Kitten natively, Kitten as a guest on Palacios using shadow paging, or Linux.

As can be seen in Figure 3.8, Palacios's pass-through virtualization imposes almost no

	Latency (μsec)	Bandwidth (Gb/sec)
Kitten (Native)	5.24	12.40
Kitten (Virtualized)	5.25	12.40
Linux	4.28	12.37

Figure 2.8: Bandwidth and latency of node-to-node Infiniband on Kitten, comparing native performance with guest performance. Linux numbers are provided for reference.

	HPCCG MFLOPS
Native CNL	588.0
Palacios/Kitten + CNL Guest	556.4
KVM/CNL + CNL Guest	546.4
% Diff Palacios vs. KVM	1.8%

Figure 2.9: Comparison of Palacios to KVM for HPCCG benchmark.

measurable overhead on Infiniband message passing. Compared to Linux, Kitten both native and virtualized using Palacios slightly outperform Linux in terms of end-to-end bandwidth, but suffers a 1 μsec/round trip latency penalty. We believe this is due to a combination of the lack of support for message-signaled interrupts (MSI) in our current Linux driver support code, as well as our use of a comparatively old version of the OpenIB driver stack. We are currently updating Linux driver support and the OpenIB stack used in in Kitten to address this issue.

2.6.7 Comparison with KVM

To get a feel for the overhead of Palacios compared to existing virtualization platforms, we ran the HPCCG benchmark in a CNL guest under both KVM running on a Linux host and Palacios running on a Kitten host. KVM (Kernel-based Virtual Machine) is a popular virtualization platform for Linux that is part of the core Linux kernel as of version 2.6.20. Due to time constraints we were not able to expose the SeaStar to KVM guest environments, so only single node experiments were performed. The same "100 100 100" test problem that was used in Section 2.6.3 was run on a single Cray XT compute node. HPCCG was compiled in serial mode (non-MPI) leading to slightly different performance results. As can be seen in Figure 2.9, Palacios delivers approximately 1.8% better performance than KVM for this benchmark. Each result is an average of three trials and has a standard deviation less of than 0.66. Note that small performance differences at the single node level typically magnify as the application and system are scaled up.

2.7 Future Work

Larger Scale Studies While our initial results show that Palacios and Kitten are capable of providing scalable virtualization for HPC environments, we intend to continue the evaluation at ever larger scales. We have completed a preliminary large scale study of up to 4096 nodes on the full Red Storm system. The preliminary results show that Palacios continues to impose minimal overhead, delivering performance within 5% as scaling increases.

Symbiotic Virtualization Based on our results to date, it is evident that the best VMM configuration is heavily dependent on the OS and application behavior inside the guest environment. In other words, *there is no singular VMM configuration suitable for HPC environments*. In order to provide the best performance for every HPC application, a VMM must be able to adapt its own behavior to the guest's. This adaptability requires that both the VMM and OS cooperate to coordinate their actions. *Symbiotic virtualization* is a new approach to system virtualization where a guest OS and a VMM use high level software interfaces to communicate with each other in order to increase performance and functionality. We are currently exploring the use of *Symbiotic Virtualization* for HPC environments.

2.8 Related Work

Recent research activities on operating systems for large-scale supercomputers generally fall into two categories: those that are Linux-based and those that are not. A number of research projects are exploring approaches for configuring and adapting Linux to be more lightweight. Alternatively, there are a few research projects investigating non-Linux approaches, using either custom lightweight kernels or adapting other existing open-source OSes for HPC.

The Cray Linux Environment [41] is the most prominent example of using a stripped-down Linux system in an HPC system, and is currently being used on the petaflop-class Jaguar system at Oak Ridge National Laboratories. Other examples of this approach are the efforts to port Linux to the IBM BlueGene/L and BlueGene/P systems [61, 6]. Since a full Linux distribution is not used, this approach suffers many of the same functionality weaknesses as non-Linux approaches. In some cases, these systems have also encountered performance issues, for example due to the mismatch between the platform's memory management hardware and the Linux memory management subsystem.

Examples of the non-Linux approach include IBM's Compute Node Kernel (CNK) [51] and several projects being led by Sandia, including the Catamount [58] and Kitten projects as well as an effort using Plan9 [50]. Both CNK and Kitten address one of the primary weaknesses of previous lightweight operating systems by providing an environment that is largely compatible with Linux. Kitten differs from CNK in that it supports commodity x86_64 hardware, is being developed in the open under the GPL license, and provides the ability to run full-featured guest operating systems when linked with Palacios.

The desire to preserve the benefits of a lightweight environment but provide support a richer feature set has also led other lightweight kernel developers to explore more full-featured alternatives [60]. We have also explored other means of providing a more full-featured set of system services [67], but the complexity of building a framework for application-specific OSes is significantly greater than simply using an existing full-featured virtualized OS, especially if the performance impact is minimal.

There has been considerable interest, both recently and historically, in applying existing virtualization tools to HPC environments [59, 24, 31, 37, 65, 66, 70]. However, most of the recent work has been exclusively in the context of adapting or evaluating Xen and Linux on cluster platforms. Palacios and Kitten are a new OS/VMM solution developed specifically for HPC systems and applications. There are many examples of the benefits available from a virtualization layer [52] for HPC. There is nothing inherently restrictive about the virtualization tools used for these implementations, so these approaches could be directly applied to this work.

