A Jade stone is useless before it is processed; a man is good-for-nothing until he is educated.

– Chinese Proverbs.

University of Alberta

INTERPROCESS COMMUNICATION MECHANISMS WITH INTER-VIRTUAL MACHINE SHARED MEMORY

by

Xiaodi Ke

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

To dad, mom, my wife and sister
For encouraging and supporting me while I am about to give up.

Abstract

In many cloud computing environments (e.g., Amazon's public Elastic Computing Cloud and Openstack for private clouds), virtual machine (VM) instances are the unit of resource allocation. When possible, VM instances can be allocated on the same physical server and many techniques (e.g., using shared memory between VMs) can be used to reduce the overhead of the inter-VM communication.

Nahanni is a novel inter-VM shared-memory mechanism. We investigate two inter-VM interprocess communication (IPC) mechanisms using Nahanni shared memory. First, minitransactions have different semantics from both traditional shared-memory programming and message-passing programming. We experimentally show that minitransactions can have better performance than traditional lock-based programming under low-contention scenarios. Second, MPI-Nahanni ports the well-known Message-Passing Interface (MPI) system to Nahanni. Using microbenchmarks and the GAMESS application, we show how MPI-Nahanni has higher bandwidth and lower latency (by up to an order of magnitude or more), and better performance than existing VM-based IPC techniques.

Acknowledgements

First and foremost, I would like to express my sincere thanks to my supervisor Dr. Paul Lu for his guidance, patience, inspiration and understanding, espcically for his help with my thesis writing. Without his support and help, this work would not be possible.

I would like to acknowledge my labmates, Adam Wolfe Gordan, Cam Macdonell, Jermey James Nickurak, for helping me set up the experiment environment and answering my "boring" questions patiently.

Last but not least, I should thank my parents who unconditionally support my study in Canada. I also need to thank my sister, Xiaomei, for her help in taking care of my parents especially when my mother was having a surgery. I am grateful to my wife, Shuyuan, for her love and continuous encouragement. Without their support and encouragement, I cannot focus on my study and get through this challenging time.

Table of Contents

List of Tables

List of Figures

List of Acronyms

ACID	Atomicity, Consistency, Isolation and Durability
API	Application Programming Interface
CLR	Common Language Runtime
CPU	Central Processing Unit
EC2	Elastic Computing Cloud
FIFO	First-In-First-Out
GAMESS	General Atomic and Molecular Electronic Structure System
HPC	High-Performance Computing
HTM	Hardware Transactional Memory
IaaS	Infrastructure-as-a-Service
IMB	Intel MPI Benchmark
I/O	Input/Output
IPC	Interprocess Communication
JVM	Java Virtual Machine
KVM	Kernel-based Virtual Machine
LMT	Large Message Transfer
MPI	Message-Passing Interface
NAT	Network Address Translation
NIC	Network Interface Card
NUMA	Non-Uniform Memory Access
OS	Operating System
PaaS	Platform-as-a-Service
RMA	Remote Memory Access
SaaS	Software-as-a-Service
SPH	Smoothed Particle Hydrodynamics
SPMD	Single Program Multiple Data
STM	Software Transactional Memory
TM	Transactional Memory
VDE	Virtual Distributed Ethernet
VM	Virtual Machine
VMM	Virtual Machine Monitor

Chapter 1

Introduction

Cloud computing usually refers to a shared pool of computational resources (e.g., hardware, software) that can be accessed via a computer network. Different cloud-computing systems can provide various services. For example, users can use the software in the software-as-a-service (SaaS) system (e.g., Google Docs, Gmail) rather than buying it themselves. The platform-as-a-service (PaaS) system (e.g., Google AppEngine, Microsoft Azure) provides a development platform and an application programming interface (API) for custom applications. The Infrastructure-as-a-Service (IaaS) system (e.g., Amazon's public Elastic Computing Cloud (EC2)) offers the computational resources in the form of virtual machines (VMs).

In an IaaS cloud-computing system, applications scale up their resource footprints by allocating more VMs, since VMs are often the unit of resource allocation. For example, Amazon EC2 allows a user to provision additional hardware resources (e.g., processors, cores, memory) by starting up new VMs on demand. Although there might be small and large VM instances available, the number of instances is the primary mechanism of scaling up the system. In fact, given the size of many cloud-computing providers, an advantage for the user is the ability to access large amounts of computing resources on demand.

If a physical server has more cores than the number of virtual processors in a VM instance, then multiple VMs might be co-located on the same server, to maximize hardware utilization. With the current trend towards more cores per server (i.e., manycores [7]), and the scheduling flexibility of VM instances with smaller number of virtual processors (i.e., a given VM can be placed or migrated to more potential servers if its virtual processor count is less than the physical core count), we expect the opportunities for co-located VMs to be significant. And, if VMs are co-located on the same physical server, we argue that those VMs should use the fastest possible mechanisms for communication.

Recently, researchers have proposed many techniques to reduce the latency and increase the bandwidth of communication between co-located VMs. For example, paravirtualization [27] and other optimizations to reduce protection domain crossings [3] have been suggested as the best approach [29] to improve communication overheads. Also, there is a large body of work on using shared memory for fast inter-VM communication (e.g., XenSocket [33], Fido [13] and MVAPICH2-ivc [21]).

Nahanni, also known as ivshmem, is a recently introduced mechanism for inter-VM shared memory in QEMU/Linux Kernel-based Virtual Machine (KVM) [25]. The shared-memory interprocess communication (IPC) supported by Nahanni can support different use-cases and applications. At one level of abstraction, Nahanni can be used like regular shared memory. It has already been demonstrated that the traditional load-store and lock-based programming method can work well on top of the Nahanni shared memory (e.g., the implementation of Nahanni memcached [31]). At a different level of abstraction, Nahanni can be layered below other APIs to support fast IPC (Chapter 4).

1.1 Motivation

One of the current goals within the Nahanni project is to explore the different methods of using shared-memory IPC. In our experience, when Nahanni is first described to a programmer, their first impulse is to, naturally, think in terms of load-store and lock-based programming models. However, other programming models can be layered on top of Nahanni, and other models can be re-implemented using Nahanni as a building block.

To this end, we investigate two APIs as representatives. The minitransaction model [6] from the transactional memory community is an example of an emerging API. Although minitransactions were originally targeting distributed systems, the same abstraction might be attractive for programming inter-VM systems. Moreover, minitransactions present a transparent way to support hybrid inter-VM and inter-server IPC (i.e., distributed memory). The well-known Message-Passing Interface (MPI) is an example of an existing and mature API. MPI is widely used and highly portable. We implement MPI-Nahanni, a port of MPICH2, on top of the Nahanni shared memory.

1.2 Nahanni Project

Nahanni, first included in QEMU/KVM v0.13 as `ivshmem` in August 2010, is a paravirtualized PCI device that allows the host and the guest VMs to share a region of memory [25]. The optional Nahanni PCI device, which looks similar to the on-board memory of a graphics card to the guest, uses a POSIX shared-memory file on the host for the backing store. Thus once Nahanni is configured, the co-resident VMs and the host can do shared-memory IPC via Nahanni.

Nahanni shared memory is just like other familiar shared memory. To use it, user-level programs inside the guest VMs open the device as a file with the `open` system call, and use `mmap` to map the memory into their address space. The whole procedure is similar to that of POSIX shared memory. After the programs initialize Nahanni, the shared-memory region can be used like other memory regions in the programs. Note that the details of the `open` and `mmap` can be hidden from application programmers within a user-level library, which is the case MPI-Nahanni.

1.3 Research Questions and Contributions

We try to answer the following research questions through our hand-on experiences: What APIs are Nahanni shared memory suitable for? What are the benefits that an API can gain from using Nahanni shared memory?

The main contribution of our work is to show that programming models and APIs, such as minitransactions and MPI, can be well-supported using Nahanni, and with high performance. Specifically:

1. We show that Nahanni is able to support new programming models and APIs, specifically the minitransaction model.

   Inspired by Sinfonia [6], we implement the minitransaction abstraction on top of Nahanni shared memory. We also port, parallelize, and evaluate the FLUID v.1 smoothed particle hydrodynamics (SPH) simulator as a case study. In the process of working with FLUID, we implement barrier synchronization, with different design choices related to supporting reliable signalling between VMs. Prior to our work with FLUID, reliable signalling had not been provided within the Nahanni environment.

2. We show that Nahanni is able to support existing programming models and APIs, specifically the well-known MPI model and library.

   One of the architectural advantages of Nahanni is that it behaves like familiar shared memory, after it is initialized. Therefore, it was by design, straightforward to port the MPICH2-Nemesis implementation of MPI to Nahanni. MPICH2-Nemesis already had a shared-memory implementation based on System V or memory-mapped shared memory. Since Nahanni is just like other shared-memory systems, much of the existing MPICH2-Nemesis code was used without change.

3. In the performance evaluation of VM-based barrier synchronization, minitransactions and MPI-Nahanni, we show that our minitransactions have better performance for low-contention cases, but can have high overheads under high contention. Also, our barrier synchronization shows good performance. Improvements to the minitransactions under high contention are for future work.

   From microbenchmarks and the GAMESS application, MPI-Nahanni shows high performance as compared to other VM-based IPC techniques. Specifically, MPI-Nahanni can have up to an order of magnitude lower latency and higher bandwidth than MPI over standard virtual networks for VMs. Leveraging Nahanni inter-VM shared memory can help MPI-Nahanni reduce the communication overheads for co-located VMs substantially.

1.4 Concluding Remarks

We discussed our motivation for investigating two APIs, minitransactions and MPI. We also briefly introduced the Nahanni project and summarized our contributions. In next chapter, we discuss some important background concepts and review some related work in this field.

Chapter 2

Background and Related Work

In the previous chapter, we introduced the Nahanni project and discussed the motivation for implementing interprocess communication (IPC) over the inter-virtual machine (VM) shared memory. In this chapter we present important concepts that relate to the design and implementation of our work. We also outline the previous related work in the field.

2.1 Background Concepts

2.1.1 Virtual Machine

There are various kinds of virtual machines. For example, the Java virtual machine (JVM) and the Microsoft .NET Common Language Runtime (CLR) are VMs that implement the virtualization technique at the programming language level. JVM and CLR execute the software-based bytecodes that are generated by the compilers. In contrast, Xen, VMware and QEMU/Linux Kernel-based Virtual Machine (KVM) are VMs that virtualize the environment at a lower level of abstraction (i.e., instruction set architecture (ISA), hardware level). Figure 2.1 and 2.2 show examples of computer architectures with and without VMs. We usually call a VM instance the *guest* and the platform that is running the VMs the *host*. The guest operating system (OS) can access the hardware resources (e.g., processor, memory, disk) as virtualized by the VM, thus the VM gives users an illusion that they own the physical machine and they can do whatever they can as if running directly on a physical machine. Currently, the well-known tools (i.e., hypervisors or virtual machine monitors (VMM)) for running VMs include QEMU/KVM [23], Xen [8] and VMware [4]. We use QEMU/KVM as our VM hypervisor in the following works.

VMs are important in Infrastructure-as-a-Service (IaaS) cloud computing and high-performance computing (HPC) environments because:

1. VMs can provide an isolated and safe environment for important services.

   Because the hypervisor treats each guest OS within a VM instance as if they are independent from other VMs running on the same host, the VM instances are isolated. In other words,

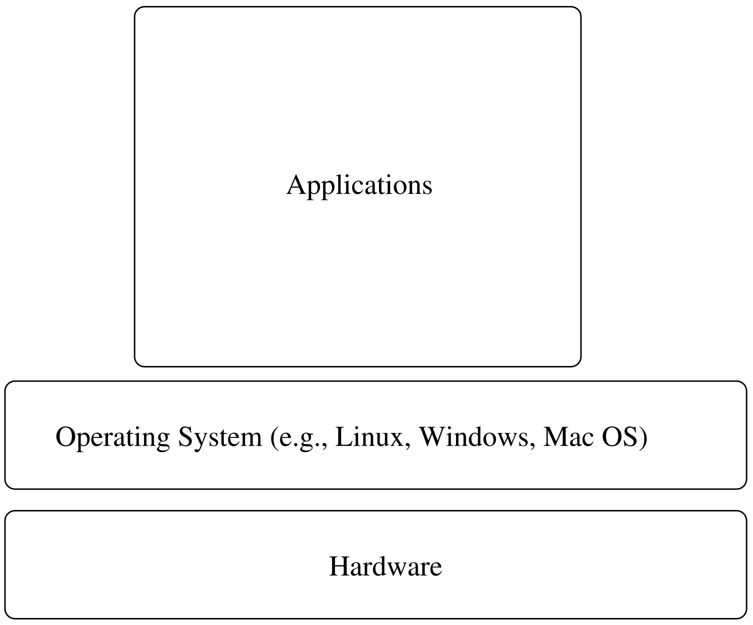


Figure 2.1: A computer architecture without VMs.

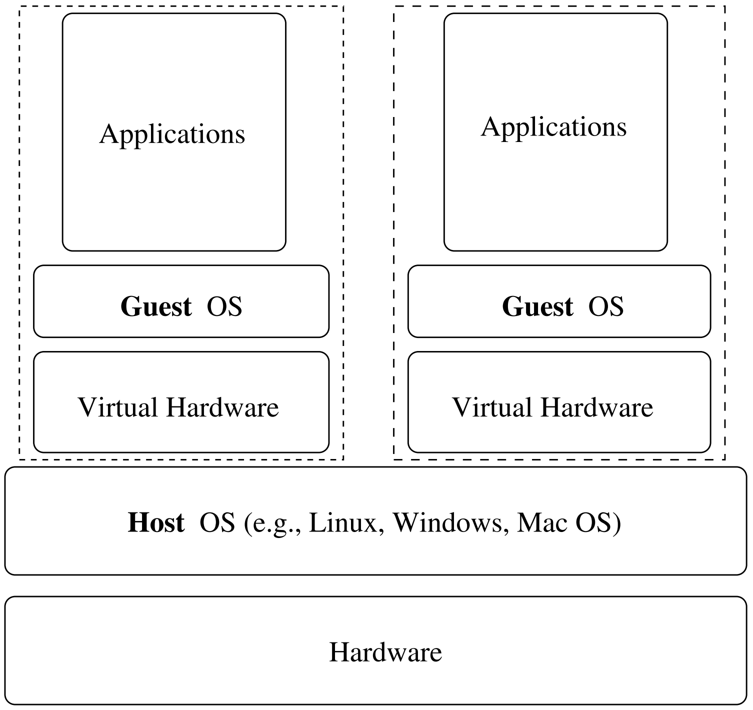


Figure 2.2: A computer architecture with VMs.

using VMs, the impact of an application failure will be contained to only one VM and the failure cannot impact other VMs. The failed application can be restarted by simply restarting the affected VM.

2. VMs can provide multiple execution environments for the user.

   Different VM instances can run different OSes on the same hardware. For example, one host server can simultaneously run independent VM instances with Linux, Windows and Mac OS. Without VMs, one would either use separate hosts for different OSes, or possibly use dual booting if those OSes do not need to run concurrently.

3. VMs can provide better utilization of the host (i.e., consolidation) and convenient management of hardware resources (e.g., VM migration).

   Using VMs is another technique for time-sharing and space-sharing hardware resources. For example, if some physical servers are underutilized (e.g., a print server, a mail server), the servers can be encapsulated inside VMs and consolidated on a single physical server to improve the utilization of that host. Also, administrators can balance the workload among multiple physical servers with the help of the VM's live migration capability. A VM can migrate from an overloaded server to a underloaded server.

Although VMs have many advantages, the potential performance overhead of VMs is one disadvantage. For example, there are overheads associated with handling privileged instructions in the guest, and with handling input/output (I/O) via emulated hardware devices. The trap-and-emulate approach to privileged instructions requires frequent context switches between privileged and non-privileged modes. Also, because multiple protection domains and software layers are crossed (shown in Figure 2.2) to access the physical hardware device (e.g., hard disk, network interface card (NIC)), I/O in VMs can have significant overheads. However, with the advent of hardware-based ISA virtualization (e.g., AMD's SVM, Intel's VT) and device paravirtualization (e,g., virtio [27] , vhost [3]), some of these overheads have been greatly reduced. In a recent study [26], the VM overhead for HPC applications (e.g., GROMACS, BLAST, HMMer) were shown to be under 6% for compute-intensive applications, and approximately 9.7% for more I/O-intensive jobs on an x86 VM platform. Therefore, the overheads are currently moderate and as the overheads of VMs are reduced by new hardware support and software techniques, the benefits of using VMs in cloud environments increase.

2.1.2 Fast Co-located Inter-VM IPC

As we mentioned in Section 2.1.1, one of the advantages of using VMs is the isolation between VMs. But this isolation barrier also becomes an obstacle when the applications in co-located VMs (i.e., guest VMs on the same physical host) need to communication with each other. The applications in different VMs have to cross multiple software layers when communicating with others. As shown in Figure 2.2, the data passes through the guest OS, the virtual hardware, and the host OS. And in Chapter 4, we also discuss how the traditional network communication for inter-VM IPC suffers significant overheads since another guest OS domain crossing is required from one VM to another VM (i.e., the datapath back up to the target's guest OS).

Moreover, the multiple data copies along the aforementioned datapath also add to the overheads. The transmitting data may be copied up to four times. The first copy is from the application's user space to the guest OS kernel space in the source VM. The second copy is from the guest OS kernel space to the host OS kernel space. The third copy is from the host OS kernel space to the target guest OS kernel space. And the final copy is from the target guest OS kernel space to the target application's user space.

However, if the VMs shared memory, some of the datapath overheads can be reduced. For example, if two VMs share a memory region, there is no need for them to pass the data through all of the guest and host software layers. The sender can write the data to the shared memory and the receiver can read the data from the shared memory. Obviously, this procedure reduces the number of times the data is copied to two: The first copy is from the source to the shared memory and the second copy is from the shared memory to the target's application buffer. In special cases, only a single copy is necessary if the system uses something like Xen's inter-VM Grant/Mapping Table framework to directly re-map the sender's buffer to the receiver's address space [20]. However, page re-mapping techniques have their own overheads (i.e., system call to the host OS to perform the re-mapping) so must be used carefully.