2.9 Conclusion

Palacios and Kitten are new open source tools that support virtualized and native supercomputing on diverse hardware. We described the design and implementation of both Palacios and Kitten, and evaluated their performance. Virtualization support, such as Palacios's, that combines hardware features such as nested paging with passthrough access to communication devices can support even the highest performing guest environments with minimal performance impact, even at relatively large scale. Palacios and Kitten provide an incremental path to using supercomputer resources that has few compromises for performance. Our analysis points the way to eliminating overheads that remain.

Chapter 3

Minimal-overhead Virtualization of a Large Scale Supercomputer

The content of this chapter was submitted for consideration to the 2011 ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'2011). The paper describes the optimizations incorporated into the virtualization layer for HPC workloads, and includes performance results obtained on up to $6,240$ *nodes of the Red Storm Cray XT4 supercomputer. We believe this to be the largest study of its kind by at least two orders of magnitude. The results confirm that the small-scale results presented in Chapter 2 continue to scale, and that* $< 5\%$ *virtualization overhead is achievable even for real, communication intensive applications.*

The authors of this paper are John Lange (Univ. Pittsburgh, formerly Northwestern Univ.), Kevin Pedretti (SNL), Peter Dinda (Northwestern Univ.), Patrick Bridges (Univ. New Mexico), Chang Bae (Northwestern Univ.), Philip Soltero (Univ. New Mexico), and Alexander Merritt (Georgia Tech).

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 with $\leq$5% overhead on a range of application- and microbenchmarks 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.

3.1 Introduction

Virtualization has the potential to dramatically increase the usability and reliability of high performance computing (HPC) systems [37, 49, 31, 53]. However, HPC systems are characterized by an overriding focus on performance. Virtualization cannot succeed in HPC systems unless the performance overheads are truly minimal and 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 minimizes overheads and asynchronous OS interference (OS noise) [28, 55], 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 can be achieved in a high-end message-passing parallel supercomputer, in this case a Cray XT4 supercomputer [3] at scales in excess of 4096 nodes. For guests, we examined the behavior of both the custom Catamount HPC OS [42] and the Cray CNL guest [41], 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 [8]. 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 keeping 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.

3.2 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 sytems and their interaction is available elsewhere [8]. We now summarize these systems.

3.2.1 Palacios

Palacios is a publicly available, open source, OS-independent VMM developed from scratch that targets the x86 and x86_64 architectures (hosts and guests) with either AMD SVM [4] or Intel VT [38] extensions. It is designed to be embeddable into diverse host OSes. 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.

3.2.2 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 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.

3.3 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, 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.

3.3.1 Hardware platform

Testing was performed during an eight hour window of dedicated system time on 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.

3.3.2 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 [42] running on the bare hardware. Some tests were also run, at much smaller scales, using Cray's Linux-derived CNL [41] 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 [7]. 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 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 [13]. 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 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 more recent versions of AMD's SVM hardware (and of Intel's VT hardware) can support much more selective interrupt exiting. If such hardware were available, we would 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 3.4.

[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.

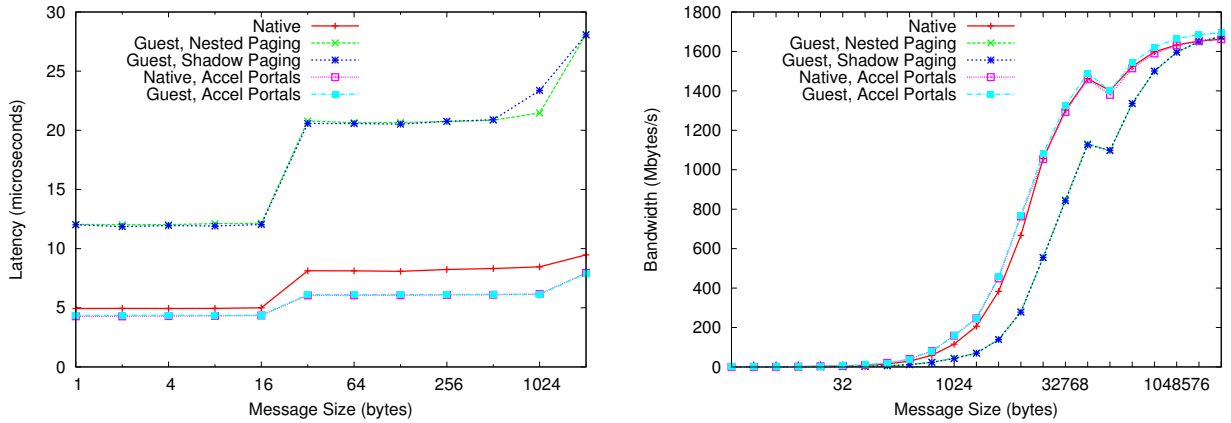
3.3.3 MPI microbenchmarks

The Intel MPI Benchmark Suite version 3.0 [39] was used to evaluate point-to-point messaging performance and scalability of collective operations.

Point-to-point performance

Figure 3.1 shows the results of a ping-pong test between two adjacent nodes. Small message latency, shown in Figure 3.1a, is approximately 2.5 times worse with nested or shadow guest environments compared to native. 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 3.1b 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 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.



(a) Latency (b) Bandwidth

Figure 3.1: MPI PingPong microbenchmark measuring (a) latency and (b) bandwidth

Collective performance

Figures 3.2, 3.3, and 3.4 show the performance of the MPI Barrier, Allreduce, 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 and Allreduce 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.

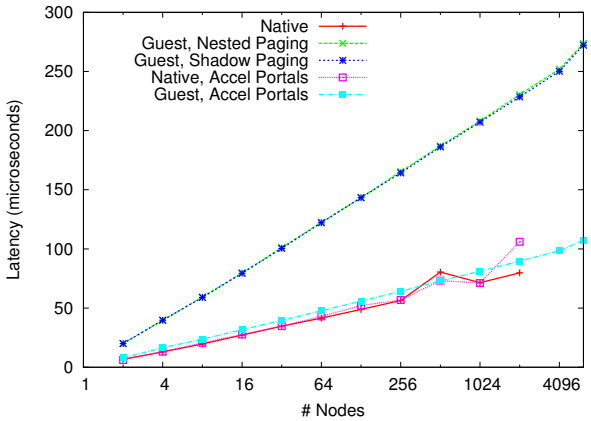


Figure 3.2: MPI barrier scaling microbenchmark results measuring the latency of a full barrier.

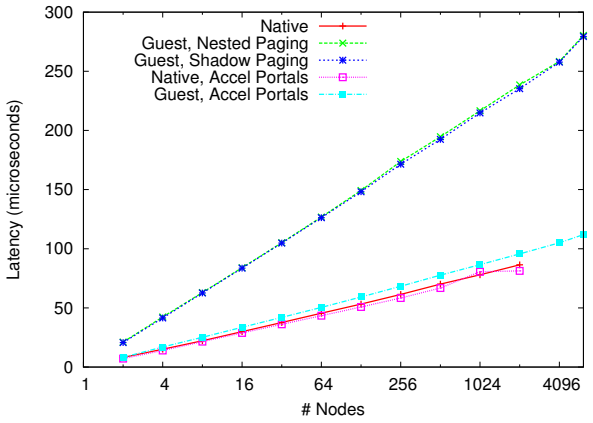


Figure 3.3: MPI all-reduce scaling microbenchmark results measuring the latency of a 16 byte all-reduce operation.

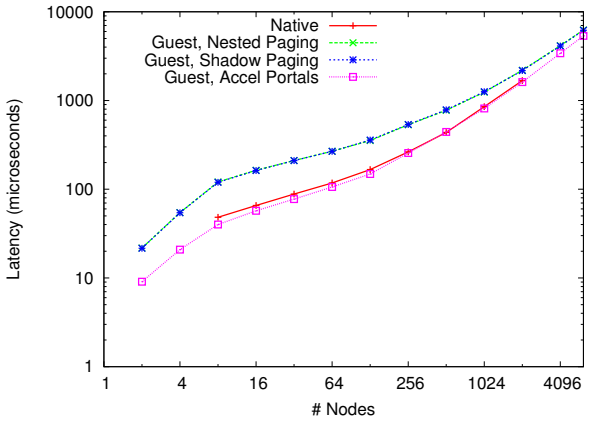


Figure 3.4: MPI all-to-all scaling microbenchmark results measuring the latency of a 16 byte all-to-all operation.

3.3.4 HPCCG application

HPCCG [35] 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" subproblem 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. 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 3.5, 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.

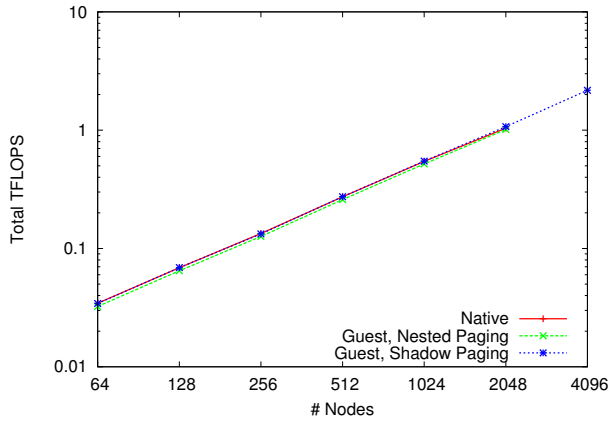


Figure 3.5: HPCCG application benchmark performance. Weak scaling is measured. Virtualized performance is within 5% of native.

3.3.5 CTH application

CTH [25] 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 [8], 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 3.6 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.

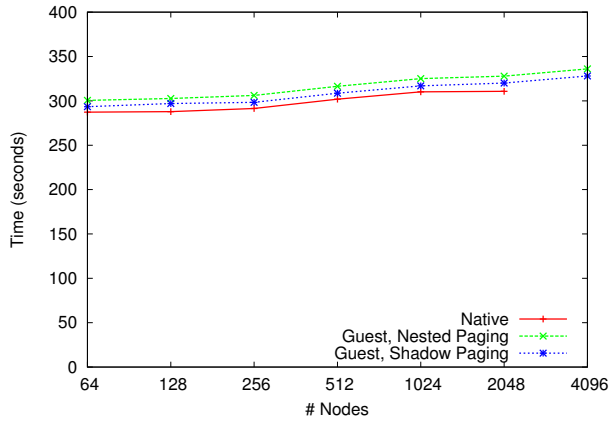


Figure 3.6: CTH application benchmark performance. Weak scaling is measured. Virtualized performance is within 5% of native.

3.3.6 SAGE application

SAGE (SAIC's Adaptive Grid Eulerian hydrocode) is a multidimensional hydrodynamics code with adaptive mesh refinement [44]. The timing_c input deck was used to perform a weak scaling study. As reported in [8], 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 3.7 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.