2.1.3 Load-Store Model for Shared Memory

The load-store model is the traditional programming model for shared memory. The first column of Figure 2.3 shows the pseudocode for updating two shared counters (i.e., `counter`, `new_counter`) using the load-store model. Note that the counters are incremented with ISA instructions (i.e., the ++ and += operators are mapped to machine instructions by the compilers of most ISAs) since the model allows the memory to be manipulated directly. When dealing with shared data, programmers are responsible for the concurrency control. Explicit locks are used to guarantee mutual exclusion and protect the access of the shared data. One the one hand, low-level primitives, such as locks and semaphores, are complicated and error-prone, especially when considering issues such as deadlock and fine-grained access. On the other hand, fine-grained locks can provide better scalability and higher concurrency for parallel programs.

2.1.4 Transactional Memory

Transactional memory (TM) [18] is a concurrency control mechanism for parallel programming and accessing shared data. It uses transactions, like database transactions, to abstract the complex locking protocols associated with shared data manipulation. Although transactions and TM will work correctly in all situations, their performance is optimized for situations where low contention for the shared data is expected, since transactions and TM are part of the larger category of optimistic locking techniques [18].

The second column of Figure 2.3 shows the pseudocode of updating two shared counters using the TM model. Note that the TM pseudocode is more succinct than using loads and stores. A transaction usually contains a set of read and write operations and these operations are applied atomically. Specifically, once the transaction is successfully committed, all the modifications of the associated operations in shared memory should be visible to others. Otherwise, the transaction should have no effect on the shared memory (i.e., all or nothing). Note that reads and writes are similar to loads and store, except that reads and writes are usually mapped to longer sequences of ISA instructions, depending on implementation details.

Hardware transactional memory (HTM) and software transactional memory (STM) are two main TM implementations [18]. HTM requires special hardware support and can provide high TM performance. STM breaks the limitations of HTM (e.g., special hardware requirement), but suffers more overhead than HTM.

Compared to the traditional load-store programming model (e.g., with explicit locks), TM has many advantages [18]: TM helps developers focus on the algorithm design instead of the complex concurrency control mechanism; TM provides balance between the scalability of and implementation effort for parallel programs; TM is deadlock free.

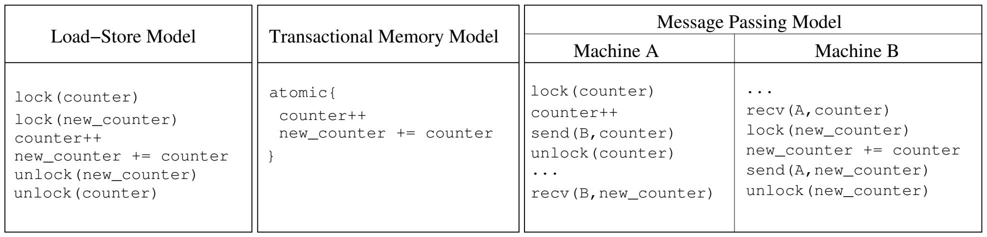
2.1.5 Message-Passing Model

Message passing is a common communication method for parallel computing and IPC. Normally, every communication endpoint has its own memory and data are packed into messages before sending. The third column of Figure 2.3 shows the pseudocode of updating two counters using the message-passing model. Machine A increases `counter` and sends it to machine B, while machine B uses the `counter` value from A to update `new_counter` and sends the new value of `new_counter` back to machine A. Usually, the message-passing model includes two classes of communication routines. First, the point-to-point communication routines (e.g., send and receive) are for pairs of endpoints. Second, the collective communication routines (e.g., broadcast) are for sets of endpoints, usually more than two.

2.2 Related Work

2.2.1 IPC Mechanism

IPC is a well-studied area of research. In particular, eliminating data copying to improve performance has been examined in several contexts and with various design goals. Brustoloni and



Load−Store Model

```
lock(counter)
lock(new_counter)
counter++
new_counter += counter
unlock(new_counter)
unlock(counter)
```

Transactional Memory Model

```
atomic{
  counter++
  new_counter += counter
}
```

Message Passing Model

Machine A

```
lock(counter)
counter++
send(B,counter)
unlock(counter)
...
recv(B,new_counter)
```

Machine B

```
...
recv(A,counter)
lock(new_counter)
new_counter += counter
send(A,new_counter)
unlock(new_counter)
```

Figure 2.3: The pseudocodes of updating two shared counters using load-store model, transactional memory model and message-passing model.

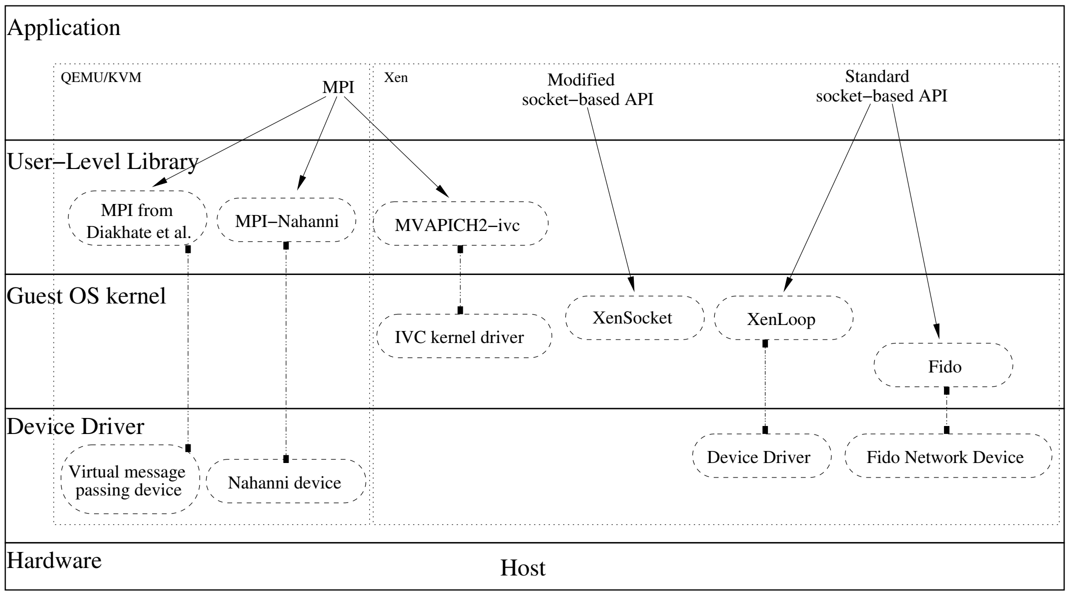


Figure 2.4: The architectural layers used by the different inter-VM shared-memory communication mechanisms.

Steenkiste [10] studied the effects of buffering semantics on I/O performance and pointed to unnecessary copies being a source of overhead leading to poor performance. Fbufs [16] are an OS mechanism for efficient data transfer within an OS kernel. Beltway Buffers [9] are an in-kernel mechanism for Linux that uses pre-allocated, shared rings for data movement for all IPC mechanisms (e.g., sockets, pipes, disks) as well as networking.

2.2.2 Inter-VM Shared-Memory Communication Mechanisms

The use of inter-VM shared memory for IPC between VMs has previously been explored. Figure 2.4 shows the architectural layers used by the various inter-VM IPC mechanisms introduced in the following discussion. The majority of inter-VM communication research has focused on the Xen hypervisor [8, 21, 30, 33, 13], as Xen is open source, widely used, and mature. However, in recent years, QEMU/KVM has emerged as another open-source hypervisor and is gaining attention as the default VM technology in the Red Hat (which bought Qumranet, Inc., the original developers of KVM) and Ubuntu Linux distributions. Architecturally, the QEMU/KVM-based systems (i.e., Diakhaté *et al.* [15], Nahanni [25]) are distinguished from the Xen-based systems by their lack of modifications to the "Guest OS kernel".

Using a combination of a user-level library and a device driver approach to implement faster IPC (i.e., the Message-Passing Interface (MPI) in this discussion) has advantages [25]: First, modifications to the "Guest OS Kernel" are not necessary. Usually, modifying the kernel is the most error-prone and complicated type of changes. Pragmatically, changes to the core kernel are also the most difficult to have accepted into the mainline code base for open-source OS projects. Second, no additional performance overheads are added to the kernel, since the kernel is unmodified. Changes to the kernel have the potential to introduce scalability bottlenecks, since they affect core OS datapaths. For example, XenLoop modifies the OS kernel to separate intra-host from inter-host traffic so as to use the appropriate mechanism (i.e., shared memory vs. network). On the one hand, XenLoop can transparently support (i.e., XenLoop is binary compatible with standard sockets-based applications) mixed intra- and inter-host data traffic. On the other hand, XenLoop introduces an extra check (and potential additional latency, however small) to every data packet. Third, for VM instances and applications that do not want to use the shared-memory IPC optimizations, the new device drivers are not loaded into the OS kernel and thus should have no performance (or other) impact at all. In contrast, the modified OS kernel must always be running in the Xen-based examples if there is any possibility of applications using the shared-memory IPC optimizations.

XenSocket [33] is a one-way inter-domain communication solution for Xen based on shared memory. The sender and receiver share two memory buffers which are used to exchange the control information and transfer the actual data. Applications using XenSocket must be modified to use a variation of the POSIX socket API, since the shared memory is not visible at the user level. With some minor changes to the socket initialization source code, most sockets-based applications can be ported to use XenSocket.

XenLoop [30] is a kernel module which provides a high-performance and fully transparent communication channel for co-located VMs. Two first-in-first-out (FIFO) data channels are created between pairwise VMs by using Xen's inter-domain shared-memory facility. Each FIFO channel is unidirectional and used for sending/receiving data to/from another co-located VM. XenLoop will inspect each outgoing packet to determine the receiver and automatically determine which datapath, either the FIFO channel or the standard network, should be used to send a given packet. XenLoop is more transparent than XenSocket, since it is binary compatible with existing sockets code

Fido [13] is another high-performance inter-VM communication mechanism. It is shared-memory-based and allows target VMs to map the entire address space of a source VM into its own address space. By doing this, Fido can achieve zero-copy data movement while transferring data between VMs. The source VM writes data into memory and the target VM can read data from the memory-mapped region directly without an extra copy. Moreover, Fido can also be used to implement a network device and a block device, which are transparent for the users to use.

Different from the aforementioned systems, MVAPICH2-ivc [21] leverages shared memory to develop a generic network device and improves MPI communication for co-located VMs. MVAPICH2-ivc benefits from a shared-memory-based VM-aware communication library, called IVC. MVAPICH2-ivc can automatically select the communication methods, between the standard network and IVC, in virtual machine environments. If two MPI processes are in different hosts, the network communication will be used. Otherwise, IVC will be activated and set up a shared-memory region for the communication between co-located VMs. MVAPICH2-ivc also supports live migration. It is transparent to the MPI program and does not require code modification.

All of the related work mentioned above are Xen-based and depend on specific Xen mechanisms, such as the grant table mechanism. The only other related work using the QEMU/KVM environment is by Diakhaté *et al.* [15]. The authors develop a shared-memory-based message-passing device which is accessed by the virtio interface. This device allocates memory from a shared-memory pool when starting a QEMU instance. Other QEMU instances should be created by forking the initial instance in order to leverage the shared-memory communication device. As compared to our work, Diakhaté *et al.* only implemented an incomplete subset of MPI functions on top of this device, such as `MPI_Irecv`, `MPI_Isend` and `MPI_Wait`, which restricts the usefulness of their system.

2.2.3 Minitransaction

The fundamental idea behind minitransactions comes from the transactional memory community [18] (Section 2.1.4). And the original implementation of minitransactions in Sinfonia [6] is targeted at distributed memory. Minitransactions are intended to be lightweight and short-lived, which usually affects implementation details and optimizations. For example, distributed minitransactions might be limited such that they fit within a single network packet.

A minitransaction comprises read items, compare items, and conditional write-items. These items can be located on different memory nodes, with the system being responsible for coordinating the global atomicity, consistency, isolation and durability (ACID) properties. Figure 2.5 shows an example of updating a counter on a remote memory node. The variable `machine_ID` specifies the memory node where the counter resides. And `addr` and `len` specify the address memory and the length of the counter, respectively. Upon execution (i.e., `commit_minitransaction()`), the runtime system retrieves the data from the locations specified by read items, and compares the data in the locations indicated by the compare items. If and only if all the comparisons are successful, the data in the conditional-write items are applied to the specified locations. The return values of `commit_minitransaction()` can be `SUCCEEDED`, `LOCK_FAILED` or `COMPARE_FAILED`. If `COMPARE_FAILED` is returned, it means that the remote counter has already been modified by other minitransactions and the `local_counter` contains an obsolete value. The user should use the data retrieved from the remote node to update local cache.

The minitransaction can be re-executed again (i.e., the `do-while` loop) until it succeeds. As

```
...
do{
     t = new Minitransaction
     new_counter = local_counter + 1
     t.add_read_item(machine_ID, addr, len)
     t.add_compare_item(machine_ID,addr,len,local_counter)
     t.add_write_item(machine_ID,addr,len,new_counter)
     succeed = t.commit_minitransaction()
     if(succeed == COMPARE_FAILED)
          local_counter = t.read_return_value()
}while(succeed != SUCCEEDED)
local_counter = new_counter
...
```

Figure 2.5: Example of using minitransactions to update a remote counter, based on a local cached value.

with transactional memory, minitransactions abstract the complexity of the concurrency control. Minitransaction programmers do not need to design the low-level message-passing and locking protocols, which are complicated and error-prone.

With minitransactions, Sinfonia can easily support complex applications. For example, a cluster file system and a group communication service are implemented with 3,900 and 3,500 lines of code, respectively [6]. Also, Aguilera *et al.* [5] built a scalable, low cost, and fault-tolerant distributed B-tree with Sinfonia and minitransactions.

2.2.4 Nemesis: Shared-Memory IPC in MPICH2

MPICH2 is an open-source implementation of MPI. Recent versions of MPICH2 contain a communication subsystem called Nemesis (i.e., the `ch3:nemesis` channel), which is designed for scalability, high-performance intra-node and inter-node communication, and multimethod inter-node communication [11, 12].

Nemesis uses shared memory for the intra-node processes communication. Each intra-node process has a receive queue and a free queue in shared memory. When a process sends a message, a new network module will determine whether it is sent to the same node or a different node. If the communication is intra-node, the network module directly inserts the message to a target process's receive queue, which is located in node's shared memory. Otherwise, the network module sends out the message over the usual mechanism (e.g., socket). With the help of a lock-free queue algorithm, Nemesis can maintain as good inter-node communication performance as other MPI implementations, and achieve better performance for intra-node communication. Also, Nemesis supports remote memory access (RMA) operations and barrier synchronization via the shared memory.

2.2.5 Nahanni Inter-VM Shared Memory

Nahanni, also known as `ivshmem`, was recently introduced into QEMU/KVM [25]. As an aside, the similarity between the names Nemesis and Nahanni is unfortunate, but unavoidable. Nahanni offers a POSIX shared-memory region between the host and the guest. Nahanni is exported to the guest as a PCI device. Users, or more likely user-level libraries (Figure 2.4), can use the traditional `open` function to open the device as a regular file and use `mmap` to map the device memory into the applications' address space. The Nahanni shared memory can also be used for fast inter-VM or host-to-guest IPC.

For example, the performance of the host-to-guest data movement with the various IPC techniques and Nahanni shared memory has been examined [25]. Data transfer can be up to 9-fold faster using Nahanni shared-memory IPC as compared to a special-purpose 9P file system and other widely used VM-based mechanisms (e.g., virtio, Netcat, SCP-HPN). Wolfe Gordon and Lu [32] proposed Nahanni memcached, a port of the standard memcached. Instead of using VM-based virtual networks to retrieve cached key-value pairs as stream data, Nahanni memcached uses the inter-VM shared memory and accesses the structured key-value data using loads and stores. Their benchmark results show that Nahanni memcached can reduce the latency of read operations (that hit in the cache) by 81% per operation, by between 29% and 45% on read-write workloads (including misses) as compared to standard memcached with the best-practise VM-based network mechanisms (e.g., vhost).

2.3 Concluding Remarks

In this chapter, we began by introducing some important concepts: virtual machines, fast inter-VM IPC, and different programming models. We introduced the advantages of using VMs and the importance of VMs in cloud computing and HPC environments. We also compared the load-store, transactional memory, and message-passing programming models and gave brief examples for each of them. Finally, we reviewed some previous work from the field of shared-memory IPC, minitransactions and MPI. They are related to our work introduced in the following chapters.

In the next chapter, we discuss the implementation details of minitransactions and barrier functions on top of Nahanni. Also, an empirical performance evaluation is presented.

Chapter 3

Minitransactions and Barriers on Nahanni

In the previous chapter, we introduced some important background concepts and related work for our shared-memory-based interprocess communication (IPC) mechanism for virtual machines (VMs). As stated earlier, our work is based on the Nahanni system for inter-VM shared memory. Therefore, the next logical questions are: What application programming interfaces (API) and synchronization mechanisms are suitable for use with Nahanni? For what applications are Nahanni useful?

As to the question of APIs and Nahanni, we want to consider programming models that are different from the typical load-store-based model of shared-memory programming. Related work with Nahanni memcached [32] already explores load-store and lock-based programming approaches with Nahanni. Instead, we implement a prototype of minitransactions [6] (Section 3.1) because it is an emerging API (from the transactional memory community [18]) and because minitransactions can support a hybrid of both distributed-memory (the original paper by Aguilera *et al.* [6]) and shared-memory implementations (this dissertation). As a prototype, the current minitransaction implementation has shown the abilities of reducing the complexity of concurrency control with good performance (Section 3.3.3).

As well, we port a smoothed particle hydrodynamics (SPH) simulator, FLUIDS v.1, to use Nahanni. We chose to port and parallelize FLUIDS as a case study because it is similar to a class of applications known as particle-in-cell (PIC) simulations, due to the importance of SPH techniques in computer graphics, and given our previous experience with the code base. Of course, no single application can represent all concerns, but SPH and FLUIDS covers many interesting domains from high-performance computing (HPC) and graphics.

At the beginning of this work, we thought that FLUIDS would need minitransactions for a proper implementation. And, in fact, there are many artificial and misleading ways to use minitransactions in FLUIDS. But, it turns out, with some sensible algorithmic design decisions, FLUIDS does not need any locking or minitransactions. Specifically, we parallelize FLUIDS using the single-program-multiple-data (SPMD) paradigm, with barrier synchronizations between four phases, and no other forms of synchronization. Since FLUIDS does not need minitransactions, we present only microbenchmarks for the minitransactions.

Our benchmarks show that minitransactions have better performance than traditional lock-based synchronization (Section 3.3.3) for low-contention scenarios. Furthermore, our benchmarks confirm the intuition that minitransactions have dramatically higher overheads as the amount of contention increases (Figure 3.11, Table 3.5), which is consistent with the intended use-cases for optimistic synchronization from the transactional memory community.

For our new barrier implementations (Table 3.1), we present both microbenchmarks (Section 3.3.2) and FLUIDS benchmarks (Section 3.3.4). We conclude that spin-based barriers within VMs can have comparable performance to barriers outside of VMs (Table 3.4). Also, we show how blocking-based barriers can be implemented between VMs, despite the involvement of multiple guest operating system (OS) kernels, but with substantially lower performance than spin-based barriers. Finally, we show that FLUIDS implemented for Nahanni, and using the new barriers, achieve reasonable speedups and performance relative to non-VM implementations (Figure 3.15).

Overall, the purpose of this chapter is to explore what APIs and applications are suitable for Nahanni, beyond the traditional load-store-based codes (e.g., Nahanni memcached). As proofs-of-concept, our implementation of minitransactions, barriers (in different forms), and FLUIDS show that Nahanni is suitable for more than just classic shared-memory programs.

3.1 Minitransaction Implementation

When manipulating data in shared memory using loads and stores (or reads and writes), programmers usually need to acquire and release locks, which is error-prone. However, minitransactions can relieve the programmer of the responsibility for explicit synchronization, as long as the operations are encapsulated within a minitransaction (Figure 2.5). Furthermore, the non-conflicting reads and writes in different minitransactions can proceed without blocking or interfering with each other, unlike traditional, typical synchronization-based techniques (Section 3.3.3).

Minitransactions usually contain three different kinds of items (Figure 3.1)—reads, compares, and conditional-writes [6]. The conditional-writes are conditional in the sense that *all* of the compare items must succeed for the conditional-write items to be applied to the shared data.

Figure 3.2 shows the steps taken upon a "User Submit" of (i.e., attempts to commit) a minitransaction. First, when committing a minitransaction, it will try to "Acquire Locks" for the three kinds of items. If any lock for any item is currently held by a different minitransaction, the acquisition is considered to have failed (i.e., no blocking for locks), and the minitransaction will drop or abort all locks it has already grabbed and return failure to the user. The minitransaction can then be re-tried (i.e., `do-while`-loop in Figure 2.5). This all-or-nothing locking algorithm prevents deadlock. If all the locks are successfully acquired, the minitransaction applies the operations indicated by the three kinds of items.

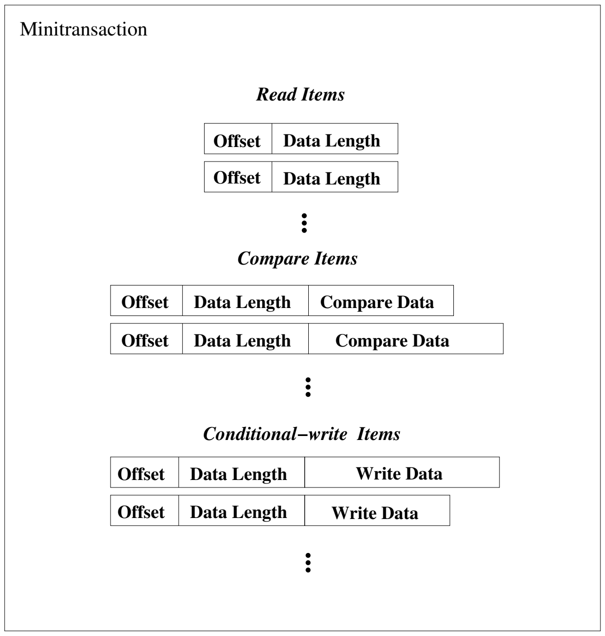


Figure 3.1: Read, compare and conditional-write items in Minitransactions.

Second, the minitransaction will retrieve the data specified by the read items. Third, the minitransaction will "Evaluate Compare Items" such that if *any* comparison fails, then the minitransaction is also aborted. Once again, the minitransaction can be re-tried, if desired. Fourth, if all the comparisons succeed, the minitransaction will apply the conditional-writes. At this point, the writes must succeed since all of the locks are properly held and all of the comparisons have already succeeded. Finally, the minitransaction releases all the locks and returns success to the user. Note that the locking and un-locking phases are completely hidden from programmers. Also, note that minitransactions can fail for two different reasons: failure to grab a lock and failure of a compare item. As a fundamental building block, the read, compare, and conditional-writes of a minitransaction can be used to implement common shared-data operations (e.g., incrementing counters, Figures 2.3 and 2.5).

3.2 Barrier Implementations

In the process of porting, parallelizing, and evaluating FLUID v.1 as a case study and proof-of-concept (Section 3.3.4), we realized that we need a reliable signalling mechanism (i.e., a reliable condition variable mechanism) between VMs. Thus, in this section we give the details of our barrier mechanism using two different implementations of condition variables.

3.2.1 Traditional Barriers

Before we present our barriers, we would like to show the traditional barrier implementation with a blocking Pthread mutex and a blocking condition variable. Figure 3.3 shows the details of the

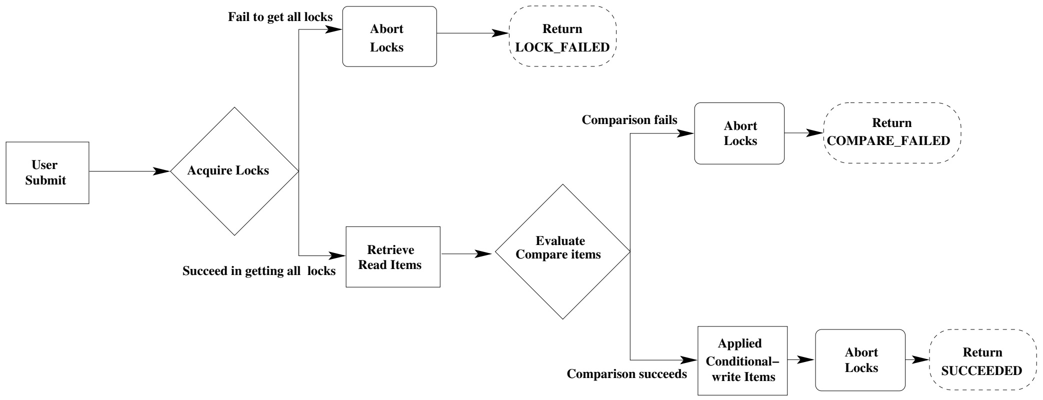


Figure 3.2: Submission steps of Minitransactions.

implementation in a multi-threaded program. `mutexc_var` and `threshold_var` are Pthread mutex and condition variables, respectively. `count` and `threadnum` are integers. `count` records how many threads have reached the barrier and `threadnum` represents the condition that should be satisfied. When a thread reaches the barrier, `count` is increased by one (line 7 in Figure 3.3). If the number of threads that have reached the barrier is less than `threadnum`, the current thread will wait on condition variable `threshold_var` (line 10 in Figure 3.3). Otherwise, the last thread that reaches the barrier will wake up other threads waiting on the condition variable (line 15 in Figure 3.3). `mutexc_var` is used to protect the critical section and make sure that only one thread can modify the variable `count` at one time.

However, since blocking mutexes and blocking condition variables are not currently supported across VM instances, we present two different barrier implementations. The first implementation uses *atomic memory-access operations* to implement a non-blocking condition variable (Figure 3.4). The second implementation uses virtio-serial, combined with `socat`, to implement a blocking condition variable (Figure 3.7).

3.2.2 Non-Blocking Condition Variable and Barrier

Unfortunately, Pthread mutex and condition variables do not work between different VM instances with Nahanni shared memory, because two different guest OS kernels are involved. Specifically, one guest OS cannot block or signal a thread belonging to another guest OS (i.e., a thread in a different VM instance) unless the two OS kernels are designed to interact, which is not the case in current Linux (and related) kernels.

Therefore, we have to replace the blocking Pthread mutex (Figure 3.3) with a non-blocking `spin_lock` and we implement our own (non-blocking) condition variable with *atomic memory-access operations*. Compare source codes in Figure 3.4 and Figure 3.3 to see the differences.

First, `spin_lock` replaces the blocking mutex lock at both the beginning and end of the function (line 3 and 21 respectively in Figure 3.4). Second, the waiting and signalling codes for condition

```
pthread_mutex_t mutexc_var;
pthread_cond_t threshold_var;

void Barrier()
{
    pthread_mutex_lock(&mutexc_var);
    count++;
    if(count < threadnum)
    {
        pthread_cond_wait(&threshold_var, &mutexc_var);
    }
    else if(count == threadnum)
    {
        count = 0;
        pthread_cond_broadcast(&threshold_var);
    }
    pthread_mutex_unlock(&mutexc_var);
}
```

Figure 3.3: Source code of traditional barrier implementation with Pthread mutex and condition variable.

variable are also changed (from line 9 to 19 in Figure 3.4). When a process waits on the condition variable, it releases the lock held by itself and enters an empty loop until all processes have arrived at the barrier and `alldone` is changed to 1. Releasing the lock is important because it gives other process a chance to enter the barrier. When the last process reaches the barrier, it modifies `alldone` to a positive value and enters another empty loop. The modification of `alldone` allows the processes that are trapped in the previous loop (line 10) to move on. Once these processes exit the loop, they use an atomic memory-access operation to modify `allexit`. The atomic memory-access operation `__syn_add_and_fetch`, a `gcc` compiler function, is able to do the addition and read of the parameter together without being interrupted. Thus, when all waiting processes exit the barrier (reaches `return` function in line 13), `allexit` will be equal to `vm_num-1`, which makes the `while` condition in line 18 false, and the last process jumps out of the loop and exits the barrier. Note that this implementation of a condition variable is non-blocking, because the two `while` loops keep the processes busy waiting until certain conditions are satisfied.

3.2.3 Blocking Condition Variable and Barrier

In the previous non-blocking implementation of a condition variable, the process occupies the central processing unit (CPU) when it busy waits for the condition to be satisfied. Busy waiting wastes CPU cycles when there are long delays in satisfying the condition, so we investigate another technique to build a blocking condition variable.

Virtio-serial was originally designed to provide a communication channel between the host and the guest. The VM will create a device and expose a serial port between guests and host. With the addition of `socat`, we can connect two serial ports together to build a communication bridge

```
void Barrier( pthread_spinlock_t* spin_lock, volatile int* count,
    volatile int* alldone, volatile int* allexit)
{
    pthread_spin_lock(spin_lock);
    (*count)++;
    (*alldone) = -1;
    (*allexit) = 0;
    if(*count < vm_num)
    {
        pthread_spin_unlock(spin_lock);
        while((*alldone) < 0)
            ;
        __sync_add_and_fetch(allexit, 1);
        return;
    } else if( *count == vm_num )
    {
        *count = 0;
        (*alldone) = 1;
        while((*allexit) < vm_num - 1)
            ;
    }
    pthread_spin_unlock(spin_lock);
}
```

Figure 3.4: Source code of our barrier implementation with `spin_lock` and *atomic memory-access operations*.

between guests. Figure 3.5 shows the architecture of two VMs connected with virtio-serial and `socat`.

Inside the VM, the serial port behaves like a regular file. The VM can use the `open` function to open the port, and use the `read`/`write` functions to receive/send data through the port. The data moves from the guest to the host along the communication channel and is forwarded from the host to another guest via `socat`. To use virtio-serial, we need special commands to start up VMs and configure the environment. In the following discussion, we use two VMs as an example to show how to set up the virtio-serial connection for these two VMs. Note that the basic technique can be extended to arbitrary numbers of VMs.

First, before starting the VMs, two socket files (i.e., `/tmp/from*` files) are created for virtio-serial (lines 1 and 2, Figure 3.6). Second, we start up the VMs with extra parameters to create the virtio-serial device (the last few parameters in line 4 and 6). Each VM uses one of the two socket files we created previously. Last, it is important to use `socat` to connect these two socket files together (commands shown in lines 8 and 9 in Figure 3.6). Otherwise, the VMs can only communicate with the host but not each other.

So far, we have described how to set up VMs with virtio-serial. Next, we show the implementation of our blocking condition variable. Inside the VMs, the virtio-serial port is exposed as a port in `/dev/vport0p1` to the guest. Now the VM is able to open this port and communicate with each other via regular `read` and `write` calls. The source code in Figure 3.7 shows the im-

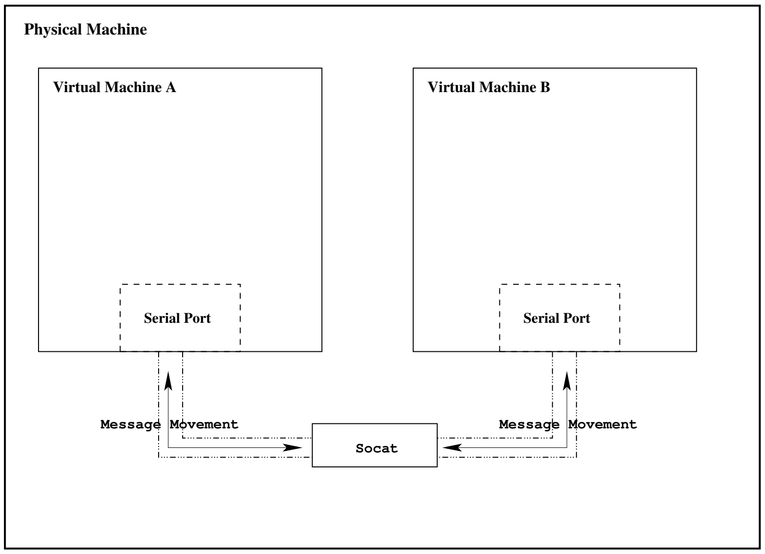


Figure 3.5: The architecture of VMs connected with virtio-serial and `socat`.

```
$ socat UNIX-LISTEN:/tmp/from_VM1_to_VM2
$ socat UNIX-LISTEN:/tmp/from_VM2_to_VM1

$ qemu-system-x86_64 -smp 4 -hda ... -chardev
    socket,path=/tmp/from_VM1_to_VM2,server,nowait,id=VM1toVM2
    -device virtio-serial -device
    virtserialport,chardev=from_VM1_to_VM2,name=VM1toVM2

$ qemu-system-x86_64 -smp 4 -hda ... -chardev
    socket,path=/tmp/from_VM2_to_VM1,server,nowait,id=VM2toVM1
    -device virtio-serial -device
    virtserialport,chardev=from_VM2_to_VM1,name=VM2toVM1

$ socat UNIX-CONNECT:/tmp/from_VM1_to_VM2 /tmp/from_VM2_to_VM1
$ socat UNIX-CONNECT:/tmp/from_VM2_to_VM1 /tmp/from_VM1_to_VM2
```

Figure 3.6: Example commands to startup two VMs with virtio-serial connection.

plementation. We replace `pthread_cond_wait` and `pthread_cond_broadcast` functions with `MyCondWait` and `MyCondBroadcast`, respectively. The functions of `poll_read` and `poll_write` are implemented with the `poll` function (line 5 and 22 in Figure 3.8). Thus, when processes wait on the condition, they actual are blocked on reading from a port. Unless the last process reaches the barrier and sends an interrupt via the serial port, the `poll` function will not return and wake up the waiting processes. Note that the code block in lines 22 to 24 of Figure 3.7 implements the same functionality as `pthread_cond_wait`: 1) release the associated lock; 2) wait until to be waken up; 3) take the associated lock again.

```
void Barrier( pthread_spinlock_t* spin, int* count )
{
    pthread_spin_lock(spin);
    (*count)++;
    printf("%d\n", *count);
    if(*count < vm_num)
    {
        MyCondWait(spin);
    }
    else if(*count == vm_num)
    {
        *count = 0;
        MyCondBroadcast();
    }
    pthread_spin_unlock(spin);
}

void MyCondWait( pthread_spinlock_t* spin )
{
    int buf;
    struct Communicate_Message msg;
    pthread_spin_unlock(spin);
    poll_read(pollgfd, &msg, sizeof(msg), -1);
    pthread_spin_lock(spin);
}

void MyCondBroadcast()
{
    struct Communicate_Message msg;
    set_message(&msg, VMID, -1, MSG_BROADCAST, "Everyone reaches
        barrier!\n");
    poll_write(pollgfd, &msg, sizeof(msg), -1);
}
```

Figure 3.7: Source code of our barrier with virtio-serial port.

```
void poll_read(struct pollfd *pollgfd, struct Communicate_Message
    *msg, int size, int timeout)
{
    int ret;
    pollgfd[0].events = POLLIN;
    ret = poll(pollgfd, 1, timeout);
    if (ret < 0)
    {
        printf("Error has occurred while polling\n");
        exit(1);
    } else if (pollgfd[0].revents & POLLIN)
    {
        read(pollgfd[0].fd, msg, size);
        printf("%s", msg->data);
    } else if (pollgfd[0].revents & POLLHUP)
        printf("hangup has occurred\n");
}

void poll_write(struct pollfd *pollgfd, const struct
    Communicate_Message *msg, int size, int timeout)
{
    int ret;
    pollgfd[0].events = POLLOUT;
    ret = poll(pollgfd, 1, timeout);
    if (ret < 0)
    {
        printf("Error has occurred while polling\n");
        exit(1);
    } else if (pollgfd[0].revents & POLLOUT)
    {
        ret = write(pollgfd[0].fd, msg, size);
        printf("Send out message\n");
    } else if(pollgfd[0].revents & POLLHUP)
        printf("hangup has occurred in write\n");
}
```

Figure 3.8: Source code of `poll_read` and `poll_write`.

Table 3.1: Technique combinations for barrier implementations

Implementations	Barrier-No VM+ Spin/Atomic Ops	Barrier-No VM+ Pthread	Barrier-VM+ Spin/Atomic Ops	Barrier-VM+ Virtio-serial
Lock Type	Spin Lock	Mutex Lock	Spin Lock	Spin Lock
Condition Variable Type	Atomic Operation	Pthread Condition Variable	Atomic Operation	Virtio-serial
Use VMs	No	No	Yes	Yes
Shared Data Location	Memory-mapped file	Global variables in an application	Nahanni shared memory	Nahanni shared memory

3.3 Application and Empirical Evaluation

In this section, we evaluate the performance of our new barrier functions and minitransactions. First, we compare four different versions of the barrier function (e.g., inside and outside VMs, blocking and non-blocking condition variable) (Table 3.1). Second, we compare the performance of minitransactions using an increment counter microbenchmark. Specifically, minitransactions are compared with explicit synchronization using non-blocking spin locks, as well as comparing the performance of running inside and outside of VMs (Table 3.2). Third, we parallelize the FLUIDS simulator using barrier synchronizations (Table 3.3). Thus, both microbenchmarks and the FLUIDS application are used to evaluate the barriers.