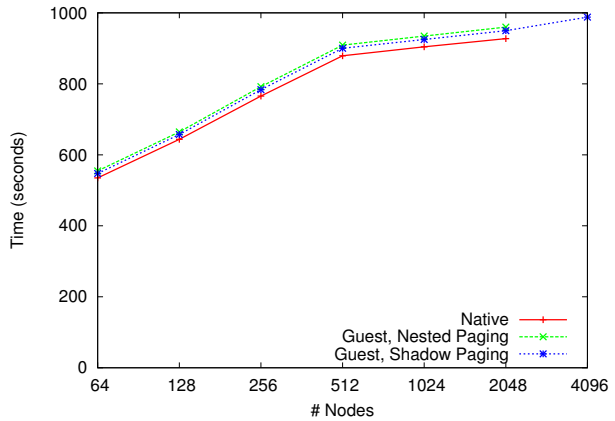


Figure 3.7: Sage application benchmark performance. Weak scaling is measured. Virtualized performance is within 5% of native.

3.4 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 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. Node here refers to some specific partition of the physical resources, be that a single CPU core, a single multicore CPU, or a collection of multicore CPUs. 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-scal 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.

3.4.1 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 instead 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 is impossible for it to 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 [47], as designed for the Xen Hypervisor, does not provide the same guarantees in terms of address space contiguousness. 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 [57] is a technique to allow device sharing without the need for a separate virtual driver domain. While self virtualization does permit direct guest interaction with hardware devices, it does so via a simplified virtual interface which places limits on the usable capabilities of the device. This approach 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 [69] 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. Such an approach 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.

3.4.2 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 3.7). 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 perform a sanity check to ensure that the calculated DMA address resides inside the guests memory space, 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.

3.4.3 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.

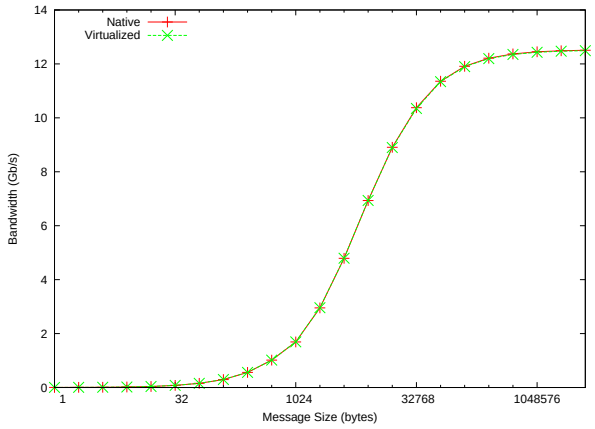


Figure 3.8: Infiniband bandwidth at message sizes from 1 byte to 4 megabytes averaged over 10000 iteration per sample. 1-byte round-trip latency both native and virtualized was identical at6.46μsec, with peak bandwidth for 4MB messages at 12.49 Gb/s on Linux virtualized with Palacios compared to 12.51 Gb/s for native Linux.

As Figure 3.8 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.

3.4.4 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 others have demonstrated [2], with the appropriate assumptions software approaches can often demonstrably operate with less overhead than hardware approaches.

3.5 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 3.3.2. 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.

3.5.1 Scaling analysis

The basic scaling results presented earlier in Section 3.3 demonstrate that the best performing technique is heavily 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, for nested paging is clearly the superior choice in this case.

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 [8] as well as the HPC context we discuss here. Figures 3.9 and 3.10 illustrate this point for HPC. Figure 3.9 shows the results of the HPCCG benchmark being run with a CNL guest environment as we scale from 1 to

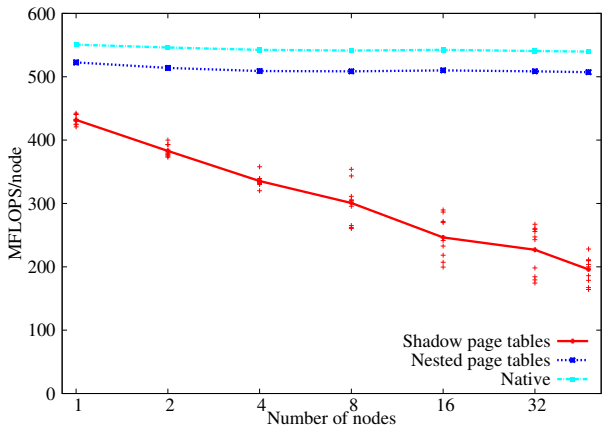


Figure 3.9: Weak scaling of HPCCG running on CNL. Nested paging is preferable, and the overhead of shadow paging compounds as we scale up. This is due to the relatively high context switch rate in CNL.

48 nodes of a Cray XT. As the results 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 3.10 shows the same benchmark run on 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.

3.5.2 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 [64, 48], an HPC-oriented benchmark suite designed to test particular elements of system performance critical to different HPC applications.