Our goal is to answer the following questions:

1. How fast are our barrier implementations (Figures 3.4 and 3.7), relative to traditional barriers (Figure 3.3)?

   The performance of "Barrier-VM+Spin/Atomic Ops" (Table 3.1) has the lowest overhead when running inside a VM (Section 3.3.2). It is almost as efficient as "Barrier-No VM+Spin/ Atomic Ops", which has no VM overheads. The traditional "Barrier-No VM+Pthread" implementation is an order of magnitude slower than "Barrier-VM+Spin/Atomic Ops". "Barrier-VM+Virtio-serial" is even slower, another order of magnitude slower than "Barrier-No VM+ Pthread", but the virtio-serial-based approach does provide true blocking and signalling between VMs.

2. How fast is a minitransaction, relative to lock-based synchronization, when used to update a shared counter?

   "Counter-VM+Minitransaction" provides the most abstraction by making the synchronization implicit, instead of explicit. Also, "Counter-VM+Minitransaction" is faster than "Counter-No VM+Spin Lock" and "Counter-VM+Spin Lock" for low-contention situations (Section 3.3.3). However, there are significant overheads when minitransactions are inappropriately used in high-contention situations.

Table 3.2: Technique combinations for increment counter implementations

Implementations	Counter-No VM+ Spin Lock	Counter-VM+ Spin Lock	Counter-VM+ Minitransaction
Mutual Exclusion	Explicit, Spin Lock	Explicit, Spin Lock	Implicit, Minitransaction
Use VMs	No	Yes	Yes
Shared Data Location	Memory-mapped file	Nahanni shared memory	Nahanni shared memory

Table 3.3: Technique combinations for FLUIDS implementations

Implementations	FLUIDS-No VM+ Pthreads Barrier	FLUIDS-VM+ Spin/Atomic Ops Barrier
Barrier Type	Barrier-No VM+ Pthread	Barrier-VM+ Spin/Atomic Ops
Use VMs	No, multiple threads in host	Yes, multiple processes over VMs
Shared Data Location	Global variables in a program	Nahanni shared memory

3. How much overhead does the barrier synchronization inside a VM add to the FLUIDS simulator?

   The “FLUIDS-No VM+Pthread Barrier” implementation is a little faster than “FLUIDS-VM+Spin/Atomic Ops Barrier” when using 2 and 4 processes or threads (Figure 3.15). In other words, there is a measurable overhead associated with using our non-blocking barrier implementation with FLUIDS when running inside a VM with 4 or fewer processes or threads. However, when we use 8 processes or threads, “FLUIDS-VM+Spin/Atomic Ops Barrier” is comparable to “FLUIDS-No VM+Pthread Barrier”.

3.3.1 Platform

Process versus Threads: In the following benchmarks, the phrase “2 processes/threads” means that the benchmark is run with 2 processes if VMs are used (i.e., one process per VM), or 2 threads if no VMs are used (i.e., run directly on the host). Similarly, “4 processes/threads” and “8 processes/threads” mean either 4 or 8 VMs (one process per VM), or 4 or 8 threads (no VMs), are used, respectively.

Software: The host OS’s Linux distribution is Fedora 11 and the guests are Ubuntu 10.04. The host has Linux kernel version 2.6.37. The guests have Linux kernel version 2.6.35. The hypervisor is the QEMU/KVM version 0.12.5, with the same Nahanni/ivshmem code as released. Each VM is configured with 2 virtual CPUs and 1 GB of RAM. The Nahanni device file is 1 GB and shared by all VMs. All benchmarks and applications are compiled using gcc version 4.4.3.

Hardware: Our host server has two Intel Xeon X5550 processors, running at 2.67 GHz, and with 48 GB RAM. There are a total of two sockets and 8 cores. There are up to 16 HyperThreads. All experiments are done within the single server.

Data Points: All data points are the average of 5 runs, and we have calculated the standard deviation. But, since the standard deviation is small, we do not use error bars in our graphs.

3.3.2 Barrier Microbenchmark

The benchmark uses a barrier synchronization within a while loop. The barrier count (i.e., number of iterations of the while loop) is a parameter, and performance is presented as both the number of barriers completed per second, and the total number of seconds required for a given barrier count (Table 3.4).

We have four different barrier implementations. The associated techniques are shown in Table 3.1. Both "Barrier-No VM+Spin/Atomic Ops" and "Barrier-VM+Spin/Atomic Ops" use the non-blocking spin lock and atomic memory-access operations (Figure 3.4). "Barrier-No VM+Spin/ Atomic Ops" uses a memory-mapped file as the shared memory and runs on the host without using a VM. By not using a VM for this configuration, we gain insight into the overheads associated with VMs. In contrast, "Barrier-VM+Spin/Atomic Ops" uses Nahanni shared memory and runs inside guest VM instances.

"Barrier-No VM+Pthread" uses the approach shown in Figure 3.3 to implement a barrier function. Since "Barrier-No VM+Pthread" uses a blocking Pthread mutex lock and a blocking condition variable, it helps us to understand the performance difference between non-blocking and blocking primitives for the lock and condition variable.

"Barrier-VM+Virtio-serial" uses virtio-serial as the communication channel between VMs (Figure 3.7). Recall that virtio-serial is the only existing mechanism within QEMU/KVM to implement a blocking barrier on top of Nahanni shared memory. Therefore, both "Barrier-No VM+Pthread" and "Barrier-VM+Virtio-serial" are blocking barriers, differing only in whether or not a VM is used.

Table 3.4 shows the total running time of different barrier implementations. Note that the times have a large range (e.g., 3.01 seconds to 1,341.71 seconds) for each process to perform 5 million barrier steps. By comparing the values of barriers per second, we can easily identify their relative performance.

First, "Barrier-No VM+Spin/Atomic Ops" and "Barrier-VM+Spin/Atomic Ops" are comparable to each other and always achieve the fastest barrier times. Therefore, the VM does not introduce significant overheads. Second, "Barrier-No VM+Pthread", which uses blocking primitives, is always an order of magnitude slower than "Barrier-No VM+Spin/Atomic Ops", which uses non-blocking primitives. Therefore, we conclude that blocking primitives have higher overheads for this particular microbenchmark. Third, "Barrier-VM+Virtio-serial" is the slowest, and another order of magnitude slower than "Barrier-No VM+Pthread". There is a trade-off between using a blocking barrier between VMs and performance. Fourth, not surprisingly, the more processes or threads that are used

Table 3.4: The run time of different barrier implementations. Note that each process completes 5 million barrier steps.

		Barrier-No VM+Spin/Atomic Ops	Barrier-No VM+Pthread	Barrier-VM+Spin/Atomic Ops	Barrier-VM+Virtio-serial
2 process or threads	Total time(s)	3.01	24.21	3.52	470.14
	barriers per second	1.66×10^6	2.06×10^5	1.42×10^6	1.06×10^4
4 process or threads	Total time(s)	7.28	81.72	7.31	870.41
	barriers per second	6.68×10^5	6.12×10^4	6.84×10^5	5.74×10^3
8 process or threads	Total time(s)	13.99	200.68	13.53	1341.71
	barriers per second	3.57×10^5	2.50×10^4	3.69×10^5	3.73×10^3

(i.e., 2, 4 or 8), the longer it takes for all implementations to perform a barrier.

As already noted, both the Pthread mutex and the condition variable used in "Barrier-No VM+Pthread" are blocking primitives. Since the microbenchmark calls the barrier within a tight while loop, busy-waiting barriers (based on a non-blocking spin lock) (e.g., "Barrier-No VM+Spin/Atomic Ops") are faster than barriers based on blocking primitives (e.g., "Barrier-No VM+Pthread"). Moreover, "Barrier-VM+Virtio-serial" always uses the virtual serial ports to send interrupt messages, which requires crossing over multiple protection domains (e.g., guest-to-host, host-to-guest) as shown in Figure 3.5. So, the overheads for "Barrier-VM+Virtio-serial" are much higher. But, no matter how low the performance of "Barrier-VM+Virtio-serial", it is the only available technique to provide a blocking barrier across VM instances.

Although other techniques might be used to further optimize the barrier performance, we leave that as future work. Also, from the application benchmark (Section 3.3.4), we show that our barrier implementation is not an significant bottleneck in the parallel FLUIDS simulator.

3.3.3 Increment-Counter Microbenchmark

To evaluate the performance of minitransactions, we implement an incremental-counter microbenchmark with three variations (Table 3.2). Both "Counter-No VM+Spin Lock" and "Counter-VM+Spin Lock" use non-blocking spin locks to protect critical sections (Figure 3.9). "Counter-No VM+Spin Lock" uses a memory-mapped file as the shared memory between processes, and runs directly on on the host. "Counter-VM+Spin Lock" uses Nahanni shared memory between VM instances, and runs one process per VM. In contrast, "Counter-VM+Minitransaction" uses minitransactions to update the counters in Nahanni shared memory, which requires no explicit lock. Figure 3.9 and Figure 3.10 provides the relevant pseudocode.

```
do N times updates
    Lock(Counter)
    Counter[index]++
    Unlock(Counter)
```

Figure 3.9: Pseudocode of incremental counter with explicit lock.

Note that we have multiple counters residing in shared memory and `index` points to the counter we want to update. The number of counters is equal to the number of processes used in the microbenchmark. For the explicit locking methods, the counters are protected by one coarse-grained lock, which is a common programming strategy. But for minitransactions, any required synchronization is handled transparently by the system. Specifically, our implementation of minitransactions uses atomic update instructions with retry, which is a sophisticated technique not used by the average programmer. Each process completes 4 million counter increments, which is `N` in the pseudocode.

Different Contention Scenarios: To better understand the use cases of minitransactions, we run the increment-counter benchmark under three scenarios—no contention, medium contention and high contention for the shared data. Figure 3.11 shows the results of the increment-counter benchmark under the different contention scenarios.

The scenario of “no contention” is implemented by making each process increment a different counter. Specifically, each process uses its process identifier as the `index` value. Since different processes update different counter values, there is no contention for the shared counter data. However, note that there is still contention for the coarse-grained lock.

The scenario of “medium contention” forces every process to increase a counter in a round-robin way. Specifically, the `index` increases by one after each counter increment, with a wrap around when `index` points beyond the last counter. Stochastically, different processes will occasionally try to increment the same counter value, resulting in some shared-data contentions.

Finally, the scenario of “high contention” is implemented by forcing all processes to use the same `index` value. Therefore, there is a single counter that all processes contend over as they try to increment the value.

From Figure 3.11, we note that the VM does not introduce significant overheads for this microbenchmark, because “Counter-VM+Spin Lock” (red bar) is comparable to “Counter-No VM+Spin Lock” (blue bar) for every data point. Another observation is that, for a fixed process count, “Counter-No VM+Spin Lock” and “Counter-VM+Spin Lock” have fairly stable performance under all three contention scenarios. But, the run times for all variations increase when the number of processes increase (e.g., 2, 4, and 8).

“Counter-VM+Minitransaction” (yellow bar) varies with different contention scenarios even if they use the same number of processes. “Counter-VM+Minitransaction” is about 2 times and 1.5

```
do N times updates
    set_minitransaction t
    do until t commits successfully
        local_counter = counter[index]
        new_counter = local_counter + 1

        /*compare item compares local_counter value with
            counter[index] in shared memory*/
        t.compareitem(index, local_counter)

        /*conditional-write item tries to update new_counter value to
            counter[index] in shared memory*/
        t.writeitem(index, new_counter)
        t.submit
```

Figure 3.10: Pseudocode of incremental counter with minitransactions.

times faster than "Counter-No VM+Spin Lock" and "Counter-VM+Spin Lock" under both "no contention" and "medium contention" scenarios, respectively. But "Counter-VM+Minitransaction" suffers high overheads under "high contention" scenario.

The stability of performance for the non-minitransaction variations, despite changes in contention, is likely because a single coarse-grained lock is used to protect all counters. "Counter-No VM+Spin Lock" and "Counter-VM+Spin Lock" always contend for the lock no matter which counter they try to increase. And the more processes that contend for the lock, the longer the lock wait times, which explains the higher run times when adding more processes/counters.

Recall that non-conflicting operations in a minitransaction will not block or interfere with other operations in different minitransactions. Therefore, under the "no contention" scenario, minitransactions have higher concurrency than non-minitransaction variations because there is no data contention, therefore the atomic update instructions have no retries. Although the "medium contention" scenario has data contention, which will lead to unsuccessful minitransaction commits (details in Figure 3.2), the amount of minitransaction failure is small as compared to the total counter increments (Table 3.5). But the failure count for minitransactions is significant under the "high contention" scenario. For example, there are a total 1.2×10^8 failures for 8 processes under the "high contention" scenario, while the total counter increments is only 3.2×10^7 (i.e., 4.0×10^6 per process). The failures-vs-number-of-increments ratio explains why the performance of minitransactions is much worse than the non-minitransaction variations for the "high contention" scenario. These results confirm what the transactional memory community has already concluded: minitransactions are not suitable for high-contention scenarios due to the higher overheads. Although a carefully designed lock-based method (e.g., using fine-grained locks) might be faster than minitransactions, minitransactions can offer an easier programming interface (i.e, implicit synchronization) without suffering significant overheads (Figure 3.11).

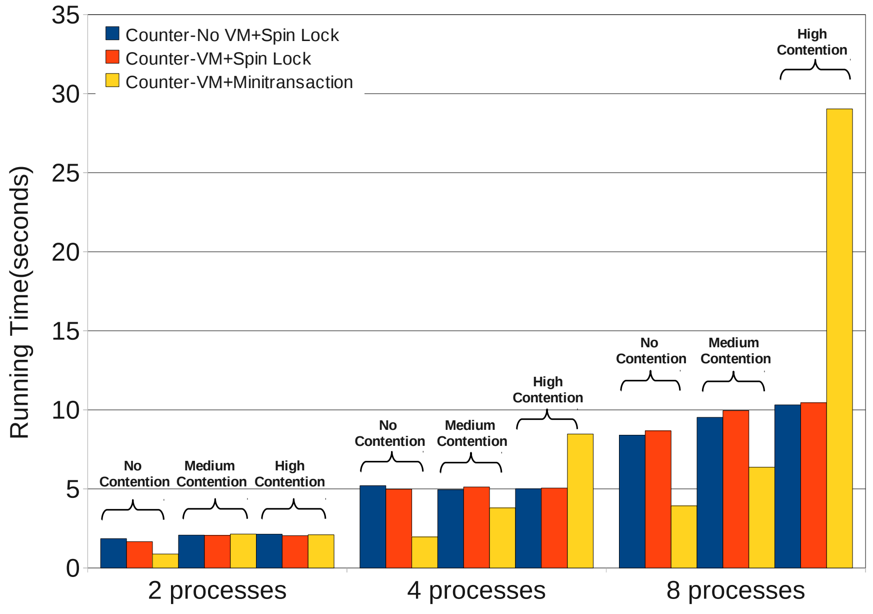


Figure 3.11: Total running time of the incremental counters under different contention scenarios. Each process updates the counters 4 million times.

Table 3.5: Minitransactions failure counts for both medium and high-contention scenario.

	Medium Contention	High Contention	Total Counter Increment (4.0×10^6 per process)
2 processes	1.9×10^6	3.6×10^6	8.0×10^6
4 processes	4.2×10^6	2.7×10^7	1.6×10^7
8 processes	7.9×10^6	1.2×10^8	3.2×10^7

3.3.4 FLUIDS Simulator

FLUIDS, developed by Rama Hoetzlein [19], is an open-source fluid simulator based on the SPH method. The basic idea of SPH is to represent a fluid as a set of discrete particles. Each particle has some properties (e.g., density, force, velocity, position) and in each time step these properties are updated according to the laws of physics, based on interactions with neighbouring particles (Figure 3.12). The solid (blue) circles represent the particles used to simulate fluid. For a given particle, represented by the red solid circle, a neighbourhood of particles (inside the large dashed circle) can influence the (red) particle during the next time step.

Figure 3.13 shows the basic computational steps for the standard SPH method. To find neighbouring particles more efficiently, a neighbourhood table is built before any computations. The space in which particles can reside is divided into adjacent cells (Figure 3.12) and every cell records the identities of particles within it. When a given particle tries to find its neighbouring particles, it

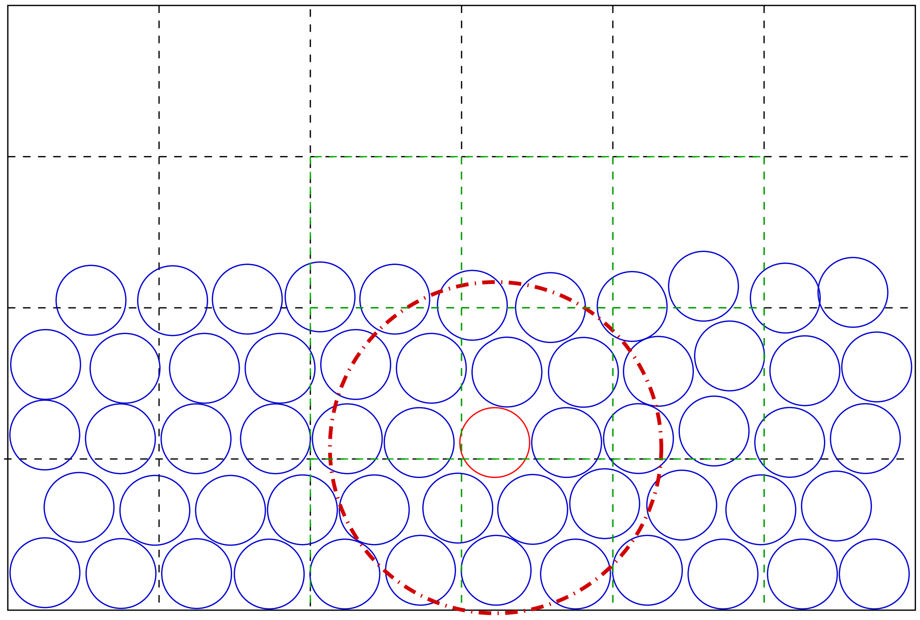


Figure 3.12: Example of fluid simulation using SPH method in two dimension.

only queries its neighbouring cells. Once the neighbouring particles are identified, the given particle can use them to compute the density, pressure, and force acting on it. Next, the velocity and position of this particle can also be updated. This basic process is repeated with each time step.

The original FLUIDS simulator was a sequential program. We parallelized the application as a case study for our VM-based barrier synchronization. Pseudocodes are shown in Figure 3.14. In the FLUIDS simulator, particles are organized as an array. Each element of the array stores the properties of one particle and the indexes of the array are used as the identifiers of particles. Embedded links within particles, cells, and the neighbourhood table make it easy to find all particles in a given neighbourhood. To parallelize the SPH computation, the particle array and the neighbourhood table are put into shared memory. Moreover, the array is partitioned into multiple non-overlapping subsets (i.e., for data parallelism). Each process/thread takes charge of one subset, shown as `PARTICLE_SUBSET` in the pseudocode. Only the specific process/thread of a given partition is allowed to update (i.e., write to) the particles that it owns, as per the owners-computes rule. Therefore, locks are unnecessary and barriers are used to synchronize between phases of the computation.

Table 3.3 shows the different techniques used to parallelize the FLUIDS simulator. “FLUIDS-No VM+Pthread Barrier” runs on the host, uses multiple threads, and uses the blocking barrier implementation. “FLUIDS-VM+Spin/Atomic Ops Barrier” runs one process per VM and uses the non-blocking barrier implementation. Both parallel implementations simulate 65,536 particles and run for 1,000 time steps, which result in 4,000 barrier synchronizations, given the four core phases.

Figure 3.15 shows the total running time for both parallel implementations with various process/thread counts. “FLUIDS-No VM+Pthread Barrier” is a little bit faster than “FLUIDS-VM+Spin/ Atomic Ops Barrier” with 2 and 4 process/thread counts. With 8 processes/thread, “FLUIDS-VM+Spin/Atomic Ops Barrier” and “FLUIDS-No VM+Pthread achieve comparable performance.

Overall, we conclude that our non-blocking barrier implementation for VMs (i.e., based on Nahanni shared memory) is sufficiently high performance for the FLUIDS application, as compared to the traditional blocking barrier for non-VM platforms. If required, our blocking barrier for inter-VM computations (i.e., based on virtio-serial) can be used, but with substantially less performance (not shown in Figure 3.15).

3.4 Concluding Remarks

When Nahanni is described as a shared-memory IPC mechanism between VMs, it is natural to think of the familiar load-store, lock-based programming model for shared memory. However, we claim that the benefits of Nahanni also extend to supporting new APIs and programming models, such as minitransactions. Therefore, we successfully implemented minitransactions on top of Nahanni. In a performance evaluation, the increment-counter microbenchmark shows that minitransactions, if used properly, are at least 1.5 times faster than non-minitransaction variations for “no contention” and “medium contention” scenarios (Section 3.3.3).

Moreover, we implemented VM-based barrier synchronizations. We also ported and parallelized the FLUIDS simulator using our VM-based barriers. The final results show that our VM-based barrier synchronization has comparable performance to non-VM-based barriers.

In this chapter, we have explored the non-traditional minitransaction model. In the next chapter, we explore MPI-Nahanni, representing the traditional message-passing model.

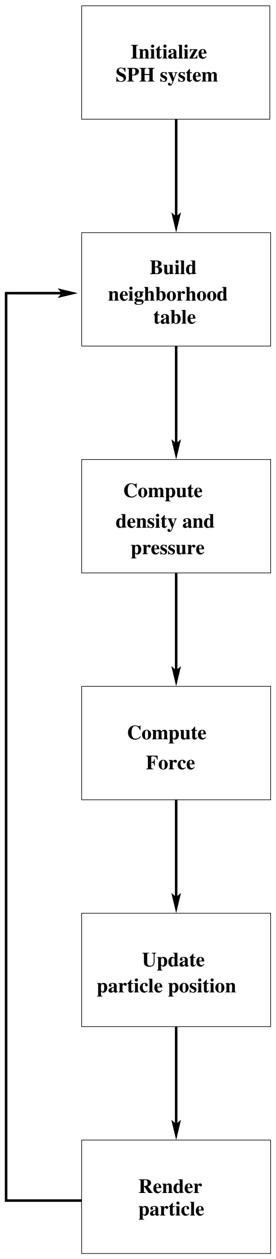


Figure 3.13: The standard SPH procedures.

```
Do N iterations
    Master process/thread builds Neighbourhood Table sequentially
    Barrier
    For each particle in PARTICLE_SUBSET
        find out neighbouring particles
        update density
        update pressure
    Barrier
    For each particle in PARTICLE_SUBSET
        find out neighbouring particles
        update force
    Barrier
    For each particle in PARTICLE_SUBSET
        update velocity
        update position
    Barrier
```

Figure 3.14: Pseudocode of parallel SPH method with lock and synchronization.

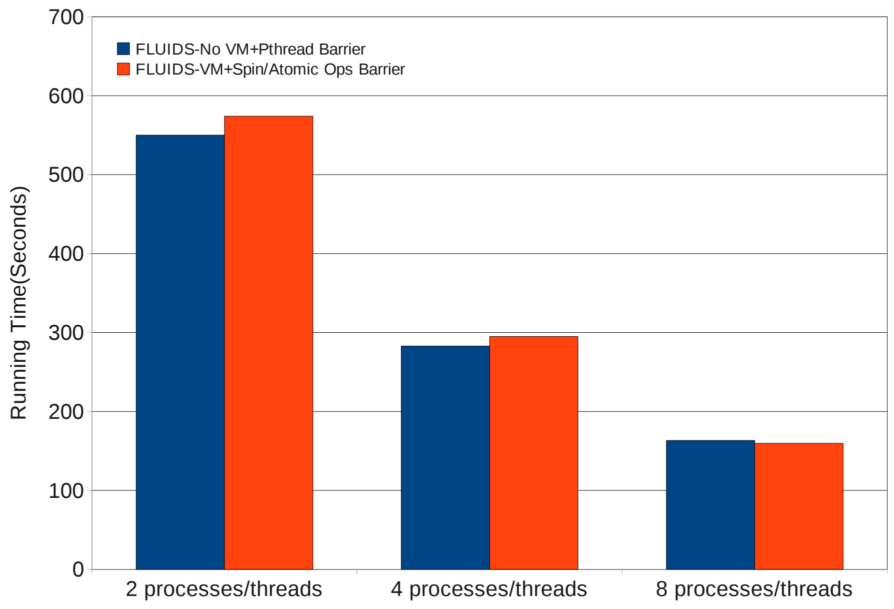


Figure 3.15: Total running times of parallel FLUIDS simulator with our barrier synchronization.

Chapter 4

MPI Nahanni

In the previous chapter, we demonstrated that Nahanni can be used for a new programming model and application programming interface (API) by showing the implementations of minitransactions and barriers for inter-virtual machine (VM) shared memory. We also presented the evaluation results of minitransactions and FLUIDS which was ported and parallelized with our barrier functions.

Now, we address the following questions: How well can Nahanni support an existing programming model or API? What performance benefits, if any, can Nahanni provide to existing models and APIs?

To answer these questions, we implement MPI-Nahanni. It is a port of MPICH2 that uses Nahanni shared memory as its inter-VM communication channel. We choose to investigate MPICH2 because it is a well-known and high-performance implementation of the Message-Passing Interface (MPI) standard. Also, since MPICH2-Nemesis [11] already has a shared-memory implementation, we show that the port of MPICH2 to Nahanni is straightforward. Furthermore, existing MPI applications can run on MPI-Nahanni *without* any code modification. Thus we can use the present microbenchmarks (e.g., NetPIPE, OSU and Intel MPI Benchmark (IMB) in Section 4.2.3) and an application (e.g., GAMESS) (Section 4.2.4) as our performance benchmarks for MPI-Nahanni directly.

The microbenchmarks and application benchmark all show that MPI-Nahanni has up to an order of magnitude (sometimes even more) better performance (both in bandwidth and latency) than the current VM-based interprocess communication (IPC) techniques (e.g., Figure 4.2, Figure 4.4). Moreover, VM-based MPI-Nahanni can achieve almost the same performance as outside VMs (e.g., Figure 4.7, Figure 4.10). In other words, MPI-Nahanni achieves nearly the same communication performance as MPI running on the host hardware directly, which is a promising result for the future of MPI-based applications in VMs.

4.1 Design Overview

MPI-Nahanni, our new MPI system, uses a new device channel `ch3:nahanni`, that is based on

MPICH2's `ch3:nemesis` channel. Since Nemesis already has support for using shared memory (either System V or memory-mapped files) for intra-node communication, porting Nemesis to Nahanni shared memory reuses almost all of the same techniques and mechanisms already in the system (Figure 4.1): Data is moved through shared memory, which helps to reduce memory-to-memory copies. And, performance-sensitive synchronization is done using shared memory (including non-blocking, lock-free algorithms), which helps to reduce protection domain context switches.

Figure 4.1 shows the architecture of MPICH2-Nemesis and MPI-Nahanni. In both Figure 4.1(a) and 4.1(b), the solid arrows indicate data communications where shared memory is used; the dashed arrows represent the communication where the network is used. Note that shared memory is only used for intra-VM communication in MPICH2-Nemesis, but Nahanni shared memory is used for both intra- and inter-VM communication in MPI-Nahanni.

A notable departure from MPICH2-Nemesis within MPI-Nahanni is the handling of large message transfers. Intuitively, optimizations to improve latency (e.g., pre-allocate all required resources) can conflict with optimizations to improve bandwidth (e.g., use additional resources as an optimization, but only allocate resources when needed to avoid unnecessary resource usage). Therefore, it makes sense to have a protocol for short messages and a separate protocol for large messages.

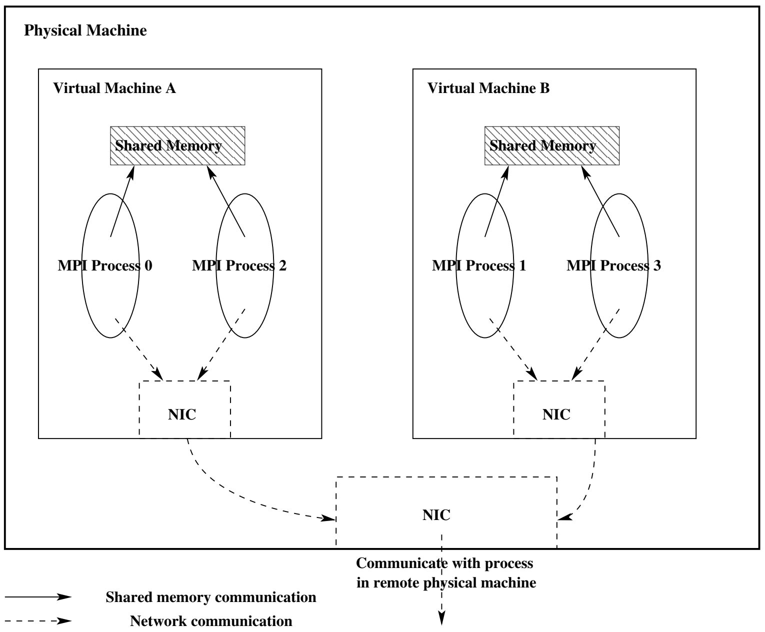
With MPICH2-Nemesis, a large message transfer (LMT) interface already exists for efficiently transferring large messages between co-located processes. Essentially, a System V shared-memory region, or a temporary memory-mapped file, is created on demand (i.e., only when large messages need to be transferred) for each pair of MPI processes that exchange a large message. The shared-memory region is a large buffer, which improves bandwidth. But, since buffers are resource intensive, no shared memory resource is allocated for LMT if a communication pair does not actually transfer large messages.

However, all Nahanni shared memory must be created at VM start-up time. Therefore, in porting MPICH2-Nemesis, we choose to use a large Nahanni shared-memory region and pre-divide the region into many non-overlapping (i.e., 256 KB) chunks. When a communication pair requires LMT, it reserves one of the pre-allocated chunks based the sender's and receiver's identifiers.

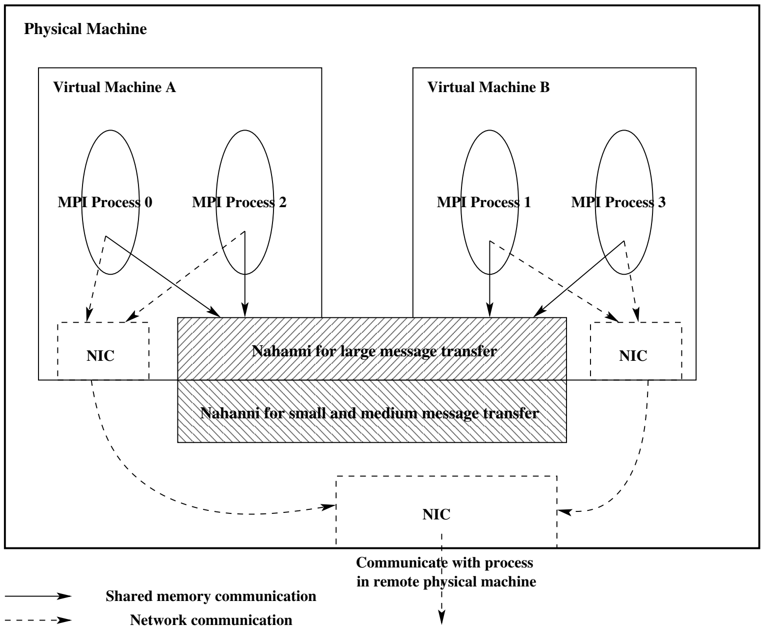
One of the architectural advantages of Nahanni is that the inter-VM shared memory looks and behaves just like shared memory between threads. Porting MPICH2-Nemesis to Nahanni is straightforward with only a few implementation curiosities, such as the LMT protocol. Therefore, we now proceed directly to addressing the performance questions.

4.2 Applications and Empirical Evaluation

In this section we evaluate the performance of our new *MPI-Nahanni* system relative to various combinations of IPC-related mechanisms (Table 4.1). Our goal is to quantify the benefits of using Nahanni shared memory, vhost paravirtualization, and Nemesis shared memory mechanisms. On the one hand, the long-standing work on using shared memory for IPC suggests that high performance



(a) Architecture of MPICH2-Nemesis. Process 0 and 2 are local and communicate with each other via shared memory; process {0, 2} and {1, 3} are non-local and communicate with each other via network.


Physical Machine
Virtual Machine A
Virtual Machine B
MPI Process 0
MPI Process 2
MPI Process 1
MPI Process 3
NIC
Nahanni for large message transfer
NIC
Nahanni for small and medium message transfer
NIC
Communicate with process in remote physical machine
Shared memory communication
Network communication

(b) Architecture of MPI-Nahanni. Process 0 and 2 are local and communicate with each other via Nahanni shared memory; although process {0, 2} and {1, 3} are non-local, they still communicate with each other via Nahanni shared memory.

Figure 4.1: Architectures of MPICH2-Nemesis and MPI-Nahanni

should be possible. On the other hand, high performance shared-memory IPC in Linux kernel-based virtual machine (KVM) has not been well-explored, and (in fact) there is some debate as to whether the vhost and virtio paravirtualization [27, 29] approach is just as fast as using shared memory. Therefore, we use a variety of benchmarks to establish that MPI-Nahanni can achieve high performance in practice. And, we show that with the current implementation and design of vhost and virtio, the performance advantage of MPI-Nahanni is significant enough to be considered.

We attempt to answer the following questions:

1. *How fast is MPI-Nahanni relative to other VM-based IPC mechanisms?*

   Our conclusion for microbenchmarks is that MPI-Nahanni always has the highest bandwidth and the lowest latency (sometimes by an order of magnitude or more) relative to other VM-based IPC mechanisms. For applications, the performance benefit of MPI-Nahanni is variable, but it can be substantial, and is proportional to how much time that application spends within the MPI libraries.

2. *How much faster can MPI-Nahanni become on our platform?*

   Pragmatically, MPI-Nahanni appears to be *almost* as fast as we can expect for many (but not all) cases, given that it achieves nearly the same performance as NoVM-MPI-Nemesis, which does not have any of the VM overheads. For the microbenchmarks (Section 4.2.2) and GAMESS (Section 4.2.4), MPI-Nahanni also tracks the performance of NoVM-MPI-Nemesis closely. Conceptually, since NoVM-MPI-Nemesis does not use VMs, it represents an upper bound on performance on our hardware platform.

4.2.1 Platform and Methodology

Software: We use MPICH2 version 1.3 for all our microbenchmarks, and the GAMESS application. All 4 systems (the columns of Table 4.1) are based on MPICH2 and are compiled with `--enable-fast=O3,nochkmsg, notiming,ndebug`.

Our MPI-Nahanni implementation is based on MPICH2 and specifically the `ch3:nemesis` channel, but MPI-Nahanni uses Nahanni shared-memory (i.e., instead of System V shared memory) for both intra-VM and inter-VM IPC, although some performance-insensitive initialization IPC is still carried over a socket. As evaluated, MPI-Nahanni also uses the vhost paravirtualization, since that is a “best practice” for Linux KVM, although vhost only benefits the socket IPC part of MPI-Nahanni.

We use the abbreviated name *MPI-vhost* for the combination of unmodified MPICH2, with the default `ch3:nemesis` channel, but with vhost paravirtualization and bridge networking in the KVM configuration. The use of vhost is expected to improve inter-VM performance, since that traffic is carried over network sockets, and is consistent with the results presented below. Note that the `ch3:nemesis` channel uses shared memory, via a memory-mapped file, for intra-VM (but not inter-VM) communication.

We use the abbreviated name *MPI-bridge* for the combination of unmodified MPICH2, with the `ch3:nemesis` channel, but *without* vhost paravirtualization. Bridge networking is used for inter-VM IPC, as opposed to using Virtual Distributed Ethernet (VDE) networking (see Sections 4.2.2 and 4.2.4), and shared memory is used for intra-VM IPC. The comparison of MPI-vhost versus MPI-bridge helps to separate the specific performance benefit of the vhost paravirtualization. Our results below show that vhost is always a net performance win.

Finally, we use the abbreviated name *MPI-socket* for the combination of unmodified MPICH2, with the default `ch3:socket` channel, and *with* vhost paravirtualization. With MPI-socket, shared-memory IPC is no longer used for intra-VM communication. Therefore, comparing MPI-vhost and MPI-socket helps to separate the benefit of Nemesis shared memory in the `ch3:nemesis` channel.

The host operating system is Fedora 11 and the guests are Ubuntu 10.04. The host has Linux kernel version 2.6.35. The guests have Linux kernel version 2.6.36. The hypervisor is the QEMU/KVM version 0.13.5. Each VM is configured with 4 virtual central processing units (CPUs) and 4 GB of RAM. Two Nahanni device files are shared by four VMs and both of them are 4 GB (Section 4.1). All guest software, including MPICH2, the microbenchmarks, and applications, are compiled using gcc version 4.4.3.