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

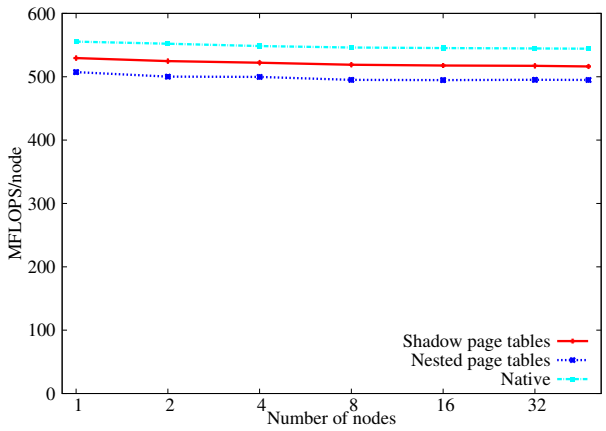


Figure 3.10: Weak scaling of HPCCG running on Catamount. Here shadow paging is preferable. This is due to the relatively low context switch rate in Catamount revealing shadow paging's better TLB behavior.

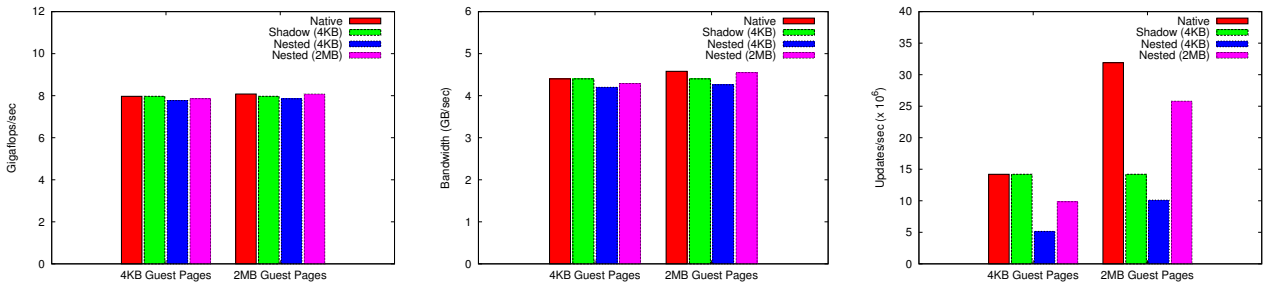
pages.

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 (in fact, using 2MB pages if necessary, this also allowed us to study the impact of these different paging choices on guests that used 4KB pages versus guests that made 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 [1], 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 3.11 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 the shadow paging. While nested paging per-



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

Figure 3.11: Performance of 2MB Nested Paging running HPC Challenge HPL, STREAM Triad, and RandomAccess benchmarks on a Cray XT4 node with a Catamount guest OS.

formance 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.

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.

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

Figure 3.12: Performance of HPC Challenge benchmark suite in Palacios for different memory virtualization approaches.

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 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 (64 bit) or 1024 (32 bit). 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.

To evaluate the potential benefits of caching and prefetching, we studied their overall performance on the HPC Challenge benchmark suite. HPC Challenge includes seven benchmarks, with two, Random Access and HPL, accounting for almost all the variation among the different paging approaches because of they exhibit a very high context switch rate. 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).

Figure 3.12 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.

3.5.3 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 incorporates a modular architecture for paging architectures. 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.

3.6 Controlled preemption

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. Palacios is designed not only minimize the amount of overhead due to virtualization, but also to concentrate necessary overheads and work into deterministic points in time in an effort to minimize the amount of noise added to the system by virtualization.

Palacios runs as a non-preemptible 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 combination of behaviors 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 impact on performance 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.

The result of this is that Palacios perturbs applications much less than 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 applcation slowdowns of orders of magnitude on large-scale systems [28, 55], keeping these overheads as small as possible is essential for virtualization to be viable in HPC systems.

3.6.1 Future extensions

An extant issue in HPC environments is the overhead induced via timer interrupts. A large 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 the timer behavior to best match the current workload.

3.7 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 3.4–3.6) relies on communication and trust across the VMM/guest interface for the mutual benefit of both entities. We might say that 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 could provide VMM↔guest information flow that could 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*.

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.

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

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

to provide extended results. Through this, a guest is able to detect the presence of 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. This address does not have to be inside the guest's allotted physical memory. Once an address has been found the guest writes it to a virtualized MSR. The VMM intercepts this operation, allocates a new page, and maps it into the guest at the location specified in the 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.

The semantic interface also includes an active component, *SymCall*, which is not used in this paper. SymCall is a mechanism by which the VMM can execute code *synchronously* in the *guest context* while the VMM is in the process of handling an exit. That is, it provides synchronous upcalls into the guest at any time.

3.8 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. Even 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. This result suggests that such machines can reap the many benefits of virtualization that have been articulated before (e.g., [37, 29]). One benefit not previously noted is that virtualization could open 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.

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

References

[1] Top 500 web site, November 2005.

[2] Keith Adams and Ole 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.

[3] Sadaf R. Alam, Jeffery A. Kuehn, Richard F. Barrett, Jeff M. Larkin, Mark R. Fahey, Ramanan Sankaran, and Patrick 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.

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

[5] Rolf Riesen Arthur B. Maccabe, Kevin S. Mccurley and Stephen R. Wheat. SUNMOS for the Intel Paragon: A brief user's guide. In *Intel Supercomputer Users' Group. 1994 Annual North America Users' Conference*, pages 245–251, 1994.

[6] Pete Beckman et al. ZeptoOS project website, `http://www.mcs.anl.gov/research/projects/zeptoos/`.

[7] Ravi Bhargava, Ben Serebrin, Francesco Spanini, and Srilatha 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.

[8] Citation removed for blind review.

[9] Ron Brightwell. A prototype implementation of MPI for SMARTMAP. In *Proceedings of the 15th European PVM/MPI Users' Group Conference*, September 2008.