Hardware: Our host server has two Intel Xeon X5550 processors, running at 2.67 GHz, and with 48 GB RAM. There are a total of two sockets and 8 cores. There are up to 16 HyperThreads. All benchmarking is done within the single server.

Data points: Unless otherwise noted, all data points for the microbenchmarks are the median of 11 runs, and the error bars indicate the range from maximum to minimum for those runs. We use median because of the presence of outlier data points for *some* tests. But, for many data points, the range is small and the error bars are not visible. As noted below (Section 4.2.4), the GAMESS data points are the average of 5 runs with the error bars indicating one standard deviation.

4.2.2 Summary of Results

We evaluate the latency and bandwidth performance of MPI-Nahanni using NetPIPE-MPI [28], the OSU microbenchmark [1], and IMB [22]. As well, we use the GAMESS quantum chemistry application [17]. We provide details of these benchmarks and applications below.

Although we evaluate 6 different combinations of software (e.g., VM or no VM) and communication mechanisms (summarized in Tables 4.1, 4.2), the main points of comparison are MPI-Nahanni (dark blue lines; our system), MPI-vhost (orange lines; “best” paravirtualized network), and NoVM-MPI-Nemesis (dark red lines; no VM overheads).

Overall, for microbenchmarks, MPI-Nahanni always has the highest bandwidth and lowest latency when using VM-instances, and is always close to the bandwidth and latency (often to the point of overlapping lines in the graphs) of NoVM-MPI-Nemesis (which represents the maximum achievable performance on our platform). Therefore, our conclusions are:

Two-Sided, Unidirectional Bandwidth (Figures 4.2(a), 4.2(b), 4.4(a)): MPI-Nahanni has the highest VM-based bandwidth, and nearly the same bandwidth as NoVM-MPI-Nemesis, *without* VM instances. For NetPIPE-MPI between cores on the **same** socket (Figure 4.2(a)) and messages less than 4 KB, MPI-Nahanni is no less than 100-times the bandwidth of MPI-vhost, the fastest of the paravirtualized combinations. For messages larger than 4 KB, MPI-Nahanni is at least 4-times faster than MPI-sockets, which is faster in turn than MPI-vhost. For the OSU microbenchmarks, MPI-Nahanni is between 3 and 33 times faster than either MPI-vhost or MPI-socket.

Based on our code inspection, the crossover in the relative performance between MPI-vhost and MPI-sockets is likely due to internal buffering parameters within the `ch3:nemesis` channel. These parameters are tunable at compile-time, but we have used the default buffering parameters.

With two exceptions (one bandwidth and one latency test), all of the NetPIPE-MPI and OSU microbenchmarks are performed after we pin the pair of communicating processes to cores on the **same** socket. To quantify the effect of pinning, we run one additional set of NetPIPE-MPI tests using cores on **different** sockets (Figure 4.2(b)) and show that MPI-Nahanni is between 5 and 60 times higher in bandwidth than MPI-vhost. Different-socket latencies are discussed below. Although intra-socket IPC represents a best-case scenario, it is a common scenario and the relative ordering of MPI-Nahanni's performance versus other mechanisms does not change. For the IMB, we pin four processes to four cores on two sockets (i.e., two cores per socket). For GAMESS, pinning is not used because it does not seem to have a meaningful impact on performance.

Two-Sided, Unidirectional Latency (Figures 4.3(a), 4.4(b), 4.3(b)): MPI-Nahanni has the lowest VM-based latency, and nearly the same latency as using NoVM-MPI-Nemesis, *without* VM instances. For NetPIPE-MPI between cores on the **same** socket, the latency of MPI-Nahanni is between 270 and 20 times lower than the fastest paravirtualized vhost combination, for messages between 0 and 64 KB, respectively. NetPIPE-MPI latencies for MPI-Nahanni between cores on **different** sockets are between 69 and 13 times faster than either MPI-vhost or MPI-socket. For OSU microbenchmarks, MPI-Nahanni is between 146 times and 22 times lower latency than MPI-vhost, which is the fastest paravirtualized vhost combination.

One-Sided Bandwidth (Figures 4.5(a), 4.6(a)): MPI-Nahanni has the highest VM-based bandwidth using *Get* and *Put* functions, and similar to the bandwidth of NoVM-MPI-Nemesis, *without* VM instances. MPI-Nahanni is at least 30 times higher in bandwidth than MPI-vhost for messages less than 4 KB. For messages larger than 4 KB, MPI-Nahanni has at least 3 times the bandwidth as MPI-sockets.

One-Sided Latency (Figures 4.5(b), 4.6(b)): MPI-Nahanni has the lowest VM-based latency using *Get* and *Put* functions, and similar latency to NoVM-MPI-Nemesis, *without* VM instances. MPI-Nahanni has between 132 and 13 times lower latency than the fastest paravirtualized vhost combination, for messages between 0 bytes and 64 KB.

Collective Operation Latency (Figures 4.7(a), 4.7(b), 4.8(a), 4.8(b), 4.9(a)): MPI-Nahanni has the lowest VM-based latency using *Allgather*, *Allreduce*, *Alltoall*, *Bcast*, and *Reduce* collective functions, and nearly the same latency as using NoVM-MPI-Nemesis, *without* VM instances. For *Allgather*, the latency of MPI-Nahanni is 110 to 5 times lower than the either paravirtualized vhost combination, for messages between 1 byte and 64 KB, respectively. For *Allreduce*, MPI-Nahanni has between 110 and 10 times lower latency than MPI-vhost, for messages between 1 byte and 64 KB, respectively. For *Alltoall*, although the performance gap decreases, MPI-Nahanni is still at least 60 and 4 times faster than either MPI-vhost or MPI-socket, for messages between 1 byte and 64 KB. For *Bcast*, the latency of MPI-Nahanni is between 80 and 6 times lower than MPI-vhost, for messages between 1 byte and 64 KB. For *Reduce*, MPI-Nahanni has 70 to 10 times lower latency than MPI-vhost. for messages between 1 byte and 64 KB.

GAMESS Application (Figures 4.10, 4.11): Although MPI-Nahanni is slower than NoVM-MPI-Nemesis and NoVM-MPI-socket (by between 2% and 15%), MPI-Nahanni still has the fastest VM-based running times using the *nic-ump2*, *si9h12*, *aza-es* and *carbaphos* GAMESS inputs (Figure 4.10). Note that, in contrast to the microbenchmarks, GAMESS is more compute-intensive, so the larger gap between VM and no-VM times are likely due to computational overheads [26] instead of communication overheads.

For *nic-ump2* and *aza-es* inputs using 8 processes across 4 VMs, MPI-vhost (the fastest paravirtualized vhost combination) is 34% and 75% slower than MPI-Nahanni, respectively. The *si9h12* and *carbaphos* have less MPI communication [25], which explains why MPI-vhost is only 2% and 9% slower than MPI-Nahanni, respectively.

One additional VM-based mechanism (i.e., VDE [14]) is shown in Figure 4.10 because a virtualized network such as VDE is sometimes used with Linux KVM [2]. The advantages of VDE include the ability to create a virtualized network across multiple host servers (although that is not relevant to our current work) and the ability to start up the VM-instances without requiring superuser privileges (of debatable relevance to our current work). In any case, although VDE is part of the Linux KVM toolset, our measured performance shows that it should not be used for high-performance applications. MPI-Nahanni is 3.56 and 5.47 times faster than MPI-VDE for *nic-ump2* and *aza-es* respectively. Even for *si9h12* and *carbaphos*, MPI-Nahanni is still 1.25 and 2.31 times faster as compared to MPI-VDE. We do not use VDE elsewhere in this chapter.

Figure 4.11 shows the performance as the number of MPI processes is varied from 4 to 16 for the different mechanisms, and just the *aza-es* input. MPI-Nahanni's performance remains slower than the non-VM combinations of mechanisms, and faster than other VM-based mechanisms.

Now, we examine each of the benchmarks in more detail.

Table 4.1: Benchmark configurations, with VM

Mechanism	Modified MPICH2	Unmodified MPICH2		
	MPI-Nahanni	MPI-vhost	MPI-bridge	MPI-socket
MPICH channel	`ch3:nahanni` (based on `ch3:nemesis`)	`ch3:nemesis`	`ch3:nemesis`	`ch3:socket`
VM	√	√	√	√
vhost	√	√		√
bridge network	√	√	√	√
Nemesis shmem		intra-VM	intra-VM	
Nahanni shmem	intra-VM, inter-VM			
Socket	initialization only	inter-VM	inter-VM	intra-VM, inter-VM

4.2.3 Microbenchmarks: NetPIPE-MPI, OSU, IMB

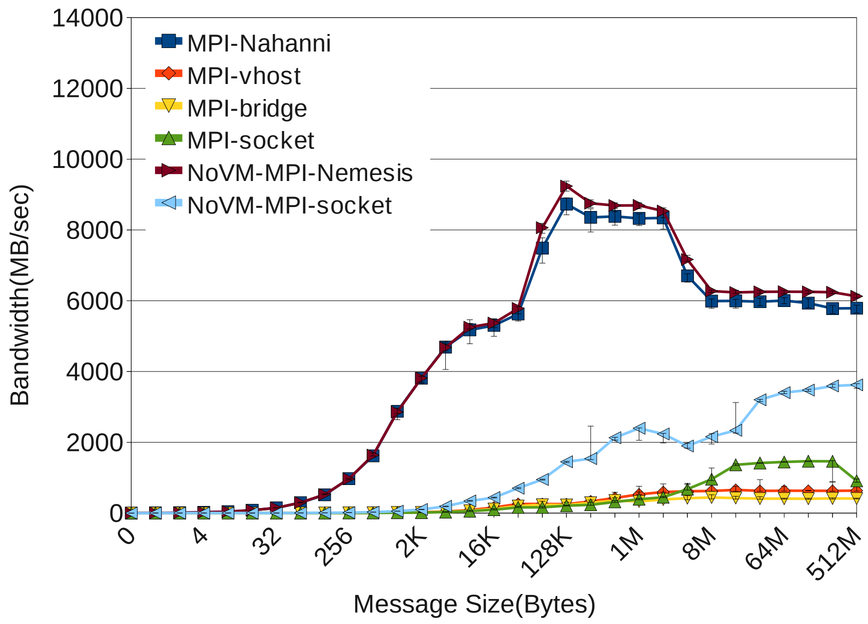
In the MPICH2-1.3 distribution [24], there is a version of the NetPIPE microbenchmark [28]. NetPIPE sends request-response messages between two nodes, and measures the bandwidth and latency with increasing message sizes. We use the benchmark as-is with one modification: the size of messages are increased by a constant multiplicative factor of two, instead of using the original step-size algorithm with perturbations. We use the term NetPIPE-MPI in this chapter to reflect the fact that our version comes from the MPICH2-1.3 distribution. As discussed elsewhere, we normally run NetPIPE-MPI with the communicating processes pinned to cores on the same socket, except for Figures 4.2(b) and 4.3(b), which use cores on different sockets.

We use the OSU microbenchmark, version 3.3, released Feb 7, 2011. For all of our tests with OSU, the communicating processes are pinned to cores on the same socket. As with NetPIPE-MPI, the OSU program measures unidirectional bandwidth and latency. Notably, the OSU microbenchmark uses *MPI_Isend* and *MPI_Irecv* whereas NetPIPE-MPI uses *MPI_Send* and *MPI_Recv*. Although changing the MPI send/receive primitives appear to affect the absolute bandwidths achieved (compare Figure 4.2(a) to Figure 4.4(a), and compare Figure 4.3(a) to Figure 4.4(b)), it does not change the partial ordering of performance, where MPI-Nahanni is always better than other VM-based mechanisms. The OSU microbenchmark also tests one-sided operations, including *MPI_Get* and *MPI_Put*.

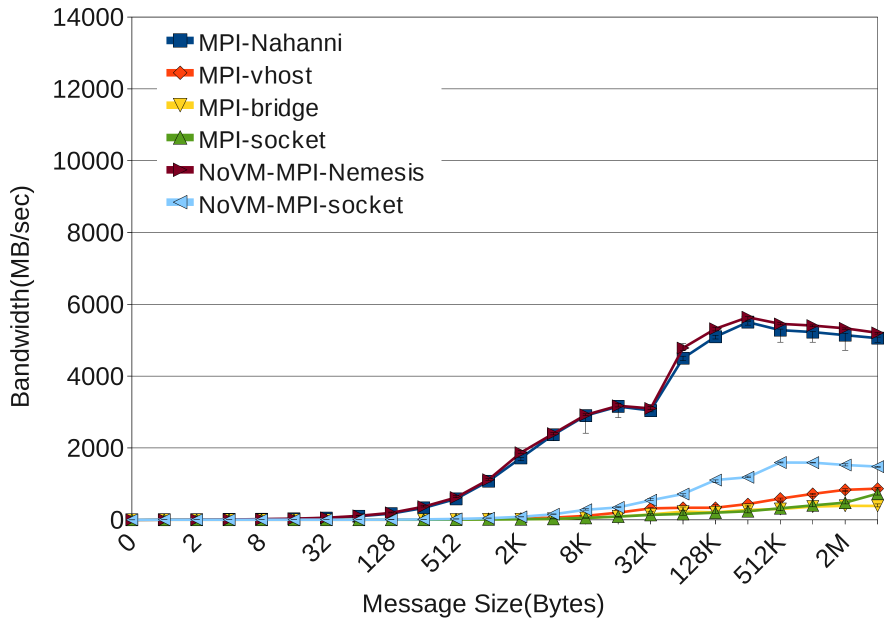
Lastly, we use the Intel MPI Benchmark (IMB) (version 3.2 update 2) to evaluate the perfor-

Table 4.2: Benchmark configurations, **without** VMs.

Mechanism	Unmodified MPICH2	
	NoVM-MPI-Nemesis	NoVM-MPI-socket
MPICH channel	`ch3:nemesis`	`ch3:socket`
Nemesis shmem	data	
Socket	initialization	all IPC

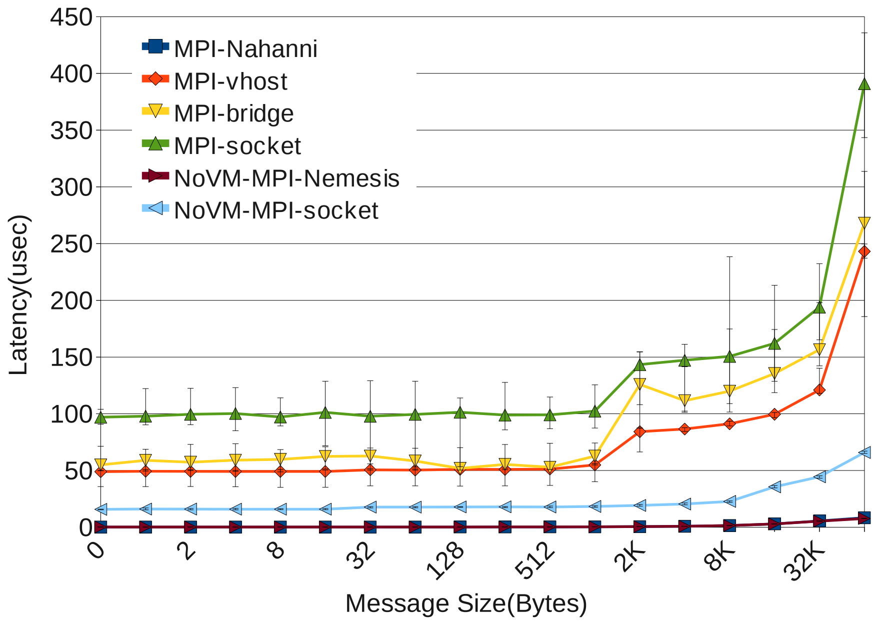


(a) NetPIPE-MPI Unidirectional Bandwidth between cores on **same** socket.

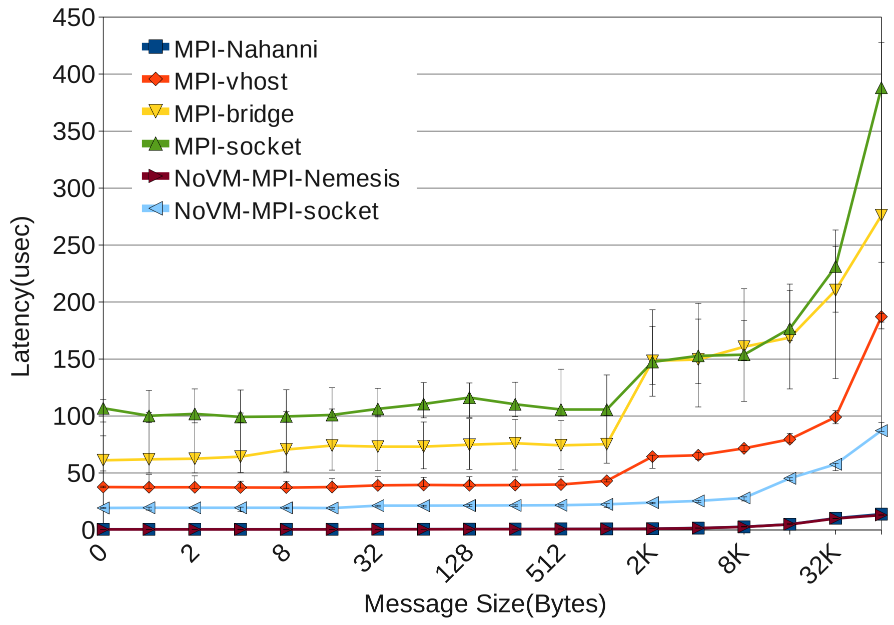


(b) NetPIPE-MPI Unidirectional Bandwidth between cores on **different** sockets.

Figure 4.2: NetPIPE-MPI Unidirectional Bandwidth on **same** and **different** sockets.

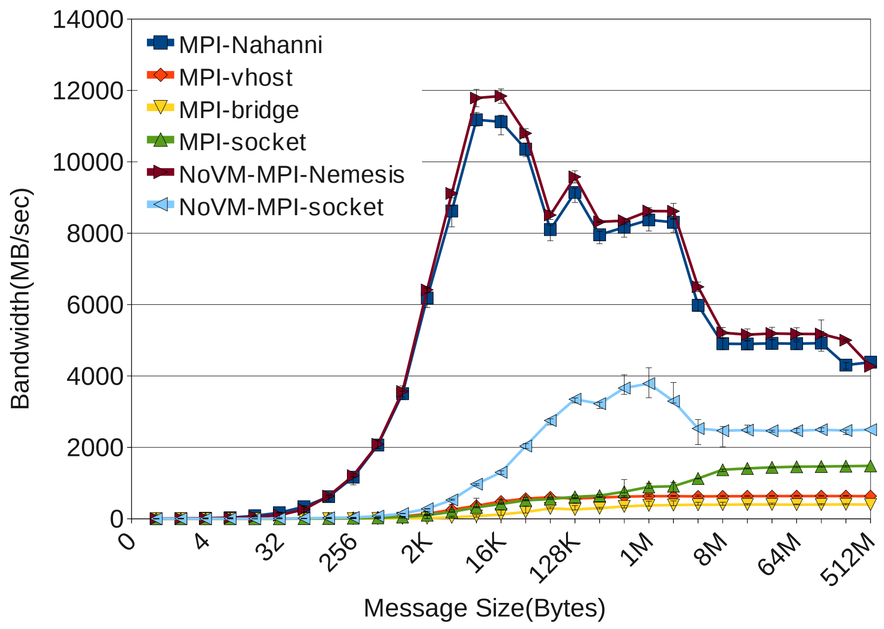


(a) NetPIPE-MPI Unidirectional Latency between cores on **same** socket.

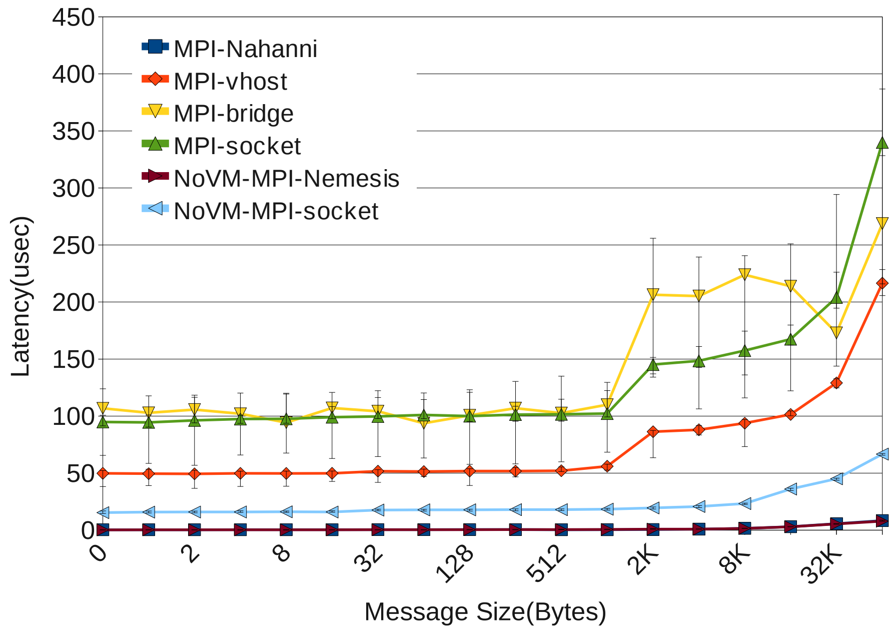


(b) NetPIPE-MPI Unidirectional Latency between cores on **different** sockets.

Figure 4.3: NetPIPE-MPI Unidirectional Latency on **same** and **different** sockets.

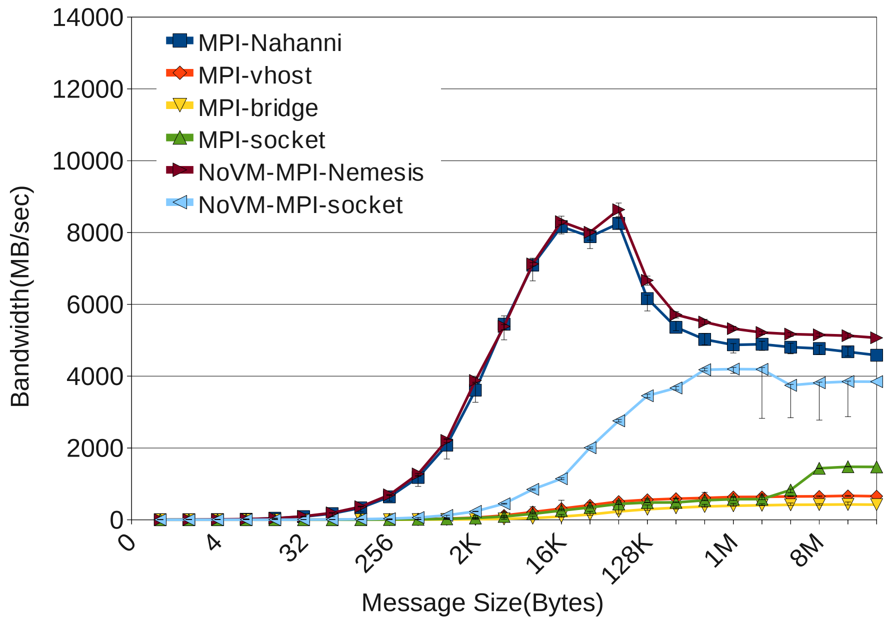


(a) OSU Unidirectional Bandwidth between cores on **same** socket.

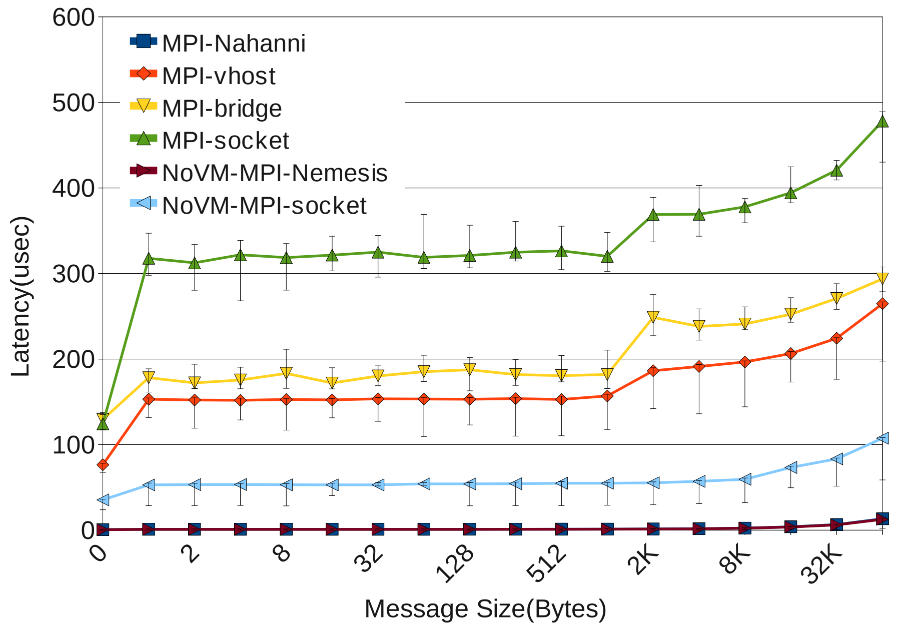


(b) OSU Unidirectional Latency cores on **same** socket.

Figure 4.4: OSU Unidirectional Bandwidth and Latency.

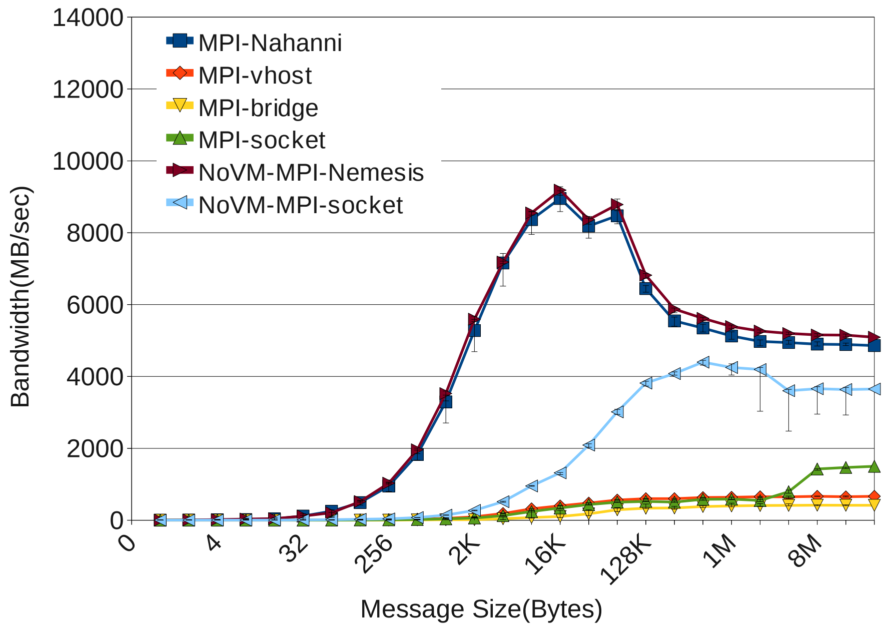


(a) OSU *Get* Bandwidth, cores on **same** socket.

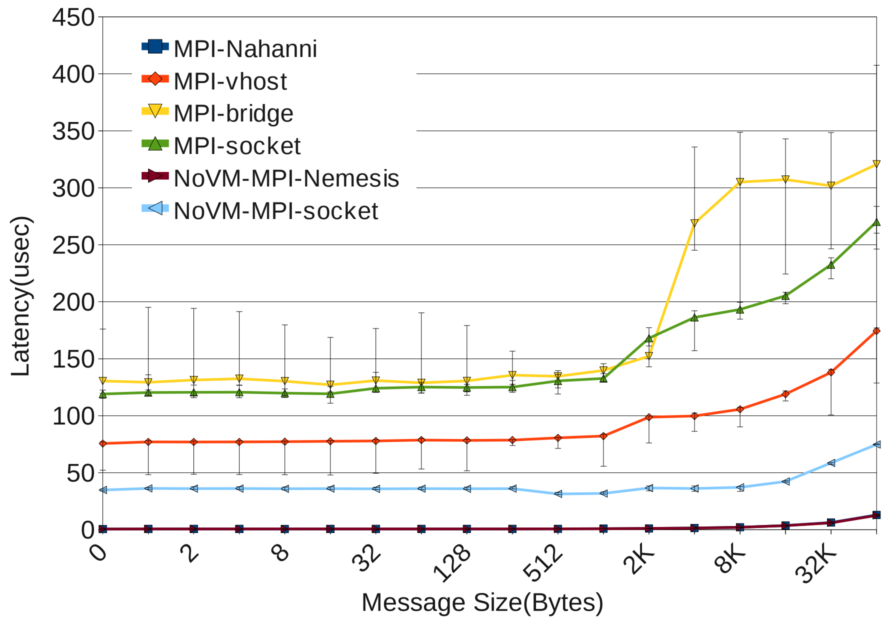


(b) OSU *Get* Latency, cores on **same** socket.

Figure 4.5: OSU Unidirectional *Get* Bandwidth and Latency.



(a) OSU *Put* Bandwidth, cores on **same** socket.



(b) OSU *Put* Latency, cores on **same** socket.

Figure 4.6: OSU Unidirectional *Put* Bandwidth and Latency.

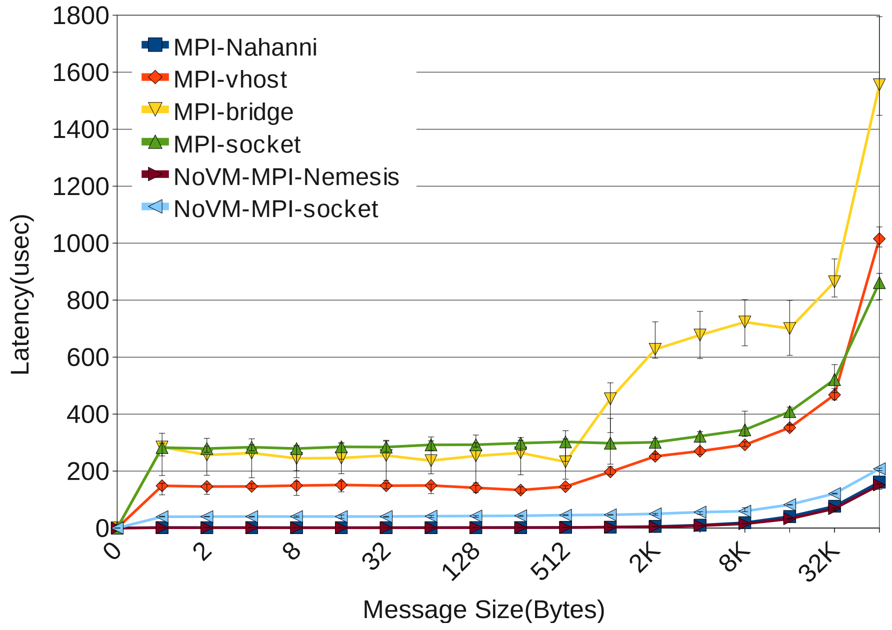
mance of MPI collective operations (Figure 4.7(a), 4.7(b), 4.8(a), 4.8(b), 4.9(a)). When we ran the IMB, we chose a communicator size of four, which requires either four VMs, or four MPI processes in the non-VM cases. All of the processes of a given VM are pinned to cores on one socket. Therefore, among the four VMs (or the 4 processes in the non-VM case), there will be both intra-socket and inter-socket communication, given our two socket hardware and four-way collective operations.

4.2.4 GAMESS: Quantum Chemistry Application

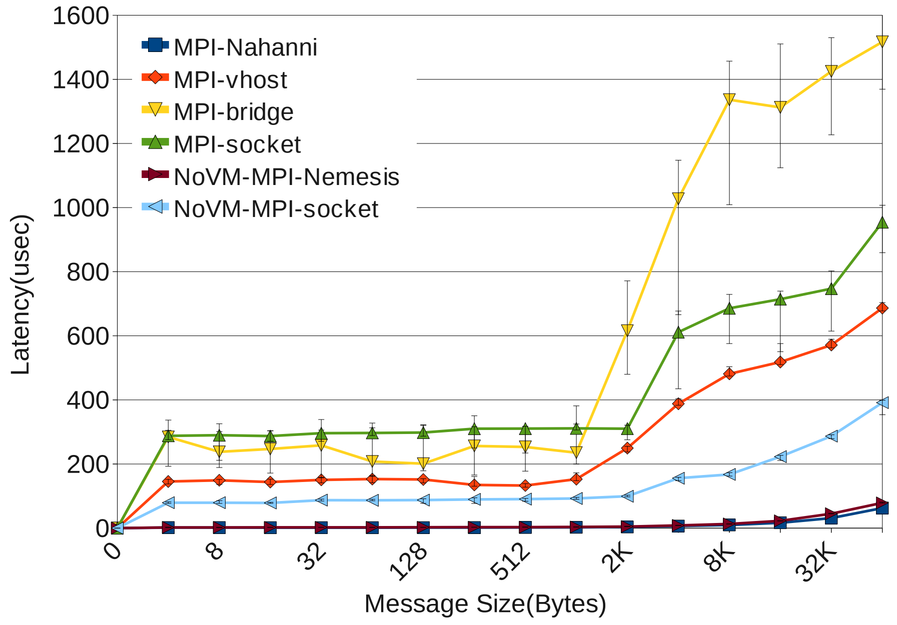
To broaden our evaluation, we tested MPI-Nahanni and the other mechanisms with a well-known computational science application: the General Atomic and Molecular Electronic Structure System (GAMESS). The availability of source code aids our benchmarking (although we did not modify the code) and facilitated the inclusion of GAMESS in the SPECfp2006 benchmark suite. We used four inputs (which are molecules) provided by a computational chemist colleague, namely *nicump2*, *si9h12*, *carbaphos* and *aza-es*. Note that we do not bind processes to cores when running the GAMESS tests. In all our tests, four VMs were used, even as we varied the number of MPI processes running within the four VMs. Our GAMESS results are the average of five runs, with a small standard deviation indicated by the error bars.

An important implementation detail of GAMESS is that it will automatically use System V shared memory for most data transfers between its own application processes, if those processes are running on the same VM or host. Therefore, MPI-Nahanni is used for inter-VM communication and GAMESS's own shared-memory-based mechanism is used for intra-VM communication. Also, for the no-VM cases, most of the communication is via System V shared memory, instead of over MPI. This explains why there is little difference between NoVM-MPI-Nemesis and NoVM-MPI-socket (Figures 4.10, 4.11). Keeping in mind that our hardware has 8 cores, the 16-MPI-process data points of Figure 4.11 are overloading the cores (but using HyperThreads), and the largest gap between NoVM-MPI-Nemesis and NoVM-MPI-socket for that data point is likely due to the use of polling within NoVM-MPI-Nemesis (an implementation detail), which has a greater detrimental effect in this overloaded case.

As discussed earlier, we show an additional mechanism for the GAMESS tests, namely the VDE system, labelled as MPI-VDE. Until we completed our benchmarking, it was unclear how VDE affects performance. We had been using VDE because it does not require superuser privileges to start the networking for the VMs, because it supports easy inter-host network virtualization, and because it provides a convenient network address translation (NAT) facility for non-high-performance-computing (HPC) applications such as Web servers. After our benchmarking, it is clear that the current version of VDE is not high performance enough for HPC applications. We include the VDE data points for completeness, but we have also been using NoVM-MPI-Nemesis and MPI-vhost as our main points of comparison throughout this chapter, so as to not exaggerate the benefits of MPI-Nahanni.

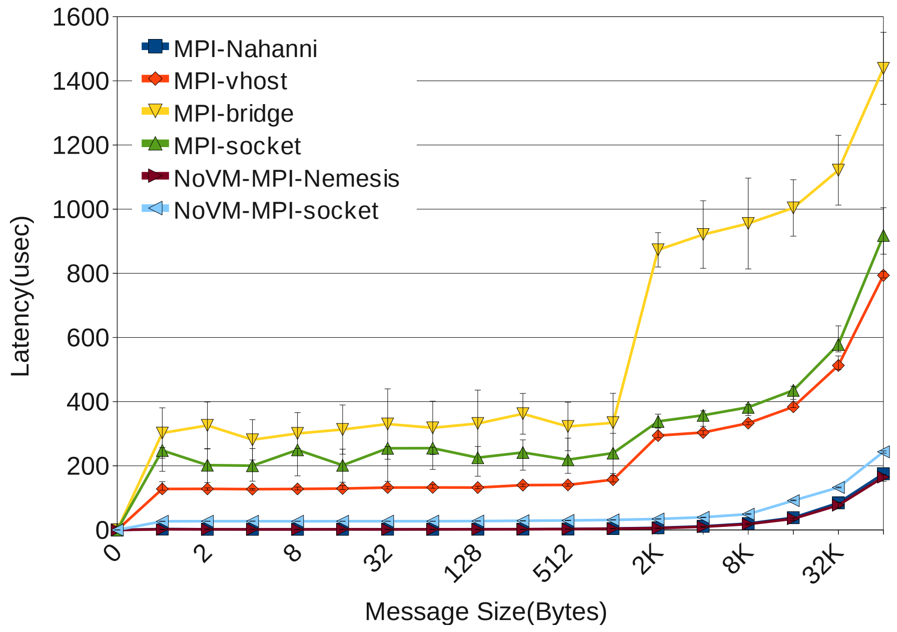


(a) Intel MPI benchmark *Allgather* Latency.