[10] Ron Brightwell, Trammell Hudson, and Kevin Pedretti. SMARTMAP: Operating system support for efficient data sharing among processes on a multi-core processor. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'08)*, November 2008.

[11] Ron Brightwell, Trammell Hudson, Kevin Pedretti, Rolf Riesen, and Keith Underwood. Implementation and performance of Portals 3.3 on the Cray XT3. In *Proceedings of the 2005 IEEE International Conference on Cluster Computing*, September 2005.

[12] Ron Brightwell, Trammell Hudson, Kevin T. Pedretti, and Keith D. Underwood. SeaStar interconnect: Balanced bandwidth for scalable performance. *IEEE Micro*, 26(3):41–57, May/June 2006.

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

[14] Ron Brightwell, Kevin T. Pedretti, Keith D. Underwood, and Trammell Hudson. SeaStar interconnect: Balanced bandwidth for scalable performance. *IEEE Micro*, 26(3):41–57, 2006.

[15] Darius Buntinas, Guillaume Mercier, and William Gropp. Data transfers between processes in an smp system: Performance study and application to mpi. In *Proceedings of the 2006 International Conference on Parallel Processing*, August 2006.

[16] Darius Buntinas, Guillaume Mercier, and William Gropp. Design and evaluation of Nemesis, a scalable, low-latency, message-passing communication subsystem. In *Proceedings of the 2006 International Symposium on Cluster Computing and the Grid*, May 2006.

[17] Darius Buntinas, Guillaume Mercier, and William Gropp. Implementation and shared-memory evaluation of MPICH2 over the Nemesis communication subsystem. In *Proceedings of the 2006 European PVM/MPI Users' Group Meeting*, September 2006.

[18] Darius Buntinas, Guillaume Mercier, and William Gropp. Implementation and evaluation of shared-memory communication and synchronization operations in MPICH2 using the Nemesis communication subsystem. *Parallel Computing*, 33(9):634–644, September 2007.

[19] William J. Camp and James L. Tomkins. Thor's hammer: The first version of the Red Storm MPP architecture. In *In Proceedings of the SC 2002 Conference on High Performance Networking and Computing*, Baltimore, MD, November 2002.

[20] Franck Cappello and Daniel Etiemble. MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks. In *Proceedings of the ACM/IEEE International Conference on High-Performance Computing and Networking (SC'00)*, November 2000.

[21] L. Chai, P. Lai, H.-W. Jin, and D. K. Panda. Designing an efficient kernel-level and user-level hybrid approach for MPI intra-node communication on multi-core systems. In *Proceedings of the International Conference on Parallel Processing*, September 2008.

[22] Suchuan Dong and George Em. Karniadakis. Dual-level parallelism for deterministic and stochastic CFD problems. In *Proceedings of the ACM/IEEE International Conference on High-Performance Computing and Networking (SC'02)*, November 2002.

[23] Peter Druschel and Larry L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. *ACM SIGOPS Operating Systems Review*, 27(5):189–202, December 1993.

[24] W. Emeneker and D. Stanzione. HPC cluster readiness of Xen and User Mode Linux. In *2006 IEEE Conference Cluster Computing (CLUSTER)*, pages 1–8, 2006.

[25] Jr. E.S. Hertel, R.L. Bell, M.G. Elrick, A.V. Farnsworth, G.I. Kerley, J.M. McGlaun, S.V. Petney, S.A. Silling, P.A. 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.

[26] Karl Feind and Kim McMahon. An ultrahigh performance MPI implementation on SGI ccNUMA Altix systems. In *Proceedings of the SGI Users' Group Technical Conference*, June 2006.

[27] Kurt Ferreira, Patrick Bridges, and Ron Brightwell. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *2008 ACM/IEEE conference on Supercomputing (SC)*, pages 1–12, 2008.

[28] Kurt B. Ferreira, Ron Brightwell, and Patrick G. Bridges. Characterizing application sensitivity to OS interference using kernel-level noise injection. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (Supercomputing'08)*, November 2008.

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

[30] Erich Focht, Jaka Močnik, Fredrik Unger, Danny Sternkopf, Marko Novak, and Thomas Grossmann. *High Performance Computing on Vector Systems 2009*, chapter The SX-Linux Project: A Progress Report, pages 79–96. Springer Berlin Heidelberg, 2009.

[31] Ada Gavrilovska, Sanjay Kumar, Himanshu Raj, Karsten Schwan, Vishakha Gupta, Ripal Nathuji, Radhika Niranjan, Adit Ranadive, and Purav Saraiya. High performance hypervisor architectures: Virtualization in HPC systems. In *1st Workshop on System-level Virtualization for High Performance Computing (HPCVirt)*, 2007.

[32] R. Goldberg. Survey of virtual machine research. *IEEE Computer*, pages 34–45, June 1974.

[33] Richard L. Graham and Galen Shipman. MPI support for multi-core architectures: Optimized shared memory collectives. In *Proceedings of the 15th European PVM/MPI Users' Group Conference*, September 2008.

[34] D. S. Henty. Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling. In *Proceedings of the ACM/IEEE International Conference on High-Performance Computing and Networking (SC'00)*, November 2000.

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

[36] David Hovenmeyer, Jeffrey Hollingsworth, and Bobby Bhattacharjee. Running on the bare metal with GeekOS. In *35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE)*, 2004.