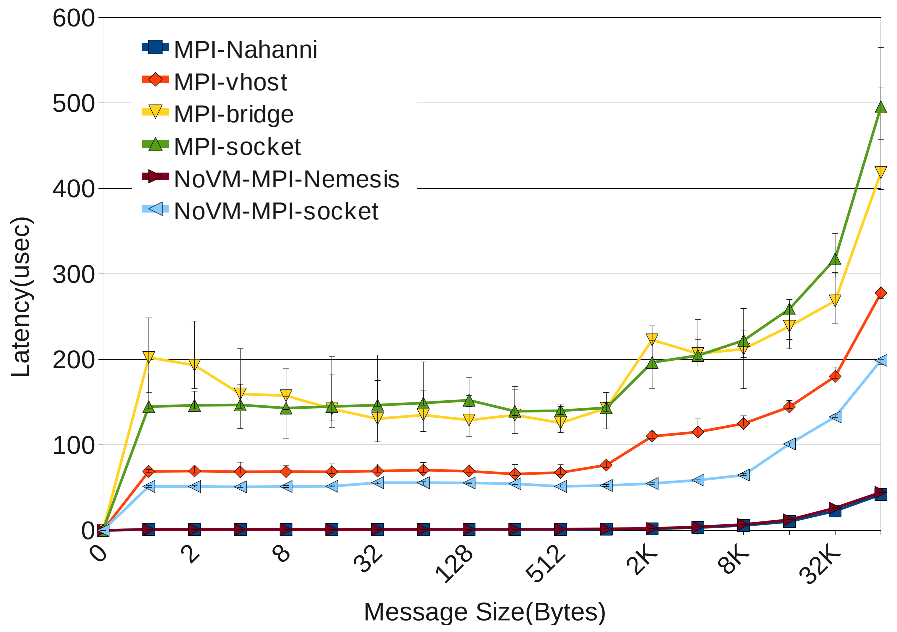


(b) Intel MPI benchmark *Allreduce* Latency.

Figure 4.7: Intel MPI Benchmark (IMB) Collective Operations I

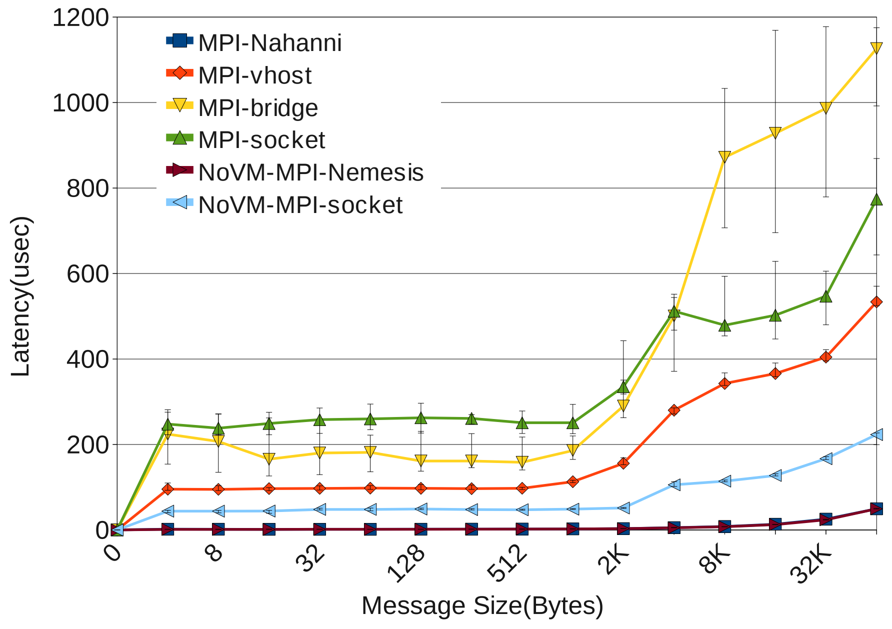


(a) Intel MPI benchmark *Alltoall* Latency.



(b) Intel MPI benchmark *Bcast* Latency.

Figure 4.8: Intel MPI Benchmark (IMB) Collective Operations II



(a) Intel MPI benchmark *Reduce* Latency.

Figure 4.9: Intel MPI Benchmark (IMB) Collective Operations III

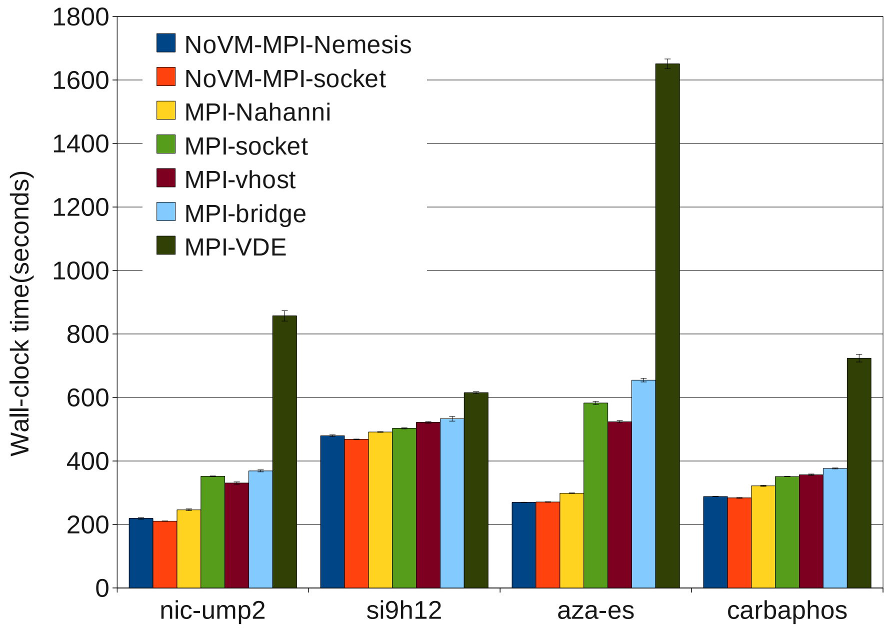


Figure 4.10: GAMESS: Wall-clock times, four inputs/test cases, 8 processes across 4 VM instances.

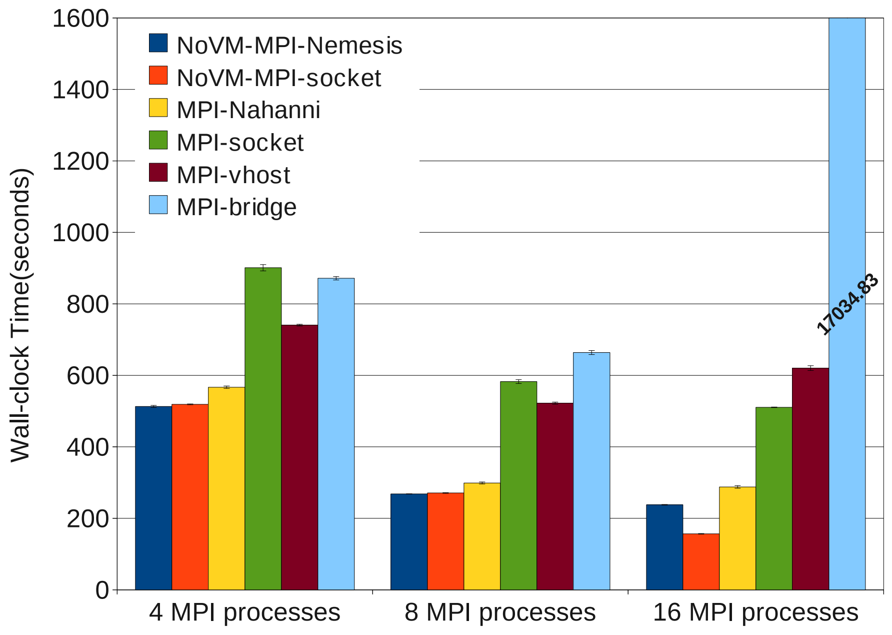


Figure 4.11: GAMESS: Wall-clock times, varying number of processes, input *aza-es*, excluding VDE mechanism.

Overall, not surprisingly, the amount MPI-based communication varies from input-to-input, with the performance benefit of MPI-Nahanni correlating closely to the amount of time each run actually spends within MPI functions [25]. Our main conclusion is that the performance benefits shown from the microbenchmarks do translate into application-level performance benefits.

4.3 Concluding Remarks

We demonstrate that Nahanni can also support existing programming models and APIs by presenting the design, implementation, and evaluation of MPI-Nahanni. MPI-Nahanni is an MPI implementation based on MPICH2's Nemesis channel for shared memory and Nahanni's support for inter-VM shared memory. Since Nahanni shared memory behaves like traditional shared memory, we also discussed how the port of MPICH2 to Nahanni is straightforward and most of the existing code in MPICH2-Nemesis can be reused.

In the empirical evaluation, we observe that the VM overheads for MPI-based program can be greatly reduced using the Nahanni shared-memory mechanism. Through a set of microbenchmarks (NetPIPE,OSU,IMB), we have shown substantial performance advantages of MPI-Nahanni over other 3 existing VM-based IPC techniques (e.g., Figure 4.2). Moreover, for many of the benchmarks, MPI-Nahanni closely tracks the performance of NoVM-MPI-Nemesis, which does not use VMs at all (e.g., Figure 4.4). Also, the results from a full-sized application, such as GAMESS, are promising. They show that MPI-Nahanni is usually faster than other existing VM-based IPC techniques (e.g., MPI-vhost) and close to NoVM-MPI-Nemesis (e.g., Figure 4.10). Conceptually, since NoVM-MPI-Nemesis does not use VMs, it represents an upper bound on performance on our hardware platform, and MPI-Nahanni's performance is usually close to that upper bound.

Chapter 5

Concluding Remarks

In this thesis, we presented the design and implementation of two interprocess communication (IPC) mechanisms on top of Nahanni inter-virtual machine (VM) shared memory under the QEMU/Linux Kernel-based Virtual Machine (KVM) environment. By implementing minitransactions and porting MPICH2-Nemesis to use inter-VM shared memory, we demonstrate that Nahanni can support both emerging programming models and existing application programming interfaces (API) and models.

We first investigated and implemented an emerging IPC mechanism, minitransactions (Section 3.1), which has the potential for supporting both shared memory and distributed memory transparently. And as compared with traditional load-store and message-passing programming models, minitransactions offer useful abstraction and semantic benefits (e.g., helping developers deal with the complicated concurrency introduced by parallel programs and shared-memory data sharing). We also implemented inter-VM-based barrier synchronizations on top of Nahanni, which was not available prior to this thesis. In the increment-counter benchmark, we showed that minitransactions have better performance (e.g., at least 1.5 times faster) than the lock-based synchronization (Section 3.3.3) for low-contention scenario. To evaluate our VM-based barrier performance, we also ported and parallelized the FLUIDS simulator with our barrier synchronization. Both the barrier microbenchmark (Section 3.3.2) and FLUIDS application benchmark (Section 3.3.4) showed that our spin-based barrier implementation has good performance and negligible overheads as compared to barriers outside of VMs.

We also studied a mature and widely used IPC interface, MPICH2, which is an open-source implementation of the Message-Passing Interface (MPI) standard. We implemented MPI-Nahanni, a new communication channel for MPICH2, which is a port of MPICH2-Nemesis to Nahanni inter-VM shared memory. Moreover, since Nahanni shared memory behaves like familiar shared memory, we discussed how the port of MPICH2-Nemesis is straightforward, and the existing MPI programs can run with MPI-Nahanni directly without any code modifications. To evaluate the benefits of using Nahanni shared memory, we used various microbenchmarks (e.g., NetPIPE, OSU microbenchmark, Intel MPI benchmark (IMB)) and an application benchmark (e.g., General Atomic and Molecular Electronic Structure System (GAMESS)) (Section 4.2). The microbenchmarks showed that both the bandwidth and latency of MPI-Nahanni are at least one order of magnitude better than other state-of-art VM-based IPC mechanisms. Finally, the GAMESS application benchmark gave us a promising result of using MPI-Nahanni in co-located VMs: MPI-Nahanni outperformances other VM-based IPC methods, with the performance benefits proportional to the percent of time spent in MPI-related functions.

Bibliography

[1] MVAPICH2 Project/OSU Micro-Benchmarks. `http://mvapich.cse.ohio-state.edu/benchmarks/`.

[2] Setting Guest Network. `http://www.linux-kvm.org/page/Networking`.

[3] UsingVhost. `http://www.linux-kvm.org/page/VhostNet`.

[4] K. Adams and O. Ageson. A comparison of software and hardware techniques for x86 virtualization. In *in ASPLOS-XII: Proceedings of the 12th international conference on Architectural*, pages 2–13. ACM Press, 2006.

[5] M. K. Aguilera, W. Golab, and M. A. Shah. A practical scalable distributed b-tree. *Proc. VLDB Endow.*, 1:598–609, August 2008.

[6] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 159–174, New York, NY, USA, 2007. ACM.

[7] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[8] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[9] W. Bruijn and H. Bos. Beltway buffers: Avoiding the os traffic jam. In *INFOCOM 2008*, 2008.

[10] J. C. Brustoloni and P. Steenkiste. Effects of buffering semantics on i/o performance. In *OSDI*, pages 277–291, 1996.

[11] D. Buntinas, G. Mercier, and W. Gropp. Design and evaluation of nemesis, a scalable, low-latency, message-passing communication subsystem. In *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '06, pages 521–530, Washington, DC, USA, 2006. IEEE Computer Society.

[12] D. Buntinas, G. Mercier, and W. Gropp. Implementation and evaluation of shared-memory communication and synchronization operations in MPICH2 using the nemesis communication subsystem. *Parallel Comput.*, 33:634–644, September 2007.

[13] A. Burtsev, K. Srinivasan, P. Radhakrishnan, L. N. Bairavasundaram, K. Voruganti, and G. R. Goodson. Fido: fast inter-virtual-machine communication for enterprise appliances. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, pages 25–25, Berkeley, CA, USA, 2009. USENIX Association.

[14] R. Davoli. VDE: Virtual Distributed Ethernet. In *First International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities*, pages 213–220. IEEE, 2005.

[15] F. Diakhaté, M. Pérache, R. Namyst, and H. Jourdren. Efficient shared memory message passing for inter-vm communications. In *4th Workshop on Virtualization in High-Performance Cloud Computing (VHPC '08), Euro-par 2008*, 2008.

[16] P. Druschel and L. L. Peterson. Fbufs: a high-bandwidth cross-domain transfer facility. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 189–202, New York, NY, USA, 1993. ACM.

[17] M. S. Gordon and M. W. Schmidt. Advances in electronic structure theory: GAMESS a decade later. In C. E. Dykstra, G. Frenking, K. S. Kim, and G. E. Scuseria, editors, *Theory and Applications of Computational Chemistry: the first forty years*, pages 1167–1189. Elsevier, Amsterdam, 2005.

[18] T. Harris, A. Cristal, O. S. Unsal, E. Ayguade, F. Gagliardi, B. Smith, and M. Valero. Transactional memory: An overview. *IEEE Micro*, 27:8–29, May 2007.

[19] R. Hoetzlein. Fluids: A fast, open source, fluid simulator. http://www.rchoetzlein.com/eng/graphics/fluids.htm.

[20] W. Huang and D. K. Panda Koop, M. J.and Panda. Efficient one-copy mpi shared memory communication in virtual machines. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing, 29 September - 1 October 2008, Tsukuba, Japan*, pages 107–115, 2008.

[21] W. Huang, M. J. Koop, Q. Gao, and D. K. Panda. Virtual machine aware communication libraries for high performance computing. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 9:1–9:12, New York, NY, USA, 2007. ACM.

[22] Intel Corporation. Intel MPI Benchmarks. `http://software.intel.com/en-us/articles/intel-mpi-benchmarks/`.

[23] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *In OLS '07: The 2007 Ottawa Linux Symposium*, pages 225–230, 2007.

[24] Argonne National Laboratory. Measuring nemesis performance. `http://wiki.mcs.anl.gov/mpich2/index.php/Measuring_Nemesis_Performance`.

[25] C. Macdonell. *Nahanni: Fast Communication for the Linux Kernel Virtual Machine*. PhD thesis, Department of Computing Science, University of Alberta, 2011.

[26] C. Macdonell and P. Lu. Pragmatics of virtual machines for high-performance computing: A quantitative study of basic overheads. In *Proceedings of High Performance Computing & Simulation Conference (HPCS'07)*, 2007.

[27] R. Russell. virtio: towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, 2008.

[28] Q. O. Snell, A. R. Mikler, and J. L. Gustafson. Netpipe: A network protocol independent performance evaluator. In *in IASTED International Conference on Intelligent Information Management and Systems*, 1996.

[29] Various. Mailing list discussion on inter-vm shared memory. `https://patchwork.kernel.org/patch/24681/`, 2009.

[30] J. Wang, K. Wright, and K. Gopalan. Xenloop: a transparent high performance inter-vm network loopback. In *Proceedings of the 17th international symposium on High performance distributed computing*, HPDC '08, pages 109–118, New York, NY, USA, 2008. ACM.

[31] A. Wolfe Gordon. Enhancing cloud environments with inter-virtual machine shared memory. Master's thesis, University of Alberta, 2011.

[32] A. Wolfe Gordon and P. Lu. Low-Latency Caching for Cloud-Based Web Applications. In *Proceedings of the 6th International Workshop on Networking Meets Databases (NetDB '11)*, Athens, Greece, 2011.

[33] X. Zhang, S. McIntosh, P. Rohatgi, and J. L. Griffin. Xensocket: a high-throughput interdomain transport for virtual machines. In *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, Middleware '07, pages 184–203, New York, NY, USA, 2007. Springer-Verlag New York, Inc.