[37] Wei Huang, Jiuxing Liu, Bulent Abali, and Dhabaleswar K. Panda. A case for high performance computing with virtual machines. In *20th Annual International Conference on Supercomputing (ICS)*, pages 125–134, 2006.

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

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

[40] Hyun-Wook Jin, Sayantan Sur, Lei Chai, and Dhabaleswar K. Panda. Limic: Support for high-performance MPI intra-node communication on Linux. In *Proceedings of the 2005 Cluster International Conference on Parallel Processing*, June 2005.

[41] Larry Kaplan. Cray CNL. In *FastOS PI Meeting and Workshop*, June 2007.

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

[43] Suzanne M. Kelly and Ron Brightwell. Software architecture of the light weight kernel, Catamount. In *Proceedings of the 2005 Cray User Group Annual Technical Conference*, May 2005.

[44] 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.

[45] John Lange, Kevin Pedretti, Trammell Hudson, Peter Dinda, Zheng Cui, Lei Xia, Patrick Bridges, Andy Gocke, Steven Jaconette, Mike Levenhagen, and Ron 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.

[46] John R. Lange and Peter A. Dinda. An introduction to the Palacios Virtual Machine Monitor—release 1.0. Technical Report NWU-EECS-08-11, Northwestern University, Department of Electrical Engineering and Computer Science, November 2008.

[47] Jiuxing Liu, Wei Huang, Bulent Abali, and Dhabaleswar Panda. High Performance VMM-Bypass I/O in Virtual Machines. In *Proceedings of the USENIX Annual Technical Conference*, May 2006.

[48] Piotr Luszczek, Jack Dongarra, and Jeremy Kepner. Design and implementation of the HPCC benchmark suite. *CT Watch Quarterly*, 2(4A), November 2006.

[49] Mark F. Mergen, Volkmar Uhlig, Orran Krieger, and Jimi Xenidis. Virtualization for high-performance computing. *Operating Systems Review*, 40(2):8–11, 2006.

[50] Ronald G. Minnich, Matthew J. Sottile, Sung-Eun Choi, Erik Hendriks, and Jim McKie. Right-weight kernels: an off-the-shelf alternative to custom light-weight kernels. *SIGOPS Oper. Syst. Rev.*, 40(2):22–28, 2006.

[51] José E. Moreira, Michael Brutman, José Castaños, Thomas Engelsiepen, Mark Giampapa, Tom Gooding, Roger Haskin, Todd Inglett, Derek Lieber, Pat McCarthy, Mike Mundy, Jeff Parker, and Brian Wallenfelt. Designing a highly-scalable operating system: The Blue Gene/L story. In *ACM/IEEE Supercomputing SC'2006 conference*, 2006.

[52] Arun Babu Nagarajan, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive fault tolerance for HPC with Xen virtualization. In *21st Annual International Conference on Supercomputing (ICS)*, pages 23–32, 2007.

[53] Hideo Nishimura, Naoya Maruyama, and Satoshi 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.

[54] Kevin T. Pedretti. Summary of multi-core hardware and programming model investigations. poster, 2007.

[55] Fabrizio Petrini, Darren Kerbyson, and Scott 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.

[56] Ioan Raicu, Zhao Zhang, Mike Wilde, Ian Foster, Pete Beckman, Kamil Iskra, and Ben Clifford. Toward loosely-coupled programming on petascale systems. In *ACM/IEEE International Conference on High-Performance Computing, Networking, Storage, and Analysis*, November 2008.

[57] Himanshu Raj and Karsten Schwan. High performance and scalable i/o virtualization via self-virtualized devices. In *16th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, July 2007.

[58] Rolf Riesen, Ron Brightwell, Patrick Bridges, Trammell Hudson, Arthur Maccabe, Patrick Widener, and Kurt Ferreira. Designing and implementing lightweight kernels for capability computing. *Concurrency and Computation: Practice and Experience*, 21(6):793–817, April 2009.

[59] Dennis M. Ritchie. A guest facility for Unicos. In *UNIX and Supercomputers Workshop Proceedings*, pages 19–24. USENIX, September 1988.

[60] Edi Shmueli, George Almasi, Jose Brunheroto, Jose Castanos, Gabor Dozsa, Sameer Kumar, and Derek Lieber. Evaluating the effect of replacing CNK with Linux on the compute-nodes of Blue Gene/L. In *roceedings of the 22nd International Conference on Supercomputing*, pages 165–174, New York, NY, USA, 2008. ACM.

[61] Edi Shmueli, George Almási, Jose Brunheroto, Jose Castaños, Gabor Dózsa, Sameer Kumar, and Derek Lieber. Evaluating the effect of replacing CNK with Linux on the compute-nodes of Blue Gene/L. In *22nd Annual International Conference on Supercomputing (ICS)*, pages 165–174, New York, NY, USA, 2008. ACM.

[62] Lance Shuler, Chu Jong, Rolf Riesen, David van Dresser, Arthur B Maccabe, Lee Ann Fisk, and T Mack Stallcup. The PUMA operating system for massively parallel computers. In *1995 Intel Supercomputer User's Group Conference*. Intel Supercomputer User's Group, 1995.

[63] Quinn O. Snell, Armin Mikler, and John L. Gustafson. NetPIPE: A network protocol independent performance evaluator. In *Proceedings of the IASTED International Conference on Intelligent Information Management and Systems*, June 1996.

[64] Shuaiwen Song, Rong Ge, Xizhou Feng, and Kirk 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.

[65] Samuel Thibault and Tim Deegan. Improving performance by embedding HPC applications in lightweight Xen domains. In *2nd Workshop on System-level Virtualization for High Performance Computing (HPCVirt)*, pages 9–15, 2008.

[66] Anand Tikotekar, Geoffroy Vallée, Thomas Naughton, Hong Ong, Christian Engelmann, Stephen L Scott, and Anthony M Filippi. Effects of virtualization on a scientific application running a hyperspectral radiative transfer code on virtual machines. In *2nd Workshop on System-Level Virtualization for High Performance Computing (HPCVirt)*, pages 16–23, 2008.

[67] Jean-Charles Tournier, Patrick Bridges, Arthur B. Maccabe, Patrick Widener, Zaid Abudayyeh, Ron Brightwell, Rolf Riesen, and Trammell Hudson. Towards a framework for dedicated operating systems development in high-end computing systems. *ACM SIGOPS Operating Systems Review*, 40(2), April 2006.

[68] Nicholas Wells. BusyBox: A Swiss Army knife for Linux. *Linux Journal*, November 2000. `http://busybox.net/`.

[69] Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gn Sirer, and Fred 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.

[70] Lamia Youseff, Rich Wolski, Brent Gorda, and Chandra Krintz. Evaluating the performance impact of Xen on MPI and process execution for HPC systems. In *2nd International Workshop on Virtualization Technology in Distributed Computing (VTDC)*, page 1, 2006.

Appendix A

External Impact

A.1 Peer-reviewed Published Papers

1. K. Ferreira, P. Bridges, R. Brightwell, K. Pedretti, The Impact of System Design Parameters on Application Noise Sensitivity, IEEE International Conference on Cluster Computing (Cluster'10), September 2010.

2. J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, R. Brightwell, Palacios and Kitten: New High Performance Operating Systems for Scalable Virtualized and Native Supercomputing, IEEE International Parallel and Distributed Processing Symposium (IPDPS'10), April 2010.

3. R. Brightwell, T. Hudson, K. Pedretti, SMARTMAP: Operating System Support for Efficient Data Sharing Among Processes on a Multi-Core Processor, ACM/IEEE Conference on Supercomputing (SC'08), November 2008.

A.2 Submitted Papers

1. J. Lange, K. Pedretti, P. Dinda, P. Bridges, C. Bae, P. Soltero, A. Merritt, Minimal-overhead Virtualization of a Large Scale Supercomputer, submitted to ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'2011), March 2011.

A.3 Other Publications

1. K. Pedretti, K. Ferreira, R. Brightwell, Exploring Memory Management Strategies in Catamount, Paper, Cray User Group Meeting, May 2008.

2. K. Pedretti, S. Kelly, M. Levenhagen, Summary of Multi-Core Hardware and Programming Model Investigations, Sandia Technical Report, SAND2008-3205, May 2008.

3. K. Pedretti, Characterization of Intra-node Topology and Locality, Poster, International Conference for High Performance Computing, Networking, Storage, and Analysis (SC07), November 2007.

A.4 Invited Talks

1. K. Pedretti, The Kitten Lightweight Kernel, Presentation at FastOS Phase II Workshop held in conjunction with ACM International Conference on Supercomputing (ICS'09), June 2009.
2. K. Pedretti, Lightweight Operating Systems for Scalable Native and Virtualized Supercomputing, Presentation at ORNL Future Technologies Colloquium Series, April 2009.
3. K. Pedretti, Quad-core Catamount and R&D in Multi-core Lightweight Kernels, Salishan Conference on High-Speed Computing, April 2008.

A.5 Service to Professional Societies

1. Kevin Pedretti – program committee member, OS and runtime – 2011 39th International Conference on Parallel Processing.
2. Kevin Pedretti – program committee member, 2010 Workshop on Micro Architectural Support for Virtualization, Data Center Computing, and Clouds, held in conjunction with MICRO-43 conference.

A.6 Patent Applications

1. R. Brightwell, K. Pedretti, T. Hudson, Direct Access Inter-Process Shared Memory, Patent Application, August 2009.

A.7 Awards

1. 2010 Employee Recognition Award – Kitten Operating System Virtualization Team. Team members: K. Pedretti, R. Brightwell, K. Ferreira, S. Kelly, M. Levenhagen, C. Vaughan
2. 2009 R&D100 Award – Catamount N-Way Lightweight Kernel, 2009 R&D100 Award. Team members: R. Brightwell, K. Ferreira, S. Kelly, J. Laros, K. Pedretti, J. Tomkins, J. Vandyke, C. Vaughan, R. Ballance, T. Hudson

A.8 New Ideas for R&D

We submitted a proposal to DOE ASCR program "X-Stack Software Research" to continue and expand on the HPC virtualization research initiated by this LDRD project. This project was selected for funding and will have a three year duration, beginning in FY11. The project is a consortium of researchers from University of New Mexico, Northwestern University, Sandia National Laboratories, and Oak Ridge National Laboratory.

We are also pursuing the use of virtualization to accelerate hardware simulation, and to expose novel architectural features (e.g., global shared memory) to system software before hardware is available. This work is in the context of the recently funded DARPA UHPC project that Sandia is leading, starting in FY11.

DISTRIBUTION:

	MS	,
		,
1	MS 0899	Technical Library, 9536 (electronic copy)

Sandia National Laboratories